Explorations in Asynchronous Teams

Sanjay Sachdev

A Dissertation
Submitted to the Graduate School
in Partial Fulfillment of the Requirements
for the Degree

Doctor of Philosophy
in
Electrical And Computer Engineering

Carnegie Mellon University
Pittsburgh, Pennsylvania

December 3, 1998

Copyright 1998 by Sanjay Sachdev

All Rights Reserved

ACKNOWLEDGEMENT

There are many people whose support was critical while I served my time in Graduate School.

My parents and siblings.

Uncles, aunts, cousins and other assorted relatives.

Friends, especially those who knew me when I was older than I am now.

Advisors.

Colleagues and committee members.

Faculty and staff.

Others, including funding agencies, who may not fall into one of the above categories.

You know who you are.

You may even belong to more than one category.

I won’t name names so as to protect the innocent.

Thank you!

ABSTRACT

The subject of this thesis is the A-Teams formalism. This formalism facilitates the organization of multiple algorithms, encapsulated as autonomous agents, into cooperating teams to solve difficult problems. The A-Teams formalism is one of many agent-based systems, and I start by providing a taxonomy of agent-based systems that allows us to see they how A-Teams relate to other agent-based systems.

A-Teams are constructed from memories that store solutions and agents that work on those solutions. A-Teams are open to the addition of new memories as well as of new agents. Sets of memories and agents can also be combined in different ways to create a variety of customized A-Teams. As new memories and agents are created, they can be added to existing repositories and reused for future applications. The automatic construction of problem-specific custom A-Teams from repositories of components has been a long standing goal of research in A-Teams. Current guidelines for A-Team construction require human interpretation. In the next section I develop a formal generative grammar for the construction of A-Teams.

A-Teams have been successfully used for many kinds of optimization problems. I provide a short overview of optimization as in introduction to the subject. Most optimization problems can be composed of problem primitives, which are subsets of the objectives and constraints of the problem. I continue by showing how specialized agents can be constructed for problem primitives and then combined in A-Teams to solve optimization problems. These specialized agents can be reconfigured in a modular fashion to compose task-specific solvers. Such a solver can be easily adapted for changes in the mix of objectives and constraints by modifying the set of specialized agents in the A-Team.

There are many optimization problems that require the pattern recognition skills of human beings as well as the numeric abilities of computers. However, humans and computers work at very different speeds. I show that, in spite of these speed differences, human beings can work as agents in A-Teams along with computer agents and the resulting A-Teams exhibit a synergy between the human and computer in solving such optimization problems.

Agents in A-Teams are of two types: constructors that add new solutions to memories and destroyers that remove undesirable solutions from memories. I show that A-Teams not only benefit from the addition of constructors but also from the addition of destroyers. Current destroyers use the quality of solutions to identify undesirable solutions. I conclude by demonstrating that destroyers that prune clusters in decision space to force exploration, and destroyers that reduce cycling in decision space by using historical data, can be used to assist the current quality oriented destroyers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The Asynchronous Teams (A-Teams) formalism is a powerful problem solving mechanism that provides a way to easily combine various algorithms and use a variety of solution representations to solve difficult computational problems [Talukdar98]. A-Teams have been developed to solve a variety of problems in design, diagnosis, control and optimization. Applications of A-Teams include nonlinear equation solving [TPG83], traveling salesperson problems [deSouza93], protein folding [LGTH97], high rise building design [Quadrel91], reconfigurable robot design [Murthy92], diagnosis of faults in electric networks [Chen92, AT96], control of electric networks [TR93], job shop scheduling [CTS93], steel [LMHM95] and paper mill scheduling [RWMTSAFAYHJ96], train scheduling [Tsen95], and constraint satisfaction [GHSTM96]. Not only do these asynchronous teams find good solutions, but they appear to be scale effective: solution quality and speed usually improve with the addition of agents and computers. This thesis extends our knowledge and understanding of A-Teams.

The A-Teams formalism is one of many agent-based systems and we would like to be able to compare A-Teams with the other systems. There has been considerable recent interest in agent-based systems and the literature is confusing because researchers use a variety of terminology for similar concepts. I identify existence, environment, behavior and structure of agents as the key concepts common to almost all agent research, and show how these concepts relate to the terminology used in the literature. I will provide a brief history of agent-based systems to identify their progenitors, the fields of artificial intelligence and distributed problem solving. I will explain why I believe that autonomy is the key characteristic of agents. I classify agent-based systems by: the characteristics of the agents in the system; the way the agents are organized; the way in which they interact with each other; and the applications for which the systems are used. I also use this taxonomy to compare the various agent-based systems with one other.

A-Teams have a modular structure and their component agents and memories can be organized in many ways to create “organizations-on-demand” that are customized for given tasks [Talukdar98]. Almost all this work is currently done by hand, and the construction of A-Teams is dependent on individual human interpretation of the guidelines provided. I develop a formal grammar that defines the atomic components of A-Teams and presents rules for combining them. This brings us closer towards the goal of automated construction of task-specific A-Teams from repositories of components, and provides more rigorous formal definitions of terms and ideas that have, so far, only been defined qualitatively.

A-Teams have been used to solve a variety of complex problems. Complex problems, containing many objectives and constraints, can be constructed from problem primitives which consist of small subsets of the objectives and constraints. I will show how complex problems can be solved by sets of specialized agents that handle the primitives of the problem. Such specialized agents can collaborate in A-Teams by taking small steps through decision space, sharing recommendations or models of constraints, or working on over-

lapping primitives. The advantages of such an approach are that: the agents are usually easier to design than algorithms that directly solve the entire complex problem; these agents can be reused for various problems that share problem primitives; and since the A-Teams consisting of such agents are modular they can be easily reconfigured in response to changes in problem requirements. I will show how such modular specialized agent A-Teams can be designed to cooperate by using ad-hoc methods or by transforming good existing iterative algorithms. I will also provide experimental evidence to show that such an approach works well for solving problems in nonlinear programming and problems in spatial layout.

A useful feature of A-Teams is that they are open to the addition of agents which makes them easy to extend. These agents have, so far, all been computer based. We would also like to be able to add humans as agents in A-Teams because they have skills that complement those of computer agents. However, humans and computers work at very different speeds. I will show that the relative speeds of agents in an A-Team may be less important than the diversity of skills they provide and will present a set of experiments that confirm the hypothesis that humans and computers can collaborate effectively in A-Teams. I provide a set of guidelines for effective human-computer collaboration based on my experiences performing these experiments.

The speed of an agent is related to its work cycle, which is the time it takes to perform a single iteration. The work cycle limits the agent's frequency of operation. The frequency of operation is also controlled by a self-contained scheduler that determine how often the agent works. I will present some results that indicate that the scheduling of agents may not affect performance significantly. These results increase our knowledge of the role of scheduling, which has received little attention until now.

The focus of most of this thesis is construction agents which generate new solutions. The last set of experiments I present in this thesis shows that diversity of agent skills is beneficial not only for construction, but also for the destruction or removal of undesirable solutions. Destruction agents usually eliminate poor quality solutions [Talukdar98]. I show how destruction agents based on the optimization techniques of clustering [TZ89] or tabu search [Glover89] can be combined with destruction agents based on the traditional technique of quality-based destruction to help A-Teams find better solutions.

2. AGENT-BASED SYSTEMS

This subject of this thesis is the A-Teams formalism that is used for a range of problems in optimization, diagnosis, control and design. A-Teams are one class of agent-based systems (ABS). Agent-based systems provide us with a variety of alternative ways to perform tasks using computers. Unfortunately, the literature on agent-based systems suffers from a confusing plethora of terminology. This chapter remedies the problem by providing a distillation of the important concepts along with a list of the commonly used terms for each. This chapter also provides an overview of several such systems and develops a taxonomy to show the relationship between the various systems with a focus on A-Teams.

2.1. A Brief History of Agent-based Systems

The design of software as agent-based systems is often referred to as agent oriented programming (AOP). The significance of the AOP paradigm in software is best understood in the context of the history of computer hardware: speed has increased and cost has decreased almost exponentially; computers have been transformed from large, expensive machines isolated in special climate controlled rooms, protected and controlled by a class of high priests to small highly interconnected ubiquitous devices; and mainframes have given way to networks. Software design has been forced to change as a result.

In spite of the phenomenal growth in hardware capability, we have never had trouble pushing it to its limit by increasing the size and complexity of computer software. Software engineering, the art and science of software development, has struggled to manage this increase in size and complexity. Programming styles and languages have evolved in steps, with each step increasing the level of abstraction to isolate the programmer from some of the complexity. Each step also increases the level of modularization to facilitate software reuse. There is almost always a price to be paid in efficiency for the advantages of abstraction, but it is usually a small price to pay. AOP is the latest step in this evolutionary process.

2.1.1. From Machine Language to Agents

The earliest computers were directly programmed in machine languages consisting of the bits and bytes that the computer executed directly. The process of software evolution started with the invention of assembly language which used mnemonics to represent machine language instructions and allowed for labels in place of memory addresses. This made programming easier for humans by clearly differentiating between instructions, data, and memory addresses in code. It also simplified code reuse. Although we almost never program directly in machine languages today, we still occasionally use assembly languages to get more optimized code.

Assembly languages are very close to the machine languages of computers and therefore each hardware architecture must have its own, unique, assembly language. This makes it difficult to port assembly language code from one platform to another. The next step in software evolution enabled portability by abstracting out

platform specific features and introduced modularity in the form of procedures, or small self-contained sections of code that larger procedures and programs were constructed from. Procedural languages specify a sequence of steps the program must perform upon data.

Data structures may be shared and processed by many procedures. In procedural languages each procedure must deal directly with the atomic elements that constitute the structure which often makes modifications to the structure difficult. Changes in the structure may result in changes in many procedures: the programmer must track every procedure that operates on the data and determine whether and how to change it. Object oriented programming eliminates a large number of these problems by shifting the focus to the data. Each data structure, or object, is associated with a set of procedures, called the interface, by which the programmer interacts with it. Provided the interface is not changed, the underlying implementation of the procedures and the data representation itself may be changed without requiring any modifications to the remainder of the code. This adds a level of abstraction that simplifies reusability of code. For example, code that operates on real numbers can easily be modified for complex numbers provided an appropriate interface for the arithmetic operators is provided.

All of the above evolutionary steps augmented the then existing paradigm with new concepts that changed the way that programs were structured. The concepts introduced at each step, mnemonics and labels, procedures, and data encapsulation, added new levels of modularity that facilitated code reuse. The older stage was not eliminated but subsumed in the new one which added a higher level of abstraction.

Agent oriented programming (AOP) provides yet another level of abstraction. The concept behind AOP is that of autonomy and a fixed inter-agent communication interface. Just as in object oriented programming there are strict interfaces for objects that allow flexibility in implementation and data representation, in agent oriented programming there are strict interfaces for autonomous program modules (agents) that may execute in parallel that allow flexibility in the implementation of the agents. Some approaches to AOP and OOP may resemble each other because they both use message-based interfaces. However in OOP the message semantics are object dependent, whereas in AOP they are agent independent.

My definition of an agent, which applies to almost all agent-based systems, is:

Agent: A piece of software that executes with significant autonomy and has a fixed interface with its environment.

where:

Autonomy: is the extent to which an agent controls its own actions.

Interface: is the way in which the agent obtains input from and generates output to the environment.

Environment: is everything, including other agents, that the agent interacts with that is not part of the agent itself.

Note that in order to be completely autonomous, an agent must stay active, looking for events to respond to, and decide for itself what it should do. In contrast, most current stand-alone software has an extremely low degree of autonomy: it is started by a human user, used for a specific task such as displaying email, and then terminated. I do not consider such a piece of software to be an agent.

Transforming software into agents can help to integrate existing programs into new applications or to enhance the functionality of systems by organizing agents in different ways. Provided that an agent's interface to its environment does not change, it can still be used in future applications long after the machine it runs on or the language it is written in becomes obsolete. Of course it might eventually become necessary to port the agent to new platforms once the legacy system becomes relatively too slow.

An important benefit of AOP is that it fits well with computer networks. Although the term "agent oriented programming" is fairly recent, in fact AOP was invented along with networked computing. I classify as agents all daemons that manage email, news, and shared resources such as printers. Agent-based systems also facilitate the sale of services across networks, in the form of data processing, in addition to the sale of code. Problems can be solved using agents that reside on machines that are distributed over the web, and customers can rent processing time rather than buying different pieces of software and the processors to run them on. Providers are already starting to make computational services available that process and return results for data provided by the users. Examples of such services include the many agent-based systems for information searches on the web such as HotBot or Yahoo and agents for optimization problems that can be found in [Neos]. Customers benefit from renting by being able to access the latest hardware and software, and by paying only for the resources they use rather than having to buy entire systems. Providers benefit by being able to protect their code from piracy. Another potential benefit of AOP is that a wide variety of software agents, having heterogeneous structures and running on different platforms, can be organized to collaborate on tasks.

2.1.2. The Growth of Agent-based Systems

Current agent-based systems have developed from research in several areas, each of which has developed its own concepts of agency (which is a word commonly used in the literature to mean the quality of being an agent). The two most important areas that have contributed to software agents are distributed problem solving (DPS) and artificial intelligence (AI). Agent-based systems that have evolved from DPS emphasize numeric and symbolic problem solving in areas such as optimization, diagnosis and control. Agent-based systems that have evolved from distributed AI (DAI) and parallel AI (PAI) emphasize negotiation and anthropomorphism.

An unfortunate result of recent interest in agent-based systems is that the term “agent” has become a buzzword that is widely abused. There is great diversity in the conceptual perspectives taken by researchers and considerable disagreement over terminology. Surveys of software agents with an AI bias such as [WJ95, Nwana96, Petrie96] agree that the term is often misused and concur that it is impossible to create a single universally accepted definition. The surveys also note a great degree of hype in the literature, and recommend that readers be very cautious about claims made. An excellent example of such content-free hype is [Chorafas98], in which the author proudly quotes a pre-publishing reviewer of the book as complaining that he “did not find between the pages the procedure needed to build an agent” before proceeding to state “Yet, the necessary procedure for building agents is not only between the pages but as well between the lines.” [page 233] This book has 382 numbered pages, between which the author not only manages to avoid explicitly stating what exactly an agent is, what it does, and how it does it, but also provides many lines between which the reader can search for clues! Unfortunately, this book is not an isolated example.

The problem of hype is exacerbated by several facts. The first is publicity in the popular media about software agents. The second is the inability of the traditional peer review and journal publishing system to cope with the rate of growth in research and development. The time lag between the introduction of new ideas in the community and the publication of articles in refereed periodicals is significant. The third fact is that many agent-based systems are network based and web (world wide web) related. Self publishing over the web is the preferred means of disseminating information. Due to the substantial lag between the appearance of papers on the web and their publication in journals or conference proceedings even the published literature, for example [Petrie96], makes many references to documents available only over the web. It is interesting to note the impact of the web on academic publications and the dissemination of information. Extrapolating from what is occurring in the agent research community we can expect that while publications in reputable academic journals will remain the preferred means of sifting and archiving good work, as well as keeping score of academic performance, the web is likely to replace printed journals as the preferred media for communicating ideas. Citations will be the means to measure the quality of work, and peer review will take the form of links to information published on the web.

This history of AI is almost as old as that of the modern computer. The Turing test, one of the foundation stones of AI, is named after Alan Turing who worked as a code-breaker during WW2. Turing claimed that if human beings were unable to distinguish between a human and a piece of software based solely on their interaction via a computer terminal, then the software could be declared intelligent. Although the AI community has expanded in many different research directions, is has been kept together by a common desire to build such intelligent anthropomorphic software. Unfortunately, AI research has also suffered from historical over-optimism about the ability to satisfy this desire. Considerable effort has been expended in this area in past decades with no significant breakthroughs. While most researchers should be aware of this, only a very few explicitly acknowledge the dismal historical record on this issue. One of these rare examples is

[Bannon97], who states that it is “seriously misguided” to imply that software agents are capable of being viewed as perfect substitutes for human agents, and that promoting such a view of agents is yet another wasteful attempt to resuscitate traditional AI.

There are many agent developers who avoid trying to explicitly mimic human intelligence but believe that intelligent behavior will somehow “emerge” from the interactions of many simple agents [Brooks91b]. They believe that numbers of simple agents acting collectively can exhibit behavior similar to that of much more intelligent entities. In the domain of robotics, such emergent behavior has been demonstrated in [Brooks91a] where an insect robot possessing simple “agents” for legs learns to walk when the legs, each of them acting independently to changes in its sensory environment, start moving in a coordinated manner. Similar emergent behavior is also demonstrated in simulated insect societies [Hiebeler94, MRLA, DG96] where sets of simple agents learn to coordinate in foraging and defense.

Trends in separating the idea of anthropomorphic intelligence from agent technology are reflected in [GAABCGOOPSW96] which clearly differentiates between the concepts of agency, defined as the degree of autonomy and amount of authority vested in the agent, and intelligence. However there is often confusion in the use of terminology. For example, although [Foner93] states that agents need not be anthropomorphic, the paper uses the human-biased terms “trustworthiness” and “personalizability” rather than the more neutral terms “reliability” and “customizability.”

An interesting hypothesis in support of taking an anthropomorphic view of agents is that it helps us to understand their behavior and interact with or use them better [Shoham94]. An example of the anthropomorphic focus can be found in [Haddadi95] which defines agents as “intentional systems” that possess “beliefs, desires and intentions” (the BDI agent). The beliefs of an agent are its knowledge base; its desires are its goals; and its intentions are its plan of action. Many definitions of AI agents expect them also to possess other human qualities such as emotions, intelligence, trustworthiness, believability, character, personality, social ability and the ability to engage in dialogs. AI agents are also usually expected to possess models of their environment, themselves and/or other agents.

A graded taxonomy of other AI inspired concepts of agency can be found in [WJ95] which distinguishes “weak” and “strong” notions of agency. The weak notion includes autonomy (operation without the direct intervention of humans or others, with some kind of control over actions and internal states), social ability (communication with others), reactivity (react to changes in environment) and pro-activeness (exhibit goal-directed behavior by taking the initiative). The strong notion of agency adds human-like properties including mentalistic notions (such as knowledge, belief, intention and obligation) and emotional states.

2.2. Characteristics of Agents

This section attempts to extract the core characteristics underlying the different definitions of agents in the literature, and to relate these characteristics to the various ways in which researchers describe them. This is necessary and useful since the lack of universal terminology makes it difficult to compare and contrast these views. The definition in [FG96] where an "autonomous agent" is defined as "a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future" provides an example which possesses most of the characteristics which are elaborated below.

2.2.1. Purpose

Agents are expected to have goals. This expectation is described in many ways, for example by saying that agents are "dedicated to a specific purpose", "exhibit goal directed behavior" or "realize goals or tasks". In order to emphasize utility, these goals are often required to be proxy goals "on behalf of a user" (which may be another software agent, with the ultimate user being human). In anthropomorphized agents, these goals are referred to as the "desires" of the agents. Such goals are usually emphasized in AI where agents are required to possess (symbolic) models of themselves and their environment. The emphasis on goals differentiates current AI agent research from traditional AI "intelligence" research which paid less attention to practical applications.

2.2.2. Existence

Almost all definitions of agents start by imposing some existential characteristic: agents must exist within some environment over some period of time. The keywords used to capture this characteristic include, for example, "inhabit environment (real world)" and "persistent / temporally continuous (stay alive)"

Since everything (even an idea) possesses these characteristics, they are only useful in distinguishing between different types of agents. For example between what the agents are made of (biological, robotic, software etc.); between the environments that the agents inhabit (computer networks, trees, water etc.); or life-span (short, long, indefinite etc.) The definitions that leave out existential characteristics (e.g. [Nwana96] where an agent is defined as "a component of software and/or hardware which is capable of acting exactingly in order to accomplish tasks on behalf of its user") probably do so since they believe existence to be self evident.

A justification for including an existential characteristic in defining agents is to distinguish them from non agent-based software that is started up, performs its task and exits. Agents are expected to stay alive, even if dormant, in order to detect triggering conditions which will reactivate them. For example, a printer daemon has this characteristic but most application programs, such as word processing packages which might use the services of such a daemon do not.

2.2.3. Environment

Another universal characteristic of agents is their ability to modify their environment. In order to do so, they must have an input-output interface with their environment. The input component is usually referred to as the “sensors” of the agent in structural definitions of agency or as “perception” for behavioral ones. Agents modify their environment using their “effectors” (structural) to perform some “action” (behavioral). This action may include motion, which may take place with or without the preservation of the internal state of the agent. Mobility in agents is a popular research topic, especially in robotics, but also in technologies that enable mobility for software agents, e.g. telescript, Java, agentTcl etc. An important type of interaction between an agent and its environment is inter-agent communication to facilitate cooperation. Communication may take the form of negotiations, transfer of information or coordination of actions. This communication need not be direct or peer-to-peer. For example, in Ant Systems [DMC96], the communication is performed by “pheromones” that the “ant” agents secrete onto components of solutions.

2.2.4. Behavior

Behavioral definitions of agents are usually more specific than structural ones. There is less consistency between the terminology used by different researchers. I group the behavioral characteristics proposed by researchers into two sets: how agents should act, and how agents should adapt or react. The aspects of action include some domain oriented reasoning, where agents are expected to “decide actions”, “interpret perceptions”, “solve problems”, or “draw inferences” by “employing knowledge and/or representations” of the “users goals or desires”. The aspects of reaction include the ability to “learn”, “degrade gracefully” on failure and the ability to adapt to available resources or the user’s objectives in a “timely fashion” [WJ95]. An even more extreme notion is the expectation that agents have human like behavioral properties, such as “believability”, “truthfulness”, “emotional states”, “rationality” etc.

2.2.5. Structure

I believe that all agents consist of a subset of the following components: input, output, processing and control mechanisms. Agents obtain input from the environment via the input mechanism. The region of the environment from which an agent obtains inputs is the ‘input space’ of the agent. Agents may modify the environment via the output mechanism: by generating output to it such as by adding a new piece of information to a shared database, or by sending a message; by removing elements from the environment; or by moving within it. The region which an agent can modify is the ‘output space’ of the agent. The output space of an agent may include its location. The processing mechanism of an agent operates on the inputs to generate the outputs of the agent. The control mechanism of an agent coordinates all the other components of an agent, and hence determines its behavior. The control mechanism determines:

(1) what the agent does, specifically:

(a) selection and acquisition of inputs from the environment, processing of the inputs (if there is more than one way it can do so), and modification of the environment (e.g. by generating some output, by communicating with other agents, or by moving).

(b) modification of internal state information (if any), including the determination of goals

(2) when the agent does it

(3) where the agent does it (if the agent is mobile)

A section of the agent community claims that proactiveness, defined as the ability of an agent to take the initiative, is the key feature of autonomy [Nwana96, WJ95]. This implies that the agent develops and maintains models of its environment, which it uses to develop plans to act upon, and that it is able to predict or anticipate the needs of a user and act upon them without being explicitly requested to do so. This requires highly complex processing and control mechanisms for agents. There is a section of the community that believes that intelligent behaviors will emerge as a result of agent interactions even if the agents are unsophisticated "reactive" entities that behave in a stimulus-response manner using a very simple processing and control mechanism [Brooks91b].

2.3. A Taxonomy for Agent-based Systems

I classify agent-based systems along four main dimensions: agent characteristics; agent organization; inter-agent interaction; and applications. The first three are further subdivided into several attributes.

2.3.1. Agent Characteristics

Degree of Autonomy: How much control over their actions do agents in the system have? Agents should have a significant degree of autonomy.

Execution Protocol: Do agents work in a synchronized, lock-step manner or do they operate asynchronously?

Mobility: Are the agents mobile? Mobile systems include robots as well as software agents that can move across computer networks.

Complexity: Are the agents fairly simple or complex? Are they adaptable? Do they exhibit anthropomorphic characteristics?

2.3.2. Agent Organization

Hierarchy: Are the agents organized hierarchically or in flat organizations?

Types and Number of Agents: Are the agents in the system homogeneous or heterogenous? If they are heterogeneous, how many different types of agents are there? Is there only one agent in the system or are there many?

Openness: Can new agents be easily added to the system? Can new types of agents be added to the system?

2.3.3. Agent Interaction

Two agents collaborate when the input space of one overlaps with the output space of another [Talukdar98]. The most common way agents collaborate is by transferring resources: for example when an agent adds a resource to the environment which another agent uses. Agents collaborate effectively if they benefit as a result of the resource transfer.

The transfer of all resources between agents is underpinned by the transfer of information. For example, consider the case of one agent (A) providing another (B) with a computer (C) to run on. This interaction can take place as follows: A informs B that it can work on machine C. B must then be able to transfer and activate a copy of itself on C. Once it is done, B must inform A and terminate its activity on C. Even though this involves providing of a real physical resource, the interactions that take place all involve transfer of information between one computer and another. The information may be messages, a copy of computer code, or problem related data.

The ability to communicate, or transfer information, is the basis of effective collaboration and the problem of facilitating effective collaboration of software agents can be posed as one of facilitating communication between software agents. Agents may collaborate indirectly if they are connected by a sequence of agents that collaborate directly. (e.g. agents A and B collaborate indirectly via an intermediate agent I if the input space of I overlaps with the output space of A and the output space of I overlaps with the input space of B). A bucket chain is an example of such indirect collaboration.

The categories of resources that can be transferred are:

(1) Physical

(a) Raw materials; e.g. a bulldozer provides a cement mixer with sand

(b) Tools/Equipment; e.g. you give your friend a hammer

(c) Space; e.g. a robot vacates a location for another robot

(2) Informational

(a) Information; e.g. a software agent tells another the location of a file

(b) Coordination

(i) Hierarchical; e.g. a supervisor coordinates the activities of a set of lower level workers.

(ii) Non-hierarchical; e.g. a group of scientists coordinate on a one-to-one basis.

Negotiation is an important aspect of communication. Negotiations often require the use of languages, often called agent communications languages (ACL), which are an important area of research. Communication between agents is facilitated by a common semantics to interpret communications and a shared protocol that agents must use to respond to the various classes of communication [LD96]. Papers in this area present strong biases and narrow views and many such as [WJ95, Nwana96] insist that agents must communicate in an ACL. For example [GK94] defines software agents as "components that communicate with their peers by exchanging messages in an expressive agent communication language" (ACL) and [Petrie96] defines agents as "typed message" agents which must use an ACL, interact via a peer-to-peer communication protocol and having some "degree of collaboration with other agents using volunteered messages".

ACLs consist of a vocabulary, a syntax to compose sentences, and a semantics to interpret those sentences [GK94]. An example of a such a language is KQML (Knowledge Query and Manipulation Language) [MLF95] which uses KIF (Knowledge Interchange Format) as the syntax. A domain dependent vocabulary (a set of words and their meaning in the domain) is referred to as an Ontology. A repository of ontologies can be found at [Ontology]. An alternative ACL which attempts to capture the beliefs, desires and intents of agents is presented in [Singh94]. This alternative seems to be limited to use in domains like planning and is not as popular as KQML.

Process models for collaboration are classified as speech-act based or input-process-output based models in [Klein98]. Speech Acts theory [Austin62] is an attempt to model and understand communication. It considers utterances to have three features: locution (physical utterance); illocution (conveying speaker's intent to hearer); and perlocutions (actions resulting from illocution). Illocutions are classified into various types, and the interaction between two or more agents consists each providing the others an appropriate response to the communication it receives. For example [Haddadi95] statements could be assertives, directives, promissives/commissives, declaritives, expressives, permissive or prohibitives. The structure of dialogues is an important area of research. Interactions between agents following a predefined negotiation script: "requesting, accepting, completing and accepting delivery of tasks" [DS83]. An example of such a negotiation dialogue structure is Contract Nets: a Manager agent sends out a call for bids to a set of Contractor agents; the Contractors return bids; the Manager then assigns tasks; and the Contractors then return the results to the Manager. [Klein98] claims that speech-acts based models contribute structure by forcing the identification

of the customer and supplier for every deliverable, but are more complex as a result. Most agent-based systems with AI origins follow the speech-acts based model. In contrast, agent-based systems with DPS origins tend to follow the input-process-output model where agents perform a “series of tasks that take several inputs (representing both control and data) and (typically using resources such as people and machinery) produce one or more outputs.” [Klein98] These outputs are then used as inputs by other agents.

There are several systems where agents collaborate without using ACLs. Communications might be performed using far simpler mechanisms such as transferring tokens such as the “pheromones” used in Ant Systems [DG96]. Alternatively, agents could transfer data without the higher level semantics attached (e.g. the values of solutions) as in A-Teams or Genetic Algorithms. An example of a system where agents collaborate without communication is the autonomous legs of some insect-like robots [Brooks91a]. ACL communications usually require synchronization, whereas the simple communications mechanisms do not. The need for synchronization among participating agents often adds overhead costs to the collaboration process.

The two key attributes of agent interaction that facilitate collaboration are:

Communications Protocol: How do agents communicate with each other? There are two types of communications protocols: message passing or peer-to-peer communications where agents communicate directly with other agents; and shared memory where agents interact via shared repositories of information.

Negotiation: What is the degree of negotiation between agents?

2.3.4. Application

The applications for agent-based systems are too many, and growing too rapidly to be completely enumerated here. Applications range from automation of personal tasks such as sorting email to distributed optimization and control. Examples of applications will be provided along with the descriptions of various agent-based systems in the next section.

2.4. A Selection of Agent-based Systems

This section provides a brief description of most of the important agent-based systems and classifies them using the taxonomy that is presented in the previous section. It is impossible to detail every instance of an agent-based system in a few pages, and therefore groups of similar systems are described using a single category along with references to representative research.

2.4.1. Assistant Agents

An assistant agent is software that supports the activity of a human user [JGAAPR96]. The main application of assistant agents is in human-computer interaction (HCI). Such applications include better user interfaces

which evolve in time to tailor themselves to the user. An example is the work of the agent group at MIT [Maes94].

A prime motivation to develop “intelligent” or “smart” agents is to create substitutes for human agents such as secretaries or travel agents. Many researchers who emphasize computer intelligence are working on assistant agents. The scope of computer intelligence is usually restricted to small, well characterized domains. These agents interact with the user to perform highly specialized, routine, repetitive (and possibly boring) tasks [SDPWZ96], and are hoped to have a similar impact on human mental labor as machines had on physical labor.

[Maes94] suggests that interface agents can act as “personal assistants” that “collaborate with the user” in one or more of the following ways: as proxies, as trainers, as assistants when users collaborate with others, and as monitors for events and procedures.

[Nwana96] emphasizes agents-as-proxies for humans and defines agents as a component which is “capable of acting exactingly in order to accomplish tasks on behalf of its user”. [Nwana96] expects agents to possess two of these three attributes: cooperation, autonomy and learning where autonomy is defined as the ability to operate without the need for human guidance. Proactivity is considered an essential characteristic of autonomy. Cooperation requires communication in an ACL. Although [Nwana96] does not define learning, the paper claims that it is a “key attribute” of an intelligent being.

2.4.2. Collaboration Support Systems

Another human-centered application is in computer supported collaborative work (CSCW). Collaboration support systems differ from assistant agents in that the emphasis is on assisting group activity and are therefore often referred to as groupware. [Ellis98] categorizes groupware agents as performing one of the following tasks: acting as a repository of information (keeper), ensuring satisfaction of precedence constraints in control flows (coordinator), supporting human-to-human communications (communicator) and assisting cooperation by acting as a social mediator or critic. An interesting application of agent assisted human interaction is described in [NSM98], where the agent (consisting of a parser, article database and associative dictionary) monitors the conversation of a group of humans and periodically interjects by contributing a section of an article from its database when the human conversation appears to have stagnated.

2.4.3. Anthropomorphic Agents

Anthropomorphic agents are designed to mimic human beings. A considerable amount of the intelligence-focused research should be classified here. The agents are designed to respond with human-like intelligence [Hermans96] or human-like emotions such as happiness, anger, frustration etc. Emotional agents such as those in the Oz project at CMU [Bates92] are targeted primarily for entertainment.

2.4.4. Mobile Agents

A mobile agent [HCK95] is a piece of software that either resides in a mobile hardware platform (robots) or a piece of software that can move from computer to computer across a network. Mobility provides the agents the ability to explore physical and virtual spaces. In addition, mobility is also desired for software agents to make them robust to hardware failures or provide them the ability to circumvent hardware limitations such as overloaded processors. Examples of such systems include the insect like robots described in [Brooks91a].

2.4.5. Actor system

Actors [Agha86] are communication processing devices that transform incoming messages into a 3-tuple consisting of a finite set of messages sent to other agents, a new internal state (behavior) and a finite set of new agents created. This transformation depends on the internal state of the agent and the message received.

Actor agents have three main characteristics: they are data driven, reacting only when they receive messages (data); they can spawn other agents which then operate in parallel with themselves; and they operate asynchronously - data is buffered in a communication channel and is communicated from one agent to another directly or by being “forwarded” by other agents.

[Agha86] provides two simple “actor” languages called SAL and Act as examples of how such a system could work. There has been relatively little work on practical applications of Actor systems.

2.4.6. Blackboards

Blackboards are, strictly speaking, not agent systems but are often considered agent systems since they consist of sets of specialized, almost agent-like pieces of software called knowledge systems (KS). Blackboards [Nii86, Corkill91] are named after the metaphor used by their component KSs to exchange information. There may be multiple solutions posted on the blackboard. There is a single blackboard controller that decides which KS will work and on what solution it will work. Only a single KS is given access to the blackboard at a time. This is the metaphor of “a single piece of chalk” used to avoid conflicts. KS’s may contain estimators, and the controller may ask a KS for an estimate of their impact on a solution before actually assigning the task. Blackboards are widely used as an underlying technology in expert systems such as those in medical diagnosis [LH98].

2.4.7. Genetic Algorithms (GA) / Evolutionary Algorithms (EA)

GA systems consist of two types of component modules called crossover and mutation operators. The GA formalism [Goldberg89, Davis91] is a meta-heuristic based on the process of evolution. Solutions are (usually) encoded as strings. A population of solutions is maintained. There are two types of operations per-

formed on the solutions - mutation (a random perturbation of the solution) and crossover (a random merge of two or more solutions) based on their counterparts in biological evolution. GA's work iteratively as follows: given a population, a new population is constructed by mutations and crossovers of its members. The selection of solutions which pass their "genes" on to the new population is performed with a bias towards those solutions with better evaluations. The new population is then evaluated and replaces the old one. GAs can be easily implemented as agent systems [ST91, PK97]. There has been a recent trend to extend the encodings used to represent solutions, as well as the types of operators used to modify solutions. These efforts are often labeled Evolutionary Algorithms, which subsume GA, since the emphasis is on using a process of selection to weed out bad solutions even though non "genetic" solution encodings are used.

2.4.8. Multi-agent Simulation Systems

Swarm [Hiebeler94, MRLA] is an agent-based system for simulating real world ecologies such as ant colonies. The idea is to move from "mathematical description of an entire ecosystem to rule-based specifications of … agents" in order to create better models of ecologies. Agents in a swarm act in a synchronized fashion. A central controller provides a synchronization signal. Upon receiving the signal, all agents execute a procedure that modifies internal state-variables, sends out messages to other agents, and acts on messages received from other agents. Swarm is one of attempts to study "emergent" intelligence, since the individual agents are often very simple. Swarm is also being used to simulate and study manufacturing activities such as supply-chain networks [LTS98].

Agentsheets [RC93, Agentsheets] is a similar system to swarm, except that the agents exist in a regular 2 dimensional grid, and respond to stimuli from neighboring cells. Agentsheets agents also respond synchronously every time step, except that they respond to the state of their neighborhoods (e.g. which cells are occupied and by what) according to a predetermined, agent-specific set of rules. Agentsheets can be considered the 90's version of Conway's game of Life.

2.4.9. Ant Systems (AS)

Ant Systems, or Ant Colony Optimization, [DG96, DMC96] consist of sets of identical agents, or ants. They are used for combinatorial optimization problems such as the Traveling Salesperson Problem (TSP) [HK70, LK73] where the objective is to find the shortest closed tour through a set of cities on a map. The agents select different parts of solutions, and mark them with "pheromones". Those parts that get a lot of pheromones are more likely to be used by other agents in constructing solutions.

The process proceeds as follows: Each agent creates new solutions by using components of existing solutions, with a bias towards selecting components with more pheromones. The agent then marks the solution with pheromones, with solutions with higher quality getting more pheromones.

Consider how the TSP is solved by Ant Systems: In the outer loop of the algorithm, each ant is started at an arbitrary city, constructs a solution (in the inner loop) and the resulting solutions are collated and a desirability assigned to each component (city-to-city edge) in the problem. The inner loop consists of n steps where each ant moves from its current city to an unvisited city in the current tour by selecting an edge in its tour. Edges from the current city is selected with a probability that is proportional to its length and to the desirability of the edge measured by the amount of "pheromone" associated with the edge.

2.4.10. A-Teams

An A-Team is an organization of autonomous cyber agents [Talukdar98]. An autonomous cyber agent is an entity whose input and output spaces are maintained by computers, and whose control system is completely self contained (i.e. it decides for itself when to work and what to work on). Cyber agents collaborate by exchanging data via shared repositories of data called memories. The organization of agents and memories can be represented as a directed hypergraph, where nodes are memories and arcs represent agents. Such a graph is called a dataflow. A dataflow is an A-Team if and only if all arcs lie in closed loops. A complete set of definitions, descriptions and prescriptions for A-Teams can be found in [Talukdar98]. Additional details and theoretical analyses of why A-Teams work can be found in [TBGD97].

The power of A-Teams lies in their modular design. Solutions in various representations, or even solutions to related problems that might provide hints at solutions for the primary problem at hand, are stored in different memories. Heuristics that operate on these solutions are encapsulated as autonomous agents. These agents work independently and asynchronously of each other. There is no central controller. Thus agents can be added or removed as needed without affecting existing agents in the team.

A variety of heuristics can be easily used in combination. In addition, agents to improve solutions and agents to evaluate solutions can all work in parallel across a network of computers. No coordination is required, making the assembly of the team easier. Parallel execution allows agents to explore more solutions in less time. Also, since A-Teams work with populations of solutions and do not require a single merit function, sets of non-dominated solutions known as a Pareto frontier can be found. This enables the user to choose the best trade-offs between the different objectives after good solutions have been found rather than having to assign weights a-priori.

2.4.11. Classification

This table provides a classification of the agent-based systems using the taxonomy developed in this chapter. As can be seen, most systems fall into one of two categories: human centric systems that use message passing or mathematical problem solving systems that use shared memory for communication.

Table 1. Classification of Agent-based Systems

System	Agent Characteristics	Agent Organization	Agent Interaction	Application
Assistant Agents [Maes94, SDPWZ96]	High Autonomy Asynchronous Not Mobile Very Complex	Usually single agent	Message-Passing Lots of Negotiation	Assist single human user. Human-computer interaction
Collaboration Support Systems [Ellis98, NSM98]	High Autonomy Asynchronous Not Mobile Complex	Hierarchical Few types Open to new agents	Message-Passing Lots of Negotiation	Assist collaboration between many humans Human-computer interaction
Anthropomorphic Agents [Bates92]	High Autonomy Asynchronous Not Mobile Very Complex	Usually single agent	Message-Passing Lots of Negotiation	Entertainment
Mobile Agents [Brooks91a, HCK95]	High Autonomy Asynchronous Mobile Fairly Complex	Usually Flat Varied agent types Open to new agents	Message-Passing Variable Negotiation	Exploration of physical and virtual spaces. Information Gathering
Actor Systems [Agha86]	High Autonomy Synchronous Not mobile Complex	Flat Single type of agent Open to new agents	Message-Passing Little Negotiation	Programming
Blackboards [Nii86, Corkill91]	Low Autonomy Synchronous Not Mobile Usually Complex	Hierarchical Multiple Types of agents Open	Shared Memory Little Negotiation	Diagnosis
Genetic Algorithms [ST91, PK97]	Low Autonomy Synchronous Not Mobile Simple	Flat Two agent types Not open	Shared Memory No Negotiation	Optimization
Simulation Systems [Hiebeler94, RC93]	High Autonomy Synchronous Not mobile Varied Complexity	Flat Variable agent types Open	Shared Memory No Negotiation	Simulation

System	Agent Characteristics	Agent Organization	Agent Interaction	Application
Ant Systems [DG96, DMC96]	Moderate Autonomy Synchronous Not mobile Simple	Flat One type of agent Open to new agents	Shared Memory No Negotiation	Combinatorial Optimization
A-Teams [Talukdar98]	High Autonomy Asynchronous May be mobile Varied Complexity	Flat Variable agent types Open	Shared Memories No Negotiation	Optimization Simulation Control Diagnosis

2.5. Summary

An agent is a piece of software that has a degree of autonomy, or the ability to control its actions: it can decide what to work on, when to work, and where to work. Agents remain alive, probing their environment for tasks to perform. In contrast non agent software is started up every time a new task is to be performed and terminates on completion.

There are two areas of research, Artificial Intelligence and Distributed Problem Solving, from which the concept of agent has evolved. There are two broad categories of agents. The first contains human centric-agents which are based on a negotiation or speech-act process model of agent activity, are used in human computer interfaces and as proxies for humans, and are implemented using message-passing for communication. The second category contains agents that are based on an input-process-output process model, are used for simulation, distributed optimization, and distribute control, and usually use a shared-memory for communication.

I identify purpose, existence, environment, behavior, and structure as the properties used to define agents in the literature. I classify agent-based systems along four main dimensions: agent characteristics; agent organization; inter agent interaction; and applications.

The A-Teams formalism is a meta-heuristic that combines features from many of the other agent-based systems. It allows the opportunistic combination of the best features of many natural and artificial systems, including agent-based ones. It uses populations of solutions (like EA's) stored in memories (resembling blackboards) worked on by teams of autonomous agents based on heterogeneous algorithms. It allows the concurrent use of both simple and complex iterative algorithms for solving optimization problems. A-Teams do not use the direct peer to peer communications or sophisticated negotiation mechanism of anthropomorphic agents.

The A-Teams formalism facilitates the synthesis and extension of existing optimization agents. It already subsumes several of the existing approaches, and benefits them by providing additional capabilities. Excellent examples are [ST91] and [PK97] where GAs are extended by using the A-Teams formalism and implemented as A-Teams with genetic operators. It appears likely that A-Teams research will be a medium of integration and cross fertilization among agent-based systems.

3. A GENERATIVE GRAMMAR FOR A-TEAMS

A-Teams enable a variety of solution representations and algorithms to be combined to solve problems. The algorithms are encapsulated as agents which can be stored in repositories and used as needed to create customized A-Teams. One of the goals of A-Teams research is the ability to automatically generate A-Teams that are tailored for any given problem. In this chapter I will develop a generative grammar for A-Teams that takes us closer to this goal.

A generative grammar consists of a set of primitives and a set of rules for constructing complex structures by combining primitives and/or other complex structures. Examples of generative grammars include the definition of programming languages such as C [KR88] and the set of architectural rules for constructing Queen Anne Houses in [Flemming87].

A high-level generative grammar for constructing A-Teams can be found in [Talukdar98] which provides a description of the components, guidelines for their construction, and rules for organizing them into A-Teams. The implementation of these high-level instructions must be done by humans, leading to possible misinterpretations. More precise definitions of the components of A-Teams and the structure of A-Teams are required to avoid such misinterpretations as well as to facilitate automated A-Team construction.

The two main entities in A-Teams are stores of solutions called memories and autonomous problem solving modules called agents. These components are reusable and can be connected together in many different ways. Thus, task specific organizations can be constructed by selecting memories and agents from repositories and organizing them into A-Teams. [Talukdar98] defines an A-Team organization as a particular kind of hypergraph, called a dataflow, where nodes represent memories and directed arcs represent agents. Nodes may overlap, and all agents must lie in closed loops. Such hypergraphs are not widely used outside of A-Teams and their properties have not been studied. An automated A-Team generator must be able to generate such hypergraphs that are also semantically justifiable as A-Teams. For example, not every agent can be connected to every memory; these restrictions on agent-memory connections are not captured in the hypergraphs. In other words, A-Teams are more than just graphs.

Any automated A-Team generator requires at least: rules to identify the relation between actual memories and agents and their representations in A-Team hypergraphs; rules for generating hypergraphs; and rules for culling those that are not valid A-Teams. While it is possible to generate A-Teams by generating all possible hypergraphs and identifying the valid A-Teams, the subset of

valid A-Teams is a small fraction of directed hypergraphs and it is far more efficient to guarantee by construction rules that the hypergraphs generated are valid A-Teams.

This chapter starts with a discussion of the issues relating to a grammar for A-Teams, presents the grammar, then provides a proof of the validity of the grammar (i.e. a proof that it generates only valid A-Teams), followed by supporting examples to show coverage (i.e. the grammar can generate most, if not all, possible A- Teams).

3.1. Designing the Grammar

I believe that the following features are especially desirable in a grammar for A-Teams: simplicity; incremental construction so that new components can be created and added to repositories; precision to facilitate automation; and validity of the dataflows generated by the grammar.

3.1.1. Primitives

A-Teams provide a way of combining heuristics into cooperative organizations. This is done by encapsulating the heuristics to create agents, and organizing the agents around populations of data that they share. Each population contains homogenous data so that all agents reading from that population can interpret the data correctly. The primitives of a grammar must include the necessary components to encapsulate heuristics into agents, and data types into memories.

3.1.2. Structure

A-Teams are a flat, non-hierarchical organization of their components. An aspect of a flat characterization can be retained by limiting the nesting levels of memories and preventing recursive combination of memories. This also prevents aliasing of memory structures: a given memory has exactly one representation, no matter how it is constructed.

3.1.3. Precision

In order for a grammar to be implementable, it must be precise. There is no scope for ambiguity in the semantics of any structure generated by the grammar. A grammar that generates A-Teams where there is no ambiguity in the connection of components is also easier to automate since no human intervention is required for disambiguation.

3.1.4. Cyclic Dataflows

Strongly cyclic dataflows, which ensure that all agents exchange information and cooperate with other agents is a key feature of A-Teams and the grammar should guarantee such dataflows.

3.1.5. Connection Rules

The connection rules state how agents and memories are connected. They ensure that no connections is made where an agent cannot access data from some part of a memory it is connected to. By doing so the connection rules also try to ensure that agents actually do share their results with each other. There is no way to guarantee that an agent will read data from, or write data to, every part of the memory it is connected to, but connection rules can be used to support the spirit of A-Team design by ensuring that an agent can access every part of any memory it is connected to.

3.2. The Grammar

I will start with a short summary of the description of A-Team structure in [Talukdar98]. An A-Team is defined as a strongly cyclic data flow consisting of agents and memories. Memories contain populations of solutions, and agents operate on solutions in the memories. A data flow is defined as a directed hypergraph, where each node is a Venn diagram of overlapping input and output memories, and each arc represents an agent. A data flow is strongly cyclic if each of its arcs is in a closed loop.

The grammar to construct such structures consists of a set or primitives, extended primitives, and a set of rules for combining the primitives. The grammar will be illustrated with hypergraphs to clarify concepts and to draw a connection between the grammar and the definitions and representations in [Talukdar98]. Note that the grammar is defined recursively, creating inter-dependencies among the various constituents, and it may be helpful to *skim through the grammar first* before reading it in detail.

3.2.1. The Primitives

Elementary Operator: A mapping from the operator input space (OIS) to the operator output space (OOS).

(a) The two spaces might be identical (i.e. OIS = OOS).

(b) A special type of operator is the **destruction operator**, which identifies undesirable elements in the OOS.

Elementary Selector: A mapping from one space (SIS: selector input space) to another (SOS: selector output space). The selector output space must be contained in the input space (SOS is a subset of SIS).

Scheduler: A procedure that outputs a signal to start work from time to time.

Elementary Memory: A repository of objects in a space (MS: memory space)

3.2.2. The Extended Primitives

Operator:

1. Operator = Elementary Operator

2. Operator = Operator × Operator

note: Consider the case of Op = Op1 × Op2. The OOS of Op1 must be the OIS of Op2. The OIS of Op is the OIS of Op1 and the OOS of Op is the OOS of Op2.

Selector:

1. Selector = Elementary Selector

2. Selector = Selector ∪ … ∪ Selector

The SIS and SOS of a union of selectors is the product of the SIS's and SOS's of the components.

3.2.3. The Functional Units

Agent(< , >): A mapping from one space (AIS: agent input space) to another (AOS: agent output space).

Symbol:

An agent connects one memory to another (possibly the same) memory. The symbol "<" is used as a placeholder for the memory from which the agent gets its input (corresponding to the AIS) and the symbol ">" is used as a placeholder for the memory to which the agent writes its output (corresponding to the AOS).

1. Agent(< , >) = Elementary Agent: Scheduler + Selector + Operator (A1)

s.t. SOS = OIS

note: (a) AIS = SIS and AOS = OOS

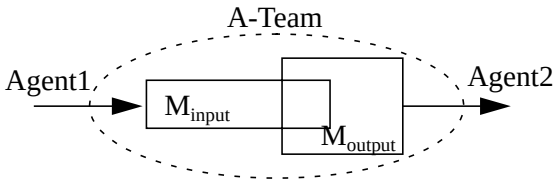
(b) an agent with a destruction operator is referred to as a **destroyer**

2. Agent(< , >) = Agent1(< , M_{output}) +> A-Team +> Agent2(M_{input} , >) (A2)

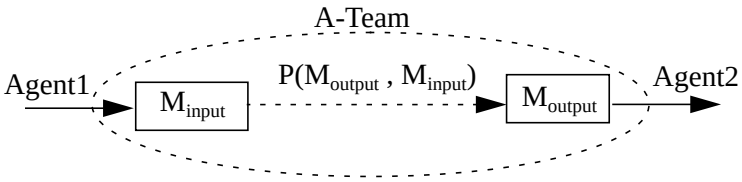
s.t. neither Agent1 nor Agent2 is a destroyer and

$M_{input} \in$ A-Team and $M_{output} \in$ A-Team and either

A2 (a) $(M_{input} \cap M_{output}) \neq \emptyset$,

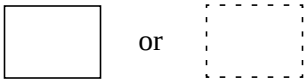


or A2 (b) there exists a path P(M_{output} , M_{input}) in the A-Team



note: the AIS of the resulting agent is the AIS of Agent1 and the output space is the AOS of Agent2.

Memory: A space (MS: memory space)

Symbol:  or

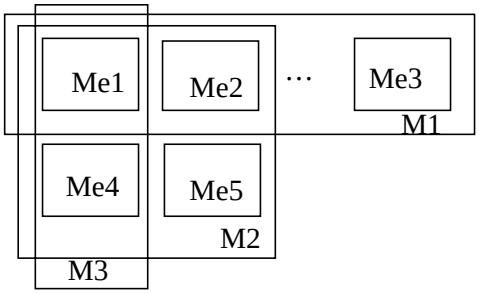
1. Memory = Elementary Memory

an elementary memory contains a collection of objects in a space

2. Memory = Elementary Memory ∪ Elementary Memory ∪ … ∪ Elementary Memory (M2)

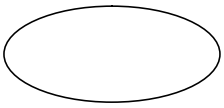
the MS of a union of Elementary Memories is the product of the MS's of the constituents

note: (a) memories may overlap and (b) nesting is at most one level deep



A-Team: A directed hypergraph (with zero or more arcs) where every arc is in a closed loop.

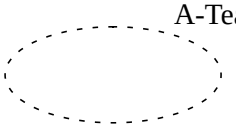
An A-Team is usually represented as a graph consisting of memories and agents; the following symbol is used here to represent an A-Team to present a high-level picture and hide possibly confusing details.

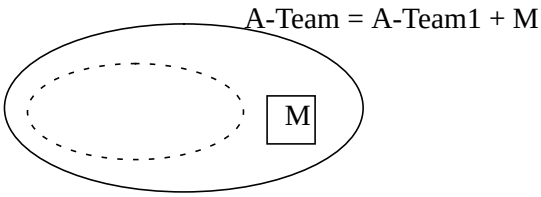
Symbol:  or 

1. A-Team = Memory (T1)

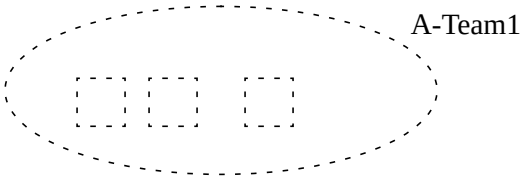
2. A-Team = A-Team1 + Memory (T2)

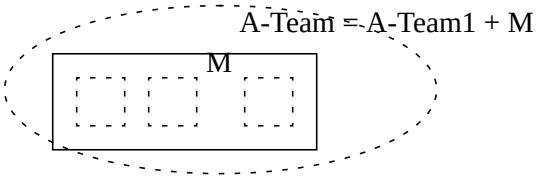
T2 (a) External memory: adding a new memory to the A-Team





T2 (b) Internal memory: creating a new memory as a union of existing elementary memories





3. A-Team = A-Team1 <+> Agent(M_{input}, M_{output}) (T3)

s.t. M_{input} and M_{output} are in the A-Team and

T3 (a) $M_{input} = M_{output}$, or

T3 (b) the Agent is not a destroyer and $(M_{output} \cap M_{input}) \neq \varnothing$, or

T3 (c) the Agent is not a destroyer and there exists a path $P(M_{output}, M_{input})$ in the A-Team

3.2.4. Auxiliary Concepts and Terms

Destruction Operator: a special type of operator that identifies elements in the OIS for elimination.

Destroyer: An (elementary) agent with a destruction operator. The input memory of a destroyer is always the same as its output memory.

Contain: memories inside memories or A-Team

(a) memory M1 $\subset$ memory M2 iff

(i) M1 is an elementary memory and M2 = … $\cup$ M1 $\cup$ … , or

(ii) all the elementary memories contained in M1 are also contained in M2

(b) memory M $\in$ A-Team T iff

(i) M is an elementary memory in T, or

(ii) $\forall$ elementary memories $M_e \subset M$, $M_e \in T$

Contain: an agent A(M1, M2) $\in$ A-Team T iff

(i) M1 $\in$ T, and

(ii) M2 $\in$ T

Equality: memory M1 = memory M2 iff

(a) they are the same elementary memory, or

(b) they are both non elementary memories consisting of the same elementary memories

Intersection of memories: (M1 $\cap$ M2) $\neq \varnothing$ iff

(a) they are identical: i.e. M1 = M2, or

(b) one contains the other: i.e. M1 $\subset$ M2 or M2 $\subset$ M1, or

(c) both contain the same elementary memory: i.e. M1 $\supset$ Mx and M2 $\supset$ Mx, where Mx is an elementary memory

Connection: Agents to Memories

If a memory M is the union of the elementary memories M1, M2, …, Mn corresponding to spaces MS_1, MS_2, …, MS_n and the agent input (output) space is S

Then the agent may be connected to the memory for input (output) only in a manner such that

$S = MS_1^{d1} \times MS_2^{d2} \times \ldots \times MS_n^{dn}$ where di ≥ 1 is an integer

MS_i^{di} corresponds to the component of its input (output) that the agent gets from (writes to) Mi.

Input Memory: of an agent is the memory to which an agent is connected for input

Output Memory: of an agent is the memory to which the agent is connected for output

Connection: Agents to A-Teams

1.((A-Team +> Agent(M_{input} , >)) or (Agent(M_{input} , >) <+ A-Team)) s.t. $M_{input} \in$ A-Team.

2. ((A-Team <+ Agent(< , M_{output})) or (Agent(< , M_{output}) +> A-Team)) s.t. $M_{output} \in$ A-Team.

3. ((A-Team <+> Agent(M_{input} , M_{output})) or (Agent(M_{input} , M_{output}) <+> A-Team)) s.t. ($M_{input} \in$ A-Team) and ($M_{output} \in$ A-Team.)

Path: connects two memories via a sequence of agents (except for destroyers) and memories, and is a way for information to flow from one memory to another.

Symbol: - - - - - - - - ►

Let P(M1, M2) be a path from memory M1 to memory M2 in a A-Team

and A(Mi, Mj) be an agent connecting memory Mi to memory Mj

P(M1, M2) = M1* A(M1*, M2*) M2*, or

M1* A(M1*, Mx) Mx P(Mx, M2), or

P(M1, Mx) Mx A(Mx, M2*) M2*

where: $(Mi^* \cap Mi) \neq \varnothing, \forall i$ (note: Mi* may equal Mi)

$Mi^* \in$ the A-Team, $\forall i$

Mx is any memory in the A-Team

note: P(. , .) $\in$ A-Team iff every constituent of P $\in$ the A-Team

3.3. Validity of the Grammar

A valid A-Team can be represented as a hypergraph where all arcs lie in closed loops [Talukdar98]. In the terminology defined in this chapter, arcs correspond to elementary agents. An elementary agent A(M1,M2) in the A-Team is in a closed loop iff there exists a path P(M2, M1) in the A-Team. This section proves that the grammar generates only valid A-Teams by showing that all elementary agents in A-Teams constructed by the grammar lie in closed loops.

3.3.1. Theorem 1: Data Flows in Agents

Theorem1: All elementary agents that constitute a non-elementary agent either lie in closed loops, or lie on a path connecting the output memory of the elementary agent on the input of the structure to the input memory of the elementary agent on the output of the structure.

Note: This theorem is required to show that non-elementary agents, created by rule A2(a) or A2(b), when added into an A-Team, create a valid resulting structure with all agents in closed loops.

Proof: This proof will be constructed inductively

Consider the construction of a non-elementary agent

A(< , >) = A1(< , M1) +> T +> A2(M2 , >)

where T is an A-Team, M1 $\in$ T, and M2 $\in$ T

(I) Base Case:

A1 and A2 are both elementary agents

all elementary agents (other than A1 and A2) in T must lie in closed loops since T is an A-Team and A(< , >) satisfies the property

(II) Inductive Cases:

There are three inductive cases: (i) A1 is elementary and A2 is non elementary; (ii) A1 is non-elementary and A2 is elementary; and (iii) both A1 and A2 are non-elementary.

Proof for case (i): A1 is an elementary agent, A2 is a non-elementary agent

all elementary agents in T lie in closed loops

let A2i(< , Mi) be input elementary agent and A2o(Mo, >) be the output elementary agent in A2 (< , >)

assume that A2 satisfies the property

all elementary agents in A2(< , >) other than A2i(< , Mi) and A2o(Mo, >) must either lie in closed loops or along a path P(Mi, Mo) in A2(< , >) since it satisfies the property

If A(< , >) is constructed according to:

Rule A2(a) : since M1 $\cap$ M2, then all agents not in closed loops must lie on the path

P(M1, Mo) = M2 A2i(M2, Mi) Mo if A2(< , >) was constructed using rule A2(a) since (Mi ∩ Mo) ≠ ∅

or P(M1, Mo) = M2 A2i(M2, Mi) P(Mi, Mo) if A2(< , >) was constructed using rule A2(b)

Rule A2(b) : there must be a path P(M1, M2) in T, and all these agents must lie on the path

P(M1, Mo) = M1 P(M1 , M2) M2 A2i(M2, Mi) Mo if A2(< , >) was constructed using rule A2(a)

or P(M1, Mo) = M1 P(M1, M2) M2 A2i(M2, Mi) Mi P(Mi, Mo) if A2(< , >) was constructed using rule A2(b).

Proof for case (ii) : A1 is a non-elementary agent, A2 is an elementary agent

The proof for this case is (almost) identical to that for inductive case (i).

Proof for case (iii): A1 and A2 are non-elementary agents.

all elementary agents in T lie in closed loops

let A1i(< , M1i), A1o(M1o , >), A2i(< , M2i) and A2o(M2o , >) be the input and output (elementary) agents in A1(< , >) and A2(< , >)

assume that A1 and A2 satisfy the property

all elementary agents in A1(< , >) other than A1i(< , M1i) and A1o(M1o, >) must either lie in closed loops or along a path P(M1i, M1o) in A1(< , >) since it satisfies the property

all elementary agents in A2(< , >) other than A2i(< , M2i) and A2o(M2o, >) must either lie in closed loops or along a path P(M2i, M2o) in A2(< , >) since it satisfies the property

If A(< , >) is constructed according to:

Rule A2(a) : since (M1 ∩ M2) ≠ ∅, all agents not in loops must lie on the path

P(M1i, M2o) = M1o A1o(M1o, M1) M1 M2 A2i(M2, M2o) M2o if both A1(< , >) and A2(< , >) were constructed using rule A2(a)

P(M1i, M2o) = M1o A1o(M1o, M1) M1 M2 A2i(M2, M2i) M2i P(M2i, M2o) if A1(< , >) was constructed using rule A2(a) and A2(< , >) was constructed using rule A2(b)

P(M1i, M2o) = P(M1i, M1o) M1o A1o(M1o, M1) M1 M2 A2i(M2, M2o) M2o if A1(< , >) was constructed using rule A2(a) and A2(< , >) was constructed using rule A2(b)

P(M1i, M2o) = P(M1i, M1o) M1o A1o(M1o , M1) M1 M2 A2i(M2 , M2i) M2i P(M2i , M2o) if both A1(< , >) and A2(< , >) were constructed using rule A2(b).

Rule A2(b) : there must be a path P(M1, M2) in T, and all agents not in loops must lie on the path

P(M1i, M2o) = M1o A1o(M1o, M1) M1 P(M1, M2) M2 A2i(M2, M2o) M2o if both A1(< , >) and A2(< , >) were constructed using rule A2(a)

P(M1i, M2o) = M1o A1o(M1o, M1) M1 P(M1, M2) M2 A2i(M2, M2i) M2i P(M2i, M2o) if A1(< , >) was constructed using rule A2(a) and A2(< , >) was constructed using rule A2(b)

P(M1i, M2o) = P(M1i, M1o) M1o A1o(M1o, M1) M1 P(M1, M2) M2 A2i(M2, M2o) M2o if A1(< , >) was constructed using rule A2(a) and A2(< , >) was constructed using rule A2(b)

P(M1i, M2o) = M1i P(M1i, M1o) M1o A1o(M1o, M1) M1 P(M1, M2) M2 A2i(M2, M2i) M2i P(M2i, M2o) M2o if both A1(< , >) and A2(< , >) were constructed using rule A2(b).

This completes the proof that all elementary agents constituting non-elementary agents lie either in closed loops or in a path from the input of the agent to the output.

3.3.2. Theorem 2: Data Flows in A-Teams

This theorem states that all A-Teams constructed using the grammar are valid (strongly cyclic dataflows). In order to prove this theorem, a set of lemmas will first be stated and proved, followed by the proof of the theorem itself.

Note: The following convention is used throughout this section in order to simplify the notation - a * is used in the naming of memories to indicate that they overlap - e.g. M and M* overlap, Mx and Mx* overlap etc.

Lemma 1: If an A-Team contains agent A(M, M*) then it lies in a closed loop

Proof: A(M, M*) lies in the loop closed by P(M*, M) = M A(M, M*) M*

Lemma 2: If an A-Team contains agents A1(Mx, My), A2(My*, Mx*) then both lie in closed loops.

Proof: A1(Mx, My) lies in the loop closed by P(My, Mx) = My* A2(My*, Mx*) Mx* and A2(My*, Mx*) lies in the loop closed by P(Mx*, My*) = Mx A1(Mx, My) My

Lemma 3: If an A-Team contains agents A1(Mx, My), A2(My*, Mz) and path P(Mz, Mx) then

(a) A1(Mx, My) and A2(My*, Mz) both lie in closed loops

(b) all agents in P(Mz, Mx) lie in closed loops

Proof: (a) A1(Mx, My) lies in the loop closed by P(My, Mx) = My* A(My*, Mz) P(Mz, Mx) and A2(My*, Mz) lies in the loop closed by P(Mz, My*) = P(Mz, Mx) A1(Mx, My) My

(b) Any agent A(M1, M2) in P(Mz, Mx) = P(Mz, M1) M1 A(M1, M2) M2 P(M2, Mx) lies in the loop closed by P(M2, M1) = P(M2, Mx) Mx A1(Mx, My) My My* A2(My*, Mz) Mz P(Mz, M1)

Lemma 4: If an A-Team contains agents A1(Mx, My), A2(Mz, Mw) and paths P(My, Mz), P(Mw, Mx) then

(a) A1(Mx, My) and A2(Mz, Mw) lie in closed loops.

(b) all agents in P(My, Mz) and P(Mw, Mx) lie in closed loops.

Proof: (a) If an A-Team contains agents A1(Mx, My), A2(Mz, Mw) and paths P(My, Mz), P(Mw, Mx) then A1(Mx, My) lies in the loop closed by P(My, Mx) = P(My, Mz) Mz A(Mz, Mw) Mw P(Mw, Mx) and A2(Mz, Mw) lies in the loop closed by P(Mw, Mz) = P(Mw, Mx) Mx A(Mx, My) My P(My, Mz)

(b) Any agent A(M1, M2) in P(My, Mz) = P(My, M1) M1 A(M1, M2) M2 P(M2, Mz) lies in the loop closed by P(M2, M1) = P(M2, Mz) Mz A2(Mz, Mw) Mw P(Mw, Mx) A1(Mx, My) My P(My, M1).

and it can be similarly proved that any agent in P(Mw, Mx) lies in a closed loop.

Theorem 2: All elementary agents in A-Teams constructed using the grammar lie in closed loops.

Proof: the proof exhaustively considers the application of all the rules in the grammar to construct an A-Team and shows that each possibility creates a valid result.

Rule T1: An A-Team consisting of a single memory has no agents. Thus there are no agents that are not in closed loops.

Rule T2: Addition of a memory to an A-Team does not change any of the agents. If all elementary agents were in closed loops prior to the addition, they remain in closed loops.

Rule T3: This rule adds Agents to A-Teams. There are two types of agents which can be added, elementary and non elementary, each of which can be added to satisfy one of the three conditions T3(a), T3(b) and T3(c).

Consider the addition of an agent A(< , >) to the A-Team T

All agents in T lie in closed loops, and remain in closed loops after the addition.

First consider the addition of an elementary agent A(< , >) to the A-Team T according to

rule T3(a): if the insertion A(M1, M1) is made to T it lies in a closed path by Lemma 1.

rule T3(b): if the insertion A(M1, M1*) is made to T it lies in a closed path by Lemma 1.

rule T3(c): if the insertion A(M1, M2) is allowed then the loop closing path P(M2, M1) must already exist in T.

Now consider the addition of non-elementary agent A(< , >).

All agents in A(< , >) that are in closed loops remain in closed loops after the addition.

Let the input elementary agent in A(< , >) be Ai(< , Mi) and the output elementary agent be Ao(Mo,>).

We need to show that all agents in A(< , >) that were not in closed loops before, Ai(<,Mi), and Ao(Mo,>) lie in closed loops after the addition of A(< , >) to T.

By theorem 1, all elementary agents (other than Ai(< , Mi) and Ao(Mo, >)) in A(< , >) that are not in a closed loop must lie on a path P(Mi, Mo).

It is possible that P(Mi, Mo) may not exist if A(< , >) was created using rule A2(a). If this is the case then Mi $\cap$ Mo.

If A(< , >) is added to T using:

rule T3(a) : If P(Mi, Mo) does not exist, then Ai(M, Mi) and Ao(Mo, M) lie in a closed loop by lemma 2. Otherwise Ai(M, Mi), P(Mi, Mo), Ao(Mo, M) exist in the new A-Team and all elementary agents included in these entities lie in closed loops by lemma 3.

rule T3(b) : If P(Mi, Mo) does not exist, then Ai(M, Mi) and Ao(Mo, M*) lie in a closed loop by lemma 2. Otherwise Ai(M, Mi), P(Mi, Mo), Ao(Mo, M*) exist in the new A-Team and all elementary agents included in these entitities lie in closed loops by lemma 3.

rule T3(c): if the insertion A(M1, M2) is allowed then the path P(M2, M1) must exist in T. If P(Mi,Mo) does not exist, then Ai(M1, Mi), A(Mo, M2) and P(M2, M1) lie in a closed loop by lemma 3. Otherwise Ai(M1,Mi), P(Mi, Mo), Ao(Mo, M2), P(M2, M1) exist in the new A-Team and all elementary agents included in these entities lie in closed loops by lemma 4.

This covers all possible ways of constructing A-Teams using the grammar, and thus completes the proof that all elementary agents in any A-Team generated by the grammar lie in closed loops and therefore any A-Team generated by the grammar must be a strongly cyclic data flow as defined in [Talukdar98].

3.4. Completeness of the Grammar

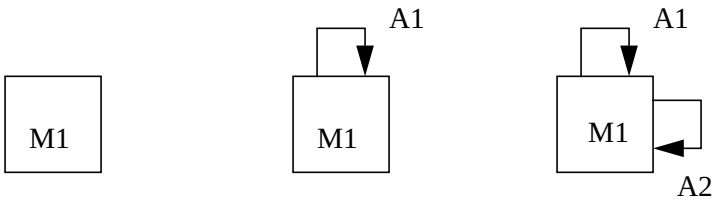
Completeness means that any valid A-Team can be generated by the grammar and therefore completeness is the dual of validity. Before completeness can be demonstrated, it is necessary to deal with the issue of memory construction since [Talukdar98] places no restriction on the level of nesting of memories. It can be argued that any highly nested memory can be represented as a single level nested memory because any non-elementary memory is identical to the union of all the elementary memories contained within it. Internal groupings merely change the internal partitioning of the memory space, but the space itself does not change. Additionally, all published A-Teams have at most one level of nesting, and can be directly represented by the grammar.

3.4.1. Demonstration of Coverage

Although I conjecture that the grammar is complete and can generate all possible A-Team dataflows, and I know of no published A-Team dataflow that cannot be generated by the grammar, I do not have a proof of completeness. In place of a proof, I will justify my conjecture by showing how a representative set of A-Teams can be generated by the grammar. The A-Teams selected for demonstration cover almost all the published A-Team structures as well as a few artificial pathological examples. This shows that interesting, useful and complex A-Teams can be generated by the grammar. The examples selected cover all the A-Teams in [Murthy92], [deSouza93], [Tsen95], [TBGD97] and [Talukdar98].

3.4.2. Example 1

The most commonly used A-Team structure, found in [Murthy92] and [Tsen95] is a single memory with a set of agents connecting the memory to itself. This can be easily constructed using rule T1 followed by repeated applications of rule T3(a).



A-Team T1 A-Team T2 A-Team T3

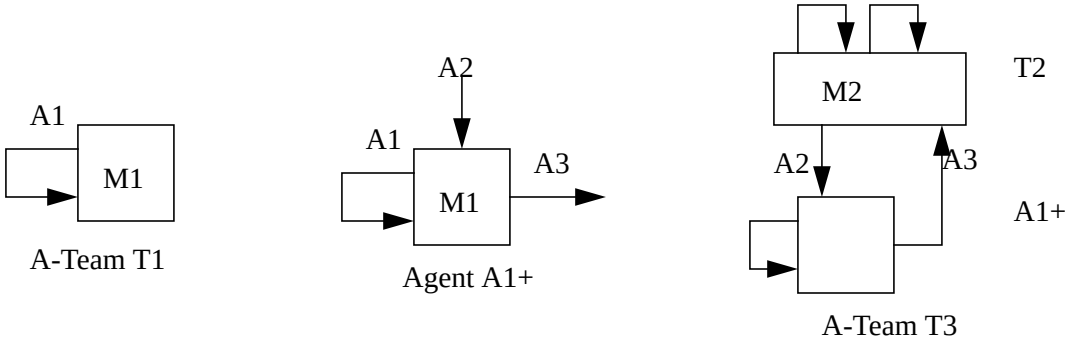
T1 = M1 (by rule T1)

T2 = T1 <+> A1(M1, M1) (by rule T3(a))

T3 = T2 <+> A2(M1, M1) (by rule T3(a))

3.4.3. Example 2

This example is from [deSouza93] and is the first in a series of increasingly complex A-Teams used to solve the Travelling Salesperson Problem (TSP)

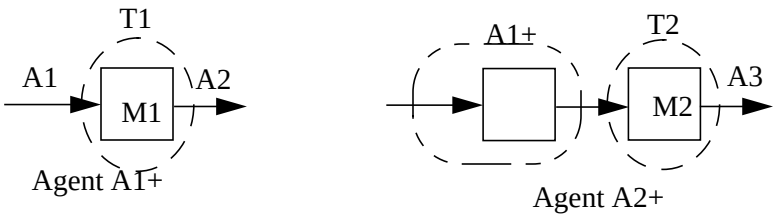


A1+(< , >) = A2(< , M1) +> T4 +> A6(M2 , >) (by rule A2(a) : M1 ∈ T2 and M2 ∈ T2)

T3 = T2 <+> A1+(M1, M1) (by rule T3(a) : M1 ∈ T2)

3.4.4. Example 3

This is an artificial example to demonstrates how the grammar can generate A-Teams consisting of long loops.

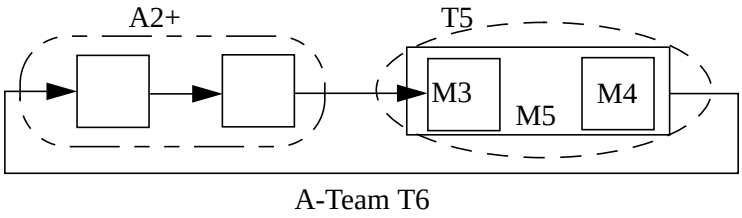


T1 = M1 (by rule T1)

A1+(< , >) = A1(< , M1) +> T1 +> A2(M1 , >) (by rule A2(a))

T2 = M2 (by rule T1)

A2+(< , >) = A1+(< , M2) +> T2 +> A3(M2 , >) (by rule A2(a))



T3 = M3 (by rule T1)

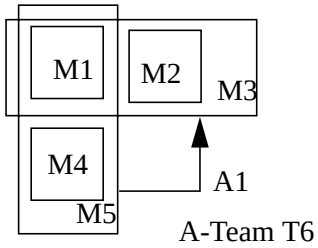
T4 = T3 + M4 (by rule T2(a))

T5 = T4 + M5 (by rule T2(b) : M5 = M3 ∪ M4)

T6 = T3 <+> A2+(M3, M5) (by rule T3(a))

3.4.5. Example 4

This is an interesting pathological case.



T1 = M1 (by rule T1)

T2 = T1 + M2 (by rule T2(a))

T3 = T2 + M3 (by rule T2(b) : M3 = M1 ∪ M2)

T4 = T3 + M4 (by rule T2(a))

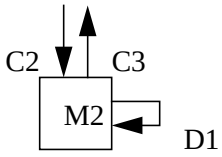
T5 = T4 + M5 (by rule T2(b) : M5 = M1 ∪ M4)

T6 = T5 <+> A1(M5, M3) (by rule T3(b) : (M3 ∩ M5) ≠ ∅)

This example shows how even a cyclic dataflow cannot guarantee that all data can flow in a cyclic manner, since data is never written to M4 or read from M2. Such an A-Team would probably be more useful if more agents were added to connect M2 to M4, or if it were used to create another agent such as A1+(<,>)=A2(< , M4) +> T6 +> A3(M3,>).

3.4.6. Example 5

This example is from [Talukdar98].

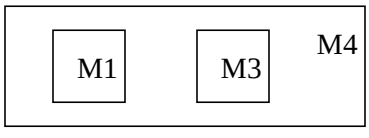


Agent A1+

T1 = M2 (by rule T1)

T2 = T1 <+> D1(M2, M2) (by rule T3(a))

A1+(< , >) = C2(< , M2) +> T3 +> C3(M2 , >) (by rule A2(a))

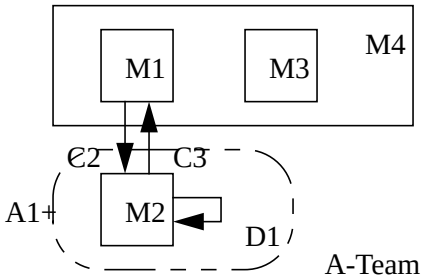


A-Team T5

T3 = M1 (by rule T1)

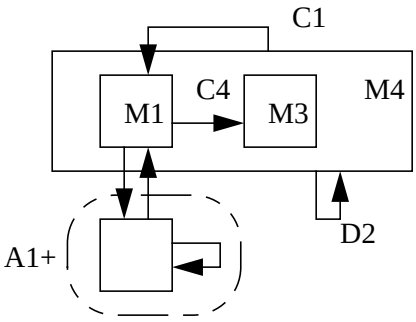
T4 = T3 + M3 (by rule T2 (a))

T5 = T4 + M4 (by rule T2(b) : M4 = M1 ∪ M3)



A-Team T6

T6 = T5 <+> A1+(M1, M1) (by rule T3(a))



A-Team T9

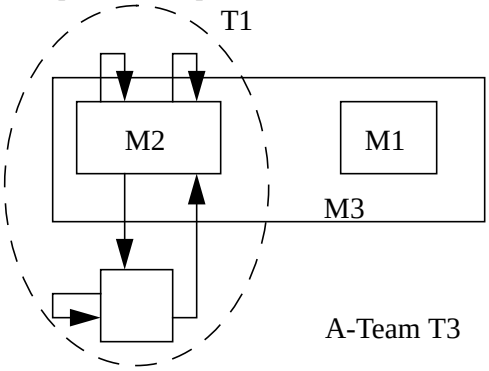
T7 = T6 <+> C1(M4, M1) (by rule T2(b) : P(M1, M4) = M4 C1(M4, M1) M1 (since (M1 ∩ M4) ≠ ∅))

T8 = T7 <+> C4(M1, M3) (by rule T2(b): P(M3, M1) = M4 C1(M4, M1) M1 (since (M3 ∩ M4) ≠ ∅))

T9 = T8 <+> D2(M4, M4) (by rule T3(a))

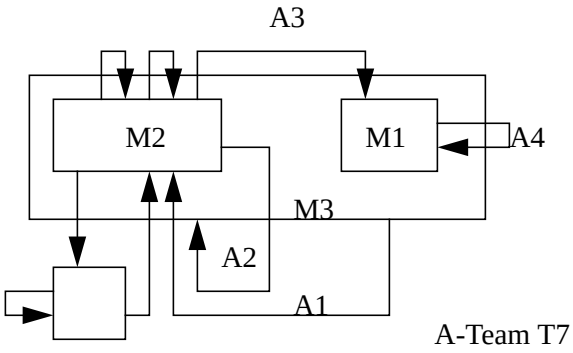
3.4.7. Example 6

A more complex example from [deSouza93]. This is an A-Team of intermediate complexity.



T2 = T1 + M1 (by rule T2(a))

T3 = T2 + M3 (by rule T2(b) : M3 = M1 ∪ M2)



T4 = T3 <+> A1(M3, M2) (by rule T3 (c) : P(M2, M3) = M3 A1(M3, M2) M2 (since (M2 ∩ M3) ≠ ∅))

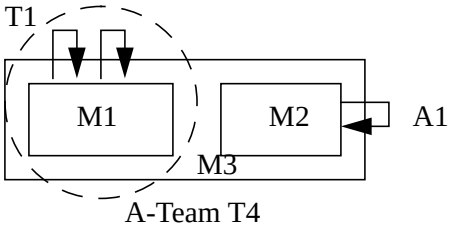
T5 = T4 <+> A2(M2, M3) (by rule T3(c) : P(M3, M2) = M3 A1(M3, M2) M2)

T6 = T5 <+> A3(M2, M1) (by rule T3(c) : P(M1, M2) = M3 A1(M3, M2) M2 (since (M1 ∩ M3) ≠ ∅))

T7 = T6 <+> A4(M1, M1) (by rule T3(a))

3.4.8. Example 7

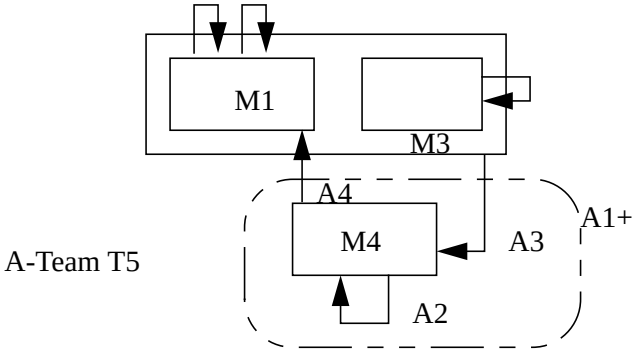
This example is the most complex from [deSouza93] and is one of the most complex A-Teams built. Although it shares some of its structure with the previous example, it is built using a slightly different sequence of steps.



T2 = T1 + M2 (rule T2(a))

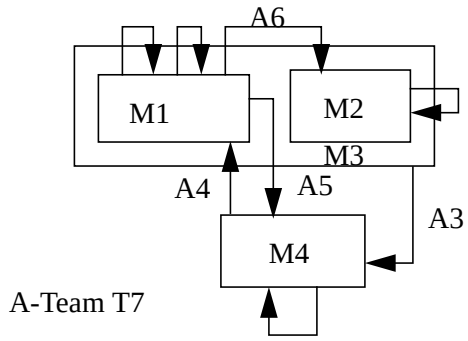
T3 = T2 + M3 (by rule T2(b) : M3 = M1 ∪ M2)

T4 = T3 <+> A1(M2, M2) (by rule T3(a))



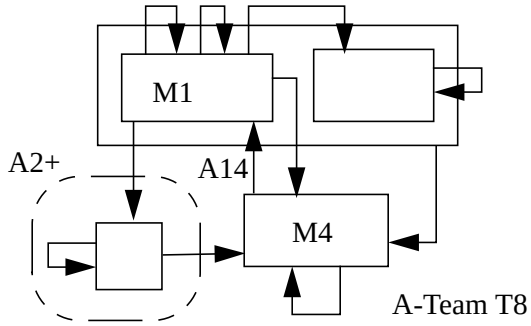
T5 = T4 <+> A1+(M3, M1) (by rule T3(c) : P(M1, M3) = M3 A1+(M3, M1) M1 (since (M1∩M3) ≠ ∅))

note: P(M1, M3) can also be written P(M1, M3) = M3 A3(M3, M4) M4 M4 A4(M4, M1) M1



T6 = T5 <+> A5(M1, M4) (by rule T3(c) : P(M4, M1) = M4 A4(M4, M1) M1)

T7 = T6 <+> A6(M1, M2) (by rule T3(c) : P(M2, M1) = M3 A3(M3, M4) M4 P(M4, M1); P(M4, M1) as for T6)



T8 = T7 <+> A2+(M1, M4) (by rule T3(c) : P(M4, M1) as for T6)

3.5. Summary

This chapter provides a grammar which facilitates the exploration of the space of A-Teams. Given a repository of primitives (Selectors, Schedulers, Operators, Elementary Memories), the grammar provides a formal set of rules to combine these elements to create a variety of memories, agents and A-Teams. There are only six construction rules in the grammar: A1 and A2 specifying the construction of agents; M2 specifying the construction of memories; and T1, T2 and T3 specifying the construction of A-Teams. The rules are restated here for convenience:

rule A1: Agent(< , >) = Elementary Agent: Scheduler + Selector + Operator

rule A2: Agent(< , >) = Agent1(< , M_{output}) +> A-Team +> Agent2(M_{input} , >)

rule M2: Memory = Elementary Memory ∪ Elementary Memory ∪ … ∪ Elementary Memory

rule T1: A-Team = Memory

rule T2: A-Team = A-Team1 + Memory

rule T3: A-Team = A-Team1 <+> Agent(M_{input}, M_{output})

The grammar attempts to provide sufficient precision to allow automation, but requires more precision in the connection rules. In implementation it is necessary to connect elementary agents to elementary memories. Agents whose input (output) space spans several elementary memories have separate input (output) channels connected to each. It is necessary to know exactly how each elementary agent connects to each elementary memory it reads from or writes to. This level of detail has never been captured in the dataflows used to represent A-Teams. For example consider an agent whose input space contains two identical elementary memories, and which gets a datum from each. There is no way to determine which of the two data the agent gets from which memory. This extra precision can be added to the grammar with some additional bookkeeping to track how exactly the agents are connected to the memories and requires no change to the conceptual structure of the grammar, or to the previously presented proofs.

The grammar is provably valid and I conjecture it is also complete since it can generate all the existing A-Teams. The grammar brings us significantly closer to the goal of automated A-Team construction from repositories of components.

4. OPTIMIZATION PROBLEMS AND METHODS

The remainder of this thesis is devoted to A-Teams for optimization. This chapter identifies some of the unsatisfied needs of real optimization problems. Later chapters show how A-Teams can meet these needs. This thesis uses nonlinear optimization and spatial layout for demonstration. An overview of problems and methods in these areas is provided here as background material. For more details, see [GMW81, TZ89, PTVF92].

An optimization problem P has the form P = (F, C, S, S_0) where F is a set of objectives, C is a set of constraints, S is the set of possible solutions and S_0 is a set of starting points. If S_0 is empty, the problem is often written P = (F, C, S). The goal of the optimization is to find an element in S that satisfies C and is desirable with respect to F.

Most solution methods have a predefined format according to which the problem must be posed. For example, the nonlinear programming methods that will be described later in this chapter require all constraints to be formulated as mathematical equality and inequality constraints, usually with continuous first derivatives. As a result of considerable research effort over the years, the solution methods are very fast and efficient for the restricted formulations they deal with. However it is often difficult to pose practical problems in the required format. In addition, reformulation of problems is often needed because of changes in requirements. It is also often necessary to re-evaluate trade-offs between the constraints and objectives during the solution process and to reformulate the problem being solved to improve the match between solutions found and the original problem.

We would like to have techniques that enable us to quickly synthesize solution procedures to keep pace with changes in requirements. The modular method approach that is developed further in this thesis provides such a way of rapidly synthesizing solution methods that are customized for each problem instance.

4.1. Modular Methods

Most optimization methods simultaneously tackle all the objectives and constraints of the problem and all the objectives and constraints must also be formulated in a similar manner. For example, sequential quadratic programming requires that all objectives and constraints must be formulated as differentiable algebraic equations. On the other hand simulated annealing does not take advantage of gradient information even if it is easily available. I label these methods monolithic to differentiate them from the modular methods that I will develop in this thesis.

Modular methods are based on a constraint oriented problem decomposition approach. Problems are solved by decomposition primarily for two reasons: smaller problems are easier to solve than large ones, and the partitioned sub-problems can often be solved in parallel for a gain in speed. Decomposition can be per-

formed in one of two ways: decomposing the decision space into subspaces, or decomposing the objectives and constraints into subsets.

Decomposition of the decision space is the traditional approach. If the problem is solved in parallel, a control processor usually distributes the sub-problems and then selects the best results. This approach is stated:

If solver i solves the problem $P_i = (F, C, S_i, S_{i\_0})$, i = 1 … n,

Then a team of n solvers tackling problems $P_1 \ldots P_n$ can solve the problem $P = (F, C, S, S_0)$

where $S = S_1 \cup S_2 \cup \ldots \cup S_n$

and $S_{i\_0} = S_i \cap S_0$, i = 1 … n.

Partitioning the decision space requires a set of essentially identical solvers that share the work. The amount of interaction between the solvers is limited once they have been assigned their tasks. An example of such an approach is branch and bound, which is described in more detail further in this chapter. Such a decomposition is not very flexible to changes in problem requirements such as addition of new constraints of reformulation of existing ones.

The alternative approach to problem decomposition, dividing the constraint set into subsets, is the basis of the modular methods that are the subject of one of the following chapters. Modular indicates that the method consists of a set of independent modules, each working on only a subset of the objectives and/or constraints, cooperating to solve the given problem. This modular approach is stated:

If each solver tackles the problem $P_i = (F_i, C_i, S, S_0)$, i = 1 … n,

Then a team of n solvers tackling problems $P_1 \ldots P_n$ can solve the problem $P = (F, C, S, S_0)$

where $C \subseteq (C_1 \cup C_2 \cup \ldots \cup C_n)$

and $F \subseteq (F_1 \cup F_2 \cup \ldots \cup F_n)$.

Such an approach is almost never used because difficulties in coordinating the solution efforts for the constraint subsets are considered too high. A means of overcoming these difficulties is developed further in this thesis. There are several advantages to such an approach which make it worthwhile. These advantages include:

(i) Customizability: If one has a set of modules that specialize in a small number (possibly just one) of constraints then customized solvers can be constructed by organizing an appropriate selection of modules. The solver can be easily reconfigured and extended by replacing or adding modules.

(ii) Maximal use of information: Each constraint can be tackled using all possible information about the constraint. For example if a large problem consists of both differentiable and non-differentiable constraints, the modules for constituent simple problems with differentiable constraints should be able to use the gradient information. The best techniques for each sub problem can be combined to solve the larger problem.

(iii) Ease of solver construction: It is usually easier to develop solution modules for simpler problems with fewer/simpler constraints.

(iv) Multiple modules: More than one module may be used for each sub-problem. Modules specializing in a particular sub-problem need not be identical, facilitating the simultaneous use of multiple methods.

(v) Parallel Execution: The modules may execute in parallel for speed. Networks of processors are often used to solve problems where a need for increased solution speed or for better solutions exists. However traditional approaches have used specially designed processor networks and matched algorithms. Most existing networks are much more loosely coupled and heterogenous. Modular methods can be executed on such networks.

Before we get into the details, I will present some background material on optimization and the problems that will be used for demonstrations in this thesis.

4.2. Classification of Optimization Problems

There are three main dimensions along which optimization problems are usually classified: Constrained vs. Unconstrained; Continuous vs. Discrete; and Local vs. Global.

4.2.1. Constrained vs. Unconstrained vs. Constraint Satisfaction

Unconstrained problems have C = Ø. Problems with F = Ø are referred to as constraint satisfaction problems.

Problems may be sub-classified in this category based on the number of objectives (single or multi-objective problems) and on the types of objectives and constraints. Examples of types of objectives include linear, quadratic, sum-of-squares, positive, concave etc. Examples of types of constraints include bound constraints, equality constraints, inequality constraints, linear or nonlinear constraints etc.

4.2.2. Continuous vs. Discrete

This indicates whether S is a continuous space, discrete, or a combination of the two. Discrete problems are often solved by first solving a continuous relaxation of the problem. Conversely, continuous global problems may be discretized to find good starting points.

4.2.3. Local vs. Global

Local problems are problems which either have a unique optimum, or the optimum sought is close to S_0. Global problems have multiple optima, and the best one is usually sought.

4.3. Nonlinear Programming Problems

Nonlinear Programming (NLP) is an important field in optimization and the class of single objective constrained optimization problems forms a significant part of nonlinear programming. Since this class of problems is used for demonstrations in this thesis, additional details are provided here. There are several popular algorithms for this class of problems as well as suite of benchmark problems to compare the algorithms. NLP problems include problems with multiple non-linear objectives and/or constraints. Many NLP problems are continuous. A typical NLP problem formulation is:

$$\begin{aligned} &\min_{x \in S} && f_k(x), k = 1 \ldots p \\ &\text{s.t.} && \begin{cases} g_i(x) \leq 0, i = 1 \ldots n \\ h_j(x) = 0, j = 1 \ldots m \\ x_l \leq x \leq x_u \end{cases} \end{aligned}$$

Problems are usually formulated to fit the best available solvers, and constraints may sometimes be posed as objectives and vice versa.

NLP problems are typically formulated with only one objective since most solution methods cannot handle multiple objectives. NLP includes linear programming where the objective and all constraints are linear, quadratic programming where the objective function is quadratic and the constraints are linear, and general nonlinear programming. In cases where there are no bound constraints on some of the decision variables, the bounds are considered to be at negative or positive infinity, which is usually represented by a large (in magnitude) number.

There are a number of equivalent formulations for such problems that are used by specialized solvers. It is possible to eliminate one form of constraint by converting those constraints to another form. Inequality constraints may be converted to equality constraints by the addition of slack variables at the cost of increased dimensionality of the decision space to be explored and equality constraints can be easily converted to pairs of inequality constraints at a cost of an increase in the number of constraints. Since it is fairly easy to transform between inequality and equality constraints, many solver implementations focus exclusively on one type.

4.4. Classification of Optimization Methods

Optimization algorithms are targeted to specific classes of problem. For example, a method might solve only constrained, single objective, continuous, local problems. In addition to being categorized according to the kind of problems they are designed to solve, optimization algorithms can also be categorized as (i) deterministic vs. stochastic and (ii) gradient based or not.

4.4.1. Deterministic vs. Stochastic

This indicates whether the algorithm introduces any randomness in its search of S. Deterministic algorithms usually have performance guarantees. Stochastic (or randomized) algorithms have been shown to find good solutions where speed is more important than quality of the solution. Stochastic methods are used most often for Global or Discrete problems.

4.4.2. Gradient based vs. Non-Gradient based

This indicates whether the algorithm uses gradient information in its search of S. Gradient based algorithms are usually applied when the functions in F and C have continuous first and possibly continuous second derivatives, and S is continuous. Gradient based algorithms are almost always much faster than non-gradient based algorithms and they are often applied to relaxations of discrete optimization problems in order to locate the neighborhood of good solutions.

4.5. Descriptions of some Optimization Methods

Each optimization method is classified according to the categories described:

(i) C for constrained, U for unconstrained and * for both

(ii) C for continuous, D for discrete, * for both

(iii) L for local, G for global, * for both

in addition, optimization methods are also classified as

(iv) D for deterministic, S for stochastic, and * for both

(v) G for gradient based, N for non-gradient based

Table 2. Classification of Common Optimization Methods

Method	Classification
Greedy Search or Hill Climbing	U,C,L,D,G or U,D,L,*,N
Simplex Method for Linear Programming (LP)	C,C,L,D,G
Quadratic Programming (QP)	C,C,L,D,G
Sequential Quadratic Programming (SQP)	C,C,L,D,G
Simplex/Complex Methods	U,C,L,D,N
Branch and Bound (B&B)	C,D,G,D,N
Genetic Algorithms (GA)	U,D,G,S,N
Simulated Annealing (SA)	U,*,G,S,N
Tabu Search	U,D,G,D,N

4.5.1. Greedy Local Search or Hill Climbing

This is the oldest, simplest and most widely used iterative improvement method. The idea is to move in the best possible direction at each iteration until an optimum is reached. The continuous version follows the gradient at each step. The discrete version replaces the iterate with the best improving point in its immediate neighborhood. If possible the entire neighborhood is systematically searched. If the neighborhood is too big it may be searched stochastically. Greedy search is fast, though will get stuck in the nearest, possibly undesirable, optimum to the starting point.

Nonlinear continuous constrained optimization problems are usually solved using iterative descent methods. Constrained problems are often first transformed into unconstrained problems by using a merit function (or penalty function) that characterizes the problem. The merit function is usually a weighted sum of the objective and (violated) constraints and is formed as shown below:

$$M(x) = f(x) + w_i \sum_{g_i(x) > 0} p(g_i(x)) + w_j \sum_{j=1}^{m} p(h_j(x))$$

where $p(y) = y^2$ or $p(y) = |y|$ are commonly used transformations.

The weights for the constraints are set sufficiently large so that the constraints are satisfied at the minimum of the merit function.

Consider an iterative descent method. Let M(x) be the merit function and x_k be the value of the solution at iteration k. The iteration consists of the following two steps:

Step 1: find a descent direction Δx_k

Step 2: determine a step length α s.t. $M(x_k + \alpha \Delta x_k) \leq M(x_k)$

Step 3: set the value of the next iterate: $x_{k+1} = x_k + \alpha \Delta x_k$

This process is repeated until either an optimum is found or a predetermined limit on the number of iterations is reached.

There are several choices for: the algorithm used to determine the descent direction; the merit function; and the algorithm used to determine the step size. The most important difference, the one that is used to characterize the various methods, is the direction determination method. Two important methods are penalty methods and sequential quadratic programming.

Penalty methods use the merit function directly to determine the descent direction by computing its gradient: $\Delta x_k = -\nabla M(x_k)$

4.5.2. Simplex Method for Linear Programming (LP)

This is an example of a gradient based single-objective multiple-constraint continuous gradient-based problem class. The objective and constraints are all linear. $S=R^q$. There is a family of deterministic, gradient based methods to solve such problems. This class of problem is probably the most widely studied and the Simplex method - a method of continuous improvement by following the edges of the polyhedron defined by the constraints is the most widely used of all optimization methods.

4.5.3. Quadratic Programming (QP)

This class of problems resembles linear programming, except that the objective is quadratic. The term QP is usually restricted to problems and methods for a non-negative quadratic objective.

4.5.4. Sequential Quadratic Programming (SQP)

This is one of the most widely used methods for solving single-objective NLP problems. The SQP algorithm has the following form:

Step 1: Generate a positive QP approximation P_{QP} to P at x_k, the current value of the iterate.

Step 2: Solve P_{QP} to find a descent direction $\Delta x_k = x^{QP} - x_k$ where x^{QP} is the solution to P_{QP}

Step 3: Determine a step length α s.t. $M(x_k + \alpha \Delta x_k) < M(x_k)$ where $M(x_k)$ is a merit function that combines the objective and constraints in P.

Step 4: Update the value of the iterate $x_{k+1} = x_k + \alpha \Delta x_k$. If $x_{k+1} = x_k$ stop, else repeat from Step 1.

The SQP typifies how difficult problems can be solved by solving a sequence of simpler problems.

Generating a positive QP approximation:

Given a problem P = (f, C, S, S_0), a positive QP approximation $P_{QP} = (f_Q, C_L, S, x_k)$ is generated at each iteration by using the Taylor Series to expand a quadratic approximation to f and a set of linear approximations to the functions in C around the iterate x_k.

$$f_Q(x) = f(x_k) + \nabla f(x_k) \cdot (x - x_k) + (x - x_k)^t \cdot B_k \cdot (x - x_k)$$
$$c_{iL}(x) = c_i(x_k) + \nabla c_i(x_k) \cdot (x - x_k)$$

The hessian (B_k) for the quadratic approximation to the objective is almost never determined analytically. It is numerically updated at each iteration of the SQP. The most popular update formula is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) [GMW81]. This formula guarantees that B_k is a positive matrix, ensuring that the quadratic approximation is convex. This generates a QP for which a convex merit function can be easily constructed making it easier to find the minimum.

Given: two iterates x_{k-1} and x_k.

Let:
$$s_k = x_k - x_{k-1}$$
$$y_k = \nabla f(x_k) - \nabla f(x_{k-1})$$

Then: $$B_k = B_{k-1} - \frac{(B_{k-1} \cdot s_k) \times (B_{k-1} \cdot s_k)}{y_k \cdot y_k} + \frac{y_k \times y_k}{y_k \cdot s_k}$$

This is guaranteed to have a positive value only iff $y_k \cdot s_k > 0$. If this condition does not hold, the approximation to the Hessian from the previous iteration is used directly.

4.5.5. Simplex Methods / Complex Methods

Simplex methods [GMW81] are an interesting class of multi-point non-gradient based methods for continuous local optimization in multi-dimensional spaces.

Pick N+1 distinct points that span the space - this is the simplex from which the method gets its name.

Step 1: Find the point with the worst evaluation P.

Step 2: Find the centroid C of the remaining points.

Step 3: Replace P with a point having a better evaluation along the line passing through P and C. If no such point is found, shrink the simplex by moving all points closer to the centroid of the simplex. If the simplex has degenerated to a single point, stop. Else go to Step 1.

Nelder and Mead's method [PTVF92] is one of the more popular instances of Simplex Methods. There are also variants that use more than N+1 points to reduce the likelihood that the simplex will lose dimensionality during the process that are referred to as Complex Methods.

4.5.6. Branch and Bound (B&B)

This is a partition and prune method, used primarily for discrete or mixed continuous-discrete problems. It is guaranteed to find the global optimum, but is often very time consuming. The B&B algorithm has the following form:

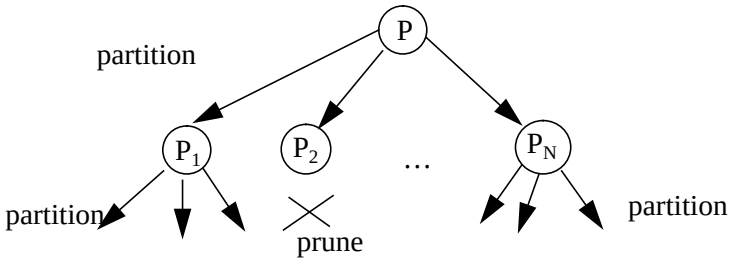
Step 1: partition S into S_1, S_2 … S_N s.t. $S = S_1 \cup S_2 \cup \ldots \cup S_N$

Step 2: identify which of the subproblems $P_i = (F, C, S_i)$ contain no good solutions

Step 3: solve the remaining subproblems $P_i = (F, C, S_i)$ in a similar fashion

The difference between B&B implementations is in the details of Step 2. The identification of regions of the space with only undesirable solutions is done by developing good upper and lower bounds on the quality of solutions in each partition of S. The better (and tighter) the quality of the bounds, the smaller the region of S that must explicitly enumerated. Most of the development effort in efficient B&B algorithms is in finding good ways of estimating the bounds.

The B&B algorithm recursively performs an implicit enumerative search over S. It can be visualized as traversing a tree of related problems. The root node is the original problem. Nodes are partitioned into sub problems. Nodes corresponding to unpromising sub-problems are pruned.



4.5.7. Genetic Algorithms (GA)

A Genetic Algorithm [Goldberg89, Davis91] is a non-gradient based stochastic method. It is widely applicable, and is used mostly to solve discrete problems. The basis of GA is the process of natural selection. GAs work as follows:

Step 0: Represent the solution as a (bit) string. Generate an initial population randomly.

Step 1: Evaluation - each solution is evaluated with respect to a merit function and assigned a “fitness” value. If the quality of the population has not improved (e.g. the best solution in the population has not changed) over many generations, then stop.

Step 2: Selection - Choose those solutions in the population which will be allowed to reproduce. Higher fitness (quality) solutions have a higher probability of being selected to reproduce.

Step 3: Reproduction or Combination of selected solutions - the standard operators are Crossover (merging of two solution) or Mutation (random permutation of a solution). The selected solutions are combined to create a new population generation.

4.5.8. Simulated Annealing (SA)

Simulated Annealing [KGC83] is another non-gradient based stochastic method based on the model of annealing metals. SA is an iterative method where solutions are subject to random perturbations. Perturbations that lead to better solutions are always accepted. Perturbations that lead to worse solutions are sometimes accepted based on a time dependent parameter called the temperature. The process is modeled after the behavior of molecules in a metal being annealed. At high temperatures, the molecules in the metal move about randomly. As the temperature decreases, the molecules tend to favor motions that result in lower energy and greater stability. Simulated annealing represents energy with the evaluation of the objective function. Perturbations to the solution are accepted with a bias towards those that improve the objective function, representing a reduction in energy. As the temperature decreases the bias against perturbations that increase energy increases. The temperature parameter is monotonically decreased as the optimization process proceeds, and SA transitions from behaving like a random walk through decision space to a stochastic descent in decision space. The algorithm has the following form:

Step 0: set the initial temperature T0, and the starting point x_0.

Step1: For some number of iterations do:

Step 1a: Generate a perturbation to the current iterate x_k.

Step 1b: If the perturbation is better than x_k accept it. If it is worse, accept it with a probability proportional to exp(-Δc/T) where Δc is the difference in the evaluations of x_k and the perturbation.

Step 2: If no perturbations are accepted, then stop. Otherwise reduce the temperature T and go to Step 1.

Simulated Annealing is fairly easy to implement, and is a popular approach for difficult problems for which good special purpose algorithms do not exist.

4.5.9. Tabu Search (TS)

Tabu Search [Glover89] is a relatively new iterative method used for discrete problems. Like SA, TS tries to avoid getting trapped in local minima by allowing worsening moves or modifications to the solution. This is done by keeping a list of recently visited solutions or, more commonly, a list of recently applied moves. This list is known as a Tabu List (hence the name Tabu Search) since the moves on the list are taboo. Tabu Search algorithms have the form:

Step 0: Set the tabu list T = Ø, the initial solution x_0

Step 1: Determine M, the set of moves applicable to the current iterate x_k.

Step 2: Determine the list of non-tabu moves N = M\T.

Step 3: If N = Ø, stop. Else apply the move m with the best resulting evaluation to x_k and update the iterate.

Step 4: If |T| = maximum allowed, remove the oldest element in T. Add m to T and go to Step 1.

The justification for the tabu list is that in order to escape from local minima, it is necessary to take worsening moves. However, taking just the best available move in the neighborhood of a point might result in cycling through a sequence of points. The tabu list prevents cycles of length less than the length of the list.

The original idea of Tabu Search was to explicitly store the values of recent iterates. However, implementations prohibiting moves were found to work better. In order to prevent marking unseen desirable solutions tabu, an *aspiration criterion* is set, and a move on T is allowed in Step 3 of the algorithm if it exceeds this criterion. A commonly used criterion is the evaluation of the best solution found so far. If the tabu move would create a better solution, it is allowed. The size of the tabu list is usually set between 7 and 12. The size is usually changed periodically to a random value within that range. An alternate heuristic: increasing the size of the list every time a tabu solution is encountered with a periodic reduction in size is suggested in [BT94].

TS is not as widely used as SA, and its applications restricted to discrete problems; mostly permutation based combinatorial optimization problems. There are no applications for continuous problems, though

some ideas on how it could possibly be done have been suggested in [Glover94]. Developing a good TS implementation is an art. A wide variety of additional ad-hoc heuristics have been suggested for guiding the search, though there is inconclusive evidence about which is best. These heuristics include adding "intensification" and "diversification" phases, corresponding to greedy local search and exploratory search.

4.6. Global Search

Global search problems are those with multiple optima in the search space, with the goal of finding the best one. I believe that almost all global search methods follow one of the following two approaches. The first approach is to follow a path through S and with a bias towards taking improving steps, but allowing occasional worsening steps to move from the neighborhood of one optimum to the neighborhood of another. The second approach is to use multiple points, which may be generated randomly or systematically, in order to get coverage of S and find the best optimum in S.

4.6.1. Path Following Methods

These methods attempt to find the best optimum by following a path through S. This path is a sequence of iterates generated by the method. The idea is to try to improve the objective value with each step, but allow (or force) occasional worsening steps to escape from bad optima. Simulated annealing is a path following method where the trajectory of the path is biased toward directions that result in improvements. Tabu search is also a path following methods where the tabu list reduces backtracking onto the path. Other methods include trajectory methods such as golf methods which are based on a physical model of particles moving with momentum through a search space. The idea being that the particle is attracted towards optima, but has sufficient momentum to escape shallow basins.

4.6.2. Multiple Point Methods

Multiple point methods check many points in S in order to find the regions of S that contain better optima. Random Multi Start (RMS) is the most commonly used multiple point method. Points are generated at random in S, and a local search method used to find the nearest optimum. The best of a set of such searches is selected. Most multiple point methods, such as genetic algorithms, make more systematic use of the population of points. Other multiple point methods include covering methods where a set of points is generated to be well spread out over S, and neighborhoods of high quality points explored in greater detail. Cluster methods attempt to identify basins of optima by finding clusters of high quality points. A single local search is then performed in each cluster, starting for example from the best point in the cluster.

4.7. Spatial Layout Problems

Spatial Layout is an important problem that appears in several areas of engineering and design, including Operations Research (OR) and Industrial Engineering (IE) where the problems are usually referred to as

“packing” problems; Electrical Engineering (EE), where the problems are called “layout” or “placement” problems; and Mechanical Engineering (ME), where the problems are related to “configuration design” problems. Layout problems are difficult to solve computationally. The problems usually have many disconnected feasible regions and many optima. In addition, for 3D problems, the computational geometry requirements are very expensive, straining the abilities of even the latest computers [LM96]. Layout problems are used as an example of challenging applications for some of the ideas developed in this thesis, and I will provide a short overview here.

The focus in Electrical Engineering emphasizes connectivity costs between objects since there is some freedom on the size and shape of objects (which may lie within some neighborhood of the given target). The problems derive from the need to transform a circuit description, consisting of a set of modules and nets (wiring connecting the modules), into a layout in 2D which is then implemented in silicon to make integrated circuits [MSA87]. There are several steps to this process, consisting of: partitioning the circuit into components; placement of modules of fixed sizes without overlap onto a 2D surface representing the chip so as to minimize wiring requirements; routing the wiring around the modules on the surface; and compaction of the final arrangement. Placement usually implies locating rectangles of fixed size. Floor-planning is placement of rectangles that may vary in size. The main concern is to fit the rectangles and minimize the length (and quality, as measured for example by the number of bends) of wiring used to connect components. A good current survey for EE layout problems is found in [ALLP97]. Problems in EE are primarily 2D due to the planar nature of the integrated circuit technology. There is some extension into the third dimension, but this consists of stacking layers of components, which can be considered as a set of 2D problems. The objects and container are rectangular in shape. Such problems consider much simpler objects than the irregular, non-convex, 3D objects found in electromechanical assemblies.

Floor-planning problems in EE are often solved by heuristics using tree representations of the layout. The trees are usually binary trees with a rectangle at each node and leaf nodes are the modules to be placed. Nodes may branch in one of two ways corresponding to a vertical or horizontal guillotine cut of the rectangle at the node. A collection of such heuristics can be found in [Lengauer90]. Genetic Algorithms that use such heuristics as decoders are also used. Placement and routing problems in EE are usually solved by Simulated Annealing, with Genetic Algorithms or Tabu Search as alternatives.

The focus in OR, IE and ME has been on maximizing packing density for problems in mechanical assembly and container packing and other constraints such as stability or accessibility are usually ignored. Much of the work in OR is on 2D problems related to theoretically oriented bin-packing or knapsack problems. This work focuses on packing rectangles in rectangular boundaries. The OR literature focuses on the development of heuristics, mainly for 2D or 3D boxlike objects in boxlike housings. There is just a little exploratory investigation into other methods such as SA or Tabu Search. Nonlinear programming methods, popular in

other OR applications, are not used due to the extreme difficulty in problem formulation and the localized nature of these methods.

A survey of the different instances and nomenclature related to packing problems can be found in [Dyckhoff90]. This survey suggests a classification of the problems into 4 main categories (i) dimension of the objects (1, 2, 3, N-D) (ii) kind of assignment (bin-packing or knapsack, depending on whether all or only a subset of the objects must be packed) (iii) type of containers (single, multiple identical, multiple different) and (iv) type of objects (few distinct, many distinct, many similar, identical). Since most of the problems deal with rectangular objects, rectangles of different aspect ratio are considered distinct, and similar if they differ by a scaling factor.

A general survey of packing problems in OR is [DD92]. It describes the main types of problems that have been considered in the literature, classifying the problems by dimension (1, 2 or 3D). This paper also describes the use of non-heuristic based methods (such as branch and bound) for 2D rectangle packing, which require considerable computation time even for small problems. The literature focuses on packing density, usually ignoring other (possibly important) constraints such as accessibility or separation requirements. The 3D problems in OR or IE usually involve packing cuboids into other cuboids (e.g. for packing boxes in containers for shipping). In addition to the packing constraints, the additional constraint of locating the center of gravity of the packed container low and near the center (in the horizontal plane) of the container is often considered.

4.8. Methods used for Spatial Layout

There are three main categories of methods for packing rigid objects: Ad-Hoc (or specialized) Heuristics, Genetic Algorithms, and Simulated Annealing. Gradient based optimization methods such as nonlinear programming methods are normally not applied to layout problems because of the difficulty in setting up the required problem formulation.

4.8.1. Ad-Hoc Heuristics

These are most widely used in the OR and IE literature. For three dimensional problems, [DD92] states that the approaches are “entirely composed of a series of ad-hoc heuristic rules derived from common sense” but that “practical experience suggests that any of [these] methods will out-perform manual methods on average.” The section on non-rectangular packing indicates the absence of methods here, with most of the work being in circle and sphere packing, primarily in the mathematical and recreational literature. It is pointed out that existing heuristics designed for rectangles are difficult to extend to these problems. Most heuristics locate the objects one at a time in the container, with some kind of layering rule to indicate how objects will be placed. The search space is usually limited to placing objects so that their corners align with already placed objects. Examples of such heuristics can be found in [Lengauer90, Aziz91, GMM90, HT90].

4.8.2. Genetic Algorithms (GA)

Genetic algorithms for spatial layout might either store the solution directly, or might make use of an encoded representation along with a decoder which maps from the solution string to an actual solution. For example, the solutions might consist of the location and orientation of the objects (direct representation), or might be an ordering of the objects (encoded representation) which is then mapped into an actual location. Alternatively, a combination of the two representations may be used such as an ordering and orientation for the objects, where the encoded part is used only to determine translations. Genetic Algorithm approaches to packing [KMK91, WF94, IBKRW97] typically use partially or completely encoded representations with a packing heuristic to locate the parts. They extend the heuristic methods by allowing for some randomization. The use of packing heuristics greatly limits the size of the search space and often ensures feasibility in terms of overlap and protrusion.

GA in 3D have been applied to packing cuboids in cuboids while optimizing the location of the center of gravity in the X-Y plane [KMK91, WF94]. Extensions to cuboids with holes (shoe-box-like objects with thick walls) have recently been tried [IBKRW97]. Related work in 3D pipe routing uses tessellated, or faceted, representations of objects to speed up the evaluations required by GA in [SCF97], allowing more complex objects to be considered.

4.8.3. Simulated Annealing (SA)

SA has been used extensively for layout problems in integrated circuit design. This led to the use of SA for packing 3D objects [SC93, SC95] which demonstrated that SA works well for packing cuboids and cylinders in cuboids. SA has also been used for packing cylinders subject to connectivity costs between objects. A recent development in this methodology is the use of multi-resolution models for fast interference detection, which reduces the time required for evaluation and facilitates the packing of non-convex objects by SA [KCR96].

4.8.4. Other Methods

An atypical approach to packing can be found in [KG91] where a nonlinear programming approach is used. The formulation of the problem is extremely complicated (the objects are represented as unions and intersection of half spaces) and the only example with 3D objects has only 3 objects, one of which is a cuboid. It appears unlikely that such an approach will ever be competitive with heuristic methods. Another approach which has been considered is Tabu Search. For example [Dowsland93] states that it would probably work better than SA. However, empirical evidence to confirm or deny this claim is lacking.

4.9. Summary

This chapter presented some of the important classes of optimization methods and solution algorithms. Good solution method exist for a restricted class of problems, and consequently problems are usually first (re)formulated to fit existing solvers. There are good greedy deterministic algorithms for solving local optimization problems. Global optimization problems are usually much harder to solve, and randomized methods can usually find good optima in reasonable time, although they cannot guarantee to find the best one. There are many good methods for single objective continuous unconstrained problems. The use of A-Teams for global problems, multi-objective problems and multi-constraint problems will be explored in this thesis.

Real world problems are often difficult to formulate and solve. Spatial layout is an example of such problems. Special-purpose and ad-hoc algorithms are usually difficult to extend as problem requirements change. General purpose algorithms such as GA and SA often ignore useful information leading to inefficiencies. Real world problems with multiple, conflicting, objectives are usually formulated as single objective problems by combining the objectives. Trade-offs must be made implicitly by modifying the merit function. A-Teams provide a way of searching for Pareto frontiers, allowing the trade-off process to be made explicit to the user. In addition, A-Teams allow independent modules that are specialized for each objective to be combined into larger solvers, facilitating the construction of customized reconfigurable solvers. This will also be explored further in this thesis.

5. MODULAR ITERATIVE ASYNCHRONOUS OPTIMIZATION

Problems can be partitioned into sets of problem primitives which consist of subsets of the objectives and/or constraints of the problem. Modular methods consist of modules that specialize in these primitives and are combined to solve the overall problem. Consider a set of modules that solve problem primitives of the form $P_i = (F_i, C_i, S)$. This chapter will demonstrate how such modules, designed to solve problems $P_1 \ldots P_n$ for example, can be organized to solve problems of the form $P = (F, C, S)$ where $C \subseteq (C_1 \cup C_2 \cup \ldots \cup C_n)$ and $F \subseteq (F_1 \cup F_2 \cup \ldots \cup F_n)$. These modules will be referred to as specialized agents [TSC97] since they specialize in part of the problem and are implemented as asynchronous autonomous agents.

These agents must be able to update their own constraints and be able to use limited information available in a standard format for the other constraints. The agents can converge to mutually acceptable solutions to the overall problem if they exchange information regularly. One way for the agents to coordinate their efforts is for each to take very small steps at each iteration. This is, however, costly in terms of speed and efficiency. A second way is for agents to specialize in overlapping primitives which would reduce the likelihood of seriously harming the work of others. A third way is for agents to exchange information in the form of guidelines or approximations to their goals. These specialized agents can be organized in A-Teams to collaborate on the overall problem. This approach also allows multiple agents specializing in the same constraints to be used together, and allows agents to work in parallel.

Two approaches to designing modular methods will be developed in this chapter. One approach is to transform a synchronous iterative process that tackles all the constraints in the problem simultaneously into a set of asynchronous iterative modular processes that handle problem primitives and thereby into a set of cooperating asynchronous agents. I believe that such an approach can be used with many synchronous iterative processes. Nonlinear programming (NLP) is an area where many good iterative processes exist, one of which is sequential quadratic programming (SQP). This chapter shows how SQP can be transformed into an asynchronous team of agents that specialize in problem primitives. This chapter proves that such transformed methods converge to the optimum under certain conditions, and experimentally demonstrates that such transformed methods converge to the optimum even if those conditions are relaxed.

The other approach to designing modular methods is the ad-hoc approach which can be applied to almost any problem. The transformation approach works better than the ad-hoc approach for NLP problems. However, the transformation approach cannot be used for many problems in design and engineering where there are no good synchronous iterative methods. In such cases, the ad-hoc approach is the only possibility, and this chapter uses spatial layout problems to demonstrate that it can be very effective compared to commonly used solution methods such as simulated annealing.

5.1. The Nonlinear Programming Problem

Nonlinear programming (NLP) has many applications, many good monolithic solution methods, and a widely used set of standard benchmark problems. A typical NLP problem formulation is :

$$\min_{x \in R^n} f(x)$$

$$\text{s.t.} \quad \begin{cases} g_i(x) \le 0, i = 1 \ldots l \\ h_j(x) = 0, j = 1 \ldots m \end{cases}$$

A merit function M(x) is selected such that the minima of the merit function are solutions (optima) of the NLP problem. Iterative methods solve the NLP problem by finding the minimum of the merit function.

The merit function usually has the form:

$$M(x) = f(x) + w_i \sum_{g_i(x) > 0} p(g_i(x)) + w_j \sum_{j=1}^{m} p(h_j(x))$$

where $p(y) = y^2$ or $p(y) = |y|$ are commonly used.

The methods generate a sequence of iterates $x_0, x_1, x_2 \ldots x_k$

Each new iterate is generated in a two step process consisting of direction determination and a stepsize selection such that such that $M(x_{k+1}) < M(x_k)$.

$x_{k+1} = x_k + \alpha_k \Delta x_k$

where Δx_k is the direction and α_k is a scalar stepsize.

Since it is relatively straightforward to transform between equality and inequality constraints, most NLP solver implementations handle only one kind. I consider only inequality constraints in the implementations of the modular methods below.

The modular methods that will be developed for NLP problems follow the two step direction-and-stepsize approach. A representative set of problems, selected from the popular benchmark problems in [HS81, FP90], is used to evaluate these methods. The table below provides a brief description of the benchmark problem characteristics. Details of the problems can be found in the appendix. Problems 4, 8, and 9 had equality constraints which were converted to a pair of inequality constraints. The table lists the total number of constraints as used to benchmark the problems and equality constraints are counted twice. The type of the

problem indicates whether the problem has practical origins or is an artificially generated problem of theoretical interest. Problems for which the objective and constraints have first and second derivatives in the entire feasible region are considered regular.

Table 3. Description of test problems

#	source, problem id.	type	objective	constraints	regularity	variables	constraints
1	FP 2.3	theoretical	quadratic	linear	regular	13	9
2	FP 3.1, HS 106	practical	linear	quadratic	regular	8	6
3	HS 100	practical	polynomial	polynomial	regular	7	4
4	HS 80	practical	general	polynomial	regular	5	6
5	HS 113	practical	quadratic	quadratic	regular	10	8
6	HS 117	practical	polynomial	quadratic	regular	15	5
7	HS 56	theoretical	polynomial	general	regular	7	8
8	HS 107	practical	polynomial	general	regular	9	12
9	HS 85	practical	general	general	irregular	5	38
10	HS 67	practical	general	general	irregular	3	14

5.2. Asynchronous Ad-Hoc Method (AAM)

This section will develop an asynchronous ad-hoc modular method (AAM) for NLP problems that is loosely based on the penalty method. An iteration of the penalty method is summarized below:

The direction is first computed: $\Delta x_k = -\nabla M(x)\,|_{x_k}$

If only inequality constraints are handled, $\Delta x_k = -\nabla f(x) + w_i \sum_{g_i(x) > 0} -\nabla g_i(x)$

The stepsize is then determined: $\alpha_k = \arg\min_a M(x_k + a\Delta x_k)$ where $a \geq 0$.

and the new iterate generated: $x_{k+1} = x_k + \alpha_k \Delta x_k$

The AAM also computes the direction at each iteration using gradients of the objective and constraints, but rather than recomputing all these values at each iteration, it uses gradients computed at previous iterations for most of the functions. The AAM consists of agents that specializing in either the objective or one of the constraints. At each iteration, an agent is selected with a bias towards those with violated constraints or those that have not worked on the iterate recently. An iteration for the module specializing in constraint j can be written:

$x_{k+1} = AAM_j(\{x_k, \ldots, x_{k-h}\}, f(x_p), \nabla f(x_p), \{g_i(x_{li}), \nabla g_i(x_{li})\})$

where: $x_{lj} = x_k$

x_p, $\{x_{li}\}$ are the most recent iterates at which the objective and constraints were updated

$\{x_k, \ldots, x_{k-h}\}$ is the recent trajectory or history of the iterates through S.

The steps followed by the module specializing in a constraint j at x:

Step 1: Compute $g_j(x)$. If the constraint is satisfied, do nothing.

Step 2: Compute the preferred direction $-\nabla g_j(x)$

Step 3: Classify the recommendations of other modules into three groups; those which will be ignored, those which will be followed, and those which the module will try not to oppose.

Step 4: Use the recent history (recent points visited in the solutions trajectory) to determine trends.

Step 5: Use the preferred direction, recommendations and trend information to determine a direction in decision space.

Step 6: Use the recommendations or history to determine a step size and take a step.

Step 7: Update the values of the iterate, the recommendation and the history.

Figure 1. The Asynchronous Ad-Hoc Method (AAM)

Modules do not communicate directly with each other, but by adding information to the iterate. In the case of the AAM, modules add a recommendation for direction, as well as a time-stamp for when the recommendation is made. The time-stamp on recommendations allows modules to discount old information and determine when they need to work. Modules update their recommendation every time they operate on a solution.

5.2.1. Implementation

This section provides details for the solution representation, as well as for four alternative implementations (AAM_1-AAM_4) for nonlinear programming problems of the Asynchronous Ad-Hoc Method (AAM) described in Figure 1.

Solution Representation

A solution consists of: $\{x_k, E, R, H, k\}$

where: x_k is the value of the current iterate

$E = \{g_0, g_1, \ldots, g_n\}$ is the latest available evaluations (here g_0 is used in place of f to simplify the notation)

$R = \{(r_i, k_i)\}$ is a list of preferred directions of other modules ($r_i = -\nabla g_i(.) / | \nabla g_i(.) |$), accompanied by a time stamp k_i which is the iteration at which this direction was last updated

$H = \{x_{k-1}, x_{k-2}, \ldots, x_{k-h}\}$ is the history of the current iterate, consisting of a list of the h most recent iterates. (The results provided below all had h=10)

k is the current iteration

Implementation (AAM_1)

The steps followed by the module for constraint i are:

Step 1: Select a solution from the memory for which the constraint g_i is violated

Step 2: Compute the preferred direction of the module. For the NLP problems, use the gradient to determine this.

$r_i = -\nabla g_i(x_k) / | \nabla g_i(x_k) |$

update R with the current value r_i and k_i

Step 3: Classify the remaining components of R into three categories R_{ignore}, R_{follow}, $R_{orthogonal}$. Ignore old recommendations, those for which the recommendations are almost orthogonal to the preferred direction, or those for which the constraint seems satisfied. Go orthogonally to those recommendations that oppose the preferred direction, or for which the constraint is near its active point. Follow the rest. The following constants used below were determined arbitrarily using what appeared to be reasonable values: the age at which a recommendation was considered outdated (200); a range of angles around 90˚ (85˚ to 95˚) and a small magnitude close to zero (1e-5).

$R_{ignore} = \{r_j \in R \mid (k_j < k - 200) \text{ or } (85^\circ < \angle(r_i, r_j) < 95^\circ) \text{ or } (g_j < -1e\text{-}5)\}$

$R_{orthogonal} = \{r_j \in R \backslash R_{ignore} \mid (\angle(r_i, r_j) > 95^\circ) \text{ or } (|g_j| < 1e\text{-}5)\}$

$R_{follow} = R \backslash (R_{ignore} \cup R_{orthogonal})$

Combine all the recommendations of other modules that will be followed.

$Q' = \sum [r_m], \ \forall \ r_m \in R_{follow}$

$Q = Q' / |Q'|$

Step 4: Combine H to capture trends in the trajectory of the solution through the decision space.

$T' = \sum [(x_k - x_{k-m})/m], (1 \le m \le W), x_{k-m} \in H$

$T = T' / |T'|$

Step 5: Compute a direction in which the module will take a step. This consists of the preferred direction perturbed by recommendations of other modules, trends in the trajectory of the solution, and a small amount of randomization.

$\Delta x_k' = r_i + U + p_r Q + p_t T$

where U is a small random vector (of length ≤ 0.1 so that it is a fraction of the magnitude of the other components of $\Delta x_k'$) and p_r and p_t are experimentally determined parameters. Since all the direction vectors are normalized, reasonable values of these parameters are on the order of 0.5

$P\Delta x_k'$ = projection of $\Delta x_k'$ into the subspace spanned by $R_{orthogonal}$

$\Delta x_k = \Delta x_k' - P\Delta x_k'$

(note: if $\Delta x_k = 0$, which occurs very rarely, use $\Delta x_k = \Delta x_k'$)

normalize direction

Step 6: Determine a step size along direction to find x_{k+1}

$\alpha_k = 1.5 * | T' |$

to facilitate convergence a limit that decreases as k increases is imposed on the stepsize:

$\alpha_k < q * e^{-0.001* k}$, where q is set to a small fraction of the smallest allowed range of the variables, to reflect the scale of the decision space

$x_{k+1} = x_k + \alpha_k \Delta x_k$

Step 7: Update H by adding x_k and removing x_{k-W}

Replace x_k with x_{k+1}, and k with k+1 in the solution

The agent for the objective uses the same algorithm as the agents for the constraints except for the following. Since there is no target "threshold" value for the objective, the objective agent activates at regular intervals and if the constraint violations are large this agent will skip steps 3 through 6.

Alternative Implementations

In order to check the sensitivity of the method to the heuristics used to compute trend, recommend and step-size, the following alternative implementations were tested.

AAM_2: Same as AAM_1, except that recommendations are discounted by their age

$Q' = \sum [r_m \, 0.95^{km - k}], \qquad \forall \, r_m \in R_{follow}$

AAM_3: Same as AAM_1, except that trends are computed from the directions used in the past discounted by age and the stepsize depends on the amount of perturbation made to the preferred direction.

$$T' = (x_k - x_{k-1}) / | (x_k - x_{k-1}) | + \sum 0.98^m [(x_{k-m} - x_{k-m-1}) / | (x_{k-m} - x_{k-m-1}) |], \text{ where } (1 \le m < h)$$

$$\alpha_k = (0.9 + 0.7\, r_i\, \Delta x_k)\, (x_k - x_{k-1}) / | (x_k - x_{k-1}) |$$

AAM_4: Same as AAM_3, except that recommendations are discounted by their age

$$Q' = \sum [r_m\, 0.95^{km - k}], \qquad \forall\, r_m \in R_{follow}$$

5.2.2. Results

An agent in the AAM works to improve an iterate with respect to a problem primitive that consists of either the objective or a constraint. These agents can be combined around memories of solutions. The experiments used a population of size one in order to measure the length of a single independent trajectory through decision space. Almost all monolithic methods follow a single trajectory and this provides us with a comparable measure of solution speed.

Several possible implementations of steps 3 to 6 of the algorithm which compute recommendations, trend and step size were tested. The differences in the implementation can be classified along two dimensions: the heuristics used to determine direction and stepsize; and the weighting factors used to combine the gradient, recommendations and trends. These weighting factors are experimentally determined parameters. Combinations of the values 0.35 and 0.7 were used to measure the sensitivity of the various implementations of the APM and the combinations of these parameters that found feasible solutions with objective values within 1% of the optimum for all problems are reported in the results.

The result values reported in Table 1 are averages over all ten standard benchmark problems. The mean number of iterations is the mean iteration count at which the best solution was found. Each iteration corresponds to an update of a single constraint (or of the objective function). The mean error is the deviation of the objective value of this solution from the reported optimum.Experimental results for several possible implementations of the algorithm produce similar results, and find good solutions across all problems over a range of the experimentally determined parameters (p_r and p_t), indicating that the method is robust over these parameters.

Table 4. Asynchronous Ad-Hoc Method: Computation required to find the Optimum (averages for 10 representative standard benchmark problems)

trial	Implementation and empirically determined parameters (p_r, p_t)	number of iterations to find best feasible solution	%difference of objective value from Optimum
1	AAM_1, 0.35, 0.7	7365.7	0.706568
2	AAM_1, 0.7, 0.35	7945.1	0.499582
3	AAM_1, 0.7, 0.7	8114.8	0.579164
4	AAM_2, 0.35, 0.7	7587.3	0.981708

trial	Implementation and empirically determined parameters (p_r, p_t)	number of iterations to find best feasible solution	%difference of objective value from Optimum
5	AAM_2, 0.7, 0.35	8100.9	0.436826
6	AAM_2, 0.7, 0.7	7801.8	0.346626
7	AAM_3, 0.35, 0.35	8034.5	0.574749
8	AAM_3, 0.35, 0.7	8318.7	0.535977
9	AAM_4, 0.35, 0.35	7828.6	0.72981
10	AAM_4, 0.35, 0.7	7867	0.185027

5.2.3. Discussion

The ad-hoc design of this modular method for NLP is very crude and based largely on heuristics. The direction determination is only loosely related to how directions are determined by the penalty method. The stepsize determination is also based on a crude heuristic compared to the traditional line search methods used to determine stepsize. The crudity of this adaptation is reflected in the performance. The number of iterations required is fairly large, requiring at least an order of magnitude more gradient evaluations than sequential quadratic programming. The algorithm also finds solutions of poorer quality. On the other hand, each iteration of the AAM requires minimal overhead and can be performed very fast. Such an adaptation is not recommended because, as we shall see next, much better modular methods can be designed by transforming good iterative monolithic methods.

5.3. Asynchronous Adaptation of Sequential Quadratic Programming (ASQP)

The transformation approach developed here is inspired by the work in [TPM83] on asynchronous distributed algorithms which describes how the solution of a set of equations X = G(X) can be determined asynchronously. Unfortunately, although [TPM83] presents convergence proofs for the fixed point problem, these proofs are not easy to apply to the asynchronous adaptation developed here.

Sequential Quadratic Programming (SQP) is one of the most popular algorithms for nonlinear programming. At each iteration it solves a quadratic programming (QP) approximation to the problem that has a quadratic objective and linear constraints. The solution of the QP provides a direction along which a ray search is conducted to generate the next iterate.

Let: $f_{Qk}(x)$, $g_{iQk}(x)$, $h_{jQk}(x)$ be quadratic approximations to the objective and constraints at x_k

$g_{iLk}(x)$ and $h_{jLk}(x)$ be linear approximations to the constraints at x_k

$M_{Qk}(x)$ be a quadratic approximation to $M(x)$ at x_k, specifically

$$M_{Qk}(x) = f_{Qk}(x) + w_i \sum_{g_{iQk}(x_k) > 0} p\left(g_{iQk}(x)\right) + w_j \sum_{j=1}^{m} p\left(h_{jQk}(x)\right)$$

Note: the quadratic and linear approximations are obtained as follows:

$$f_{Qk}(x) = f(x_k) + \nabla f(x_k) \cdot (x - x_k) + (x - x_k)^t \cdot B_k \cdot (x - x_k)$$

$$f_{Lk}(x) = f(x_k) + \nabla f(x_k) \cdot (x - x_k)$$

B_k is (an approximation to) the hessian of f(x) at x_k

The direction at iteration k is determined by solving a QP approximation to the problem at x_k

$\Delta x_k = x_{QP} - x_k$

where: $x_{QP} = \arg \min_{x \in R^n} M_{Qk}(x)$

s.t. $g_{iLk}(x) \le 0, i = 1 \ldots l$

$h_{jLk}(x) = 0, j = 1 \ldots m$

The stepsize is then determined: $\alpha_k = \arg \min_a M(x_k + a\Delta x_k)$ where $a \ge 0$.

and the new iterate generated: $x_{k+1} = x_k + \alpha_k \Delta x_k$

The asynchronous modular method which is developed in this chapter is based on transforming the SQP and computing the approximations to the objective and constraints asynchronously. The ASQP is similar to the SQP, excepting that the quadratic and linear approximations used for the objective and constraints may have been computed at previous iterations.

Let: $f_Q^*(x)$, $g_{iQ}^*(x)$, $h_{jQ}^*(x)$ be the latest available quadratic approximations to the objective and constraints

$g_{iL}^*(x)$ and $h_{jL}^*(x)$ be the latest available quadratic approximations to the constraints

$M_{Qk}^*(x)$ be a quadratic approximation to M(x), specifically

$$M_{Qk}^*(x) = f_Q^*(x) + w_i \sum_{g_{iQ}^*(x_k) > 0} p\left(g_{iQ}^*(x)\right) + w_j \sum_{j=1}^{m} p\left(h_{jQ}^*(x)\right)$$

The direction at iteration k is determined by solving a QP approximation to the problem at x_k

$$\Delta x_k = x_{QP} - x_k$$

where: $x_{QP} = \arg \min_{x \in R^n} M_{Qk}^*(x)$

s.t. $g_{iL}^*(x) \leq 0, i = 1 \ldots l$

$h_{jL}^*(x) = 0, j = 1 \ldots m$

The stepsize is then determined: $\alpha_k = \arg\min_a M_k^*(x_k + a\Delta x_k)$ where $a \geq 0$.

$M_k^*(x)$ lies somewhere between $M_{Qk}^*(x)$ and $M(x)$. For the objective and constraints, if possible, the exact value is used as in $M(x)$, otherwise the quadratic approximation is used as in $M_{Qk}^*(x)$.

and the new iterate generated: $x_{k+1} = x_k + \alpha_k \Delta x_k$

The term asynchronous implies that there is no coordination between the updates for the various constraints, and that there is no predetermined order in which the updates are performed. This asynchronicity adds a stochastic character to the behavior of the ASQP, which may follow different trajectories through the decision space depending on the order in which the constraints are updated. In contrast, the synchronous SQP always follows the same trajectory through the decision space from a given starting point.

5.3.1. Convergence

The ASQP can be shown to converge under a fairly limiting set of conditions, the key of which is that the starting point must be in a convex basin of the merit function and that the stepsize determination mechanism guarantees that only improving steps with respect to the merit function $M(x)$ will be taken. This is the case if $M(x)$ is used in the determination of the stepsize.

Another important condition is that the synchronous SQP can always find improving directions from every point in the basin in which the process is started. This means that the SQP never gets stuck at any point other than the optimum. The ASQP, on the other hand, may temporarily get stuck at a point in decision space where it fails to find an improving direction and there may be several identical consecutive values of the iterates in the asynchronous method: $x_k = x_{k-1} = x_{k-2}$ etc. However, if a constraint is only updated if the most recent point where it was updated is different from the current iterate, and if the algorithm gets stuck at a point in decision space, after a finite number of iterates, equal in number to $|C| + 1$, the direction determined by the asynchronous method will be identical to that determined synchronously and the ASQP will proceed.

This implies that if the process is started in a region of the space that contains the optimum and the synchronous method can reach the optimum from every point in that region, then the asynchronous method can also reach the optimum.

Let: x^* be the optimum

$X_k = \{x : M(x) \leq k\}$

Theorem:

If: $x_0 \in X_k$ (1)

$x^* \in X_k$ (2)

X_k is connected and $\forall\ X_j$, $j < k$, $X_j \cap X_k$ is connected (3)

$\forall$ very small δx: $x \in X_k$ and $M(x+\delta x) < M(x) \Rightarrow x+\delta x \in X_k$ (4)

$\forall$ x in X_k s.t. $x \neq x^*$ and Δx is generated at x by the synchronous algorithm:

$\alpha > 0$ and $M(x) > M(x + \alpha\, \Delta x)$ (5)

$\forall$ x in X_k and any direction δx:

$\alpha \geq 0$ and $M(x + \alpha\, \delta x) \leq M(x)$ (6)

the objective or a given constraint is updated in the asynchronous method only if the point at which it was most recently updated is not identical to the current iterate. (7)

Then: The sequence of iterates generated by the asynchronous iterations method will converge to x^*

Proof:

Because of conditions (1) through (6) the algorithm will find a sequence of points $x_1, x_2, \ldots x_\infty$ s.t.

$k \geq M(x_0) \geq M(x_1) \geq M(x_2) \geq \ldots \geq M(x_\infty)$

corresponding to the sets $X_{k0} \supseteq X_{k1} \supseteq X_{k2} \ldots$ where $k_i = M(x_i)$

The sequence may contains subsequences of identical sets. However, because of condition (7) the asynchronous iterations will get stuck at a non-optimal point for at most $(|C| + 1)$ iterations. If we eliminate all duplicates in the sequence, we are left with a sequence

$X_{k0} \supset X_{ki} \supset X_{kj} \supset \ldots \supseteq X_{k\infty}$

This sequence consists of sets that are strict subsets of their predecessors, except if the set is x^*, where the algorithm must converge.

This proof only applies if the step size determination method prevents increases in the merit function which implies that the step size determination must be done synchronously. The empirical results presented next show that asynchronous methods converge even if this constraint is relaxed.

The steps followed by an module specializing in a constraint:

Step 1: Check if the constraint is violated. If not, do nothing.

Step 2: Compute the gradient for the constraint at the current point. Update the approximation to the Hessian using the BFGS formula

Step 3: Solve a QP to determine a direction.

Step 4: Determine the step size using the latest available approximations. Since only approximations are used, there are lower bounds (to prevent getting stuck) and upper bounds (to avoid moving too far away) on the step size. These bounds may be modified adaptively.

Figure 2. The Asynchronous Sequential Quadratic Programming Method

5.3.2. Implementation

Solution Representation

A solution consists of: $\{x_k, A, \alpha_{min}, \delta x_{max}\}$

where

x_k : the value of the current iterate

$A = \{f_Q(x), g_{Q1}(x), \ldots, g_{Qn}(x)\}$: the latest available approximations to the objective and constraints

$g_{Qi}(x) = g_i(x_+) + \nabla g_i(x_+)\,\Delta x + \Delta x\, H_i(x_+)\,\Delta x$, where $\Delta x = (x - x_+)$ and x_+ is the iterate of the latest update for constraint i.

α_{min} : lower bound on the step size (as a multiple of the QP step) $\alpha_{min} \in [0..1]$

δx_{max} : upper bound on the step size

Implementation of Method

The steps followed by the module specializing in constraint i are:

Step 1: Select a solution from the memory for which the constraint i is violated

Step 2: Update the approximation $g_{Qi}(x)$ by computing the evaluation of the constraint $g_i(x_k)$ and its gradient $\nabla g_j(x_k)$ and using the BFGS formula to update the approximation to the hessian $H_i(x_k)$

Step 3: Set up a QP using A, and solve to determine Δx^*

Step 4: Perform a search on the segment defined by x_k and $x_k+\Delta x^*$ to find the new iterate x_{k+1} such that the stepsize is less than δx_{max}.

Step 5: Update the bound δx_{max}. If the current step is at the bound limit, increase it slightly, otherwise decrease it.

keeping α_{min} fixed at 0.05 works well.

5.3.3. Results

The ASQP was tested on the same set of standard benchmark NLP problems used for the AAM and its performance was compared to the SQP. The quality of solutions found by the ASQP is significantly better than those found by the AAM; since it is adapted from the SQP, it finds as good solutions as the SQP does.

The effort required to find the optimum by NLP algorithms is usually measured by the number of function and gradient evaluations required to find the optimum. A starting point is provided with each benchmark problem in [HS81]. The following experiments were each run with the provided starting points. The SQP is a deterministic algorithm and follows the same trajectory through decision space for every run. The ASQP on the other hand is stochastic in nature, and the trajectory it follows through decision space depends on the sequence of agents that operate. Ten trials were run for each problem for the ASQP. Each trial has a different sequence of agents and hence follows a different trajectory through decision space from the rest. The best, average (median) and worst case performance is shown in the table below. The ASQP on average takes slightly fewer gradient computations as the SQP and considerably fewer function evaluations than the SQP.

Table 5. Asynchronous SQP vs. SQP: Computation required to reach the Optimum

prob. #	SQP			Asynchronous Algorithm (10 trials/problem)								
				#QP (solutions)			#eval(uations)			#grad(ients)		
	#QP	#eval	#grad	best	avg.	worst	best	avg.	worst	best	avg.	worst
1	4	120	13	3	3	18	12	12	28	10	10	13
2	62	4368	251	162	228	297	170	274	331	113	143	186
3	111	7495	555	374	454	567	378	458	571	378	458	571
4	154	15785	1078	116	209	235	216	394	449	216	394	449
5	148	18414	891	345	460	724	363	473	754	240	300	482
6	12	336	72	93	131	157	98	136	162	98	136	162
7	6	234	78	78	150	187	159	300	373	159	300	373
8	455	273780	16383	459	798	1049	533	889	1171	477	763	1023
9	6	315	80	109	117	133	123	131	147	103	117	131
10	54	2926	378	112	138	216	209	266	412	209	266	412

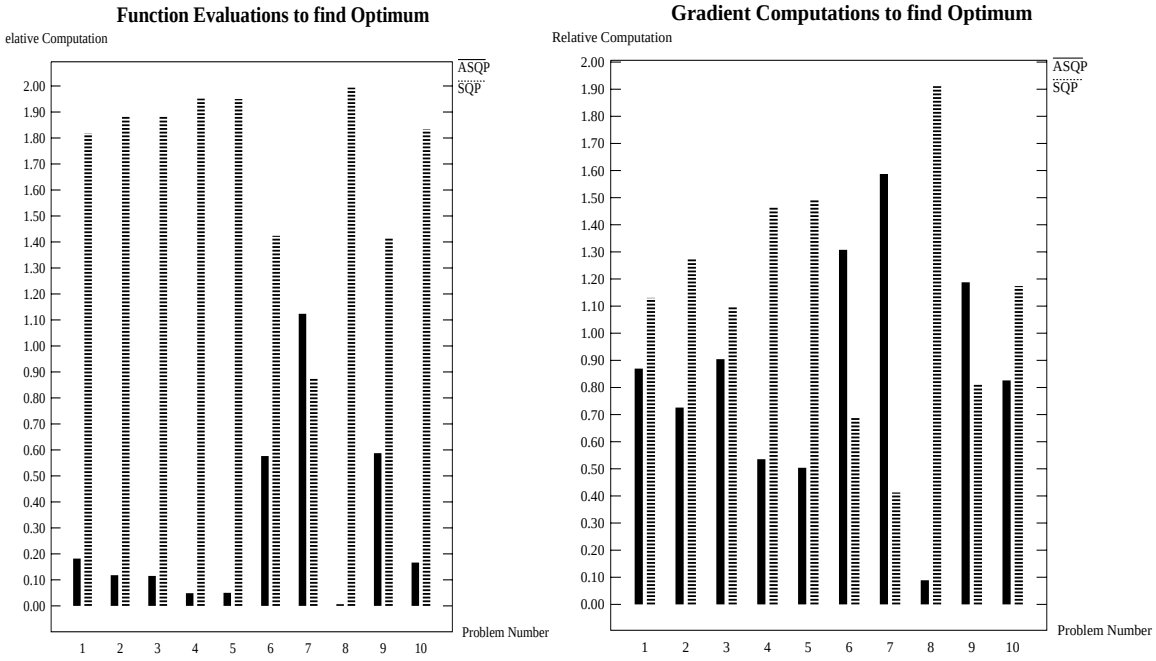


Figure 3. Comparative computation required to find optimum by SQP and Asynchronous SQP.

A summary of the performance of the ASQP compared to the SQP, averaged over the entire set of problems is shown below. Since the ASQP uses approximations and outdated information, it takes 2.5 to 5 times as many QP solutions as the SQP. On the other hand, the use of approximations results in far fewer function evaluations and fewer gradient computations than the SQP.

Table 6. Asynchronous SQP vs. SQP: Comparison of effort to find the Optimum

	(ASQP effort)/(SQP effort)
iterations	3.39276
function evaluation	0.0619182
gradient evaluations	0.669476

5.3.4. Discussion

The ASQP performs better on average than the SQP measured by the number of function and gradient evaluations. However, it takes many more QP solutions. For problems with simple objective and constraint functions, which are easy to evaluate and differentiate, it is therefore better to use the SQP. For problems with complicated objective and constraint functions the ASQP has the advantage.

The ASQP has other benefits because of its modular design. The agents in the experiments were combined in an A-Team with a memory size of 1. A-Teams allow multiple solutions to be considered and several agents to work simultaneously. Multiple copies of agents can be easily added to the team to increase speed. In addition, if multiple copies of solutions are available, agents have the option of using the nearest approximations in solution space for other constraints in place of the most recent, which can help improve solution speed.

5.4. The Spatial Layout Problem

There are many problems that cannot be easily formulated mathematically and for which good solution algorithms like SQP do not exist. Such problems are common in practical design and engineering domains. Requirements for such problems are subject to modification, and constraints and objectives change fairly often. The modular approach provides a means of tackling such problems. I shall now focus on one such problem, spatial layout of 3D components, and show how a modular approach can work in this case.

I will consider problems in Mechanical Engineering domains such as design, prototyping and manufacturing which typically require packing rigid components with complicated shapes, subject to a variety of constraints. The objects are usually required to be fit into a given housing with no mutual overlap. There may also be other objectives to be satisfied, such as connectivity costs between objects, the location of the center of gravity of the packed assembly, the accessibility of certain objects etc. Examples of applications include the design of satellites, the design of cars, packing for material handling, and 3D nesting for rapid prototyping which is a rapidly growing emerging technology [Hinzman95, Prototype1, Prototype2]. Most spatial layout problems have many different conflicting constraint and objective functions which make them very challenging, and the software currently used for solving such problems is difficult and expensive to develop.

Spatial layout problems in 3D are usually solved using ad-hoc heuristics, genetic algorithms (GA) or simulated annealing (SA). Ad-hoc heuristics take advantage of structure in the problem and are fast but fragile.

GA and SA are more robust to differences in problem structure but much slower. The A-Teams formalism enables all these different methods to be combined to take advantage of their complementary strengths.

A-Teams have several other advantages that are desirable for solving 3D spatial layout problems [SPGT98]. Agents in A-Teams can also work in parallel across a network of computers, allowing the exploration of more solutions and saving time. A-Teams do not require a single merit function, and sets of non-dominated solutions (i.e. a Pareto frontier) can be found. Thus the user does not have to provide a set of weights a-priori and can choose the best trade-offs between the different objectives after solutions have been found.

Specialized heuristics and algorithms that handle problem primitives are relatively easy to construct compared to those that must handle the entire problem. These specialized heuristics, encapsulated as agents, can be reorganized for different tasks. Whenever a new type of problem needs to be solved, the appropriate set of specialist agents can be assembled from a repository and solvers can be customized by composition of an appropriate mix of agents.

5.4.1. Problem Definition

A small set of constraints was selected to develop and demonstrate the modular A-Team spatial layout tool. The constraints were selected to represent several different types of constraints found in practice, discontinuous, continuous-differentiable, and procedural. The agents for each constraint can use the best methods for dealing with the constraint, for example gradient-based methods can be used for differentiable constraints.

A typical electromechanical assembly design problem is used as an instance of the class of problems under consideration. The task consists of packing a set of components into a housing unit subject to a set of constraints. There are accessibility constraints (for components such as knobs, connectors, heat sinks etc.); connectivity requirements (for components that may have many connections between them); and separation constraints (for components that may interfere electromagnetically or thermally if placed too close together). The problem is stated as follows:

Given: a solid model H for the housing packable volume or interior of the housing

$\{C_i\}$: a set of solid models for the components

$\{f\}$: a set of planar faces on components in $\{C_i\}$ that need to be accessible

$\{W_{ij}\}$: a set of costs for connecting components C_i and C_j

$\{s_{ij}\}$: a set of minimum required separation distances between components C_i and C_j

Let: Distance(C_i, C_j) be the distance between the two components

SweepFace(f) be the solid body generated by sweeping face f, a face on one of the components, to infinity in the direction of its normal

Volume(C) be the volume of a solid body C

Find: a placement of the objects in order to optimize

Connectivity objective: minimize $\sum_{i,j} W_{ij} * \text{Distance}(C_i, C_j)$

Accessibility objective: minimize $\sum_{i,k} (\text{Volume}(\text{SweepFace}(f) \cap H))$

s.t.: Protrusion constraint: $C_i \cap H = C_i, \forall\ i$

Overlap constraint: $C_i \cap C_j = \varnothing, \forall\ i, j; i \neq j$

Separation constraint: $\text{Distance}(C_i, C_j) \geq s_{ij}$

Accessibility constraint: $\text{SweepFace}(f) \cap C_i = \varnothing, \forall\ f, i$

The connectivity objective measures the total cost of establishing physical connections between components. Separation constraints keep components apart that might interfere with each other electromagnetically or thermally. Certain faces on some of the components are required to be accessible (i.e. not blocked by any other component, for instance the battery compartment of a portable radio needs to be accessible). The accessibility constraint for a face is defined as the ability to sweep the face along the direction of its normal without the swept volume intersecting with any other object. The accessibility objective is used to demonstrate the reconfigurability of the A-Team based tool. The alternate problem definition measures accessibility not only by the ability to sweep the required face freely, but also by the distance of the face from the boundary of the housing packable volume measured by the intersection between the swept face volume and the housing packable volume.

The connectivity objective is designed to represent differentiable and continuous objectives. The separation constraint represents discontinuous constraints with many regions of zero violation. The accessibility constraint represents procedurally defined constraints. Both separation and accessibility constraints have highly discontinuous derivatives, which makes them difficult to solve using gradient based methods.

The artifact used for demonstration is the optics sub-assembly of a missile seeker. There are ten components in the sub-assembly: a ccd camera unit, a microprocessor, a digital signal processor, two analog-digital converters, a voltage stabilizer, serial and parallel connectors and two amplifiers. There are connectivity requirements between the microprocessor, digital signal processor, camera and connectors. The amplifiers must be kept suitably far from the camera and microprocessor. Additionally, the connectors must be accessible from outside the housing, and there must be no component blocking the front of the camera lens. The housing is non convex, and contains holes. The components cover a range of shapes and sizes.

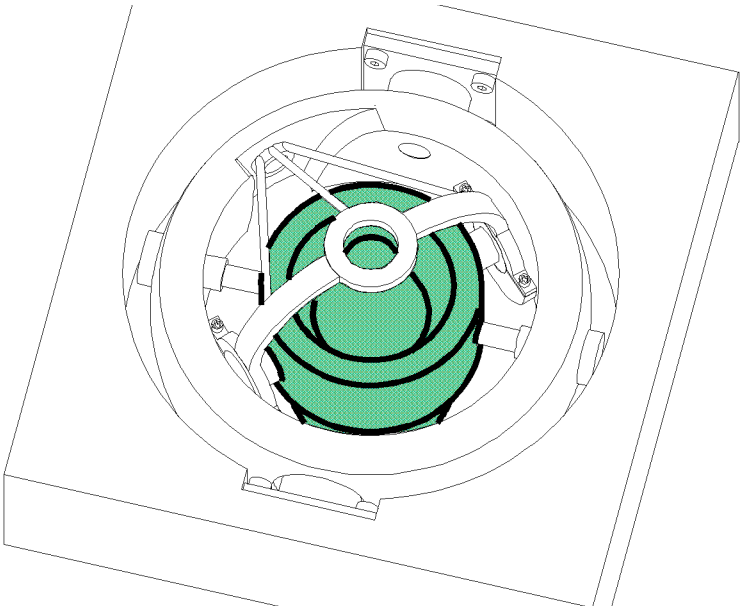


Figure 4. The Seeker Assembly for a Missile (The optics housing is colored)

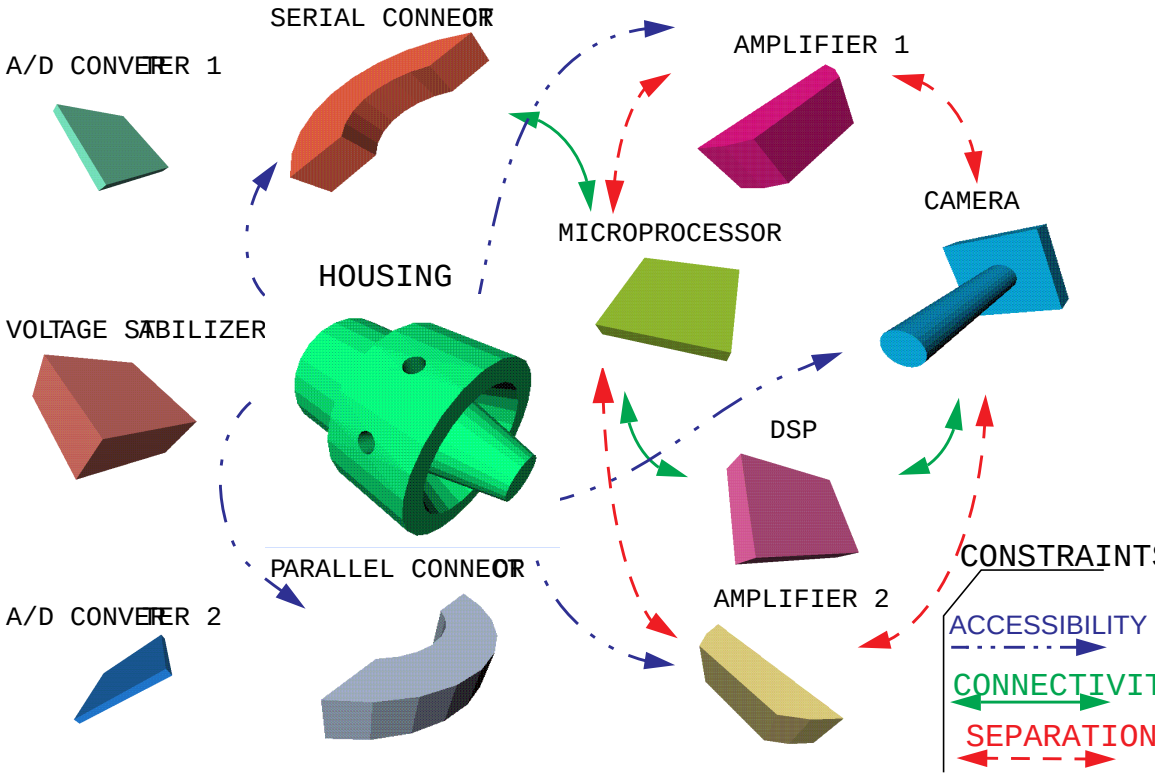


Figure 5. Problem Description: Optics Housing and Components. The arrows indicate the presence of accessibility, connectivity and separation constraints between objects.

The objects are represented by 3D CAD models in ACIS format. The ACIS 3.0 class library and API are used for manipulation and processing of the 3D models. ACIS 3.0 is used to evaluate the interference, protrusion, and accessibility requirements. OpenInventor is used for display and rendering of layouts.

5.5. Modular Ad-Hoc A-Team Tool for Spatial Layout

The approach used to coordinate the actions of agents is to use subsets of overlapping constraints for agents and for each agent to take a small step in each iteration.

5.5.1. Implementation

An ad-hoc approach was used for developing the tool, and a set of agents specializing in small subsets of overlapping objectives and constraints was developed. Since the protrusion and overlap constraints are common to most layout problems, all the agents were designed to avoid increasing the violation of these constraints.

Solution Representation

A solution is represented as a list $\{X_i\}$: where X_i is the translation of object i

Two secondary representations are also used by some of the agents:

$\{L_i\}$: where L_i is the location of object i on a predefined grid in the bounding box of the housing

$\{P_i\}$: a permutation of the integers from 1 to $|\{Ci\}|$ corresponding to the order in which the objects will be placed in the housing by a construction heuristic

Agent Implementations

The construction agents are of three types: (1) Creators or seeders, which generate new solutions to populate the memories; (2) Modifiers which create new solutions by modifying existing solutions in the memories; and (3) Evaluators which evaluate the solutions with respect to the objectives and constraints. There is an independent evaluator for each constraint and objective. Evaluators start work whenever they detect new unevaluated solutions in the memory. Since they are independent, they can simultaneously update evaluations for new solutions or even work on different solutions.

There is no algorithm or heuristic that simultaneously attempts to optimize the entire problem. The A-Team is composed of agents that each specialize in a different set of objectives or constraints. Since interference and protrusion requirements are almost universal in layout problems, most of the specialized agents take these constraints (referred collectively as the packing requirement) into account. The packing specialists focus only on improving packing, ignoring all other constraints. The other specialized agents consider one of the accessibility, connectivity or separation requirements as well as the packing requirement. For exam-

ple, the accessibility agents attempt to improve the accessibility measure, ignoring the connectivity and separation requirements. The connectivity and separation agents behave in a similar manner.

The specialized agents can use a variety of heuristics, restricted only by the availability of good methods. As an example, one heuristic for connectivity in the A-Team uses the gradient of the connectivity cost to improve solutions by sliding objects around their current locations. Since the A-Team approach allows multiple specialists for a single constraint, another agent to improve connectivity that tries to relocate an object in a different part of the housing can also cooperate.

The creators can be classified into three groups: one that uses an ordering of the objects combined with a decoder (similar to packing heuristics found in the Operations Research literature and in Genetic Algorithms), one that uses random placement of the objects (similar to that used in Simulated Annealing or Tabu Search) and one that uses a discretized grid to locate objects. A description of one of the creators provides an example of how secondary representations can be used to assist in optimization: An A-Team that uses a discretized grid representation ignores protrusion and interference, trying to place accessible objects near the grid boundary. Separation and connectivity are approximated by integral manhattan distances on the grid. The only goal of this subsidiary A-Team is to generate good starting configurations.

A destroyer erases solutions that are dominated by (and hence are worse than) others in the memory. If the memory is full of non-dominated solutions, an arbitrarily selected solution may be destroyed to make space.

A separate agent was built for each constraint and objective. Each agent used one or more heuristics to improve solutions with respect to its goals.

Discretized Layout Heuristic:

Step 1: Partition the components into two groups depending on whether they have accessibility requirements. The objects requiring accessibility will be placed first. Generate a random pemutation of the objects in each group.

Step 2: Generate a discrete grid in the bounding box of the housing

Step 3: Use the permutation to place the objects one at a time in the housing at the location on the grid where the new object has the smallest overlap with the already placed objects.

Packing Constraints:

Step 1: Compute the packing violation (overlap + protrusion).

Step 2: For a small number (100) of trials: Generate a small perturbation to the solution. If the packing violation increases, undo the perturbation, otherwise replace the old solution with the new one.

Accessibility Constraint:

Step 1: Identify objects requiring access that are blocked. Remove all blocked and blocking objects. The blocked objects will be replaced first, followed by the blocking objects. Generate a random permutation of the objects in each group to be replaced.

Step 2: Generate a discrete grid in the bounding box of the housing

Step 3: Place the objects one at a time in the housing at the location on the grid where the new object has the smallest overlap with already placed objects.

Separation Constraint:

Step 1: Identify a pair of objects that violates the constraint

Step 2: Move the objects slightly away from each other in a direction determined by the line between their centroids.

Connectivity Objective:

Step 1: Compute the gradient of the connectivity cost

Step 2: Move the solution a small amount in the direction opposite to the gradient to lower connectivity cost.

Accessibility Objective:

Step 1: Select an object randomly that needs to be accessible.

Step 2: Determine the normal to a face that needs to be accessible on this object.

Step 3: Move the object a small distance along the normal computed in Step 2.

5.5.2. Results

The tool was first used to find a placement of the objects to minimize only the connectivity objective subject to all the constraints. The organization of the A-Team is provided below.

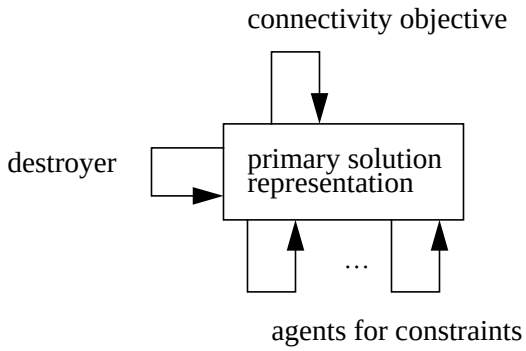


Figure 6. A-Team organization for connectivity objective only

The ability to customize the solver was demonstrated by adding the accessibility objective. It was considered insufficient for the appropriate face of the connectors to just be reachable from outside, and accessibility was redefined to give preference to placements where the connectors and the heat sinks of the amplifiers were closer to the housing boundary. The reorganization of the team required the addition of new agents for the accessibility objective.

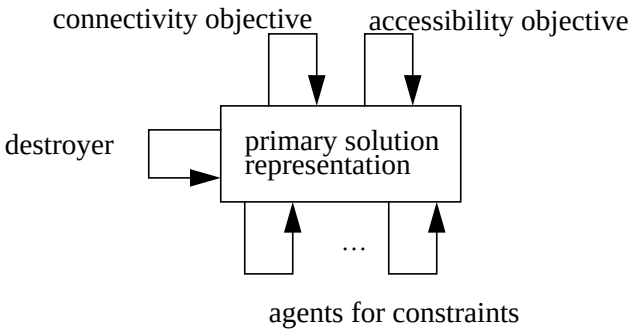


Figure 7. A-Team organization after the addition of the accessibility objective

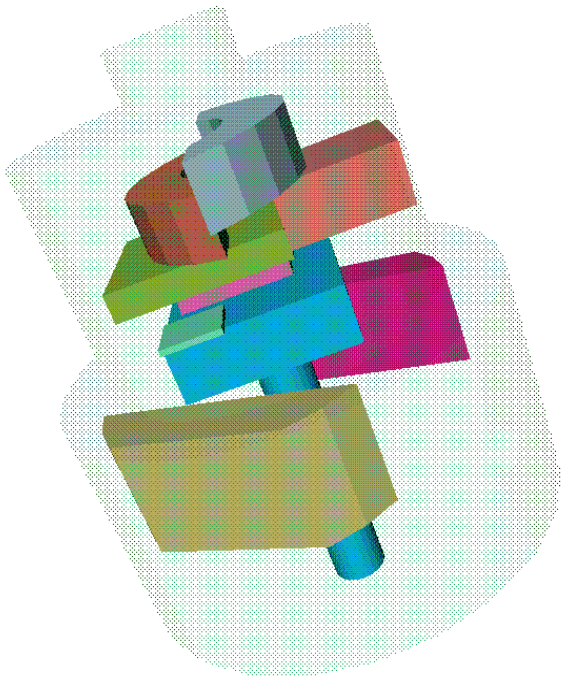


Figure 8. Layout for the Optics Assembly with Connectivity objective only

Since the A-Team is composed of independent specialized agents, it is easy to reorganize the team to match new task requirements. In the case of the new accessibility objective, modifying the A-Team required only the addition of agents for accessibility. The remaining agents were unchanged and, in fact could continue working during the team reorganization. A solution found by the modified team is shown for comparison. Some of the components have been moved closer to the housing.

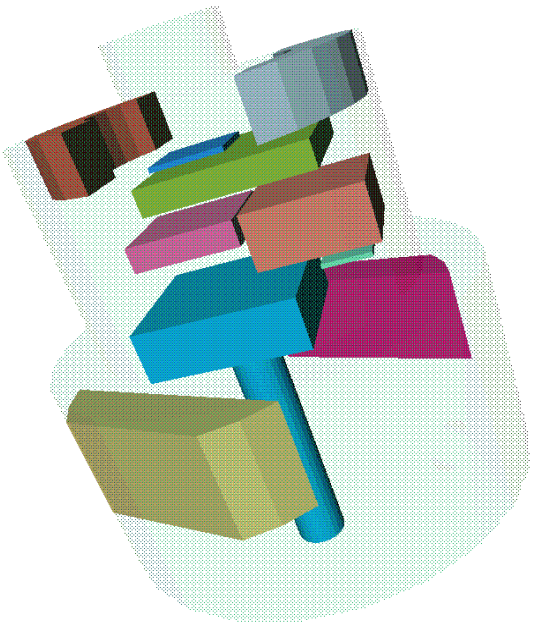


Figure 9. Layout for the Optics Assembly with Connectivity and Accessibility objectives.

Benchmarking the Performance of the Modular A-Teams Approach

In order to obtain a quantitative measure of A-Team performance, a benchmark algorithm was selected as a comparison. Initial choices included heuristics, Genetic Algorithms and Simulated Annealing. Heuristics were ruled out since there are no good heuristics tailored for the problem under consideration. Heuristics tend to be limited in scope, and fragile to changes in problem specifications. Genetic Algorithms were eliminated since they are traditionally built around a heuristic acting as the decoder. This left Simulated Annealing, a powerful general purpose algorithm which works well and has been proposed as a good candidate for spatial layout problems. The Simulated Annealing algorithm was also selected because it is conceptually simple and easy to implement.

In order to develop a good simulating annealing implementation, the algorithm was tuned prior to gathering the benchmark data. Several trials were run to tune the choice of a merit function. The merit function used was:

M(layout) = 1.0e6*accessibility violation + 1.0e3*interference violation + 1.0e3*protrusion violation + 100*separation violation + connectivity objective + accessibility objective

The move set consisted of a range of perturbations (perturb single object/multiple objects; make small or medium perturbations relative to the housing size) and object swaps.

The Simulated Annealing Algorithm used was:

Step 0: Set initial temperature manually so that about 95% of all trial moves are accepted. Set Failures = 0.

Step 1: Try perturbations at the given temperature. Go to next step when 15 perturbations have been accepted or 3000 perturbations have been tried.

Step 2: If the merit function improved in Step 1 set Failures = 0, else increment Failures.]

Step 3: If Failures = 3 then stop, else lower the temperature: $t_{new} = 0.9 * t_{old}$ and go to Step 1.

Five trials were performed. Each took between 22 and 25 hours before freezing on a dedicated Sparc Ultra machine. Of these, 2 froze with infeasible packings. Three were able to find good solutions.

Single Processor Comparison

The entire A-Team was first executed on a single processor to establish a lower bound on A-Team performance. Five trials were performed, each took approximately 12 hours to find solutions on a dedicated Sun Sparc 20, which was an older and slightly slower computer than the Ultra used for the Simulated Annealing. The pareto surface of feasible solutions was compared with the results from the Simulated Annealing. All five trials produced feasible solutions, unlike the SA which failed in two cases. The Pareto surface of feasible solutions contained 2 to 8 solutions.

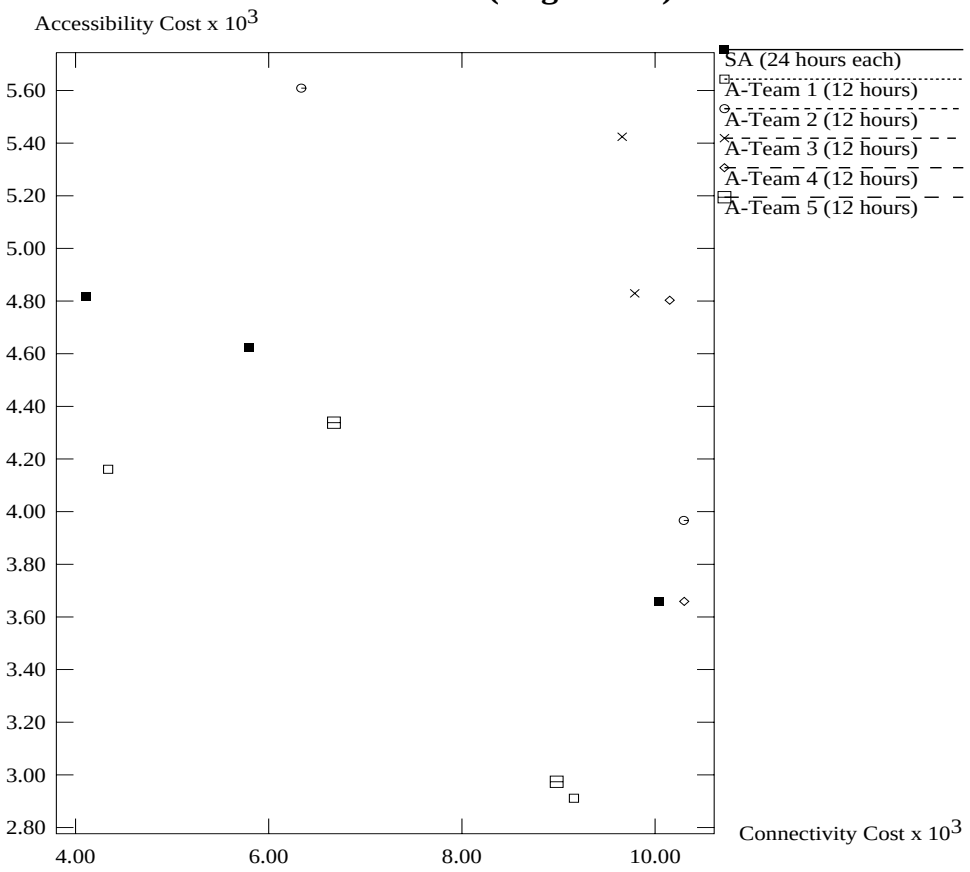


Figure 10. Comparison of Pareto Surfaces for experiments on single processor. Results for Simulated Annealing are for the four trials where a solutions was found. Time for SA is 24 hours. Results for A-Team trials are shown individually, with the two extreme points on the Pareto surface. Time for A-Teams is 12 hours.

This experiment was constructed to determine how competitive an A-Team consisting of a set of specialized agents of heuristics is compared to a popular general purpose method. The Simulated Annealing algorithm was tuned for good performance by empirically selecting the move set, merit function and starting temperature. The same code was used for evaluating solutions for both the Simulated Annealing and the A-Team. Additionally, the Simulated Annealing was run with two to three times the cpu power (twice the amount of time on a much faster computer).

Table 7. A-Team vs. Simulated Annealing: Comparison for a single computer

	A-Team	Simulated Annealing
Failures (out of 5 trials)	0	2
clock time (dedicated computer)	12 hours	24 hours
computer	Sparc 20, 75Mhz, 64M RAM	Sparc Ultra, 143Mhz, 128M RAM
# of solutions found per trial	2 to 8	0 or 1

Network of Processors Comparison

In order to demonstrate parallelization of A-Teams, the A-Team was executed on a distributed network of 5 computers. The computers included two 2-processor Sparc 20s, two moderate speed Sparc Ultras and a fast 2-processor Sparc Ultra for a total of 8 processors. Five trials of 2 hours each were performed, each of which found 2-8 feasible solutions. In comparison, if each of the five SA trials had been run simultaneously on fast Sparc Ultras, the SA benchmark would have found 3 non-dominated solutions in 24 hours. The quality of the A-Team results is comparable to those of the SA as can be seen in the plot of the Pareto frontiers. In addition, a distributed A-Team that was run for 12 hours found a solution that clearly dominated all those found by the SA.

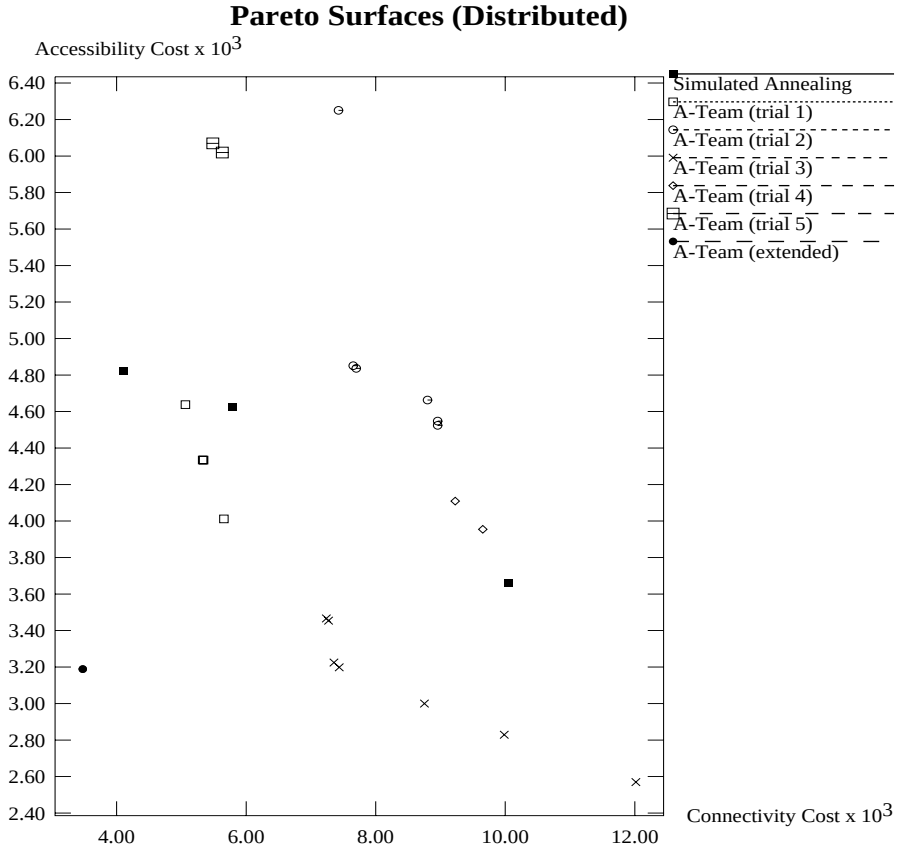


Figure 11. Comparison of Pareto Surfaces for experiments on multiple processors. The SA results are for 5 runs of 24 hours each. The Pareto Surfaces found for 5 A-Team trials are shown, each trial running for 2 hours on a network of 5 computers. The extended A-Team trial is for 12 hours.

Table 8. A-Team vs. Simulated Annealing: Comparison for a computer network

	A-Team	Simulated Annealing
clock time	2 hours	24 hours
computers	2 Sparc 20s, 3 sparc Ultras	5 Sparc Ultras
Pareto Frontier Size	1 to 8	3

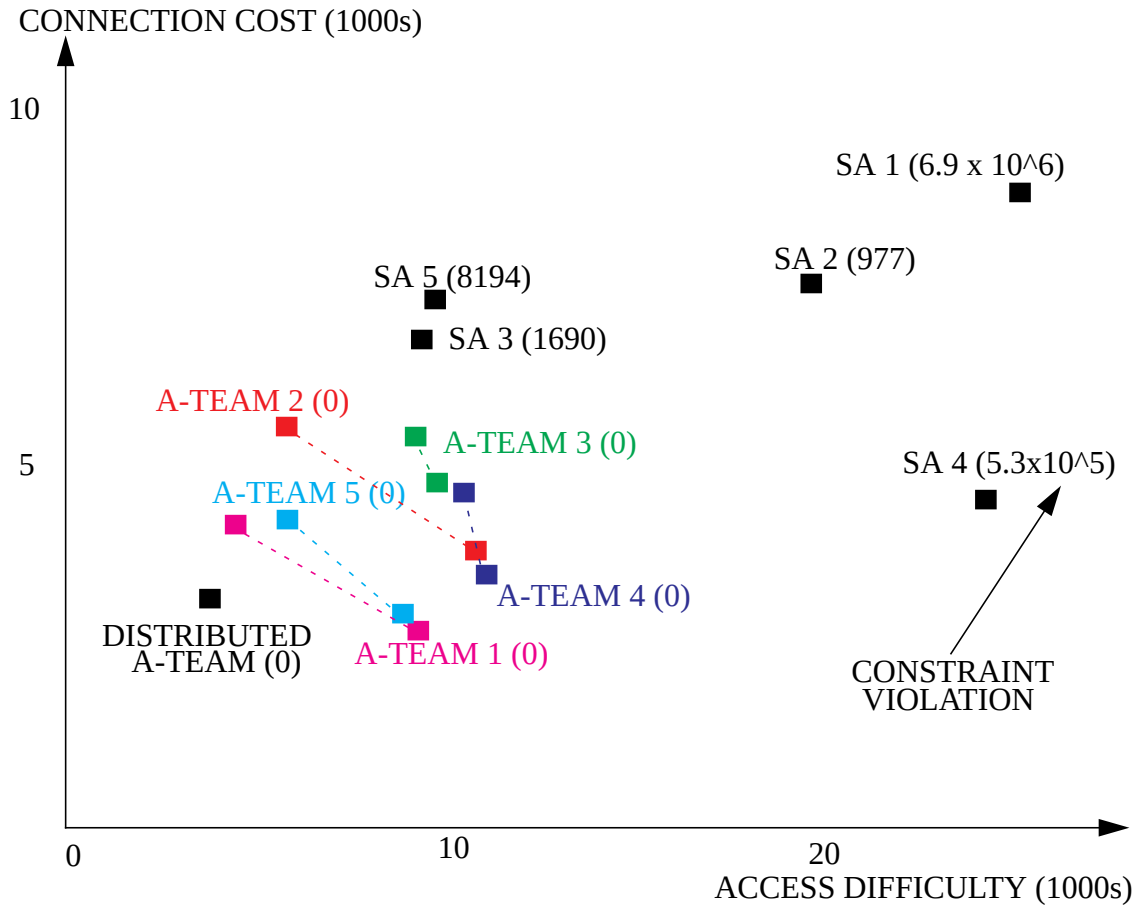


Figure 12. Comparison of solutions found in 12 hours by 5 A-Teams and 5 Simulated Annealing trials on a single computer. The results for a distributed A-Team running on a network of 5 computers is also shown.

5.5.3. Discussion

The modular method can solve problems faster than simulated annealing, which is one of the more popular approaches to solving layout problems. This is partly because the modular method uses more information about the properties of constraints to improve solutions. The modular method also usually finds several alternative layouts whereas the simulated annealing finds just one.

5.6. Summary

Asynchronous algorithms are well suited for execution on networks of heterogeneous, loosely coupled processors. Standard optimization algorithms can be transformed to operate asynchronously by using approximations. It is possible to develop crude, heuristic based adaptations of optimization methods that can find near optimal solutions. More sophisticated adaptations using better approximations perform significantly better. They are able to find optima to the numerical resolution of the base algorithms. In addition, the use of approximations reduces the number of function and gradient evaluations required.

It is not difficult to construct asynchronous algorithms using the A-Teams formalism. The asynchronous algorithm is constructed of a set of modules, each of which focuses on a separate constraint and is encapsulated as a specialized agents. These agents are reusable for families of problems that share constraints by choosing the appropriate set of agents to organize into an A-Team. These A-Teams are also open to the addition of other modification heuristics such as the crossover operator used in GAs. This helps make the A-Teams more robust. Another factor that contributes to the robustness of the A-Team is the use of multiple solutions and approximations. This allows the team to opportunistically explore different regions of the space and reduce the likelihood of getting stuck.

A tool to solve spatial layout problems for packing arbitrary 3D objects subject to constraints was developed using the A-Team formalism. The tool includes an expandable repository of specialized agents, each specializing in a subset of the constraints. The tool demonstrates that task-specific solvers can be constructed for optimization problems by composing teams from an appropriate set of specialized agents. The A-Team tool can be reconfigured for different tasks by changing the mix of agents in the team.

The A-Team was benchmarked using a Simulated Annealing algorithm. The A-Team takes considerably less time to find solutions of comparable quality even when run on a less powerful processor. A-Teams are by nature parallelizable. The A-Team for spatial layout was distributed over a network of processors to achieve even greater speedups, taking less than a tenth of the time of the SA.

The spatial layout problem exemplifies engineering optimization problems where it is difficult, if not possible to develop good monolithic algorithms or heuristics. Traditional optimization methods are also often not applicable. For instance, this problem cannot practically be formulated as a NLP problem. General purpose heuristics are the only recourse. Ad-hoc heuristics, which are often very efficient, can only be developed for problem primitives taking into account only some of the constraints. A-Teams allows such ad-hoc heuristics to be encapsulated as specialized agents and facilitates their combination into a task-specific, reconfigurable solver. As the requirements change for the problem, the solver can be updated by adding the appropriate specialists. Specialized agents can be stored in a repository and incorporated in teams as needed.

6. HUMAN-COMPUTER COLLABORATION

Computers and humans have complementary skills: computers are extremely good at some skills compared to human beings, and extremely poor at others. There are many problems that require both kinds of skills. However, computers and human beings operate at very different speeds. Can they collaborate effectively if organized in A-Teams?

This chapter will address three issues related to human-computer collaboration: diversity of agent skills is more important than the speed of agents; asynchronous collaboration can be more effective than synchronous collaboration; and humans and computers work synergistically when combined in A-Teams.

6.1. Benefits of Human-Computer Collaboration

This section explores when human experts and computer tools can complement each others skills and identifies characteristics of suitable problems for human-computer collaboration. A considerable amount of effective human-computer cooperation is based on combining the speed of computers at handling numeric constraints and their ability to explore many alternatives with the pattern recognition skills of the humans. Layout problems are used here as representative examples of problems that require both kinds of skills. Layout problems require finding the optimal location and orientation of a set of objects subject to constraints and occur in several areas: layout of modules on integrated circuits or layout of components on printed circuit boards in electrical engineering; layout of pieces to be cut out of sheets of fabric or metal (called nesting problems) [DM95] in industrial and mechanical engineering; and layout of rooms in buildings in architecture [Harada97].

The computer usually has the advantage when there are large numbers of constraints and objectives or many possible solutions. The human mind can actively consider only a very small number of alternatives because it can only simultaneously track a small number of distinct pieces of information at a time [Miller56]. As a result, humans are prone to get stuck exploring only around one, or very few, regions of the decision space. The computer, on the other hand, can rapidly explore many alternatives and computers dominate in layout problems in electrical engineering where there are too many constraints for humans to keep track of. Even in such cases, it should be possible for humans to contribute by identifying local improvements. For example there appear to be obvious improvements to the solutions found by the simulated annealing algorithm in [OF93].

Although computers are far superior to humans at most numerically oriented tasks it is not always possible to tailor the problem formulation to match the available algorithms: because the formulation would be a poor reflection of the actual problem; because it would be too difficult to create; or because the formulation would too complex to solve easily using the algorithm. Ad-hoc and general purpose heuristics must be used in such cases. In the context of layout problems, standard numerical algorithms such as sequential quadratic

programming are impractical: The problem formulation is extremely complicated, and the resulting problem very large. An attempt at such an approach can be found in [KG91] which makes the problems with this approach obvious. Where the objects are fairly simple, layout can be done by computer using either ad-hoc heuristics or simulated annealing. Simulated annealing is more popular than heuristics because it can usually find better solutions but it may still require certain algorithm parameters to be tuned by a human for each problem [PTVF92]. Humans can contribute even when the problems are more suited to computer solutions.

Computers are weak at problems with objects that have very complex shapes because of the computational complexity of recognizing shapes and identifying likely matings of different components. For example the algorithm for packing 2D non-convex polygons inside non-convex boundaries presented in [DM97] is considered practical only for up to four polygons by the authors since its complexity increases exponentially. Other examples are the problems in [SC95] which are very difficult to solve using a computer, but which could be easily solved by a human. Computers abilities are improving rapidly, but there are still many problems where humans could be very useful as part of a problem solving team.

Another issue that is left for future work is the collaboration of humans with computers for problems where some of the constraints cannot be formulated mathematically, for instance aesthetic considerations in architecture. I believe that humans-computer teams are likely to be extremely effective for such problems but considerable thought will be required in designing the teams.

6.2. The Experimental Setup

A set of layout problems was created to study the issues in human-computer collaboration. The test problems are representative of practical problems found in various application domains such as VLSI or architectural layout and nesting.

A set of objects was designed. The objects have axis-aligned edges to help speed up intersection detection and are rotationally asymmetric and non-convex in order to make the problems challenging and avoid symmetries in the solution space. A few example shapes are shown below

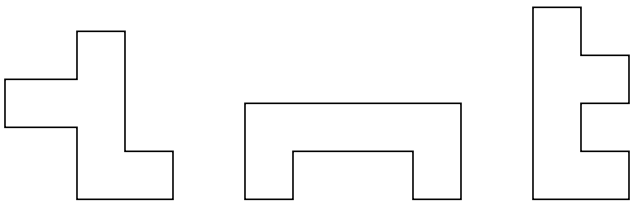


Figure 13. Example shapes of objects for layout

In each problem instance, a set of 2D objects is to be packed in as small an area as possible, while trying to minimize connectivity costs between pairs of objects and satisfying separation requirements along the coordinate axes between pairs of objects.

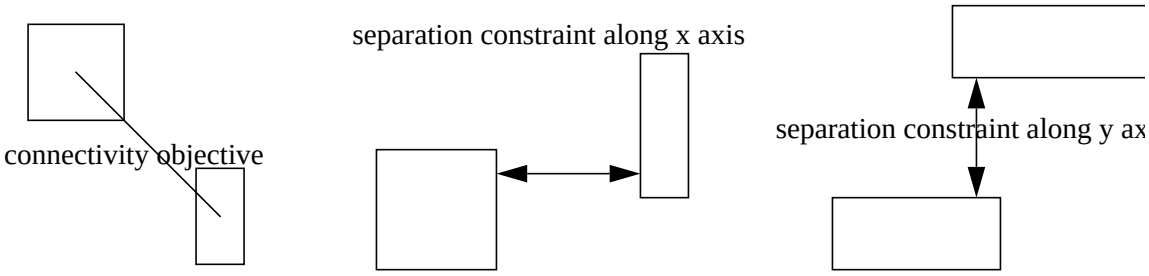


Figure 14. Examples of the Connectivity objective and the Separation constraints between bounding boxes of objects

A range of problem sizes was selected for the demonstration, the smallest being 10 objects, and the largest 35. A problem instance was generated by randomly selecting a set of objects from the repository and randomly assigning constraints between pairs of objects. A set of 10 problems was randomly generated for each problem size to make up the test bed.

The solution representation used was the translation and orientation (90 degree rotations) of the objects. A merit function to be minimized was defined to take into account all the objectives and constraints.

6.2.1. The Problem

Given: $\{A_i\}$: A set of 2D objects

$\{c_{ij}\}$: A set of connectivity costs between A_i and A_j

$\{Sx_{ij}\}$: A set of separation requirements along the x axis between A_i and A_j

$\{Sy_{ij}\}$: A set of separation requirements along the y axis between A_i and A_j

Let: $(X, O)_i$ be the translation and orientation (90 degree rotations only) of A_i

xl_i, xh_i, yl_i, yh_i define the bounding box of A_i

d_{ij} be the distance between the centers of the bounding boxes of A_i and A_j

$sx_{ij} = \max (xl_i, xl_j) - \min (xh_i, xh_j)$

$sy_{ij} = \max (yl_i, yl_j) - \min (yh_i, yh_j)$

$BBarea = (\max_i (xh_i) - \min_i (xl_i)) * (\max_i (yh_i) - \min(yl_i))$

$Overlap = \sum_{i,j} area (A_i \cap A_j)$

Connectivity = $\sum_{i,j} c_{ij} * d_{ij}$

SeparationViolation = $\sum_{i,j} (\max (Sx_{ij} - sx_{ij}, 0) + \max(Sy_{ij} - sy_{ij}, 0))$

Find: A layout of the objects $\{(X, O)_i\}$

To: minimize (BBarea + Connectivity + 10^4 * SeparationViolation + 10^6 * Overlap)

6.2.2. The Agents

Four algorithms of the kind typically used for such problems, encapsulated as agents, were implemented. All algorithms generate solutions with the representation defined above.

1. simulated annealing

Simulated annealing (SA) is commonly used in placement problems, especially in VLSI design. The SA agent was initially constructed using a geometric cooling schedule: the temperature is lowered by a fixed fraction at each iteration. A good cooling schedule was experimentally determined. The cooling schedule was then fixed, with temperature being determined as a function of time elapsed since starting the SA.

temperature $\alpha\ 0.99^{\text{time elapsed}}$

2. heuristic constructor (HC)

This constructor creates new solutions and is representative of the construction heuristics used for such problems. It starts with a random permutation of the objects to be placed. The objects are placed one at a time, with the candidate locations being restricted to vertices on the top-right profile of the existing layout as shown in the figure below.

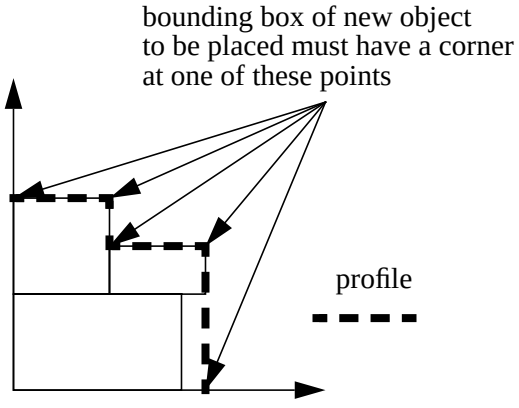


Figure 15. Profile of bounding boxes of 3 objects in a partial solution

Step 1: set profile to be arbitrary point in 2D space (e.g. the origin)

Step 2: generate a permutation of the objects.

Step 3: If all objects are placed, stop.

Step 4: Try placing object at all points on the current profile. Pick the placement with the lowest increment to the merit function.

Step 5. Update the profile taking into account the new object. Repeat from Step 3.

3. genetic algorithm (GA)

There are two approaches for packing used in GA. The first is to use an implicit solution representation combined with a decoder. The solution could be a permutation or ordering in which objects are to be placed by the decoder. The decoder is usually a construction heuristic similar to HC. However, it is difficult to construct decoders that can find optimal solutions. The second approach, which is used here, is to use an explicit solution representation. The GA agent has a maximum population size of 20 and a work cycle consists of running 50 generations. When connected to an A-Team memory, it selects 10 good solutions from the A-Team memory, runs 50 generations, and returns the 10 best solutions to the A-Team memory.

4. realign (R)

This is an example of an improvement heuristic. It modifies layouts by aligning objects so that they fit better. The agent takes a placement, arbitrarily selects two objects and moves one of them so that they overlap along an edge.

Each agent was assigned to a different computer so that they did not interfere with each other. For the duration of the experiments, care was taken to ensure that the agents did not compete with any other programs for computer resources.

6.3. Diversity, Work Cycles and Scheduling

Although humans and computers have complementary skills, there is a large gap between the speed at which the two operate. The time it takes an agent to run, to read solutions from its input memory, modify it and write the results back, is called the work cycle of the agent. The work cycle of computer agents might be milliseconds. Human beings cannot work at these speeds. Can agents with very different work cycles collaborate asynchronously and effectively? In order to answer this question, we need to study the effects of agent scheduling on A-Team performance.

Agents in A-Teams are constituted of three components: an operator, a selector and a scheduler. The operator (a heuristic or algorithm that has been encapsulated into an autonomous agent) is usually the main component of the agent. The selector determines what the agent will work on. The most commonly used selection mechanism is quality-based selection [BAT97]. Another alternative which has been proposed for multi-objective optimization is means-ends selection [Murthy92]. The scheduler is the component that determines when the agent will work.

Scheduling has not received much research attention and there are many open questions about scheduling: Are there optimum scheduling strategies for each agent? What are the penalties of using sub-optimal strategies? etc. All the A-Teams that have been built so far use fixed-frequency scheduling: the agent works periodically, with a fixed time interval between iterations. There are, of course, many alternate scheduling policies still to be explored. As an example of a policy, consider an agent that only makes small local adjustments to improve already good solutions. This agent might be more useful towards the end of an optimization process and should probably start with a low scheduling frequency which increases with time.

Almost all agents in current A-Teams use fixed frequency scheduling. This policy is closely related to the work cycle of agents, and the effect of this policy on the performance of A-Teams will be studied here.

6.3.1. Diversity and Work Cycles

Do agents with very different work cycles contribute in A-Teams? The analytic model of A-Teams [TBGD97] assumes that all agents have identical work cycles, but in practice this is usually not the case. The work cycle of the computer agents used for the experiments in this chapter varies considerably as can be seen in the table below. We would like to identify whether they all contribute to performance in spite of differences in work cycles.

Table 9. Work Cycles of the Computer Agents used for the layout problems.

Agent	work cycle (msec)
SA	250
HC	800
GA	2000
R	100

The first experiment is to determine whether agents with a range of work cycles all contribute, at least in some instances, when combined in A-Teams. The performance of the team with all agents was compared to the performance of the team with a single agent removed. Since A-Teams are stochastic in nature, and results can vary from one run to the next, ten trials were run for each A-Team for each problem instance. Each A-Team was run for 5 minutes and the best solution found was taken as the result of that trial. Consider the case of an agent X.

Let: $B = \{r_1 \ldots r_{10}\}$ be the merit function values of the best solutions found for 10 trials of the entire A-Team.

$B_x = \{r_{x1} \ldots r_{x10}\}$ be the merit function values of the best solutions found for 10 trials of the A-Team with agent X removed.

μ be the mean of B

σ be the standard deviation of B

μ_x be the mean of B_x

σ_x be the standard deviation of B_x

The agent is labeled critical if removing it from the team resulted in degradation in performance, where performance was considered to have degraded if $\mu_x > (\mu + \sigma)$ and $\mu < (\mu_x - \sigma_x)$.

Table 10. Is Agent Critical? Problem size = 20

Problem Number	Agent			
	SA	HC	GA	R
0	No	No	No	No
1	No	Yes	No	No
2	No	Yes	No	Yes
3	No	No	No	Yes
4	No	Yes	No	Yes
5	No	Yes	Yes	Yes
6	No	No	No	Yes
7	No	No	No	No
8	No	Yes	No	Yes
9	No	No	No	Yes

Table 11. Is Agent Critical? Problem size = 25

Problem Number	Agent			
	SA	HC	GA	R
0	No	Yes	Yes	Yes
1	No	No	No	No
2	No	No	No	Yes
3	No	No	No	No
4	No	No	No	Yes
5	Yes	No	No	Yes
6	No	No	No	Yes
7	No	No	Yes	Yes
8	Yes	No	No	Yes
9	No	No	No	Yes

Table 12. Is Agent Critical? Problem size = 30

Problem Number	Agent			
	SA	HC	GA	R
0	Yes	No	Yes	Yes
1	No	No	No	Yes
2	No	Yes	No	No
3	No	Yes	No	Yes
4	Yes	Yes	Yes	No
5	Yes	Yes	No	Yes
6	Yes	Yes	Yes	Yes
7	No	Yes	No	No
8	No	No	Yes	Yes
9	Yes	No	No	Yes

Table 13. Is Agent Critical? Problem size = 35

Problem Number	Agent			
	SA	HC	GA	R
0	Yes	Yes	Yes	Yes
1	No	Yes	Yes	Yes
2	No	No	No	No
3	No	No	No	No
4	No	Yes	No	Yes
5	Yes	Yes	Yes	Yes
6	No	Yes	Yes	Yes
7	Yes	No	Yes	Yes
8	Yes	Yes	Yes	Yes
9	Yes	Yes	Yes	Yes

The set of critical agents varies considerably from problem instance to problem instance even though the problems were all generated by the same problem generator with approximately the same level of complexity as measured by the number of constraints and the range of sizes of individual objects to be packed. It seems very difficult to predict for a given problem instance which agents are critical. Even though the work cycles of the agents differ significantly, each of them is critical in some subset of the problems, and there are several problems where all of them are critical, implying that agents with a range of work cycles can contribute when combined in A-Teams.

6.3.2. Sensitivity to Scheduling Period.

The traditional approach to scheduling is for agents to run as often as possible. These experiments attempt to measure how the scheduling period of an agent affects the performance of an A-Team. A subset of 6 problems was chosen for testing. The scheduling period of the agents was increased geometrically (one agent at a time) by adding delays between agent execution to create periods of 2, 4, 8, 16 and 32 times the work cycle.

As in the previous set of experiments, the A-Team was then allowed to run for 5 minutes, with the result being the merit function value of the best solution found within that time. Ten trials were run for each set of scheduling parameters and the mean value of the results are plotted below. Performance was also measured by measuring the time taken for the A-Team to first find a solution with a merit function value less than a predetermined threshold.

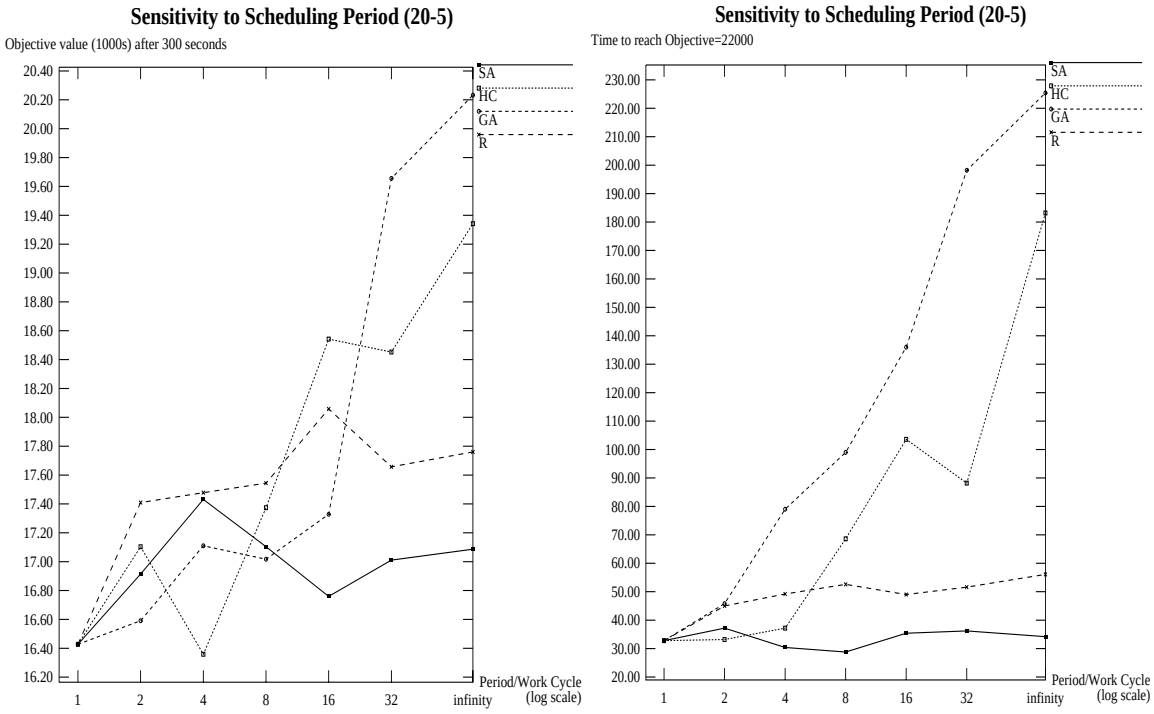


Figure 16. Performance vs. Scheduling Period (problem size = 20, problem number 5)

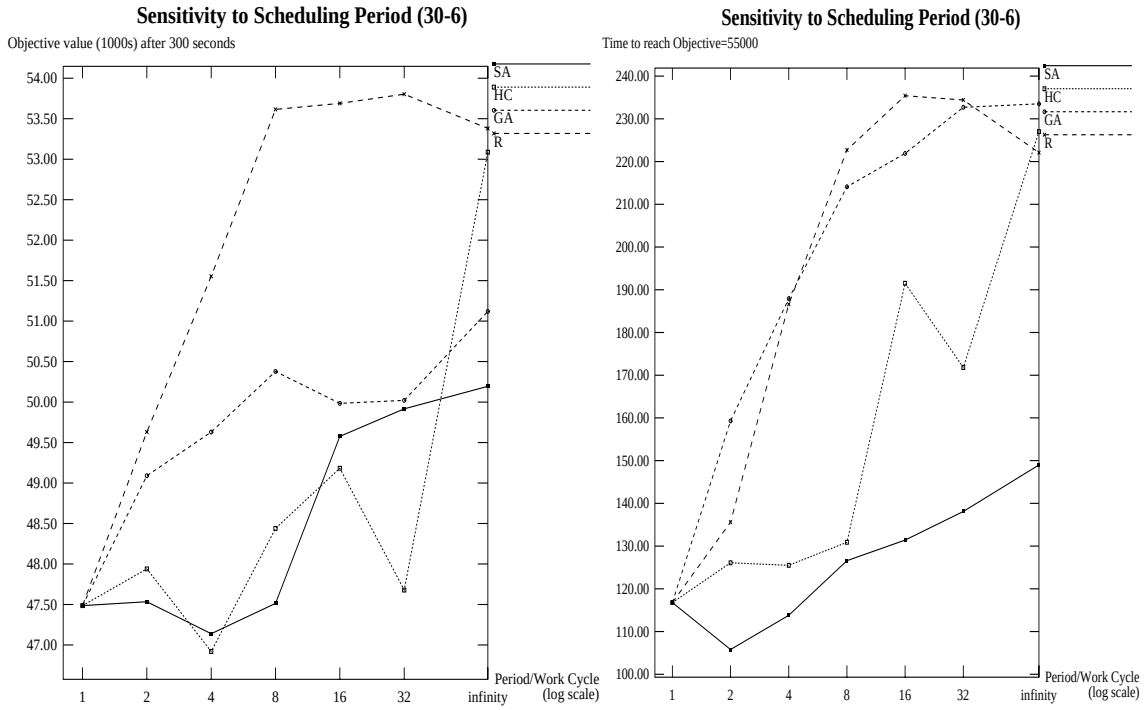


Figure 17. Performance vs. Scheduling Period (problem size = 30, problem number 6)

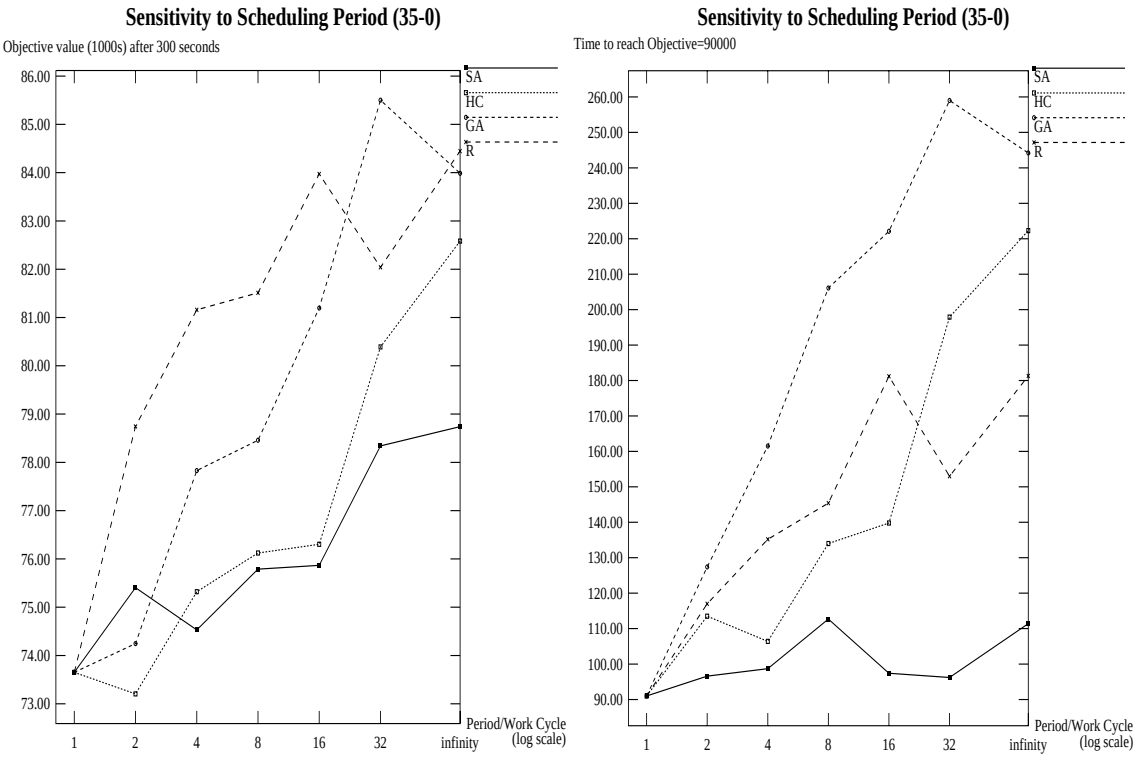


Figure 18. Performance vs. Scheduling Period (problem size = 35, problem number 0)

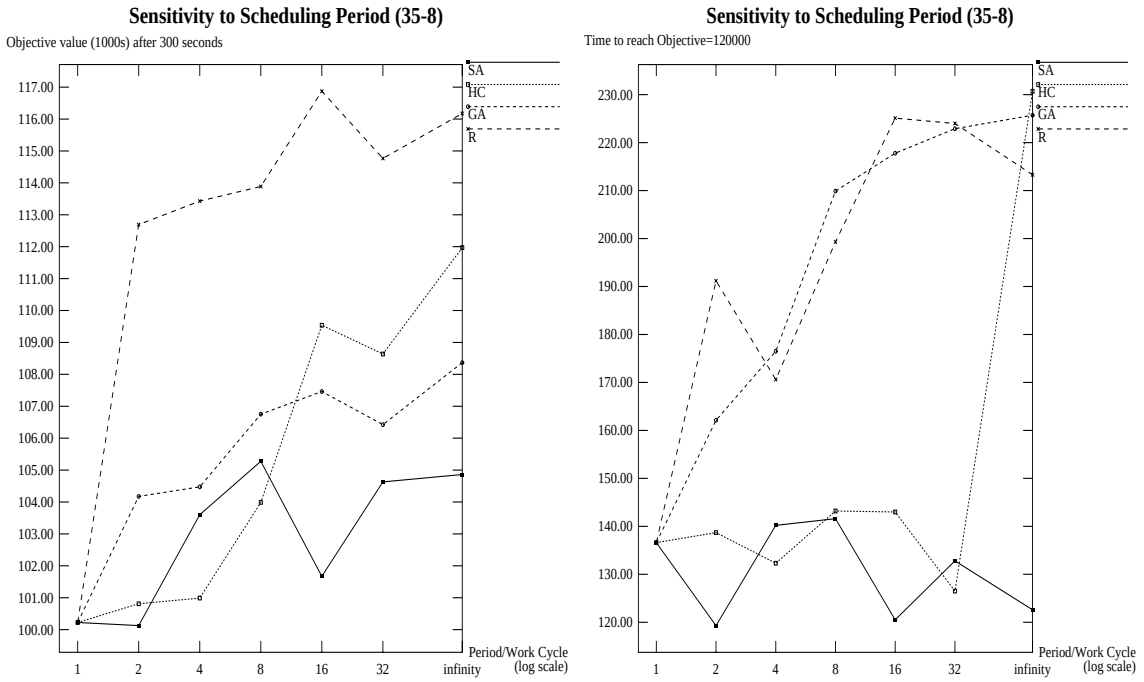


Figure 19. Performance vs. Scheduling Period (problem size = 35, problem number 8)

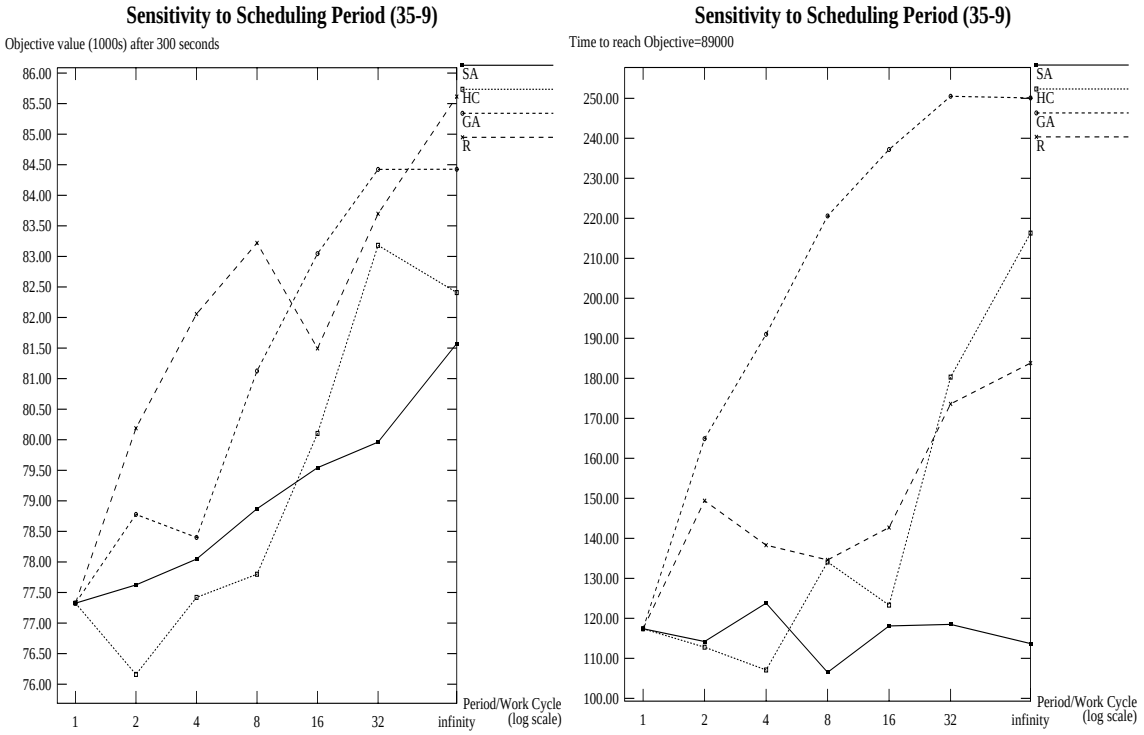


Figure 20. Performance vs. Scheduling Period (problem size = 35, problem number 9)

The graphs indicate that performance does not degrade significantly as scheduling period is increased. At worst, the performance degrades proportionally with the log of the scheduling period. These graphs also show that performance does not degrade significantly as scheduling period is increased. Again, in the worst case, the performance degrades proportionally with the log of the scheduling period.

6.3.3. Discussion

These results lead to two conclusions. The first is that diversity is desirable in A-Teams. Since the set of critical agents can vary unpredictably from one problem instance to the next and since it does not hurt performance to add agents, it is better to add agents to a team if they are likely to contribute.

The second conclusion is that the scheduling period for fixed frequency scheduling is not a critical issue in A-Teams. A single agent in a team can often be slowed down by adding delays between execution without significantly affecting performance, determined by measuring both quality-in-time and time-to-quality, which degrades gradually with the log of the scheduling period. The graphs for performance vs. scheduling period appear to have three regions: saturated performance regions when the scheduling period is very high or very low and a gradual transition region in the middle. It appears that when an agent is executing sufficiently often, there is little benefit from running more often and that it pays to speed up an agent only to the point where it reaches the saturation region.

A note of caution is required here: the experiments do not measure the impact of simultaneous changes for multiple agents which may be more significant. The experiments also do not reflect on the impact of performance of agents which are crucial to the team. If an agent performs a task necessary for solving the problem, and there is no other agent (or combination thereof) that can perform the same task (albeit less efficiently) then the effect of scheduling can be expected to be more significant. In the case of the A-Team used, there was no agent without which the problem could not be solved. Thus degradations in contributions of any of the agents could be compensated for by the others, reaffirming the robustness of A-Teams.

6.4. Synchronous vs. Asynchronous Human-Computer Interaction

We have determined that diversity is beneficial to A-Teams and that scheduling period has a small impact on performance. This means that human beings should be able to collaborate effectively with fast computer agents in A-Teams. How does asynchronous interaction between human and computer compare to traditional methods of interaction between humans and computers?

In the areas of design and optimization, humans usually interact with computers in a sequential fashion: the human makes changes to a solution (or to the problem formulation) - runs a computer tool - gets results - makes more changes - and the cycle is repeated until a satisfactory result is obtained. At any given time,

only one of them controls the modifications made to the solution; each must wait while the other works. Such cooperation schemes are sequential and synchronous in nature.

For example, consider the case of layout for VLSI design. One methodology is for the designer to use a computer tool to generate a set of high-level floor-plans, select one and run another tool to perform more detailed layout on it. “The floor planning step and the detailed placement and routing steps are usually applied iteratively until the best solution is found.” [Burich87] Once the best solution is found, it is usually passed through a “compactor” which pushes the blocks closer to each other to further increase packing density. The human and computer-tools work alternately under the control of the human.

The class of asynchronous cooperation schemes remains to be developed and studied. Autonomous asynchronous cooperation in (computer based) A-Teams of software agents has been previously demonstrated to work well in several domains, include design, scheduling and optimization. Such cooperation could be extended by allowing human beings to participate as equal members. This would facilitate incorporating of human abilities that are difficult, if not impossible, to encode algorithmically into the problem solving process. The computer methods would continue to function autonomously as in current A-Teams of software agents.

The A-Teams formalism provides a framework for parallel asynchronous human-computer cooperation. An experimental demonstration will show how the complementary skills of humans and machines can be synergistically combined.

6.4.1. Asynchronous Human-Computer Cooperation (HCC) and A-Teams

Sequential synchronous HCC has two main characteristics: (i) only a single solution is considered at any given time and (ii) the human is usually in control, deciding when and what computer method to apply to the candidate solution.

A typical example of such synchronous HCC is found in [OC94], where the performance of a human alone, a software “agent” alone, and a combination of the two are compared. The task is to generate schedules for a simulated shipping task. Goods are to be transported from port to port, which act as bottlenecks where ships must queue up for loading or unloading. Given a partial schedule, and current location of the ships, the agent predicts future bottlenecks and attempts to route ships around them, thus incrementally constructing a schedule. In the combination case, the human constructs the schedules with the help of the agent, which acts as a critic and points out potential bottlenecks and recommendations for the incremental construction of the schedule. The human then decides what to do next (and may completely disregard the agents recommendation). Four measures of schedule quality are used. The results for the combination of human and computer lie between those of either alone on all four measures.

The process follows the trajectory of a single solution through the decision space (DS). At each point in time the entity (human or computer) most likely to benefit the solution the most is selected by the human to work on the iterate.

An asynchronous approach to combining human and computer skills is found in [Shahroudi97]. In this case, problems are formulated as NLP problems, which are then solved by a traditional SQP algorithm. The human has real-time control over: the problem formulation (numeric values of the bound constraints, and weights on different components of the merit function); the solution (decision variables); and on internal variables of the SQP package (tolerances etc.). The human being guides the automatic optimizer through decision space and assists to the SQP algorithm to: reach convergence; satisfy constraints and bounds; and satisfy optimality conditions. This also provides the human with the ability to: observe the trade-off between different components of the merit function; study the impact of bound constraints; and force the search into different regions of the decision space to find alternative minima.

The model of cooperation proposed in this chapter considers multiple solutions simultaneously, exploring more of the decision space in parallel, and free the human from having to coordinate the computer methods by augmenting each of them with self contained control mechanisms to guide their operation. The use of populations in A-Teams allows human and computers, both working at considerably different speed scales, to collaborate without obstructing each other. Neither is forced to wait for the other, but can work to improve whatever solutions are available in the shared memories. Of course, if it is necessary for the two to alternate (i.e. the human can only work on computer-generated solutions and vice-versa) then one may turn out to be a bottleneck, but since A-Teams are open to the addition of agents, duplicates (either several humans, or multiple copies of computer agents) can also be added to compensate for the speed differential. Populations of solutions also facilitate the simultaneous development of multiple trajectories in, and the simultaneous exploration of several promising regions of, the decision space.

Humans collaborating in A-Teams decide for themselves when and what to work on just as the computer agents do. Humans may participate as advisors for agents by affecting the control mechanisms of the agents, but we will not explore this issue here and will focus on humans as agents. Humans may function as constructors, creating and modifying solutions to add to the memory and/or as destroyers removing solutions from the memory.

6.4.2. The Human-Computer Interface

An interface is required between the human and the A-Team memory to allow the human to participate. The interface should allow the human to browse the contents of the memory in order to select what to change or destroy. If the human is acting as a constructor, an interface (possibly the same as the browse interface) is also required that enables the human to manipulate solutions. The interface to the memory could be static or

dynamic. A dynamic interface would always show the latest contents of the memory, a static one would allow the user to take snapshots of the memory with which to work.

A dynamic interface keeps the human constantly updated on changes to the population but adds considerable overhead since it has to continuously poll the memory, pull the latest information and update the display. It can also be confusing to the human if the memory is changing very rapidly. Although a static interface may get outdated and requires that the human periodically obtain a new snapshot, it acts as a buffer to isolate the human from rapid changes created by the computer agents. For this reason, a static interface was developed for the demonstration. Additionally, the ability to manipulate solutions was incorporated into the interface, allowing the human to act as a constructor as well as a destroyer in the A-Team.

6.4.3. Synchronous vs. Asynchronous Collaboration

In order to compare synchronous with asynchronous collaboration, an experiment was performed on a problem of size 30. The experiment was run for 20 minutes per trial. Three solvers were used: A computer A-Team; the A-Team in a synchronous collaboration with the expert human; and the A-Team in an asynchronous collaboration with an expert human agent. Since I had designed the system, I acted as the test “expert” subject.

In the synchronous case, the human tracks the performance of the computer A-Team, and intervenes whenever progress appears to be leveling off. Intervention includes purging of bad solutions from the memory, as well as improving solutions and adding them to the memory. In the asynchronous case, the human acts in the same capacity as the other agents, primarily to modify and improve solutions.

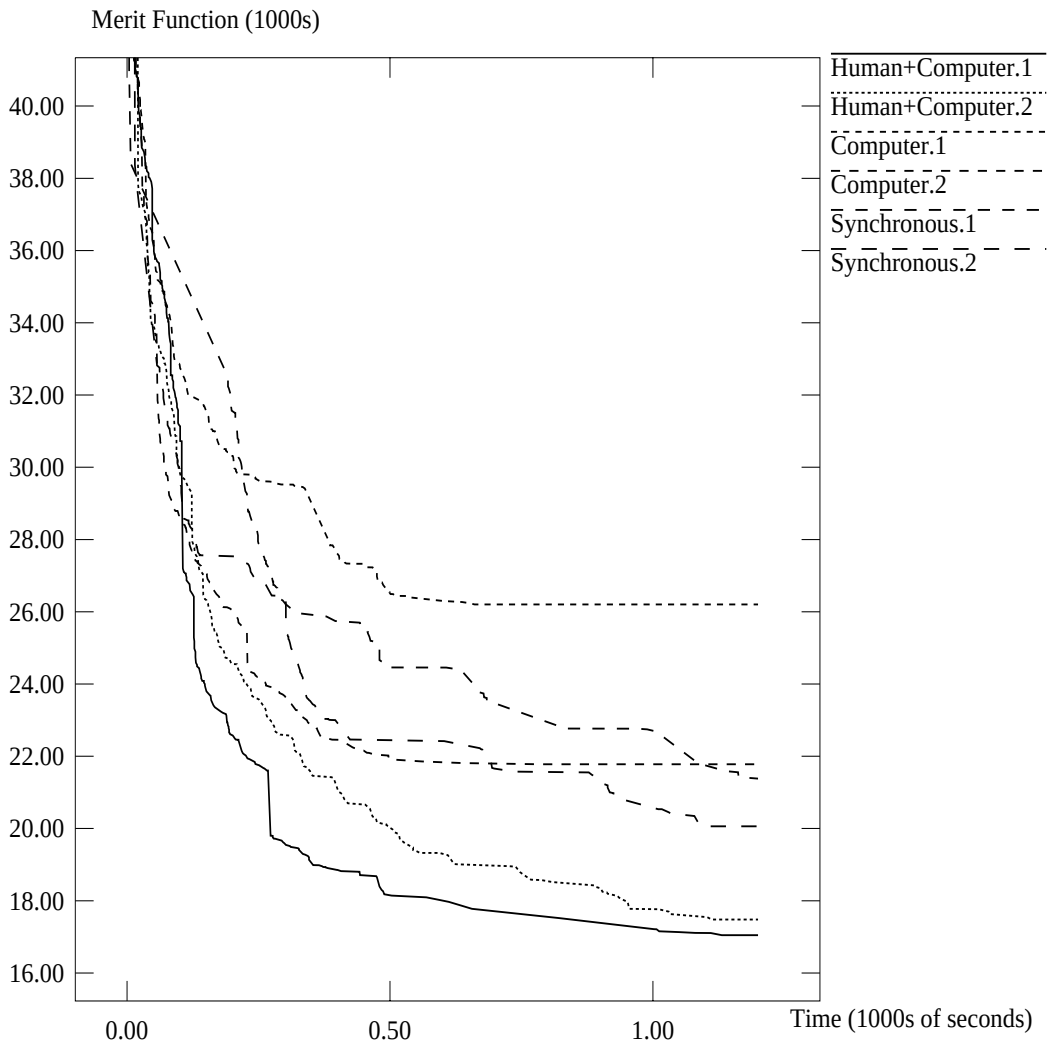


Figure 21. Performance of computer alone, synchronous human-computer collaboration and asynchronous human-computer collaboration.

Two trials were run for each scenario, and the value of the merit function plotted as a function of time. Initial progress was similar. The performance of the computer A-Teams leveled off within the first half of the experiment. Both made little or no progress beyond this point. In comparison, all solvers involving humans continued to make progress in the second half of the experiment.

In the synchronous case, we observe a step-like curve with the flat regions corresponding to when the computer progress has slowed and/or the human is working. In spite of the human periodically slowing down progress, overall progress is better than without the human. There is usually a rapid improvement after the human has worked indicating that the human has helped provide good inputs for the computer A-Team. In the asynchronous case, the combined A-Team including the human performs better than both computer A-

Team alone and the synchronous collaboration between the A-Team and the human. This is because the computer agents can continue to make progress even while the human is working, and the performance curve is smooth compared to that for synchronous cooperation.

6.5. Synergy in Human-Computer Collaboration

We have determined that humans and computer agents can collaborate effectively while interacting asynchronously as members of an A-Team. This set of experiments tries to determine whether they are both contributing, or whether all the work is being done by either the human alone or the computer agents alone. For purposes of this demonstration, the effectiveness of a solver was measured by the quality of the best solutions found by it in a fixed time interval. The answer to whether an A-Team consisting of humans and computer based agents find better results than either component must be inferred from the following: Given an optimization/design task and a set of computer agents, does an A-Team consisting of the computer agents and a human find better solutions than an A-Team of just the computer agents or the human alone.

Three solvers were used to solve all the benchmark problems.

C: Computer based agents in an A-Team

H: Human

T: Human + Computer based agents in an A-Team

Each solver was given five minutes to solve each problem. Two test humans were used. I acted as the "expert" human, and in order to corroborate the results, I asked another graduate student in the ECE department to volunteer as a test "novice" subject.

6.5.1. Results

Comparison of results for team T (human and computer based agents) to team C (computer based agents) and to H (human) shows that there is clearly a synergistic effect for the larger, and more complex, problems. The combined team T performs better than both H and C in most of the individual trials. Details of performance for each of the larger problem instances are also presented below. As can be seen, the combined team almost always finds better solutions than either of its individual components, and in a large number of cases the combined A-Team has a significant advantage.

problem size	% by which solutions for T are better	
	T vs. C	T vs. H
10	5.8654	1.60269
20	13.9038	-5.5534
25	23.8944	16.8327
30	9.10957	20.0804
35	11.1256	14.5266

Table 14. Comparison of results for team T (human and computer based agents) to team C (computer based agents) and to H (human) for Asynchronous Cooperation. The results are averages over 10 problem instances for each problem size.

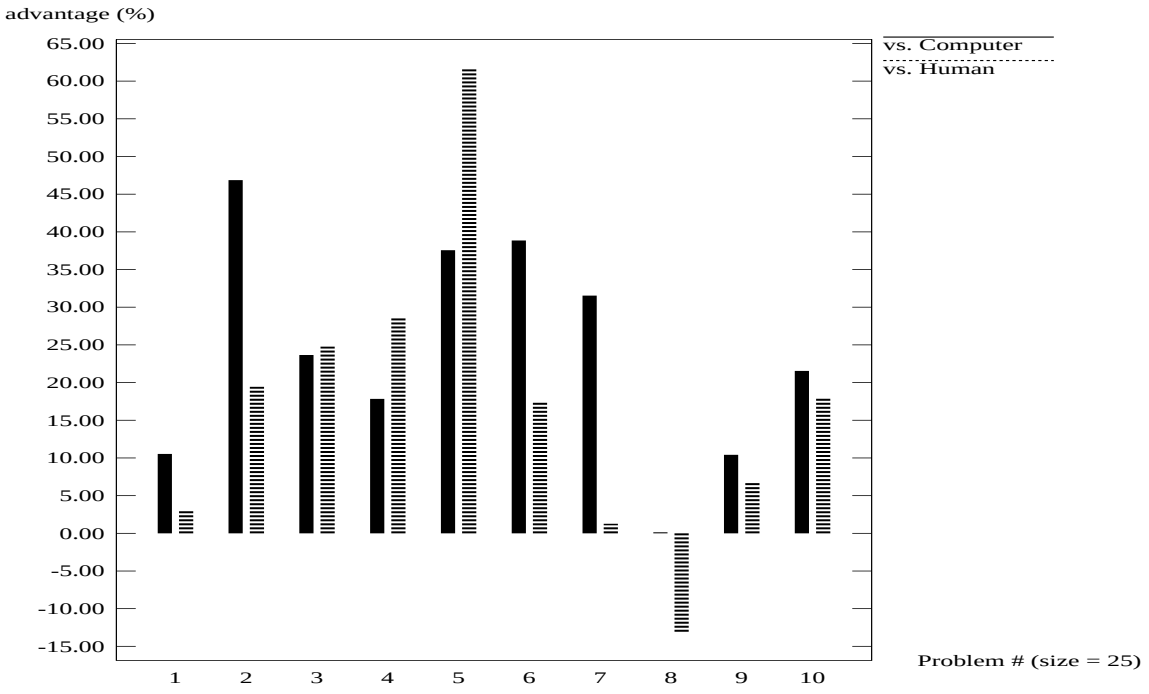


Figure 22. Comparison of Performance: The percentage by which solutions found by a combined human-computer A-Team are better than those found by either a human or a computer A-Team alone for ten sample problems of size 25.

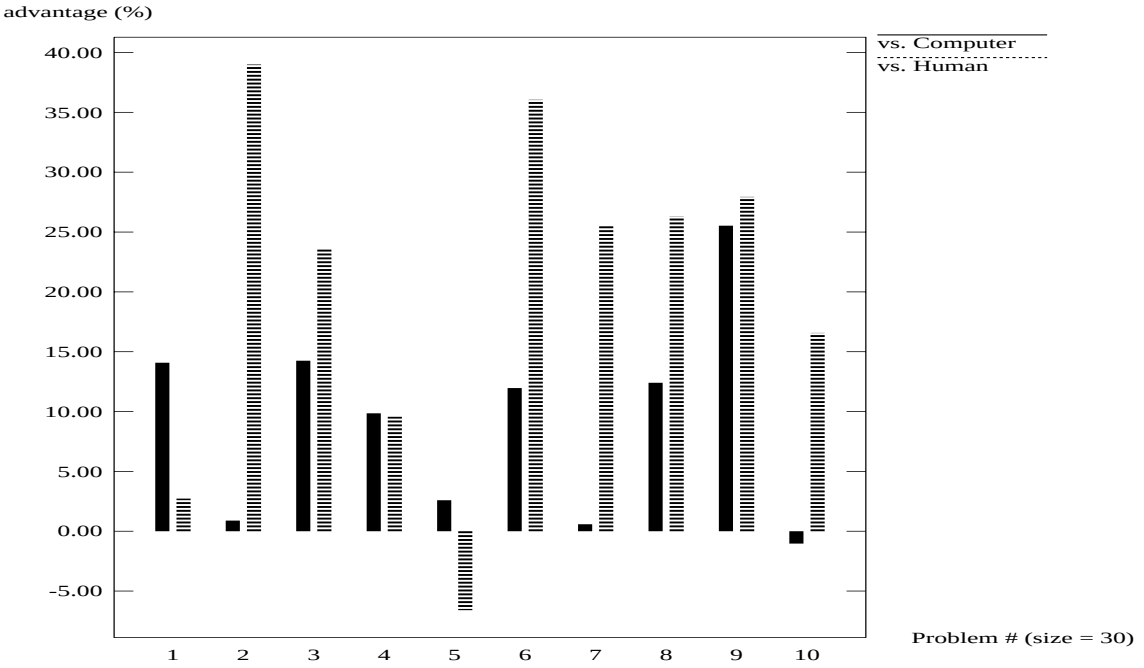


Figure 23. Comparison of Performance: The percentage by which solutions found by a combined human-computer A-Team are better than those found by either a human or a computer A-Team alone for ten sample problems of size 30.

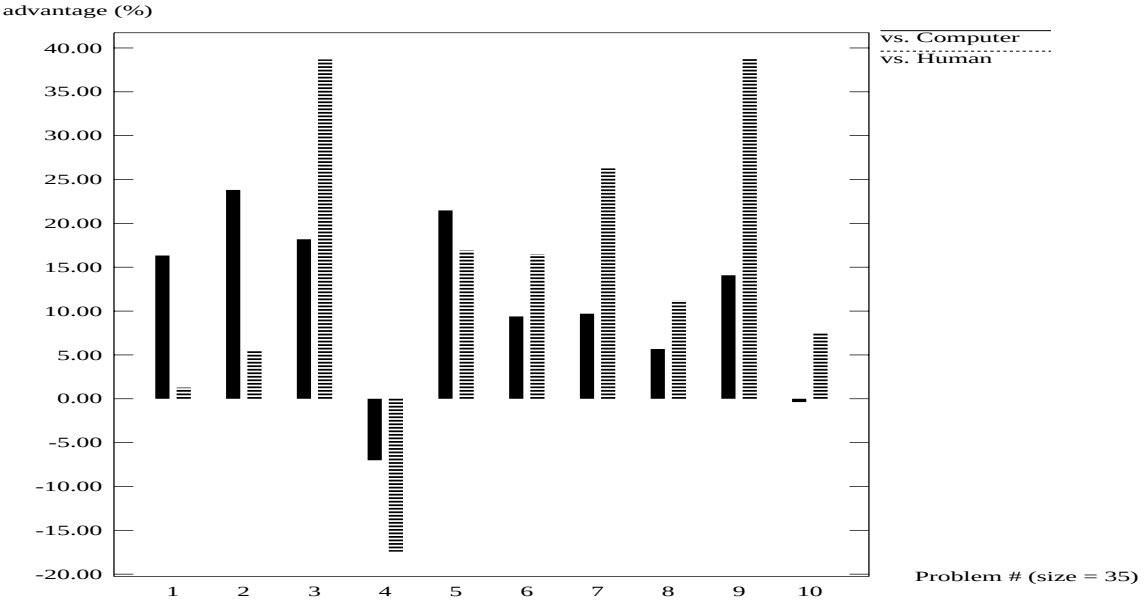


Figure 24. Comparison of Performance: The percentage by which solutions found by a combined human-computer A-Team are better than those found by either a human or a computer A-Team alone for ten sample problems of size 35.

These results show that humans and computers can effectively collaborate in order to solve such layout problems. The human can contribute to the solution process even for problems where the human would have

extreme difficulty in solving the problem alone. For the smaller problems, the human alone can find solutions that are close to the global optima. As the problems get larger and more complex, and the number of minima increases, the human finds it increasing difficult to find good solutions. Another factor that makes it difficult for the human is the interaction between constraints.

In order to corroborate the results, a novice subject who was exposed to the system for the first time was requested to participate. Since the experiments require time and attention from the human, only the ten problems of size 30 were used. The novice subject (a graduate student in the ECE department) was simultaneously learning to use the user interface and strategies to follow while performing the task. He refused (after attempting two problems and failing to find feasible solutions) to attempt the task unaided. His performance is shown below.

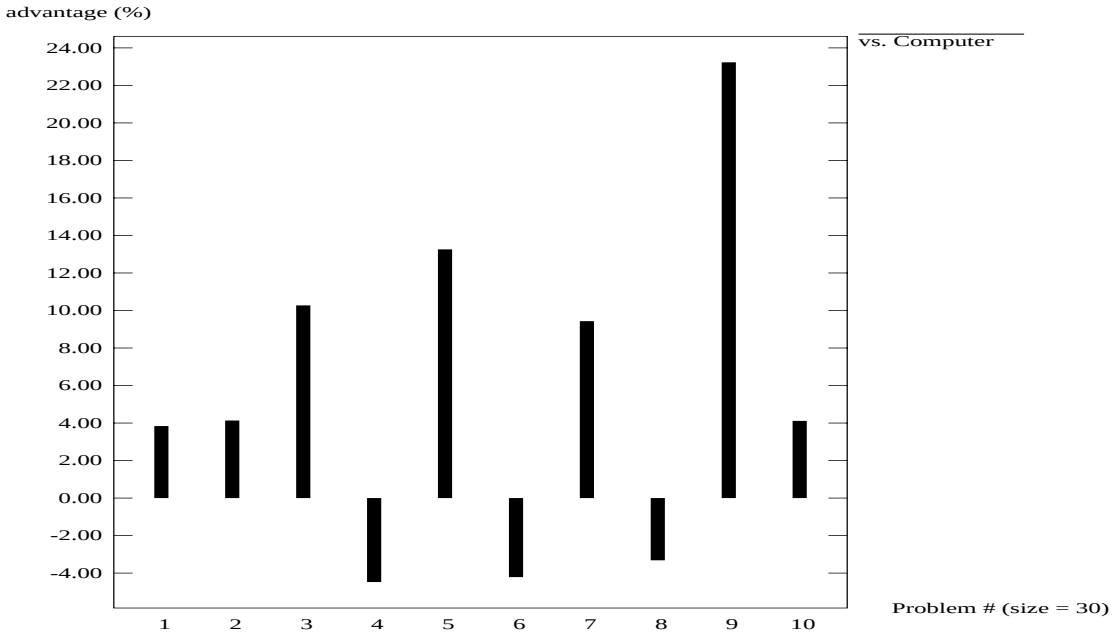


Figure 25. Comparison of Performance: The percentage by which solutions found by a combined (novice) human-computer A-Team are better than those found by computer A-Team alone for ten sample problems of size 30.

Although the combination with the novice did not perform as well as the one with the expert, it performed better than the computer A-Team alone. Where the combination did worse than the computer alone, in 3 of the 10 cases, the solutions were still fairly close (on average 4.0% worse). In the remaining 7 cases where the combination did better than the computer alone, the solutions were much better (on average 9.8% better). Overall, the combination found solutions that were 5.6% better than the computer alone. In contrast, the combination with the expert performed 9.1% better on average for the same problems, finding better solutions in 9 out of the 10 cases, doing only marginally worse in the last case.

6.5.2. Strategies for Effective Collaboration

In order to understand how the human was contributing to the solution process it is helpful to list some of the effective strategies used and some of the problems encountered.

1. Keep all objectives in mind. Don't focus on only one.

There were two conflicting component objectives in the merit function for the demonstration problems: bounding box area and connection cost. It was not always obvious for a given solution whether the merit function would decrease more by reducing the bounding box area (and a possibly increasing connection cost) or by reducing the connection cost (and possibly increasing in bounding box area).

2. Even minor improvements help.

Another important discovery was that even minor improvements helped the overall process. The strategy here was to make any obvious improvements since the computer agents did not find those improvements as obvious.

3. Don't get stuck on a single solution.

This is a major obstacle to human contribution. It is easy for humans to get stuck on one solution and try all possible ways of improving it rather than looking at the other solutions available in the solution memory.

4. Share results with other agents.

Cyclic dataflow is a key characteristic of A-Teams - and this experiment reaffirmed the importance of this cyclic characteristic. It was found to be important for the human to update the snapshot of the A-Team memory often. And even more important to add solutions to the memory often - even after making only a few changes - as this allows other agent to use them.

5. Try many alternatives.

Since the computer can quickly evaluate and eliminate solutions - it was found to be useful to add all the obvious (possibly improving) variations on a solution and let the computer agents pick the best ones. Thus the human could avoid having to evaluate and compare evaluations for all the variations. If there were several locations/orientations in which to place an object, the best actions would be to try them all rapidly (i.e. place and add, place and add …).

6. Avoid adding violations.

This strategy reflects the weakness in selection technology (which solutions an agent should work on or destroy). A biased-by-quality strategy was used here - and since violations were heavily penalized in selection - infeasible solutions (with overlaps or separation violations) were more likely than not discarded. Taking the trouble to actually align the objects closely and shorten the connections as much as possible greatly increased the likelihood that the solution would get worked on by other agents and lead to the new best solution

6.6. Summary

There are many problems that can benefit from combining human and computer based solvers. An example of such a class of problems is packing or layout problems subject to constraints. Human beings are very good at packing objects in 2 or 3 dimensions, and considerably better than computers if the objects are non-convex. They are especially good at spotting improvements to solutions such as identifying open spaces where objects might fit. They are very weak, however, if the number of objects is very large or if there are additional numeric objectives and constraints to satisfy.

Human and computer skills can be combined effectively using the A-Teams formalism. Humans and software agents cooperate around shared memories, and work asynchronously and independently of each other. The human acts as an equal to the computer based agents, working to improve solutions in the shared repository. The human is freed from having to match software algorithms with solutions or having to control the software agent. The team of human and computer agents performs better than either alone. The computer contributes the ability to check many alternatives while the human contributes the ability to spot good fits of the different objects.

The asynchronous arrangement allows the computer agents to work independently of the much slower human. It also frees the human from having to direct the operation of the computer algorithms, resulting in smoother and faster progress to better solutions.

The key to successful human-computer asynchronous cooperation in the demonstration is iterations: usually lots of small improvements lead to better outcomes than a few major improvements. The human must focus on human skills such as pattern recognition and the ability to see how pieces can be rearranged to fit better. Collaborating with software agents in an A-Team is a very different experience from using software tools and the graphical user interface for the human must be designed appropriately. The human must also get used to such an interaction which at least in the case of the demonstration took some getting used to, but were not unpleasant. It is important for the human participant to keep in mind that iterations are very important, and to frequently share results with the computer agents.

Such asynchronous human computer collaboration might help in transitioning from human based solutions to problems to computer based ones. Algorithms can be developed and applied to problems alongside human experts. This characteristic could be very useful in areas where human experts are still dominant, but computer based solutions are starting to become feasible.

7. DIVERSITY IN CONSTRUCTION AND DESTRUCTION

Diversity of construction agents has been shown to beneficial to A-Team performance in several cases including the previous chapter and [deSouza93]. This chapter will present another example of diversity in construction to show how algorithms can compensate for each others weaknesses if combined in A-Teams. Traditional destruction mechanisms in A-Teams have focused on quality based destruction. This chapter will show how quality based destroyers can be augmented with destroyers based on clustering techniques [TZ89] or tabu search [Glover89] to improve A-Team performance.

7.1. Diversity in Construction

The following experiment studies the effect of combining SQP and GA into an A-Team. Although both can be used for the same types of NLP problems, SQP is very fast near optima, but is slow or may fail altogether further away. It also finds only those optima that are near the starting point. The GA on the other hand is better at exploring the decision space and finding better quality solutions, but is much slower than the SQP at finding the optima once it gets close. An A-Team consisting of a GA and an SQP provides a trade-off between the robustness of the GA and the speed of the SQP.

An A-Team having a GA and an SQP and a memory size of 20 was used to solve seven global optimization benchmark problems from [FP90] and [Ramesh94]. Both the GA and SQP agents used selectors that had a bias towards selecting the better solutions in the memory for the agents to work on. The SQP agent used a single point selected from the memory as a starting point and returned the optimum found by the SQP. The GA started with about 10 solutions from the memory, ran 200 iterations with a population size of 50, and returned the 10 best solutions found. Each agent executed on average 10 times per trial.

Two comparisons were made using exclusively one or the other of the constituent algorithms (agents) of the A-Team. The SQP comparison consisted of using a random multi start SQP (RMS-SQP), where the best of 20 random starts for the SQP was selected as the result of each trial. The GA comparison consisted of running the GA for the equivalent of 8 times the amount of iterations of the GA in the A-Team.

Table 15. SQP vs. A-Team: Results for 10 trials per problem

	random multi start SQP		A-Team	
	average	best	average	best
FP23	-13.8281	-13.8281	-15	-15
He6	-194.7	-298	-310	-310
Gr10	1.12016e-2	0	5.54322e-3	0
Gr15	1.81949e-3	0	4.5024e-4	0
Gr20	1.11649e-4	0	1.54211e-5	0
D1	FAIL	***	0	0
D2	FAIL	***	0.347991	0.347991

The RMS-SQP converges faster than the GA, and in problems Gr10, Gr15 and Gr20 it can find solutions that are almost as good as those of the A-Team in slightly less time. However, it is not very robust and fails to find the optimum for FP23, He6, D1 and D2. In fact it was never able to find feasible solutions for D1 and D2, even when it was left running for hundreds of trials.

Table 16. GA vs. A-Team: Results for 10 trials per problem

	Genetic Algorithm		A-Team	
	average	best	average	best
FP23	-14.89998	-15	-15	-15
He6	-309.989	-310	-310	-310
Gr10	4.07867e-2	0	5.54322e-3	0
Gr15	4.91301e-3	1.72672e-3	4.5024e-4	0
Gr20	1.56984e-3	1.9739e-4	1.54211e-5	0
D1	0	0	0	0
D2	0.347991	0.347991	0.347991	0.347991

The GA is much more robust than the RMS-SQP but considerably slower than the A-Team. In the first five problems, the quality of the solutions were poorer than those of the A-Team even after the GA had run for 5 times as long as the A-Team and progress had almost stopped. For D1 and D2 it took between two and three times as long as the A-Team to find the optima.

A comparison of the best solutions found by the three methods is presented below. The A-Team finds the optimal in all cases. The RMS-SQP fails to find the optimal in four of the seven problems. In two of the problems it fails to find even feasible solutions. The GA can find close to optimal solutions, but it takes very long to do so. For example, in two of the cases it does not locate the exact optimum even after executing for about five times as long as the A-Team.

Table 17. Comparison of best solutions found (note: A-Team finds optimal in all)

Problem	RMS-SQP	GA	A-Team
FP23	-13.8281	-15	-15
He6	-298	-310	-310
Gr10	0	0	0
Gr15	0	1.72672e-3	0
Gr20	0	1.9739e-4	0
D1	FAIL	0	0
D2	FAIL	0.347991	0.347991

These results show the advantage of opportunistic application of agents with diverse skills to solutions. For solutions that are far from the desired optima, and on which the SQP might fail, the GA provides robustness

and brings the solutions closer to the desired optima. For solutions that are close to optima and where the GA is very slow, the SQP provides speed to rapidly reach the optima.

7.2. Diversity in Destruction

Destroyers perform two functions in A-Teams. The first is to keep the populations of memories in check and prevent them from growing indefinitely. The second is to assist constructors by identifying and eliminating unpromising solutions. In this role, destroyers force the team to search for better solutions, improving the results of the A-Team as time progresses. This section will present some alternative destruction mechanisms that can be added to A-Teams to improve performance.

7.2.1. Destruction Policies

There are several possible policies that can be used in the design of destroyers. However, almost all destroyers constructed to-date use just one of them, quality-based destruction. I will describe some of the possible policies here.

1. Distance Based Destruction (DBD)

This policy is based on the model used to study A-Teams [Talukdar98]. The goal is the region of the decision space consisting of acceptable final solutions. The rest of decision space is divided into regions based on the shortest path to the goal measured in number of agent applications assuming that the optimal sequence of agents operates on each solution. DBD is primarily of theoretical interest since it is usually not possible to measure distance from the goal.

2. Quality Based Destruction (QBD)

This is the policy used by almost all destroyers constructed so far by users of A-Teams. QBD forces the A-Team to improve the average quality of the solutions in the memory by removing solutions with worse evaluations. QBD resembles the process of eliminating solutions used in GA, SA and hill climbing. QBD is analogous to the selection of the breeding population in GA, except that (i) the destroyers work asynchronously of the constructors rather than in a sequence of “generations,” and (ii) the destroyers always leave some of the population, whereas with the GA the entire population may be replaced with none of the original population surviving. There are several ways in which the QBD destroyers have been biased. Examples include: destroying the worst solutions in the memory resembling elitist selection in GA; destroying worse solutions with a bias proportional to their quality known as roulette wheel destruction in GA; or destroying the worst of a small number of solutions selected randomly.

QBD may not always be the best means of destruction, especially when the correlation between the evaluation of a solution and its distance from the goal is not high. For example, consider the case of optimizing a

multi-modal function where most of the optima are not acceptable. QBD is likely to prevent solutions from escaping the neighborhoods of such unacceptable optima.

3. Tabu List Destruction (TLD)

Tabu Search [Glover89] follows the trajectory of a single solution through decision space. It behaves much like hill climbing, except that a list of solutions is marked Tabu. These solutions are the most recent solutions visited. The list size is usually between 7 and 12. The list changes with each iteration. The best possible non-Tabu solution generated from the latest iterate is selected for the next iteration. The tabu list prevents cycling, and can force the solution trajectory out of local optima, provided the tabu list is sufficiently large. Tabu lists can also be used to mark previously visited regions of the solution space to prevent cycling over longer intervals. Such lists are referred to as medium term and long term lists.

An A-Team using QBD may easily get trapped in unacceptable optima. Escapes from such neighborhoods are facilitated by the tabu lists in Tabu Search. This suggests that Tabu Lists could be used by destroyers in A-Teams. Tabu List Destruction (TLD) can also be used to retain a “memory” of the search to help reducing cycling.

4. Cluster Based Destruction (CBD)

One of the advantages of populations of solutions is the exploration of many regions of the decision space. If there is a large number of optima, QBD is likely to get stuck in just a few, leaving the rest unexplored. Clustering is a technique used in global optimization to locate a large number of optima [TZ89]. Solutions that are clustered close together are likely to like near the same optimum, whereas solutions that are further away would lie near different optima. Cluster membership can be used to eliminate solutions that are likely to lead to the same optimum, and to preserve solutions that would lead to different optima. Cluster Based Destruction (CBD) can be used to help in the exploration of the decision space.

7.2.2. Experiments

A set of 30 randomly generated 20-city TSP problems - 10 euclidean, 10 (non-euclidean) symmetric and 10 asymmetric was used to test the effects of augmenting a typical A-Team using QBD by adding destroyers using CBD or TLD. The memory capacity was set to 50. The same construction agents were used in all trials. Each trial was run for 100,000 tour evaluations.

Ten problems were used in each category to average out problem dependent behavior. Ten different initial populations were used per problem to average out the effect of starting points. Each trial was also run for sufficiently long so that further improvements were unlikely.

A single implementation of TLD and two different CBD implementations were tried.

Let: (x, xid) be the solution and a unique id tag for each solution in the memory

d(x, y) be a measure of distance between solution x and solution y in decision space.

M = {(x, xid)} be the current memory of solutions

Tabu List Destroyer (TLD):

Step 1: Eliminate Tabu Solutions.

If T = {(x, xid)} is the tabu list of solutions

For all (x, xid) $\in$ M, and (y, yid) $\in$ T: if d(x, y) < $d_{allowed}$ and xid $\neq$ yid then M = M\(x, xid)

where $d_{allowed}$ defines a small region around solutions

Step 2: Update Tabu List.

T = T + B, where B = {(x, xid)} is a list of the 5 best solutions in M\T

while |T| > 200, T = T \ T_{oldest}, where T_{oldest} is the oldest item in T

Cluster Based Destroyer (CBD1):

This destroyer finds a fixed number of variable sized clusters in the memory and leaves a single solution in each cluster.

Step 1: Identify Clusters

Let C_i = {(x, xid)} be a cluster of solutions in M

and W_i = max(d(x, y)) is a measure of the width of C_i where x, y $\in$ C_i

Find C_1 … C_5 to minimize $\sum_i W_i$ s.t. $C_i \cap C_j = \varnothing$ for i $\neq$ j.

Step 2: Prune Clusters

If $C_{iw} = C_i \backslash C_{ib}$ where C_{ib} is the best element in C_i

$M = M \setminus \{C_{1w} \cup \ldots \cup C_{5w}\}$

Cluster Based Destroyer (CBD2):

This destroyer eliminates solutions that are close the the best solutions in the memory with a high probability.

Step 1: Identify Clusters

let A = M

while $A \neq \varnothing$

C_i = best solution in A + all solutions within a given radius of the best solution

$A = A \backslash C_i$

Step 2: Prune Clusters

$C_{iw} = C_i \setminus C_{ia}$ where C_{ia} is an arbitrarily selected element in C_i

$M = M \setminus \{C_{1w} \cup \ldots \cup C_{nw}\}$

Two different scheduling frequencies were tried for each destroyer: with similar scheduling for the QBD destroyer and the additional destroyer; and with the additional destroyer operating ten times as often as the QBD destroyer.

7.2.3. Results

The scores used to compare the various destruction policies were generated as follows. Consider the euclidean problems. Ten different starting solution populations were generated for each of the ten euclidean problem. Each A-Team was used with each starting population, for a total of 100 trials. Each A-Team was assigned a rank for each trial based on the quality of the solutions found in each trial, where the best A-Team got a rank 1 and the worst got a rank of 7. In the case of ties, the A-Teams were assigned the same rank number.

The overall score for A-Team i is $\sum_j r_{ij} / 100$

where: r_{ij} is the rank for A-Team i in trial j; $1 \leq i \leq 7$ and $1 \leq j \leq 100$

The destroyers and their relative scheduling frequency in each A-Team are indicated in the left column. The A-Teams were scored separately for each problem type.

Table 18. Scores for combinations of destroyers for the three types of problems.

A-Team	Destroyers	Score		
		euclidean	symmetric	asymmetric
1	QBD	2.02	3.23	4.28
2	QBD + TLD	3.02	5.5	4.43
3	QBD + 10 x TLD	1.76	3.06	3.81
4	QBD + CBD1	2.62	4.99	4.11
5	QBD + 10 x CBD1	2.52	5.34	4.38
6	QBD + CBD2	2.02	2.81	3.51
7	QBD + 10 x CBD2	1.82	3.01	3.48

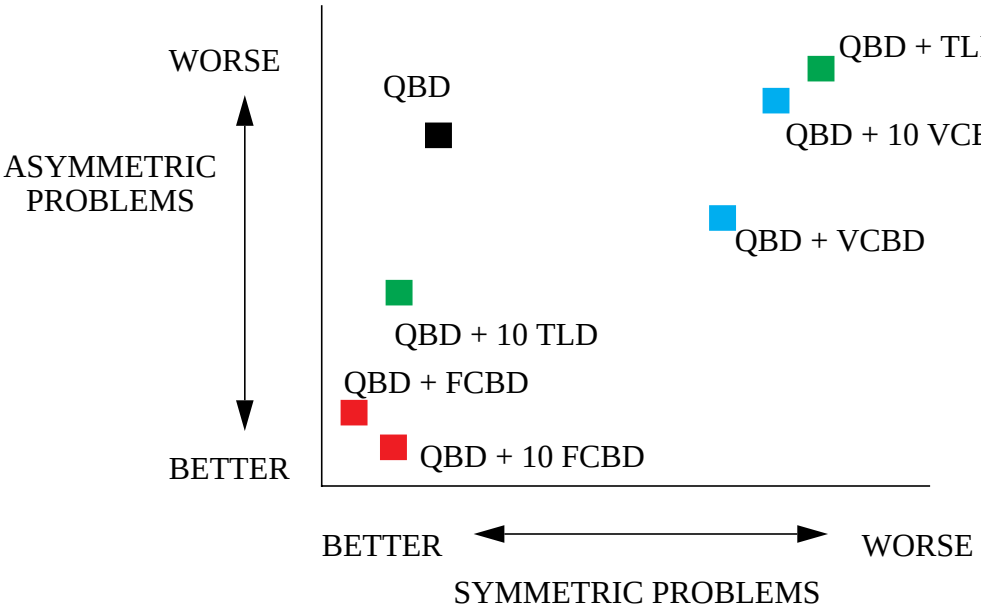


Figure 26. Comparison of A-Team performance for Asymmetric vs. Symmetric Problems.

The benchmark A-Team is A-Team 1. A-Teams 4 and 5 seem to be consistently worse than the others indicating that variable sized clusters do not work well. A-Teams 2 and 3 performed worse and better than A-Team 1 respectively. This indicates that scheduling frequency is important for the QBD-TLD destroyer combination. QBD with CBD2 performs better than QBD alone for both scheduling ratios tested. Adding TLD and CBD can both improve performance. The performance improvement is most significant in the asymmetric tours.

In 3D Pareto space (the axes representing ranking for the three classes of problems) for the A-Teams, the A-Teams lie in the following layers: {QBD + 10*TLD, QBD + CLD2, QBD + 10*CLD2} dominate {QBD, QBD + CBD1} dominate {QBD + 10*CBD1} dominate {QBD + TLD}.

7.3. Summary

A combination of a fast but fragile construction agent with a slow but robust construction agent was tested. The combination provides a trade-off between speed and robustness. The resulting combination has the robustness of the slower agent, but is closer in speed to the fast agent.

Several implementations of destruction policies were tested, as were various scheduling frequencies. Two conclusions can be drawn from these experiments. The first is that implementation of destruction policies is very critical to performance. For example, adding CBD destroyers using variable sized clusters resulted in worse scores for the A-Teams, but adding CBD destroyers using fixed size clusters improved the scores. The second conclusion is that destroyer scheduling might be very important in contrast to constructor scheduling. While the A-Team with the high-frequency scheduled TLD destroyer was on the Pareto frontier and scored better than the benchmark A-Team on all three classes of problems, the A-Team with the low-frequency scheduled TLD destroyer was dominated by all the other A-Teams in all classes of problems.

8. CONCLUSIONS

This thesis extends the state-of-the-art in A-Teams in several areas. These include: a taxonomy of agent-based systems that relates A-Teams to other agent-based systems; a generative grammar for A-Teams; the use of modular-by-constraint agents to solve multi-criterion problems using A-Teams; the identification of human-computer collaboration as a promising application of A-Teams; and the introduction of new destruction policies for A-Teams. I will now describe the main contributions of each chapter in this thesis and identify avenues for future exploration:

8.1. A Critical Survey of Agent-Based Systems

A large amount of the literature on the subject is misleading and confusing. The chapter on agent-based systems identified the key characteristics of agents and relates them to the ideas presented in the literature. It also provided a taxonomy of agent-based systems. This taxonomy was used to compare a representative set of such systems.

8.2. A Generative Grammar for A-Teams

A generative grammar for A-Teams was designed in this thesis. The grammar provides a formal definition of A-Team related concepts, and a set of rules that guarantees the construction of valid A-Teams from given repositories of components. This grammar brings us a step closer to the goal of automatic A-Team construction. Several other steps still need to be made. The next steps include the implementation of the grammar into a compiler that can take heuristics that are provided as code and data structures for solutions and convert them into agents and memories and the construction of a computer tool to generate A-Team dataflows from repositories of agents.

8.3. Modular Optimization

Complex problems can be composed of problem primitives consisting of subsets of objectives and constraints. This thesis showed how specialized agents for these primitives can be combined in a modular fashion to construct task specific solvers for the larger problems. Two approaches for constructing such modular optimizers were presented: the ad-hoc approach and the transformation approach where a good monolithic algorithm is used as the framework for the modular method.

A secondary contribution of the chapter on modular optimization is the development of a tool to solve 3D spatial layout problems. The tool finds sets of Pareto optimal solutions and allows the user to view these solutions to determine trade-offs and select the most acceptable ones. The tool also facilitates intervention by the user who can guide the optimization into regions of the decision space where the trade-offs are more acceptable, as well as to check and eliminate solutions with undesirable, unquantifiable characteristics such as aesthetic requirements.

8.4. Human-Computer Collaboration

The main contribution of the chapter on human computer collaboration was to show that humans and computers can be combined in A-Teams to solve problems better than either can alone. An important advantage to the A-Team approach is that the human and computer agents can work at their own speeds without holding each other up. The asynchronous interaction of human and computers appears to be better than the traditional, alternating, synchronous interaction. A list of successful strategies for human agents in A-Teams was also developed.

Secondary contributions of this chapter are the experimental results on diversity, work cycles and scheduling. There has been little work in scheduling of agents, and this chapter showed that at least in the experiments conducted there, the frequency of scheduling did not play a very important role in scheduling. This indicates that there is little benefit to be gained from small speedups of agents. The experiments showed that it is difficult to predict exactly which agents will contribute in an A-Team and that it is better to add all available agents that are likely to help.

Human-Computer collaboration has been shown to be very promising in 2D layout problems. This leads to the belief that such collaboration can be applied to 3D layout problems too. Other domains where such collaboration could be useful remain to be identified and explored. This demonstration also opens up a whole field of issues that need to be addressed such as: the identification of traits that make a person more likely to perform well in such collaboration with computers; the design of the human computer interfaces for such collaboration; the simultaneous participation of multiple human agents; and the use of humans to work on unquantifiable aspects of the problem.

8.5. Diversity of Construction and Destruction Agents

This chapter showed that diversity of skills is beneficial in both construction as well as destruction agents in A-Teams. Construction agents can be combined to support each others weaknesses and build more robust A-Teams. Destroyers in A-Teams have so far used a quality-based policy for eliminating solutions. Global optimization techniques such as clustering and tabu search suggest alternative destruction policies. This thesis showed how destroyers based on the ideas from these techniques could be used along with quality-based destruction.

It is important to design the destroyers carefully, and this chapter barely skims the surface of alternative destruction policies. For example, the research in Tabu Search indicates that tabu list size is an important factor in performance. These alternatives remain to be explored.

GLOSSARY

ABS: Agent-Based System

ACL: Agent Communication Language

ACO: Ant Colony Optimization

AI: Artificial Intelligence

AIS: Agent Input Space

AOP: Agent Oriented Programming

AOS: Agent Output Space

AS: Ant System

B&B: Branch and Bound

BDI: Beliefs, Desires and Intentions

BFGS: Broyden-Fletcher-Goldfarb-Shanno formula used in SQP

CSCW: Computer Supported Collaborative Work

DAI: Distributed AI

DS: Decision Space

EA: Evolutionary Algorithms

EE: Electrical Engineering

GA: Genetic Algorithms

HCC: Human Computer Cooperation

HCI: Human Computer Interfaces

IE: Industrial Engineering

iff: if and only if

KIF: Knowledge Interchange Format

KQML: Knowledge Query and Manipulation Language

LP: Linear Program(ming)

ME: Mechanical Engineering

MS: Memory Space

NLP: Nonlinear Programming - a class of optimization problems consisting of a (set of) nonlinear objectives and a set of nonlinear (equality and inequality) constraints.

OIS: Operator Input Space

OOP: Object Oriented Programming

OOS: Operator Output Space

OR: Operations Research

PAI: Parallel AI

PBIL: Population Based Incremental Learning

QP: Quadratic Program(ming) - a class of optimization problems with a quadratic objective function and set of linear (equality and/or inequality) constraints. See also: SQP, NLP.

RMS: Random Multi Start

SA: Simulated Annealing.

SIS: Selector Input Space

SOS: Selector Output Space

SQP: Sequential Quadratic Programming - a method of solving NLP problems by solving a series of QP approximations to the NLP problem. See also: QP, NLP.

s.t.: such that

TL: Tabu List

TS: Tabu Search

TSP: Travelling Salesperson Problem.

VLSI: Very Large Scale Integration

APPENDIX

This appendix provides the NLP problems used in this thesis. Problems with numbers starting with HS have been scanned from [HS81]. The rest have been scanned from [Ramesh94].

Problem HS106

OBJECTIVE FUNCTION:

$$f(x) = x_1 + x_2 + x_3$$

CONSTRAINTS:

$$1 - .0025(x_4 + x_6) \geq 0$$

$$1 - .0025(x_5 + x_7 - x_4) \geq 0$$

$$1 - .01(x_8 - x_5) \geq 0$$

$$x_1x_6 - 833.33252x_4 - 100x_1 + 83333.333 \geq 0$$

$$x_2x_7 - 1250x_5 - x_2x_4 + 1250x_4 \geq 0$$

$$x_3x_8 - 1250000 - x_3x_5 + 2500x_5 \geq 0$$

$$100 \leq x_1 \leq 10000$$

$$1000 \leq x_i \leq 10000 \quad , \quad i=2,3$$

$$10 \leq x_i \leq 1000 \quad , \quad i=4,\ldots,8$$

START: $x_0 = (5000,5000,5000,200,350,150,225,425)$

Problem HS100

OBJECTIVE FUNCTION:

$$f(x) = (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 + 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7$$

CONSTRAINTS:

$$127 - 2x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 \geq 0$$

$$282 - 7x_1 - 3x_2 - 10x_3^2 - x_6 + x_7 \geq 0$$

$$196 - 23x_1 - x_2^2 - 6x_6^2 + 8x_7 \geq 0$$

$$-4x_1^2 - x_2^2 + 3x_1x_2 - 2x_3^2 - 5x_6 + 11x_7 \geq 0$$

START: $x_0 = (1, 2, 0, 4, 0, 1, 1)$

Problem HS80

OBJECTIVE FUNCTION:

$$f(x) = \exp(x_1x_2x_3x_4x_5)$$

CONSTRAINTS:

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 10 = 0$$

$$x_2x_3 - 5x_4x_5 = 0$$

$$x_1^3 + x_2^3 + 1 = 0$$

$$-2.3 \le x_i \le 2.3 \quad , \quad i=1,2$$

$$-3.2 \le x_i \le 3.2 \quad , \quad i=3,4,5$$

START: x_0 = (-2 , 2 , 2 , -1 , -1) (not feasible)

Problem HS113

OBJECTIVE FUNCTION:

$$f(x) = x_1^2 + x_2^2 + x_1x_2 - 14x_1 - 16x_2 + (x_3 - 10)^2 + 4(x_4 - 5)^2 + (x_5 - 3)^2 + 2(x_6 - 1)^2 + 5x_7^2 + 7(x_8 - 11)^2 + 2(x_9 - 10)^2 + (x_{10} - 7)^2 + 45$$

CONSTRAINTS:

$$105 - 4x_1 - 5x_2 + 3x_7 - 9x_8 \ge 0$$

$$-10x_1 + 8x_2 + 17x_7 - 2x_8 \ge 0$$

$$8x_1 - 2x_2 - 5x_9 + 2x_{10} + 12 \ge 0$$

$$-3(x_1 - 2)^2 - 4(x_2 - 3)^2 - 2x_3^2 + 7x_4 + 120 \ge 0$$

$$-5x_1^2 - 8x_2 - (x_3 - 6)^2 + 2x_4 + 40 \ge 0$$

$$-.5(x_1 - 8)^2 - 2(x_2 - 4)^2 - 3x_5^2 + x_6 + 30 \ge 0$$

$$-x_1^2 - 2(x_2 - 2)^2 + 2x_1x_2 - 14x_5 + 6x_6 \ge 0$$

$$3x_1 - 6x_2 - 12(x_9 - 8)^2 + 7x_{10} \ge 0$$

START: x_0 = (2,3,5,5,1,2,7,3,6,10) (feasible)

Problem HS117

OBJECTIVE FUNCTION:

$$f(x) = -\sum_{j=1}^{10} b_j x_j + \sum_{j=1}^{5}\sum_{k=1}^{5} c_{kj} x_{10+k} x_{10+j} + 2\sum_{j=1}^{5} d_j x_{10+j}^3$$

CONSTRAINTS:

$$2\sum_{k=1}^{5} c_{kj} x_{10+k} + 3d_j x_{10+j}^2 + a_j - \sum_{k=1}^{10} a_{kj} x_k \geq 0 \quad , \quad j=1,\ldots,5$$

$$0 \leq x_i \quad , \quad i=1,\ldots,15$$

$a_{ij}, b_j, c_{ij}, d_j, e_j$: cf. Appendix A

START: x_0 = .001(1,1,1,1,1,1,60000,1,1,1,1,1,1,1,1)

data for problem HS117

j	1	2	3	4	5
e_j	-15	-27	-36	-18	-12
c_{1j}	30	-20	-10	32	-10
c_{2j}	-20	39	-6	-31	32
c_{3j}	-10	-6	10	-6	-10
c_{4j}	32	-31	-6	39	-20
c_{5j}	-10	32	-10	-20	30
d_j	4	8	10	6	2
a_{1j}	-16	2	0	1	0
a_{2j}	0	-2	0	4	2
a_{3j}	-3.5	6	2	0	0
a_{4j}	0	-2	0	-4	-1
a_{5j}	0	-9	-2	1	-2.8
a_{6j}	2	0	-4	0	0
a_{7j}	-1	-1	-1	-1	-1
a_{8j}	-1	-2	-3	-2	-1
a_{9j}	1	2	3	4	5
a_{10j}	1	1	1	1	1
b_j	-40	-2	-.25	-4	-4
b_{5+j}	-1	-40	-60	5	1

Problem HS56

OBJECTIVE FUNCTION:

$$f(x) = -x_1x_2x_3$$

CONSTRAINTS:

$$x_1 - 4.2\sin^2 x_4 = 0$$

$$x_2 - 4.2\sin^2 x_5 = 0$$

$$x_3 - 4.2\sin^2 x_6 = 0$$

$$x_1 + 2x_2 + 2x_3 - 7.2\sin^2 x_7 = 0$$

START: $x_0 = .(1, 1, 1, a, a, a, b)$

Problem HS107

OBJECTIVE FUNCTION:

$$f(x) = 3000x_1 + 1000x_1^3 + 2000x_2 + 666.667x_2^3$$

CONSTRAINTS:

$$.4 - x_1 + 2cx_3^2 - x_5x_6(dy_1 + cy_2) - x_5x_7(dy_3 + cy_4) = 0$$

$$.4 - x_2 + 2cx_6^2 + x_5x_6(dy_1 - cy_2) + x_6x_7(dy_5 - cy_6) = 0$$

$$.8 + 2cx_7^2 + x_5x_7(dy_3 - cy_4) - x_6x_7(dy_5 + cy_6) - 0$$

$$.2 - x_3 + 2dx_5^2 + x_5x_6(cy_1 - dy_2) + x_5x_7(cy_3 - dy_4) = 0$$

$$.2 - x_4 + 2dx_6^2 - x_5x_6(cy_1 + dy_2) - x_6x_7(cy_3 + dy_4) = 0$$

$$-.337 + 2dx_7^2 - x_5x_7(cy_3 + dy_4) + x_6x_7(cy_3 - dy_4) = 0$$

$$0 \le x_i, \ i=1,2 \quad , \quad .90909 \le x_i \le 1.0909, \ i=5,6,7$$

$$y_1 = \sin x_8 \ , \ y_2 = \cos x_8 \ , \ y_3 = \sin x_9$$

$$y_4 = \cos x_9 \ , \ y_5 = \sin(x_8 - x_9) \ , \ y_6 = \cos(x_8 - x_9)$$

$$c = (48.4/50.176)\sin.25 \ , \ d = (48.4/50.176)\cos.25$$

START: $x_0 = (.8, .8, .2, .2, 1.0454, 1.0454, 0, 0)$

Problem HS85

OBJECTIVE FUNCTION:

$$f(x) = -5.843E\text{-}7y_{17}(x) + 1.17E\text{-}4y_{14}(x) + 2.358E\text{-}5y_{13}(x) + 1.502E\text{-}6y_{16}(x) + .0321y_{12}(x) + .00423y_5(x) + 1.E\text{-}4c_{15}(x)/c_{16}(x) + 37.48y_2(x)/c_{12}(x) - .1365$$

CONSTRAINTS:

$1.5x_2 - x_3 \geq 0$

$y_1(x) - 213.1 \geq 0$

$405.23 - y_1(x) \geq 0$

$y_{j-2}(x) - a_{j-2} \geq 0$, $j=4,\ldots,19$

$b_{j-18} - y_{j-18}(x) \geq 0$, $j=20,\ldots,35$

$y_4(x) - .28/.72y_5(x) \geq 0$

$21 - 3496y_2(x)/c_{12}(x) \geq 0$

$62212/c_{17}(x) - 110.6 - y_1(x) \geq 0$

$704.4148 \leq x_1 \leq 906.3855$

$68.6 \leq x_2 \leq 288.88$

$0 \leq x_3 \leq 134.75$

$193 \leq x_4 \leq 287.0966$

$25 \leq x_5 \leq 84.1988$

$y_j(x)$, $c_j(x)$, a_j , b_j : cf. Appendix A

START: $x_0 = (900 , 80 , 115 , 267 , 27)$

data for problem HS85

Let $y_i = y_i(x)$, $c_i = c_i(x)$,

$y_1 = x_2 + x_3 + 41.6$

$c_1 = .024x_4 - 4.62$

$y_2 = 12.5/c_1 + 12$

$c_2 = .0003535x_1^2 + .5311x_1 + .08705y_2x_1$

$c_3 = .052x_1 + 78 + .002377y_2x_1$

$y_3 = c_2/c_3$

$y_4 = 19y_3$

$c_4 = .04782(x_1 - y_3) + .1956(x_1 - y_3)^2/x_2 + .6376y_4 + 1.594y_3$

$c_5 = 100x_2$

$c_6 = x_1 - y_3 - y_4$

$c_7 = .95 - c_4/c_5$

$y_5 = c_6c_7$

$y_6 = x_1 - y_5 - y_4 - y_3$

$c_8 = (y_5 + y_4).995$

$y_7 = c_8/y_1$

$c_9 = 3798$

$y_8 = y_7 - .0663y_7/y_8 - .3153$

$y_9 = 96.82/c_9 + .321y_1$

$y_{10} = 1.29y_5 + 1.258y_4 + 2.29y_3 + 1.71y_6$

$y_{11} = 1.71x_1 - .452y_4 + .58y_3$

$c_{10} = 12.3/752.3$

$c_{11} = (1.75y_2)(.995x_1)$

$c_{12} = .995y_{10} + 1998$

$y_{12} = c_{10}x_1 + c_{11}/c_{12}$

$y_{13} = c_{12} - 1.75y_2$

$y_{14} = 3623 + 64.4x_2 + 58.4x_3 + 146312/(y_9 + x_5)$

$c_{13} = .995y_{10} + 60.8x_2 + 48x_4 - .1121y_{14} - 5095$

$y_{15} = y_{13}/c_{13}$

$y_{16} = 148000 - 331000y_{15} + 40y_{13} - 61y_{15}y_{13}$

$c_{14} = 2324y_{10} - 28740000y_2$

$y_{17} = 14130000 - 1328y_{10} - 531y_{11} + c_{14}/c_{12}$

$c_{15} = y_{13}/y_{15} - y_{13}/.52$

$c_{16} = 1.104 - .72y_{15}$

$c_{17} = y_9 + x_5$

i	a_i	b_i
2	17.505	1053.6667
3	11.275	35.03
4	214.228	665.585
5	7.458	584.463
6	.961	265.916
7	1.612	7.046
8	.146	.222
9	107.99	273.366
10	922.693	1286.105
11	926.832	1444.046
12	18.766	537.141
13	1072.163	3247.039
14	8961.448	26844.086
15	.063	.386
16	71084.33	140000
17	2802713	12146108

Problem HS67

OBJECTIVE FUNCTION:

$$f(x) = -(.063y_2(x)y_5(x) - 5.04x_1 - 3.36y_3(x) - .035x_2 - 10x_3)$$

$y_i(x)$: cf. Appendix A

CONSTRAINTS:

$$y_{i+1}(x) - a_i \geq 0 \quad , \quad i=1,\ldots,7$$

$$a_i - y_{i+8}(x) \geq 0 \quad , \quad i=8,\ldots,14$$

$$1.E{-}5 \leq x_1 \leq 2.E3$$

$$1.E{-}5 \leq x_2 \leq 1.6E4$$

$$1.E{-}5 \leq x_3 \leq 1.2E2$$

a_i : cf. Appendix A

START: x_0 = (1745 , 12000 , 110) (feasible)

data for problem HS67

Let $y_i = y_i(x)$ - the functions are described by a subprogram:

$y_2 = 1.6x_1$
10 $y_3 = 1.22y_2 - x_3$
$y_6 = (x_2 + y_3)/x_1$
$y_{2c} = .01x_1(112 + 13.167y_6 - .6667y_6^2)$
if $|y_{2c} - y_2| \leq .001$ goto 30 else goto 20
20 $y_2 = y_{2c}$
goto 10
30 $y_4 = 93$
40 $y_5 = 86.35 + 1.098y_6 - .038y_6^2 + .325(y_4 - 89)$
$y_8 = 3y_5 - 133$
$y_7 = 35.82 - .222y_8$
$y_{4c} = 98000x_3/(y_2y_7 + 1000x_3)$
if $|y_{4c} - y_4| \leq .001$ goto 60 else goto 50
50 $y_4 = y_{4c}$
goto 40
60 stop

i	a_i	i	a_i
1	0	8	5000
2	0	9	2000
3	85	10	93
4	90	11	95
5	3	12	12
6	.01	13	4
7	145	14	162

Problem He6

$$\text{Min} \quad -(25(x_1-2)^2+(x_2-2)^2+(x_3-1)^2+(x_4-4)^2+(x_5-1)^2+(x_6-4)^2)$$

$$\text{s.t.} \quad x_1, x_2 \geq 0;\ 1 \leq x_3 \leq 5;\ 0 \leq x_4 \leq 6;\ 1 \leq x_5 \leq 5;\ 0 \leq x_6 \leq 10;\ 2 \leq x_1+x_2 \leq 6;\ -x_1+x_2 \leq 2;$$

$$x_1-3x_2 \leq 2;\ 4 \leq (x_3-3)^2+x_4;\ 4 \leq (x_5-3)^2+x_6$$

There is a global minimum at $x^* = (5, 1, 5, 0, 5, 10)$ with value -310. There are about twenty local minima.

Problem Gr10, Gr15, Gr20

$$\text{Min} \sum_{i=1}^{N} \frac{x_i^2}{d} - \prod_{i=1}^{N} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

$$\text{s.t.} \quad -600 \leq x_i \leq 600,\ i = 1, 2, \ldots, N; \quad \sum_{i=1}^{N} x_i^2 \leq 1800^2$$

For Gr-10, d = 4000, N = 10
For Gr-15, d = 80,000, N = 15
For Gr-20, d = 800,000, N = 20

All three of these problems have a global minimum at the origin with value zero. There are several thousand local minima.

Problem D1

$$\text{Min} \sum_{i=1}^{20} \cos^2\left(\frac{\Pi}{2}\left(\frac{x_i}{(i \bmod 5)+1}\right)\right)$$

$$\text{s.t.} \quad (x_{i+1}-x_i)^2 \geq 1,\ \sin^2\left(\frac{\Pi}{2}\left(\frac{x_i}{(i \bmod 5)+1}\right)\right) \geq 0.95,\ i=1,2,\ldots,20$$

$x_i^* = \pm n\ ((i \bmod 5)+1)$, n odd. $\Rightarrow X^*$ is any vector composed of combinations of $x_i^* \leq 20$.

X^* is thus a combination of the subsets {2,3,4,5,1}, {6,9,12,15,3}, {10,15,20,25,5} etc.

Lots of minima with zero as the global optimal value, each minimum belonging to a distinct disconnected feasible region. Random starting points were generated within the hypercube centered on the origin and having edges of length 10 units.$(0 \leq x_i \leq 10)$

Problem D2

$$\text{Min} \sum_{i=1}^{20} \left(\frac{x_i^2}{5^4} + \cos^2\left(\frac{\Pi}{2}\left(\frac{x_i}{(i \bmod 5)+1}\right)\right)\right)$$

$$\text{s.t.} \quad \sin^2\left(\frac{\Pi}{2}\left(\frac{x_i}{(i \bmod 5)+1}\right)\right) \geq 0.95,\ x_i^3 \geq 0.9^3,\ x_i^3 \leq 5.1^3,\ i = 1,2,\ldots,20$$

$$x_i^3 \leq 1.5^3,\ i=5n,\ n=1,2$$

$$x_i^3 \leq 4^3,\ i=5n,\ n=3$$

There are 6 disconnected feasible regions, each with a unique minimum. Random starting points were generated within hypercube $0 \leq x_i \leq 6$.

REFERENCES

[Agha86] Gul A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge MA, 1986.

[AL97] E. Aarts, J. K. Lenstra (editors), *Local Search in Combinatorial Optimization*, John Wiley & Sons, New York, NY, 1997.

[ALLP97] E. H. L. Aarts, P. J. M. van Laarhoven, C. L. Liu, Peichen Pan, "VLSI layout synthesis", in [AL97] Ch 12, 1997.

[AT96] P. Avila-Abascal and S. N. Talukdar, "Cooperative Algorithms and Abductive Causal Networks for the Automatic Generation of Intelligent Substation Alarm Processors", *Proceedings of ISCAS-96*.

[Austin62] John L. Austin, *How to do things with words*, Oxford University Press, Oxford, UK, 1962.

[Aziz91] N. M. Aziz, "A computer-aided box stacking model for truck transport and pallets," *Computers in Industry*, v17, 1991, pp. 1-8.

[Bannon97] L. J. Bannon "Problems in human-machine interaction and communication," Design of Computing Systems: Cognitive Considerations. *Proceedings of the Seventh International Conference on Human-Computer Interaction* (HCI International '97) San Francisco, CA, USA; 24-29 Aug. 1997, Elsevier, Amsterdam, Netherlands, pp. 47-50.

[BAT97] L. Baerentzen, P. Avila, and S. Talukdar, "Learning Network Design for Asynchronous Teams," *Multi-Agent Rationality*, Lecture Notes in Artificial Intelligence, Springer-Verlag, vol. 1237, pp. 177-196, May 1997.

[Bates92] Joseph Bates. "Virtual Reality, Art and Entertainment," *Presence: The Journal of Teleoperators and Virtual Environment*s, v. 1 no. 1, MIT Press, Winter 1992, pp. 133-138.

[Brooks91a] Rodney A. Brooks, "New Approaches to Robotics," *Science*, v. 253, Sept, 1991, pp. 1227-1232,

[Brooks91b] Rodney A. Brooks, "Intelligence without representation," *Artificial Intelligence Journal*, Sept, 1991, v. 47, pp. 139-159.

[BT94] Roberto Battiti and Giampietro Tecchiolli, "The Reactive Tabu Search," *ORSA Journal on Computing*, v.6 no.2, 1994.

[Burich87] Misha R. Burich, “Design of Module Generators and Silicon Compilers”, in [MSA87], pp. 403-437.

[Chen92] C.L. Chen, “Bayesian Nets and A-Teams for Power System Fault Diagnosis,” Ph. D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Chorafas98] Dimitris N. Chorafas, *Agent Technology Handbook*, McGraw Hill, NY, 1998,

[CN98] Wolfram Conen and Gustaf Neumann (Eds.), *Coordination Technology for Collaborative Applications: Organizations, Processes, and Agents*, Springer-Verlag, Berlin, Lecture Notes in Computer Science, no. 1364, 1998.

[Corkill91] Daniel D. Corkill, “Blackboard Systems,” *AI Expert*, v. 6, no. 9, Sept, 1991, pp. 40-47.

[CTS93] S. Y. Chen, S. N. Talukdar, N. M. Sadeh, “Job-Shop-Scheduling by a Team of Asynchronous Agents,” *IJCAI-93 Workshop on Knowledge-Based Production, Scheduling and Control*, Chambery, France, 1993.

[Davis91] L. Davis (editor), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[DD92] K. A. Dowsland, W. B. Dowsland, “Packing Problems,” *European Journal of Operational Research*, v.56, 1992, pp 2-14.

[deSouza93] P. de Souza, “Asynchronous Organizations for Multi-Algorithm Problems,” Ph. D. dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1993.

[DG96] Marco Dorigo and Luca Maria Gambardella, “Ant Colony System: A Cooperative Learning Approach to the Travelling Salesman Problem,” Technical Report TR/IRIDIA/1996-5, Universite’ Libre de Bruxelles, Belgium, 1996.

[DM95] K.M. Daniels and V.J. Milenkovic, “Multiple Translational Containment: Approximate and Exact Algorithms”, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms(SODA)*, January 22-24, 1995, pp. 205-214.

[DM97] K.M. Daniels and V.J. Milenkovic, “Multiple Translational Containment. Part I: An Approximate Algorithm”, *Algorithmica: special issue on Computational Geometry in Manufacturing*, v. 19, 1997, pp. 148-182.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni, “The Ant System: Optimization by a colony of cooperating agents,” *IEEE Transaction on Systems, Man and Cybernetics*, Part-B, v. 26, no. 1, 1996, pp. 1-13.

[Dowsland93] K. Dowsland, “Some Experiments with Simulated Annealing Techniques for Packing problems,” *European Journal of Operational Research*, v68, 1993, pp. 389-399.

[DS83] Randall Davis and Reid G. Smith, “Negotiation as a metaphor for distributed problem solving,” *Artificial Intelligence*, v. 20, 1983, pp. 63-109.

[Dyckhoff90] H. Dyckhoff, “A typology of cutting and packing problems,” *European Journal of Operational Research*, v.44, 1990, pp. 145-159.

[Ellis98] Clarence A. Ellis, “A Framework and Mathematical Model for Collaboratoin Technology,” in [CN98], pp. 161-176.

[FG96] Stan Franklin and Art Graesser, “Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents,” Technical Report, Institute for Intelligent Systems, U. of Memphis, March, 1996.

[Flemming87] U. Flemming, “More than the sum of parts: the grammar of Queen Anne houses,” *Environment and Planning B: Planning and Design*, v. 14, 1987, pp. 323-350.

[Foner93] L. Foner, “What’s an Agent, Anyway? A Sociological Case Study,” Agents Memo 93-01, MIT Media Lab, 1993.

[FP90] C.A. Floudas and P.M. Pardalos, *A Collection of Test Problems for Constrained Global Optimization Problems*, Springer-Verlag, 1990.

[GAABCGOOPSW96] Don Gilbert, Manny Aparicio, Betty Atkinson, Steve Brad, Joe Ciccarino, Benjamin Grosof, Pat O’Connor, Damian Osisek, Steve Pritko, Rick Spagna, and Les Wilson, “The Role of Intelligent Agents in the Information Infrastructure,” Technical Report, IBM Corporation, Research Triangle Park, NC, 1996.

[GHSTM96] S. R. Gorti, S Humair, R. D. Sriram, S. Talukdar, S. Murthy, “Solving Constraint Satisfaction Problems Using A-Teams,” *AI-EDAM*, v. 10, 1996, pp. 1-19.

[GK94] Michael R. Genesereth and Steven P. Ketchpel, “Software Agents,” *Communications of the ACM*, v. 37, no. 7, 1994, pp. 48-53.

[Glover89] F. Glover, “Tabu Search-Parts I and II,” *ORSA Journal of Computing*, Vol. 1. No. 3, Summer 1989 and Vol. 2, No. 1, Winter 1990.

[Glover94] F. Glover, “Tabu search for nonlinear and parametric optimization (with links to genetic algorithms),” *Discrete Applied Mathematics*, v. 49, 1994, pp. 231-255.

[GMM90] H. Gehring, K. Menschner, M. Meyer, “A Computer Based Heuristic for Packing Pooled Shipment Containers,” *European Journal of Operational Research*, v44, 1990, pp. 277-288.

[GMW81] P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization*, Academic Press, San Diego, 1981.

[Goldberg89] David E. Goldberg, *Genetic Algorithms: in Search, Optimization and Machine Learning*, Addison-Wesley Publishing Company, Inc., Reading MA, 1989.

[Haddadi95] Afsaneh Haddadi, *Communication and Cooperation in Agent System: A Pragmatic Theory*, Lecture Notes in Artificial Intelligence 1056, Springer-Verlag, Berlin Heidelberg, 1995.

[Harada97] Mikako Harada, “Discrete / continuous design exploration by direct manipulation,” Ph.D. Dissertation, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA, 1997.

[HCK95] Colin G. Harrison, David M. Chess, Aaron Kershenbaum, “Mobile Agents: Are they a good idea?,” Technical Report, IBM Research Division, T.J.Watson Research Center, NY, March, 1995,

[Hermans96] Bjorn Hermans, Intelligent Software Agents on the Internet: an inventory of currently offered functionality in the information society & a prediction of (near-)future developments, Thesis, Tilburg University, Tilburg, The Netherlands, July 9, 1996. (http://www.hermans.org/agents)

[Hiebeler94] David Hiebeler, “The Swarm Simulation System and Individual-based Modeling,” *Proceedings of Decision Support 2001: Advanced Technology for Natural Resource Management*, Toronto, 1994.

[Hinzman95] B. Hinzman, “The Personal Factory,” *Rapid Prototyping Journal: Internet conference. Dec. 1995. ISSN 1355 2546*. http://www.mcb.co.uk/services/conferen/dec95/rapidpd/hinzmann/backgnd7.htm

[HK70] M. Held and R.M. Karp, “The Traveling-Salesman Problem and Minimum Spanning Trees,” *Operations Research*, Vol. 18, 1138-1162, 1970.

[HS81] W. Hock and K. Schittkowski, *Test Examples for Nonlinear Programming Codes*, Springer-Verlag, 1981.

[HT90] R. W. Haessler, F. B. Talbot “Load Planning for shipments of low density products,”, *European Journal of Operational Research*, v. 44, 1990, pp. 289-299.

[IBKRW97] I. T. Ikonen, W. E. Biles, A. Kumar, R. K. Ragade, J. C. Wissel, “A Genetic Algorithm for Packing Three-Dimensional Non-Convex Objects Having Cavities and Holes,” *Proceedings of the 7th International Conference on Genetic Algorithms*, Michigan State University, East Lansing, MI, 19-23 July, 1997. http://www.spd.louisville.edu/~itikon01

[JGAAPR96] Peter C. Janca (with Don Gilbert, Manny Aparici, Betty Atkinson, Steve Pritko, and Joe Rusnak), “Practical Design of Intelligent Agent Systems,” Technical Report, IBM Corporation, Research Triangle Park, NC, June, 1996.

[KCR96] A. Kolli, J. Cagan, and R. Rutenbar, “Packing of Generic, Three-Dimensional Components based on Multi-Resolution Modeling,” *Proceedings of the 1996 ASME Design Engineering Technical Conferences*, 96-DETC/DAC-1479, Irvine, California, 1996.

[KG91] J. J. Kim and D. C. Gossard, “Reasoning on the Location of Components for Assembly Packaging,” *Transactions of the ASME, Journal of Mechanical Design*, v. 113, December 1991, pp. 402-407.

[KGC83] S. Kirkpatrick, C.D. Gelatt, and M.P. Cecchi, “Optimization by Simulated Annealing,” *Science*, Vol. 220, Number 4598, May, 1983.

[Klein98] Mark Klein, “Coordination Science: Challenges and Directions,” in [CN98], pp. 161-176.

[KMK91] T. Kawakami, M. Minagawa, Y Kakazu, “Auto Tuning of 3-D Packing Rules Using Genetic Algorithms,” *IEEE/RSJ International Workshop in Intelligent Robots and Systems IROS ‘91*, Osaka, Japan, pp. 1319-1324.

[KR88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, ANSI-C Version, 2nd edition, Prentice-Hall Inc, NJ, 1988.

[LD96] Moises Lejter and Thomas Dean, “A Framework for the Development of Multiagent Architectures,” *IEEE Expert/Intelligent Systems & Their Applications*, v. 11, no. 6, Dec. 1996, pp. 47-59.

[Lengauer90] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, Chichester (England), 1990.

[LGTH97] J. A. Lukin, A. P. Gove, S. N. Talukdar and C. Ho, “Automated Probabilistic Method for Assigning Backbone Resonances of (^{13}C, ^{15}N)-Labelled Proteins,” *Journal of Biomolecular NMR*, 9, 1997.

[LH98] Jan Eric Larsson, Barbara Hayes-Roth, “Guardian: An intelligent autonomous agent for medical monitoring and diagnosis,” *IEEE Expert/Intelligent Systems & Their Applications*, v. 13, no. 1, Jan. 1998, pp. 58-64.

[LK73] S. Lin and B.W. Kernighan, “An Effective Heuristic Algorithm for the Traveling-Salesman Problem,” *Operations Research*, Vol. 21, 1973, pp. 498-516.

[LM96] M. C. Lin and D. N. Manocha (editors), *Applied computational geometry: Towards Geometric Engineering*, ACM Workshop on Applied Computational Geometry 1996, selected papers, Springer, NY, 1996.

[LMHM95] H. Lee, S. Murthy, W. Haider, D. Morse, “Primary Production Scheduling at Steel making Industries,” IBM report, 1995

[LTS98] Fu-Ren Lin, Gek Woo Tan, M.J. Shaw, “Modeling supply-chain networks by a multi-agent system,” P*roceedings of the Thirty-First Hawaii International Conference on System Sciences*, Kohala Coast, HI, USA, Jan. 1998.

[Maes94] Pattie Maes, “Agents that Reduce Work and Information Overload,” *Communications of the ACM*, July, 1994, v. 37, no. 7.

[Miller56] George A. Miller, “The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information,” Psychological Review, v. 63, no. 2, 1956, pp 81-97.

[MLF95] James Mayfield, Yannis Labrou, Tim Finin, “Evaluation of KQML as an Agent Communication Language,” *Intelligent Agents Vol II - Proceedings of the 195 Workshop on Agent Theories, Architectures, and Languages*, (M. Wooldridge, J.P. Muller and M. Tambe, editors), Lecture Notes in Artificial Intelligence, Springer Verlag, 1996, (http://www.cs.umbc.edu/lait/papers/kqml-eval.ps).

[MRLA] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi, The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations, Technical Report, Santa Fe Institute,

[MSA87] G. De Micheli, A. Sangiovanni-Vincentelli and P. Antognetti (editors), D*esign Systems for VLSI Circuits*, Martinus Nijhoff Publishers, Dordrecht, The Netherlands, 1987.

[Murthy92] S. Murthy, “Synergy in Cooperating Agents: Designing Manipulators from Task Specifications,” Ph.D. dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Nii86] H. P. Nii, “Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, Parts I and II, *AI Magazine*, 7:2 and 7:3, 1986.

[NSM98] Kazushi Nishimoto, Yasuyuki Sumi, and Kenji Mase, Enhancementof Creative Aspects of a Daily Conversation with a Topic Development Agent, in [CN98], pp. 161-176.

[Nwana96] Hyacinth S Nwana, “Software Agents: An Overview,” *Knowledge Engineering Review*, Oct/Nov, 1996, v. 11, no. 3, pp. 205-244, (http://www.cs.umbc.edu/agents/introduction/ao.ps).

[OC94] Tim Oates and Paul R. Cohen, “Humans Plus Agents Maintain Schedules Better than Either Alone,” Computer Science Technical Report 94-03, Dept. of Computer Science, University of Massachusetts, 1994.

[OF93] J.F.C. Oliveira and J.A.S. Ferreira, “Algorithms for Nesting Problems,” *Applied Simulated Annealing* (R.V.V. Vidal, editor), Lecture Notes in Economics and Mathematical Systems, no. 396, Springer-Verlag, 1993, ch. 12, pp. 255-274.

[Petrie96] Charles J. Petrie, “Agent-Based Engineering, the Web, and Intelligence,” *IEEE Expert/Intelligent Systems & Their Applications*, Dec. 1996, v. 11, no. 6, pp. 24-29.

[PK97] Christiaan J.J. Paredis and Pradeep K. Khosla, “Task-Based Design of Manipulators: An Agent-Based Approach,” P*roceedings of ASME Design Engineering Technical Conference 1997*, Sacramento, California, September, 14-17, 1997, no. DETC97-DAC3853,

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numerical Recipes in C: The art of Scientific Computing* (second edition), Cambridge University Press, 1992.

[Quadrel91] R.W. Quadrel, “Asynchronous Design Environments: Architecture and Behavior,” Ph.D. dissertation, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA, 1991.

[Ramesh94] V.C. Ramesh, “Inertial Search and Asynchronous Decompositions,” Ph.D. dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1994.

[RC93] A. Repenning, and W. Citrin, “Agentsheets: applying grid-based spatial reasoning to human-computer interaction,” *Proceedings 1993 IEEE Symposium on Visual Language*, Bergen, Norway, 24-27 Aug. 1993,

[RWMTSAFAYHJ96] J. Rachlin, F. Wu, S. Murthy, S. Talukdar, M. Sturzenbecker, R. Akkiraju, R. Fuhrer, A. Aggarwal, J. Yeh, R. Henry and R. Jayaraman, “Forest View: A System For Integrated Scheduling In Complex Manufacturing Domains,” IBM report, 1996.

[SC93] S. Szykman and J. Cagan, “Automated Generation of Optimally Directed Three Dimensional Component Layouts,” *Advances In Design Automation*, DE-Vol. 65-1, 1993, pp. 527-537.

[SC95] S. Szykman and J. Cagan, “A Simulated Annealing-Based Approach to Three-Dimensional Component Packing,” *Transactions of the ASME, Journal of Mechanical Design*, v. 117, June 1995, pp. 308-314.

[SCF97] S. Sandurkar, W. Chen, and G.W. Fadel, “Gaprus: Three-Dimensional Pipe Routing using Genetic Algorithms and Tessellated Objects,” *Proceedings of 1997 ASME Design Engineering Technical Conferences*, DETC97/DAC-3982, Sacramento, California, 1997.

[SDPWZ96] Katia Sycara, Keith Decker, Anandeep Pannu, Mike Williamson, and Dajun Zeng, “Distributed Intelligent Agents,” *IEEE Expert/Intelligent Systems & Their Applications*, Dec. 1996, v. 11, no. 6, pp. 36-46.

[Shahroudi97] K.E. Shahroudi, “Design by continuous collaboration between manual and automatic optimization,” Technical Report CWIS SEN-R9701, Centrum voor Wiskunde en Informatica (National Research Institute for Mathematics and Computer Science, Amsterdam, NL, Feb. 1997.

[Shoham94] Yoav Shoham, “Multi-Agent Research in the Knowbotics Group,” *Artificial social systems, 4th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW ‘92*, Lecture Notes in Artificial Intelligence 830, Springer Verlag, 1994.

[Singh94] Munindar P. Singh, *Multiagent Systems: A Theoretical Framework for Intentions, Know-How, and Communications*, Lecture Notes in Artificial Intelligence 799, Springer-Verlag, Berlin Heidelberg 1994.

[SPGT98] S. Sachdev, C.J.J. Paredis, S.K. Gupta, S.N. Talukdar, “3D Spatial Layouts Using A-Teams,” DETC98/DAC-5628, *Proceedings of ASME 24th Design Automation Conference*, Sep. 1998.

[ST91] P.S. de Souza and S.N. Talukdar, “Genetic Algorithms in Asynchronous Teams,” *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, Los Altos, CA, 1991.

[Talukdar98] S.N. Talukdar, “Autonomous Cyber Agents: Rules for Collaboration,” *Proceedings of the thirty-first Hawaii International Conference on System Sciences* (HICSS-31), Hawaii, January 1998.

[TBGD97] S. N. Talukdar, L. Baerentzen, A. Gove, P. de Souza, “Asynchronous Teams: Cooperation Schemes for Autonomous Agents”, *Journal of Heuristics*, 1997.

[TPG83] S. N. Talukdar, S. S. Pyo and T. Giras, “Asynchronous Procedures for Parallel Processing,” *IEEE Trans. on PAS*, Vol. PAS-102, NO 11, Nov. 1983.

[TPM83] S.N. Talukdar, S.S. Pyo, R. Mehrotra, *Designing Algorithms and Assignments for Distributed Processing*, Electric Power Research Institute, EL3317, Project 1764-3, Nov. 1983.

[TR93] S. N. Talukdar, V.C. Ramesh, “A Parallel Global Optimization Algorithm and its Application to the CCOPF Problem,” *Proceedings of the Power Industry Computer Applications Conference*, Phoenix, May, 1993.

[TSC97] S.N. Talukdar, S. Sachdev, E. Camponogara, “Collaboration among Asynchronous Autonomous Specialist Agents,” *Proceedings of the SPIE, Symposium on Intelligent Systems & Advanced Manufacturing*, Oct. 1997.

[Tsen95] C. K. Tsen, “Solving Train Scheduling Problems Using A-Teams,” Ph.D. dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1995.

[TZ89] Aimo Torn and Antanas Zilinskas, *Global Optimization*, Lecture Notes in Computer Science 350, Springer-Verlag, Berlin, 1989.

[WF94] J. R. Wodziak, G. M. Fadel “Packing and Optimizing the Center of Gravity Location using a Genetic Algorithm,” http://www.vr.clemson.edu/dmg/papers/Opt-papers.html, Design Methodology Group, Clemson University, Clemson SC.

[WJ95] Michael Wooldridge and Nicholas R. Jennings, “Intelligent Agents: Theory and Practice,” *Knowledge Engineering Review*, v. 10, no. 2, 1995, pp. 115-152,

[Prototype1] http://ltk.hut.fi/archives/rp-ml/0584.html

[Prototype2] http://nak.iis.u-tokyo.ac.jp/rpjp/stuts/index.html

[Neos] Neos Server (for solving problems over the web) - http://www.mcs.anl.gov/home/otc/Server/

[Agentsheets] http://www.cs.colorado.edu/~l3d/systems/agentsheets/

[Ontology] KSL Ontology Server, (http://www-ksl-svc.stanford.edu:5915/).