Abstract

Knowledge-based search is a set of concepts that can help software developers build better search systems. Until recently, developers often build sequential knowledge-based search systems (making use of only a single processor core) because sequential systems are easier to develop. For further speed improvements, developers could rely on new processors to run at a faster clock rate. However, with the recent trend in processor design, clock rates have remained stagnant while processors are moving towards having multiple processing cores. To make use of these additional processing capabilities, developers now need to *distribute* their search to the available processing cores.

Building distributed based search systems is a difficult and time consuming process because not only are distributed systems difficult to develop, but knowledge-based search systems are not readily able to be distributed. Fortunately, there are different distribution paradigms that provide guidelines as to how the search process can be distributed.

This thesis introduces DisSLib:ICA, a software library for building distributed knowledge-based search systems based on the improving on the competition approach paradigm. The main goal of DisSLib:ICA is to allow developers to build distributed search systems in the same manner, and with the same amount of effort, as it would normally take to build a sequential search system. It achieves this by handling the communication and multi-threading tasks along with providing developers a skeleton structure of a search system that can be extended to fit the developer’s concrete search problem. To evaluate DisSLib:ICA, we have built three search systems that solves different problems using the library. Our results have shown that the library allows developers to build distributed systems with approximately the same amount of effort as it would take to build a sequential system. In addition, our experiments show that by using the improving on the competition approach paradigm, the library produces synergistic speedups.
Acknowledgements

I am truly blessed to have the support and encouragement of many wonderful people throughout this entire journey.

I would like to thank my supervisor Dr. Jörg Denzinger for all the support and guidance that he has given me. This thesis would not have been possible without his guidance.

I would like to thank my parents Anh and Phuong for their encouragement and for providing me with all the resources that I required.

I would like to thank my wife Leina for all the support and encouragement she has given me and for putting up with me using the weekends that we were supposed to spend time together to work on this thesis.

Finally, I would like to thank my examination committee for their time and effort.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>ii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Goals</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Overview</td>
<td>5</td>
</tr>
<tr>
<td>2. Knowledge Based Search</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Formal Search Definitions</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 Search Model</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 Search Instance</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3 Search Process</td>
<td>10</td>
</tr>
<tr>
<td>2.1.4 Search Derivation</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Knowledge Based Search Paradigms</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Sets</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 And-Trees</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3 Or-Trees</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Programmatic Interface of Search</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1 Search Model</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Search Instance</td>
<td>23</td>
</tr>
<tr>
<td>2.3.3 Search Process</td>
<td>24</td>
</tr>
<tr>
<td>2.3.4 Search Derivation</td>
<td>26</td>
</tr>
<tr>
<td>2.3.5 Environment</td>
<td>28</td>
</tr>
<tr>
<td>3. Distributed Search Systems</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Basic Definitions</td>
<td>30</td>
</tr>
<tr>
<td>3.1.1 Search Agent</td>
<td>31</td>
</tr>
<tr>
<td>3.1.2 Distributed Search</td>
<td>32</td>
</tr>
<tr>
<td>3.1.3 Time Frame and search course protocol</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Distributed Search Paradigms</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1 Using a Common Search State</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2 Dividing the Problem into Sub-problems</td>
<td>38</td>
</tr>
<tr>
<td>3.2.3 Improving on the Competition Approach (ICA)</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Improving on the Competition Approach Paradigm: Formal Description</td>
<td>43</td>
</tr>
<tr>
<td>3.3.1 Communication Structure</td>
<td>43</td>
</tr>
<tr>
<td>3.3.2 Search Agent</td>
<td>44</td>
</tr>
<tr>
<td>3.3.3 Distributed Search System ICA</td>
<td>44</td>
</tr>
<tr>
<td>3.3.4 Time Frame and Search Course Protocol</td>
<td>45</td>
</tr>
<tr>
<td>3.4 Changes to the Programmatic Interface of ICA</td>
<td>45</td>
</tr>
<tr>
<td>4. System Components of an ICA System</td>
<td>50</td>
</tr>
<tr>
<td>4.1 Search Component</td>
<td>53</td>
</tr>
<tr>
<td>4.2 Configuration Component</td>
<td>58</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>This table shows the core search classes and the number of lines of code added during the conversion process.</td>
<td>93</td>
</tr>
<tr>
<td>6.2</td>
<td>(a) shows number of lines of code for the core search classes of the branch and bound system. (b) Compares the total lines of code for each sequential system.</td>
<td>96</td>
</tr>
<tr>
<td>6.3</td>
<td>The lines of code that is required for sharing information between the BB and GA agents.</td>
<td>99</td>
</tr>
<tr>
<td>6.4</td>
<td>The total lines of code to implement a GA sequential course scheduling system using DisSLib:ICA</td>
<td>106</td>
</tr>
<tr>
<td>6.5</td>
<td>The total lines of code to implement a BB sequential course scheduling system using DisSLib:ICA</td>
<td>110</td>
</tr>
<tr>
<td>6.6</td>
<td>The lines of code that is required for adding information sharing to the course scheduling system.</td>
<td>114</td>
</tr>
<tr>
<td>6.7</td>
<td>The total lines of code that are required for adapting distributed systems built in DisSLib:CC to DisSLib:ICA.</td>
<td>116</td>
</tr>
<tr>
<td>7.1</td>
<td>Hardware configuration of the virtual machine.</td>
<td>119</td>
</tr>
<tr>
<td>7.2</td>
<td>The average total soft constraint violations of the best solution produced by the original exam scheduling system versus the adapted DisSLib:ICA exam scheduling system running in standalone mode.</td>
<td>121</td>
</tr>
<tr>
<td>7.3</td>
<td>Comparisons of the quality of the solution between the sequential agents versus the distributed.</td>
<td>121</td>
</tr>
<tr>
<td>7.4</td>
<td>Comparisons of the quality of the solution between the sequential agents versus the distributed.</td>
<td>123</td>
</tr>
<tr>
<td>7.5</td>
<td>Comparisons between the time it took to find the same solution between sequential and distributed agents.</td>
<td>124</td>
</tr>
<tr>
<td>7.6</td>
<td>Comparisons in the time it took to find the optimal solution for two instances of the package delivery problem.</td>
<td>124</td>
</tr>
<tr>
<td>7.7</td>
<td>Comparisons in the time it took to find the optimal solution at different sharing intervals.</td>
<td>126</td>
</tr>
</tbody>
</table>
List of Figures and Illustrations

2.1 Class diagram showing the interface of the SearchState. The search state defines two methods: selectTransition and integrateResult. The selectTransition method has a search control as an argument to allow for multiple search controls to be used during a search run. Usually, selection of transitions is delegated to the search control but it can also be accomplished by the search state. Once a transition is processed, the transition result is integrated back into the search state through the integrateResult method. .......................... 22

2.2 The SearchResult is a data container that holds the result of the processing done by the search control. The data contained in the search result is dependent on the search problem and needs to contain all the necessary data in order for the search state to advance to another state. .......................... 23

2.3 The SearchTransition object holds information regarding how a state can transition to another state. Its implementation is empty because its internal representation is dependent on the search state and the concrete search problem. .......................... 23

2.4 The ProblemInstance object is part of the search instance component. It is responsible for creating the initial start state through the generateStartState method. .......................... 24

2.5 The SearchObserver is responsible for deciding whether a state is finished through the canContinue method. This method determines whether a specified search state is either a goal state or no further transitions are available. .......................... 24

2.6 The SearchControl object requires two methods to be implemented: getNextTransition and processTransition. The getNextTransition selects a transition based on the current search state that is passed in as an argument. Once the transition has been selected, processTransition method will be called to process transition, producing a search result that is to be integrated back into the search state. .......................... 26

2.7 The SearchProtocol interface consists of four methods that must be implemented to allow the search protocol to record all the necessary information of the search run. The recordSearchState, recordTransition, recordResult records the state, transitions, and results of the processing respectively. The recordSearchEnd method records a search state and then determines the solution by processing all the recorded states and transitions. .......................... 27

2.8 The environment object is responsible for the external data outside of the search model. It does not expose any methods because its design is dependent on the search system and the search problem at hand. .......................... 28

3.1 Agents A and B are homogenous agents. In this sample run A and B execute at the same time but no information is shared between the two agents. It takes A 4 transitions to arrive at the goal state, while B requires 6 transitions. .......................... 39
3.2 After each transition is processed, the search state is shared between the two agents. Having additional information allows each agent to select better transitions at each step of the search.

3.3 The **Information** interface represents the data that is sent between the agents. This interface is bare of methods and properties because it is dependent on the concrete problem. The information interface is further divided into the four types of information: **PositiveControlInformation**, **NegativeControlInformation**, **PositiveStateInformation**, and **NegativeStateInformation**. A search system does not need to support all types of information, only the ones that are supported by the agent.

3.4 The **SendReferee** looks at the current search state and control to determine whether or not new information should be sent to its respective agent. The information that is sent over should be information that it believes to be of benefit to the receiving agent.

3.5 The **SearchControl** is now responsible for handling information (both positive and negative state and control information) received from other agent’s send referee. In order to understand the information sent, the receive referee must first filter the information through the corresponding filter methods.

3.6 The programmatic interface of the **SearchState** has been modified from its sequential form to allow for integration of positive and negative information filtered by the **SearchControl**. Since the integration of positive and negative results needs to be handled differently, the integration is divided into two methods: **integratePositiveResult** and **integrateNegativeResult**.

4.1 The 6 components that make up DisSLib:ICA. The bars represent how much effort is required of the developer for each of the components.

4.2 A sequential search run is made up of a loop that consists of 4 main actions.

4.3 The send and receive referee actions can occur at any point during the search loop.

4.4 The communication channel of a three agent system.

4.5 A class diagram showing the relationships of the classes that make up the mail delivery system.

4.6 Class diagram showing the relationships between the classes that make up the event component.

5.1 Class diagram of the start agent.

6.1 Crossover and mutation operations in genetic algorithms.

7.1 A line chart showing the best solutions of each search agent at a specific point in time.

7.2

7.3
Chapter 1

Introduction

Since the introduction of the microprocessor software developers have taken for granted the fact that the clock rate of computer processors (CPU) increase with each new processor generation. Thus, with no work required on their end, software developers can expect their sequential software to run faster on the latest processor. Unfortunately, this “free lunch” is coming to an end. Processor designers have reached a limit when it comes to increasing the clock rate due to the growing energy consumption and excessive heat produced by high clock rate CPUs. Instead, for the foreseeable future, the trend is to produce highly energy efficient processors that contain multiple processing cores (multi-core processors) while maintaining the current clock rate.

Multi-core processors are faster than their single core counter-parts at the same clock rate because they can execute multiple programs in parallel, thus enabling each program to have full access to a CPU rather than share it. Unfortunately, sequential software programs (one that is designed to only run on a single CPU core) generally does not see much improvements running on multi-core processors since it cannot take advantage of the extra processing capabilities. To fully realize the benefits of multi-core processors, sequential software needs to be modified to be able to execute concurrently on different CPU cores.

In addition to being able to distribute work to the additional CPU cores, the rise of cloud computing has also given developers unprecedented access to computers. If a developer needs access to 20 computers he can easily do so by renting from cloud providers at an hourly rate. However, to fully capitalize on all these extra computers, the developer needs to convert his sequential system to a distributed system.

Unfortunately, modifying software to take advantage of the extra processing capabilities
is difficult because concurrent and network programming introduces extra implementation burden on developers (they need to deal with multiple threads of execution and the communication overhead required in distributed computing). However, we are now seeing industry and researchers trying to make this process easier by building software libraries (such as Parallel Task Library \cite{CJMT10}, MPJ-Express \cite{SM09}, and Hadoop \cite{Whi09}) that allow developers to build distributed software without requiring them to write multi-threaded and network code. While these libraries work great for certain types of applications others cannot easily make use of them. For example, web pages do not interact with each other so it is natural for web browsers to distribute the rendering process of each web page to separate processing cores. On the other hand, software applications that require information to be shared among all the processes are harder to distribute because they require information to be synchronized among the processes which can leave the system in a state of deadlock.

One type of application that is difficult to distribute is knowledge-based search. Knowledge-based search systems try to solve hard problems by using knowledge of the problem to pick through the decision possibilities. Since such a system does not know the correct path, it will need to decide what to do next based on the knowledge it has gathered so far. One naive way to distribute a knowledge-based search system is to divide up the solution space and distribute each part to each CPU core. However this would mean that the work the other cores would be wasted because an optimal solution can potentially exist in only one of the divided solution space parts. Also, since the information a search gathers along the way is crucial to guide the direction of the search, each processor core would operate with only parts of the whole information, thus potentially making decisions it would not have otherwise made. This poor decision making can potentially make the distributed search system run slower than the sequential system.

Fortunately, there are distributed search paradigms that provide developers with guidelines as to how to structure a search system for distribution. These paradigms require that
the search processes communicate with one another and are general enough to allow for most search problems to be distributed. Unfortunately, these distribution paradigms only act as a guideline and developers are still required to implement the system. This makes the transition from a sequential system to a distributed system quite difficult, as it involves the developer having to write multi-threaded code and implement communication between the search processes. To build support for communication, the developer needs to know about computer networking and deal with multi-threaded programming. The developer also needs to decide what information to communicate and when to communicate the information. This process is time consuming for developers who have many search systems or developers who wish to test different models as they are required to build a new system each time.

1.1 Goals

This thesis presents a distributed search library for the improving on the competition approach paradigm (DisSLib:ICA). By using DisSLib:ICA, developers can implement their distributed search systems based on the improving on the competition approach paradigm. The improving on the competition approach paradigm is a distributed paradigm that allows heterogeneous agents to cooperate with each other through the sharing of relevant information. The agents use this information to make better choices. DisSLib:ICA takes care of all the communication between the search processes and handles all the multi-threaded programming, thereby allowing the developer to focus on the issues related to the search problem. The library also provides utilities that allow developers to configure their systems and log relevant information related to the search.

DisSLib:ICA was inspired from DisSLib:CC ([Ken10]) and is designed based on the same definitions of search (discussed in Chapter 2) as DisSLib:CC. The end result is that both libraries share similar skeleton interfaces that allow developers to extend the library to their particular search problem. Where the DisSLib:CC and DisSLib:ICA differ is the paradigms
that each library supports. DisSLib:CC supports the central common central search state paradigm whereas DisSLib:ICA supports the improvement on the competition approach paradigm. The central common search state paradigm can only support homogeneous agents, that is, agents that have the same search model. Also, the results produced from the central common search state paradigm can at best be a linear speedup from the sequential system. The improving on the competition approach, on the other hand, allows for heterogeneous agents and synergistic speedups (speedup greater than linear). In addition, the two paradigms require different communication channels for the agents to communicate. As a result, DisSLib:ICA shares with DisSLib:CC only 5 skeleton interfaces that describes a sequential search system. The remaining components that make up the DisSLib:ICA library (such as the communication component) are different from DisSLib:CC.

The main goal of DisSLib:ICA is to allow developers to build distributed search systems based on the improving on the competition approach in the same manner and also with the same amount of effort as it would take to build a sequential system. It achieves this by providing developers with the structure of a search system through its programming interface and allows developers to build search systems as they would build a sequential system.

To evaluate DisSLib:ICA we have built three different search systems using DisSLib:ICA. With these systems we compared the effort it takes to a distributed system using DisSLib:ICA versus a sequential system. In addition, we have also evaluated the performance improvements of the distributed system over the sequential system. The result of our experiments showed that it takes a similar amount of effort to build a distributed search system using DisSLib:ICA as it normally would take to build a sequential search system. Our experiments also show that the improving on the competition approach paradigm allows systems built with DisSLib:ICA to produce synergistic results. That is, solutions produced by DisSLib:ICA can potentially be much better (in either the quality of the solution or the speed at which the system arrive at the solution) than the expected linear improvement.
1.2 Overview

This thesis is organized as follows:

Chapter 2 introduces the concept of knowledge based search and provides formal definitions of the components that make up knowledge-based search systems. The programmatic interfaces of the library that are derived from the formal definitions are also introduced.

Chapter 3 continues with formal definitions of the components that make up a distributed search system. It will also discuss three known distributed search paradigms while focusing on the improving on the competition approach paradigm. The programatic interfaces that are required for distribution are also described.

Chapter 4 takes an in-depth look at all the system components that make up DisSLib:ICA. It goes through the design of the library and all the components that a developer would need to know in order to use DisSLib:ICA. Chapter 5 shows how a developer can begin using DisSLib:ICA by looking at the components and the code that is required. Chapter 6 describes the three search systems that we use in our case studies. We also compare the effort that is required to build these distributed search systems against building a sequential system.

Chapter 7 reveals the performance gains of the distributed search systems built using the DisSLib:ICA we were able to achieve. Chapter 8 looks at other works relating to distributed search libraries. Finally, Chapter 9 concludes the thesis with a look at future improvements that can be made to DisSLib:ICA.
Chapter 2

Knowledge Based Search

Search is a critical function of our everyday life. It is ingrained into our behavior and we use it without much consideration for it. Similarly, most software programs also employ search in their programming. The search employed by these software programs can be simple (looking through a list of contacts to find a specific person) to complex (searching through medical records to find the most likely cause matching a patient’s symptoms). We often attribute these complex search programs as being intelligent; however, a computer is only capable of storing data and computing machine instructions. These operations are not what we normally consider intelligent. Instead, the intelligence lies in the programming, more specifically, the ability to search through very large amounts of data to find a solution.

Take for example the artificial intelligence in a chess program. For a software program to compete against human players at the game of chess, the program needs to calculate all the possible moves that it can make in a round. However, as chess is a game of strategy, planning for only one round is insufficient. To be successful, the chess program needs to compute all the possible moves along with all possible counter moves that the opponent can make to each of those moves, for not only the current round, but as many rounds as possible. Given the number of possible moves each piece on the board can perform and the possible number of responses an opponent can make, it shows how hard a problem like chess is to solve computationally. With the limited time available for the chess program to make a decision having a faster processor to perform more computations will not significantly impact the speed of the search. Instead of relying on processor improvements to speed up the search we can improve the search method by applying knowledge based search ([Pea81]).

Knowledge based search is a set of concepts that can help developers to build better
search systems. This improvement can be either in the form of finding a better solution or finding a solution faster. For most problems, if we do not have knowledge of the problem then it is very difficult to even know where to start. For example, suppose that we have misplaced our keys. With no knowledge of the keys (where they were last seen and who might have had it last) we would have to search through every location on the planet. With knowledge about the keys, however, we can make better decisions as to where to search thus limiting our search area. Knowing who might have seen the keys last allows us to use that person’s knowledge to help us find the keys. Having knowledge greatly reduces the number of places we have to search and speeds up the search process.

We can obtain knowledge through a few different methods. Knowledge can be gained through trial and error. As children, we knew nothing of the physical world we live in. We had to interact with it to get to know it. Touching a hot stove will burn our fingers but after that incident we know not to touch it while hot. Another way to gain knowledge is to have it taught to us. Much of our early life is spent in school having knowledge passed on to us from our teachers. It is through this type of learning that we receive most of the knowledge that we could not have learned through trial and error.

Search programs can gain knowledge through the same means. In a game of chess, the beginning and middle are where there are the most number of possible actions. To improve the search, developers can add knowledge by programming in known openings, positions, and counters. This will greatly reduce the space in which the program needs to search through, thus improving the speed of the search.

In the next section, we provide a formal definition of search and describe how a search system differs from other types of software. Following the definitions, we present programmatic interfaces that have been developed from these formal definitions.
2.1 Formal Search Definitions

On the hardware level search and computation are the same, as they are simply machine instructions to be processed. On the software and performance level, however, differences between the two are noticeable. When a program performs computation, it processes an action which results in more actions and the cycle repeats until there are no more actions to be processed. Software developers prefer this linear execution because it is easier to understand and debug, as there is only a single path to follow and it always guarantees that a solution will be found if it exists.

However, when dealing with combinatorial problems it is not feasible to process every possible action. A better strategy is to take the best path out of many possible paths that might lead to a solution. That is, after computing an action, there are many possible actions available to choose from. Ideally, we want to choose the best possible action but this is often not the case, since if the program knew the right action to take it would not need to use search. Instead the program needs to search through many possible actions to find the best action. Often a search program gets far into the search only to realize that the path chosen leads to a dead end. In this case the program needs to explore different paths. As a result, the nature of search “requires” that there may be many unnecessary actions.

For example, a software program that allows users to look for a specific contact is performing computation. When a user enters a contact’s name to search for the program can start from the first contact and search through the entire list until it finds the contact. Each action that the program takes is guaranteed to find a contact or to find out that there is no such contact.

A chess program, by contrast, is performing search. After a human player makes a move, the chess program can respond with many different actions. Choosing an action from those possible actions will lead to many possible responses from the player. The software needs to consider all these different scenarios in order to have a chance at winning.
A search system can be described using four different components: search model, search instance, search process, and search derivation. The next section looks at the formal definitions of each component. In the following sections, we present formal definitions of search based on definitions from [Den99].

2.1.1 Search Model

When trying to solve a problem it is often helpful to model the problem to allow us to understand the problem better. One way to accomplish this is to simplify the problem. With search problems, we have observed that although each problem might require different algorithms, these algorithms exhibit certain commonalities. More specifically, they can be represented by states and these states are operated on by performing state transitions.

Definition 2.1.1. (Search Model)

A pair $\mathcal{A} = (S, T)$ with $T \subseteq S \times S$ is called a search model. The elements of $S$ are called states, the relation $T$ is the set of transitions.

If $s, s' \in S$ and there are $s_1, ..., s_n \in S$ such that $s_1 = s, s_n = s'$ and $(s_i, s_{i+1}) \in T$ for all $i = 1, ..., n - 1$, then $s'$ is reachable from $s$.

The set of states $S$ consists of all the states a search can be in for a given search problem. The search can transition from one state to another by processing transitions. Search systems need to be designed to solve many instances of a problem. For example, we can model the problem of finding the shortest path to three cities or finding the shortest path to $n$ cities. The former is more specific but it can only be used to solve instances where the number of cities is three. The latter is general and can be applied to any shortest path problem.

For a specific instance of a problem, $S$ and $T$ would have to be defined more concretely. For example, to model a chess game, the states can be represented as the position of each chess piece on the board. The transitions describe how each piece can move from one position on the board to another.
2.1.2 Search Instance

In order for a search system to solve a specific problem instance, the users must be allowed to input data regarding the specific instance they wish to solve. This is accomplished through the search instance component. The search instance is comprised of two parts: the initial search state and the goal condition.

**Definition 2.1.2. (Search Instance)**

Let $\mathcal{A} = (S, T)$ be a search model. A search instance to $\mathcal{A}$ is a pair $\mathcal{Ins} = (s_0, \mathcal{G})$, where $s_0 \in S$ is the start state of the search and the predicate $\mathcal{G} : S \rightarrow \{yes, no\}$ is the goal condition that evaluates those states $s \in S$ to yes in which the search can be successfully terminated.

The starting point of a search system is the start state $s_0$. A problem instance can have many possible goal states, so a predicate $\mathcal{G}$ is required to determine which state is a goal state. A search is finished when the current state is deemed a goal state by the predicate $\mathcal{G}$ or when there are no further possible transitions.

Regarding a chess program, the start state is the starting board placement. The goal condition of a chess program is when either one of the players has put the other in checkmate or the game ends in a draw.

2.1.3 Search Process

The search process is at the heart of the search system. It is a critical component that can affect the effectiveness of a search. It consists of a search control that is responsible for selecting and processing transitions.

**Definition 2.1.3. (Search Process)**

Let $\mathcal{A} = (S, T)$ be a search model. Further, let $\mathcal{Env}$ be a set of structures, describing the possible situations that the environment of the search may be in. The triple $\mathcal{P} = (\mathcal{A}, \mathcal{Env}, \mathcal{K})$
is a search process, if $\mathcal{K} : S \times \text{Env} \rightarrow S$ is a function for which $\mathcal{K}(s,e) \in \{s'|(s,s') \in T\}$ holds for all $s \in S$ and $e \in \text{Env}$. $\mathcal{K}$ is called the search control of the search process.

Normally a search control only requires the search state and the set of possible transitions available to the search state to be able to decide which transition to take. However, for some problems, such as trying to move a robot around the street, the environment can play a big role in the transitions available to the search control. Suppose that the possible moves for a robot can be straight, left, right and back. In this example, there are 4 transitions available to the robot at any given time. However, if an accident suddenly occurs and cars are now blocking the path in front of the robot, the robot now only has 3 transitions to choose from. The search control would require this information from the environment to be able to make the correct choice.

For distributed search systems, having the search control being able to choose transitions based on a situation from $\text{Env}$ means that the search process can handle information from other search processes. Instead of just choosing from its internal state, the search control $\mathcal{K}$ may choose its next state based on the information from the environment. But allowing the control to choose the next action from the environment can also be dangerous. The definition of the search control states that both $s$ and the successor state $s'$ needs to be an element of $T$.

Regarding a chess program, there would not be any influence coming from the environment. All the data required to make the decision resides on the board and so it is possible to ignore the environment and concentrate on the search state and transitions. In a chess game, the number of transitions available at a particular state depends on the number of pieces that are still in play. The control would need to determine all the moves available and evaluate each of those moves and select the best one.
2.1.4 Search Derivation

The search control of a search is rarely optimal the first time it is implemented. There will be many caveats that the developers have not anticipated. Some controls perform great on a particular instance of a problem while performing poorly on others. To analyze and understand the reason, a transcription of the entire search run is required. This is handled by the search derivation, the last component of a search system.

**Definition 2.1.4.** (Search Derivation)

Let \( \mathcal{A} = (S, T) \) be a search model and \( \mathcal{P} = (A, Env, K) \) a search process to \( A \). Further, let \( \mathcal{I}ns = (s_0, G) \) be a search instance to \( A \) and \( e_0 \in Env \) the environment to \( \mathcal{P} \) at the start of the search. The sequence \( \mathcal{A} = (s_i)_{i \in \mathbb{N}} \), such that \( s_{i+1} = K(s_i, e_i) \) is called the search derivation to \( A, \mathcal{P} \) and \( \mathcal{I}ns \). If there is a \( k \in A \) such that \( G(s_k) = \text{yes} \), then \( A \) is called successful. The partial sequence \( \mathcal{A} = (s_i)_{i \leq l} \) with \( l = \min_k(G(s_k) = \text{yes}) \) is called the solution sequence to \( \mathcal{I}ns \).

The search derivation keeps track of all the search states that were traversed since the start of the search run until the search is finished or all states have been evaluated. The ability to review a search run allows the developers to evaluate the effectiveness of the search and where it can be improved. The improvements can be achieved by tweaking the search control or the search model. The search derivation is also capable of determining what the solution to the problem is. Sometimes the search can reach a goal state and halts but the user may not know the solution before analysing the sequence of states that leads to the solution.

2.2 Knowledge Based Search Paradigms

The definitions presented in the previous section are intended to be very general to fit most types of search problems. However, the definitions do not provide any guidelines for building
a good search system. Take for example a search system that enumerates all the possible states. This solution is perfectly valid but it is very inefficient. What is required instead are definitions that, while not as broad as the first set of definitions, define how knowledge based search components should be built.

Knowledge can be used in two ways to improve the efficiency of a search. First, knowledge can help to reduce the number of transitions available in a search thus reducing the space that needs to be searched through. Second, it can allow the search process to select better transitions to take, thus reducing unnecessary steps that waste CPU cycles. The knowledge imbued into a search is not guaranteed to improve for all search problem instances. For one problem instance the knowledge might improve the effectiveness of the search, while it might be worse for another problem instance.

To explicitly define knowledge, software developers often turn to data. Since each person may represent their knowledge differently, there can be many data structures to solve a particular search problem and each of those data structures would be considered valid. Data structures can range from simple to complex (where data structures may point to other data structures). It is not in our interest to define all the possible data structures as they are quite specific to the concrete problem. Instead, we will focus on the general data structures that can represent many types of search. The complex data structures can then be built on top of these general data structures to represent the problem specific knowledge. The thesis [Den99] describes different data structures including sets, and-trees, or-trees, and-or-trees, and graphs. In this thesis we will focus on three of the data structures described in [Den99]: namely set, and-tree, and or-tree data structures.

2.2.1 Sets

The search state can contain many different types of information. The simplest way to represent this information in a data structure is to store it as facts. These facts together make up the set data structure. The only requirement of the set is that no fact occurs more
than once in the set. The definition of a set-based search model is as follows.

**Definition 2.2.1. (set-based search model)**

Let $\mathcal{F}$ be a set of facts and $\mathcal{E}xt \subseteq \{A \rightarrow B \mid A,B \subseteq \mathcal{F}\}$ be a set of extension rules to $\mathcal{F}$. We call the pair $\mathcal{A}_{set} = (S_{set}, T_{set})$ a set-based search model, if the following holds:

- $S_{set} \subseteq 2^\mathcal{F}$ and
- $T_{set} = \{(s,s') \mid \text{there is } A \rightarrow B \in \mathcal{E}xt \text{ with } A \subseteq s \text{ and } s' = (s-A) \cup B\}$.

The search state is now represented as facts in the set-based definition of the search model. These facts are not associated with any other facts and there are no hierarchies or relationships between the facts. The extension rules extend the states from one to another by either adding or removing facts from the state. With set-based search there can be infinitely many facts about a search problem thus introducing infinitely many transitions.

**Definition 2.2.2. (set-based search instance)**

Let $\mathcal{A}_{set} = (S_{set}, T_{set})$ be a set-based search model. Let further be $s_0, s_{goal} \in 2^\mathcal{F}$. The set-based search instance $\mathcal{I}ns_{set} = (s_0, G_{set})$ to $\mathcal{A}_{set}$ and $s_0, s_{goal}$ is defined by

$G_{set}(s) = yes$, if and only if $s_{goal} \subseteq s$ or there is no extension applicable in $s$.

A set-based search is considered finished once a particular set of facts is contained within the search state or no further extensions are possible (which suggests that there are no solutions for the instance). For certain problems, using a set-based search might not guarantee that a solution can be found. In this case, the goal condition is not connected to the problem instance but instead a time limit is used to guarantee that the search does end.

In the general definition of the search process, the search control is responsible for choosing what it believes to be the best transition, however, how it determines the best transition is left to the developers to define. Having states represented as sets of facts and transitions as extension rules for those facts allows for a more precise definition of the set-based search process compared to the general search process definition. The control can now base its
decision as to which facts to extend by measuring the quality of the facts that are involved in the extension.

**Definition 2.2.3.** (set-based search process)

Let $\mathcal{A}_{\text{set}} = (S_{\text{set}}, T_{\text{set}})$ be a set-based search model and $\mathcal{E}_{\text{nv}}$ an environment. $\mathcal{P}_{\text{set}} = (\mathcal{A}_{\text{set}}, \mathcal{E}_{\text{nv}}, \mathcal{K}_{\text{set}})$ is a set-based search process, if the following holds: $\mathcal{K}_{\text{set}}(s, e) = (s - A) \cup B$, where

- $A \rightarrow B \in \mathcal{E}_{\text{xt}}$,
- $A \subseteq s$,
- for all $A' \rightarrow B'$ with $A' \subseteq s$ holds: $f_{\text{Wert}}(A, B, e) \leq f_{\text{Wert}}(A', B', e)$ and
- $A \rightarrow B = f_{\text{select}}(\{A' \rightarrow B' \mid f_{\text{Wert}}(A', B', e) \leq f_{\text{Wert}}(A'', B'', e)\text{ for all } A'' \rightarrow B'' \in \mathcal{E}_{\text{xt}}\text{ with } A'' \subseteq s\}, e)$,

for two functions $f_{\text{Wert}} : \mathcal{F} \times \mathcal{F} \times \mathcal{E}_{\text{nv}} \rightarrow \mathbb{N}$ and $f_{\text{select}} : 2^{\mathcal{F} \times \mathcal{F}} \times \mathcal{E}_{\text{nv}} \rightarrow \mathcal{F} \times \mathcal{F}$.

The function $f_{\text{Wert}}$ determines the quality of the facts involved in the extension. Having this capability gives the search control the means to determine the quality of each possible transition. In cases where $f_{\text{Wert}}$ is not accurate enough to determine whether one transition is better than another, $f_{\text{select}}$ can be used to help the control select the best transition. For example, if $f_{\text{Wert}}$ gives two transitions the same value, it is up to $f_{\text{select}}$ to break the tie and pick the facts to extend. Ideally we want to select an extension that produces a successor state containing a smaller set of facts, as that will allow for fewer possible extensions.

The definition of search derivation does not need to be refined for the set-based search because the process of tracking the transitions and states does not change by adding knowledge to search systems.
2.2.2 And-Trees

Set based search systems require that facts are independent from each other. While this can make it easier to model, in cases where several pieces of information are connected it would be best to represent them as connected entities. When we have pieces of information that are connected by ways of alternate paths, or when they are connected such that certain parts of the problem are required to be solved in order for the parent problem to be solved then a tree-based data structure is preferable to a set-based data structure.

The and-tree based data structure allows for problems to be divided into sub-problems. The search starts with the full problem and divides it into smaller, more manageable sub-problems until it can find solutions to the sub problems. If the solutions to all the sub-problems are found then the larger problem is also potentially solved (as long as the solutions are compatible).

The method of dividing a problem into sub-problems is entirely dependent on the problem instance. When dividing the problem into sub-problems the goal is to create a division such that all the sub-problems are required to be solved to get a solution to the parent problem. Since we are dealing with a tree structure the root node is the initial problem instance and the child nodes are sub-problems of their parent nodes. The updated definition of the and-tree based search model is more complex than its set-based counterpart as it needs to cover how to divide the problems into sub-problems.

**Definition 2.2.4.** (and-tree based search model)

Let $\mathcal{Prob}$ be a set of problem descriptions and $\mathit{Div} \subseteq \mathcal{Prob}^+$ a division relation. Then the set $\mathit{Atree} = \mathit{Atree}(\mathcal{Prob})$ is recursively defined as follows:

- $(\mathit{pr}, \mathit{sol}) \in \mathit{Atree}$ for $\mathit{pr} \in \mathcal{Prob}$, $\mathit{sol} \in \{\text{yes,?}\}$
- $(\mathit{pr}, \mathit{sol}, b_1, \ldots, b_n) \in \mathit{Atree}$ for $\mathit{pr} \in \mathcal{Prob}$, $\mathit{sol} \in \{\text{yes,?}\}$, $b_1, \ldots, b_n \in \mathit{Atree}$.

Let further $Erw_\wedge$ and $Erw_\wedge^*$ be relations on $\mathit{Atree}$ defined by

$Erw_\wedge((\mathit{pr},?), (\mathit{pr}, \text{yes}))$, if $\mathit{pr}$ is solved
\( Erw_{\Lambda}((pr,?), (pr, ?, (pr_1,?), \ldots, (pr_n,?))) \), if \( Div(pr, pr_1, \ldots, pr_2) \) holds

\( Erw_{\Lambda}((pr,?,b_1,\ldots,b_n),(pr,?,b'_1,\ldots,b'_n)) \), if it holds for an \( i \) that \( Erw_{\Lambda}(b_i,b'_i) \) and \( b_j = b'_j \) for all \( j \neq i \).

and

\( Erw_{\Lambda} \subset Erw_{\Lambda}^* \)

\( Erw_{\Lambda}((pr,?,b_1,\ldots,b_n),(pr,?,b'_1,\ldots,b'_n)) \), if for all \( i \) either \( Erw_{\Lambda}^*(b_i,b'_i) \) or \( b_i = b'_i \) holds.

The pair \( \mathcal{A}_{\Lambda} = (S_{\Lambda}, T_{\Lambda}) \) is called an and-tree-based search model if

- \( S_{\Lambda} \subseteq Atree \) and
- \( T_{\Lambda} = \{(s_1,s_2) \mid s_1, s_2 \in S_{\Lambda} \text{ and } Erw_{\Lambda}(s_1,s_2) \text{ or } Erw_{\Lambda}^*(s_2,s_1)\} \).

Modelling with an and-tree data structure starts by recursively dividing the problem (accomplished with the division relation \( \text{Div} \)) into sub-problems, until the problem cannot be further divided. The model also allows the problem to be backtracked (described by \( Erw_{\Lambda}^* \)) in case the sub-problems are not solvable or the solutions are not compatible.

The fact that problems can be divided into sub-problems presents new challenges in how to determine whether a state is a goal state. Solutions to the problem are only valid if the solutions to the sub-problems are compatible with each other. To determine the goal state all the leaf nodes must be tested to determine that their solutions are compatible to ensure that the parent nodes can be solved. Determining whether nodes are compatible is dependent on the actual problem, thus makes it hard to define.

**Definition 2.2.5.** (and-tree based search instance)

Let \( \mathcal{A}_{\Lambda} = (S_{\Lambda}, T_{\Lambda}) \) be an and-tree-based search model. Let further \( pr \in \mathcal{P}rob \) be a problem to solve. The and-tree-based search instance \( \mathcal{I}ns_{\Lambda} = (s_0, \mathcal{G}_{\Lambda}) \) to \( \mathcal{A}_{\Lambda} \) and \( pr \) is defined by

\[ s_0 = (pr,?) \]
\[ G_\land(s) = yes, \text{ if and only if } s=(pr', yes) \text{ or } s=(pr', ?, b_1, ..., b_n), \quad G_\land(b_1) = ... = G_\land(b_n) = yes \text{ and } b_1, ..., b_n \text{ are compatible with each other or there is no transition that has not been tried out already.} \]

In this definition of the and-tree based search instance \( G_\land \) recursively checks each node to determine if the solution is valid, i.e. the child nodes must be solved and the solutions are compatible. If no solution is found and the tree cannot be divided any further or differently, then it means that there are no solutions for the problem instance.

The definition of an and-tree based search process cannot be more specific than the general definition. However, generally it is a good idea to divide the search control into two parts: the selection and the extension. The selection determines which leaf in the tree should be processed next. The extension selects which transition should deal with the selected leaf. Similarly, the definition of the and-tree based search derivation cannot be more specific than the general definition previously given.

### 2.2.3 Or-Trees

In a search it is rare to have a single path leading to an optimal solution. Instead, there may be many possible alternative paths. The or-tree data structure allows the search to deviate and try alternate paths in the hope that it will find a solution. With or-trees, each successive node adds additional information to the previous node. This differs from and-trees (which try to break down information) while or-trees try to add information until a leaf node contains the solution to the problem.

**Definition 2.2.6.** (or-tree-based search model)

Let \( \mathcal{Prob} \) be a set of problem descriptions and \( \text{Altern} \subseteq \mathcal{Prob}^+ \) a solution alternative relation. Then the set \( Otree = Otree(\mathcal{Prob}) \) is recursively defined as follows:

\[(pr, sol) \in Otree \text{ for } pr \in \mathcal{Prob}, sol \in \{yes, ?, no\} \]

\[(pr, sol, b_1, ..., b_n) \in Otree \text{ for } pr \in \mathcal{Prob}, sol \in \{yes, ?, no\}, b_1, ..., b_n \in Otree.\]
Let further be $Erw_\lor$ a relation on $Otree$ defined as

$Erw_\lor((pr,?), (pr, yes))$, if $pr$ is solved

$Erw_\lor((pr,?), (pr, no))$, if $pr$ is unsolvable

$Erw_\lor((pr,?), (pr,?, (pr_1,?), \ldots, (pr_n,?)))$, if $Altern(pr, pr_1, \ldots, pr_n)$ holds

$Erw_\lor((pr,?, b_1,\ldots, b_n), (pr,?, b'_1,\ldots, b'_n))$, if for an $i$ $Erw_\lor(b_i, b'_i)$ holds and $b_j = b'_j$ for all $j \neq i$.

Then the pair $A_\lor = (S_\lor, T_\lor)$ is called an or-tree-based search model, if

- $S_\lor \subseteq Otree$ and
- $T_\lor = \{(s_1, s_2)| s_1, s_2 \in S_\lor \text{ and } Erw_\lor(s_1, s_2)\}$

Or-trees start with a problem description and recursively travel the tree until a solution is found. Backtracking is not required with or-trees because all possible alternatives are represented in the search state. When a path does not lead to a solution the search can select from other alternative paths rather than backtracking. The transitions of an or-tree based search model are defined by $Altern$ which specifies all the alternative paths that the search can take at each node.

**Definition 2.2.7. (or-tree-based search instance)**

Let $A_\lor = (S_\lor, T_\lor)$ be an or-tree-based search model. Let further $pr \in Prob$ be a problem to solve. The or-tree-based search instance to $A_\lor$ and $pr$ is defined by

$s_0 = (pr,?)$ and

$G_\lor(s) = yes$ if and only if $s=(pr',yes)$ or $s=(pr',?,b_1,\ldots,b_n)$ and $G_\lor(b_i) = yes$ for an $i \in \{1,\ldots,n\}$ or if all leafs of $s$ have either the sol-entry "no" or cannot be processed using Altern.

An or-tree-based search is finished when a leaf node is a solution or the problem is deemed to not be solvable (when all leafs have been processed and no solution is found).

Similar to and-trees, the search process and search derivation definitions for an or-tree based search cannot be made more specific than that of the general definition.
2.3 Programmatic Interface of Search

One of the main goals of our search library is to be as general and flexible as possible so it can be extended to solve most search problems using most search paradigms. The first step in achieving this goal is to base the design of the library on very general definitions of search. Even when knowledge is added only very basic data structures are used so that they can accommodate most search problems. Problems that require more complex data structures can build on top of these basic data structures. The second step, which is discussed in this section, is to take these abstract definitions and turn them into very general software components that can be used and adapted to solve many different search problems. To achieve this, we made use of the programmatic interfaces provided by the Java programming language.

The Java programming language was chosen for three main reasons. The first is that it is available on all major operating systems. When developing a software library, we want to attract as many potential users as possible. Using a programming language that allows the software to be built only once and then run on any supported operating system means that we can attract more developers than compared to a language that is only supported on a few operating systems or must be recompiled for that specific operating system. The second reason is that Java is currently one of the most popular programming languages. Again, this allows for a greater potential adoption of our library as opposed to a relatively unknown language. The last reason is that Java is an object oriented language. Object oriented programming is the industry standard in writing software programs. The topic has been well studied and many works have been written on how best to develop software using object-oriented programming. As object-oriented programming is the industry standard, it is safe for us to assume that most software developers already know how to program in an object-oriented manner so that, even if they have never used Java before, they can apply this existing object-oriented knowledge to Java quite easily.
In object oriented programming, one of the most general constructs is the programmatic interface. One can think of the interface as a contract that enforces objects (which are instantiations of a class) that implement the interface to adhere to the methods and functionality defined in the interface contract. An object is not limited to implementing one interface but can implement as many interfaces as it needs. Thanks to this flexibility, the interface is used in most software libraries and frameworks to ensure the structure of the system is followed. In our library, each search component that was defined in previous sections is represented as a programmatic interface. When designing the interface, our goal was to try to keep it in-line with good software design. To achieve this, some components were broken up into multiple interfaces. The programmatic interfaces discussed in the next section are shown as Unified Modelling Language (UML) class diagrams.

Another goal we had for our library was for it to be compatible with other distributed search libraries that implement other distributed search paradigms. Ideally, we want the developer to only have to develop the search program once and be able to distribute it using our library or other distributed search libraries without having to make many changes to the code. To that end, we have chosen to use the same skeleton interfaces as DisSLib:CC [Ken10] so that the structure of the search system will be similar.

2.3.1 Search Model

The formal definition of a search model consists of two entities that search systems need to define: the set of states S and the set of transitions T. We can enforce this structure by creating interfaces based on these entities. The SearchState interface (Figure 2.1) represents the current state of the search.

The search state interface defines two methods: integrateResult and selectTransition. It does not force the library users to use a specific data structure but instead relies on the developers to implement the underlying data structure that is relevant to the problem instance. The selectTransition method chooses the next transition to be processed. This
Figure 2.1: Class diagram showing the interface of the **SearchState**. The search state defines two methods: *selectTransition* and *integrateResult*. The *selectTransition* method has a search control as an argument to allow for multiple search controls to be used during a search run. Usually, selection of transitions is delegated to the search control but it can also be accomplished by the search state. Once a transition is processed, the transition result is integrated back into the search state through the *integrateResult* method.

```
<SearchState>
  + integrateResult(result:transitionResult)
  + selectTransition(control:SearchControl):searchtransition
```

method depends on a search control that is passed in as a parameter. The reason that the search control is an argument is so that the search state can use multiple search controls during a search run. For example, if during a search run, the search control being used is not providing very good results, that control can be swapped with another search control that might perform better.

The way *selectTransition* is designed gives the developers flexibility on how to choose the next transition. One possibility is to have the search state itself select the next transition. This is ideal when the search state already knows the possible transitions due to the processing from previous rounds. This way the search state can use the information from its internal state to select the next transition rather than waste CPU cycles on having the search control decide. In normal cases though the search state usually delegates to the search control since the control has knowledge of how to select a transition.

The other method a search state requires is *integrateResult*. This method is executed after a transition is processed by the search control. When the search control processes a transition, it produces a **SearchResult** (Figure 2.2). The search result is a data container that can contain any necessary information the search state requires in order for it to transition to the next state.

In addition to implementing the two methods, the developers also need to define how the internal state will be stored. This usually includes the data structure used to model the search and possibly any caching or pre-computed results in order to speed up some of the
Figure 2.2: The **SearchResult** is a data container that holds the result of the processing done by the search control. The data contained in the search result is dependent on the search problem and needs to contain all the necessary data in order for the search state to advance to another state.

```
«SearchResult»
```

operations that are performed in the `selectTransition` and `integrateResult` methods.

In addition to the search state, developers also need to define the structure of a search transition. Much like the **SearchResult** interface, the search transition interface (Figure 2.3) does not require any methods to be implemented. Again, this is because the search transition is a data container that is dependent on the concrete search problem. It is up to the developers to define what specific information is necessary for a search state to transition to another state. When a search state selects a transition through the `selectTransition` method, it will create a **SearchTransition** object and returns it to the search control to be processed. The search control uses this information to produce a **SearchResult** to be integrated by the search state.

Figure 2.3: The **SearchTransition** object holds information regarding how a state can transition to another state. Its implementation is empty because its internal representation is dependent on the search state and the concrete search problem.

```
«SearchTransition»
```

2.3.2 Search Instance

The formal definition of the search instance states that it has two responsibilities. The first responsibility is to provide the initial start state to the concrete search problem. The second is identifying when the search has finished by checking to see if the current state is a goal state. Since it is generally a bad design to have objects with more than one responsibility, we have separated the responsibilities of the search instance component into two interfaces:
the ProblemInstance and the SearchObserver interface.

The ProblemInstance (Figure 2.4) interface is responsible for providing the initial start state to the search system through its generateStartState method. Normally, to provide flexibility, this method needs to get its input (the data regarding the problem instance) from an external source (database, text file, etc...) before it can generate a search state. Once the problem instance data has been gathered, the method creates the start state of the search.

Figure 2.4: The ProblemInstance object is part of the search instance component. It is responsible for creating the initial start state through the generateStartState method.

<table>
<thead>
<tr>
<th>«ProblemInstance»</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ generateStartState():SearchState</td>
</tr>
</tbody>
</table>

The SearchObserver (Figure 2.5) can identify when the search is finished through its canContinue method. This method is passed, as an argument, to the current search state and is used to determine if a goal state has been found. In addition, it also checks whether there are any more transitions available for the search system. When either the goal state has been found or there are no more transitions the method will return false and the system will stop.

Figure 2.5: The SearchObserver is responsible for deciding whether a state is finished through the canContinue method. This method determines whether a specified search state is either a goal state or no further transitions are available.

<table>
<thead>
<tr>
<th>«SearchObserver»</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ canContinue(state:SearchState):bool</td>
</tr>
</tbody>
</table>

2.3.3 Search Process

At the heart of the search process is the SearchControl (Figure 2.6). The search control is responsible for two things: selecting transitions based on the current search state and processing selected transitions to return a search result to be integrated back into the search state. It defines two methods to accomplish these tasks: getNextTransition and processTran-
situation. These two methods are two of the most important methods in the search system since they determine the quality and speed of the search.

Selecting the next transition to be processed is handled by the method `getNextTransition`. This method takes a search state as an argument to be used to determine the best transition to take with respect to the current search state. Sometimes the search state contains more information than is available to the search control so getting the next transition can be achieved by calling the `selectTransition` method of the search state instead. If this method is used, the developers need to ensure that either the search control delegates to the search state or the search state delegates to the search control otherwise an infinite loop can occur as shown in Listing 2.1. However, this is often a non-issue in a normal search system as either only the search control or search state would have enough information to select the next transition.

Listing 2.1: This example shows how an infinite loop can occur when both the search control and search state delegates the transition selection process to each other.

```java
/* selectTransition method from the SearchState interface */
public SearchTransition selectTransition(SearchControl control) {
    return control.getNextTransition(this); //delegates to the control
}

/* getNextTransition method from the SearchControl interface */
public SearchTransition getNextTransition(SearchState state) {
    return state.selectTransition(this); //delegates to the search state
}
```

When implementing the `getNextTransition` method, not only do the developers need to consider the method of selecting the next transition but they also need to consider the performance of the method. Selecting a good transition is very important since it can lead to the goal state faster. However, if it takes a really long time to determine the next transition then the entire search process will still be slow. The developer needs to consider the trade-offs between a really good quality transition and the speed at which the program selects those transitions.
Figure 2.6: The SearchControl object requires two methods to be implemented: getNextTransition and processTransition. The getNextTransition selects a transition based on the current search state that is passed in as an argument. Once the transition has been selected, processTransition method will be called to process transition, producing a search result that is to be integrated back into the search state.

<table>
<thead>
<tr>
<th>SearchControl</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNextTransition(state: SearchState): SearchTransition</td>
</tr>
<tr>
<td>processTransition(transition: SearchTransition): TransitionResult</td>
</tr>
</tbody>
</table>

The processing of the selected transition is handled by the processTransition method. This method returns a search result that needs to be integrated back into the search state. It might seem odd that a transition needs to be processed before it can be integrated since the formal definitions do not define such a process. However, while the definition does not explicitly say so, there is an implicit processing that must be done to the transition before it can be integrated. A search control, as defined, is a function that takes a search state and selects a transition producing another state. For example, in tree-based search, each node is either (depending on whether it is an or-tree or and-tree) divided into sub-problems or extended to produce alternative solutions. Either way the node needs to be processed in order to produce child nodes that will transition the current state to the next state. The processTransition method is responsible for the processing of the transitions.

2.3.4 Search Derivation

The search derivation is more than a log of the search process. It keeps track of the transitions and states throughout the search run and is also capable of determining the solution to the problem instance (from the recorded states and transitions). The search derivation is represented by the SearchProtocol (Figure 3.6).

The search protocol defines four methods that enable it to record the properties of the search run and to find the solution at the end. The method recordSearchState takes a state as an argument. As the name suggests, the search state is recorded to the protocol’s internal
data structure. The data structure used by the protocol is dependent on the problem but should be designed to be as small as possible to limit memory usage. Similarly, `recordTransition` records the transition that is passed in as an argument. The `recordResult` method records the search result of the processing done by the search control on the selected transition. Finally, the `recordSearchEnd` method records the final state that has been selected and also processes the recorded states and transitions to determine the solution.

Figure 2.7: The `SearchProtocol` interface consists of four methods that must be implemented to allow the search protocol to record all the necessary information of the search run. The `recordSearchState`, `recordTransition`, `recordResult` records the state, transitions, and results of the processing respectively. The `recordSearchEnd` method records a search state and then determines the solution by processing all the recorded states and transitions.

<table>
<thead>
<tr>
<th>«SearchProtocol»</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordSearchState(state:SearchState)</td>
</tr>
<tr>
<td>recordTransition(transition:SearchTransition)</td>
</tr>
<tr>
<td>recordResult(result:TransitionResult)</td>
</tr>
<tr>
<td>recordSearchEnd(state:SearchState)</td>
</tr>
</tbody>
</table>

When designing the search protocol, the developers need to keep in mind the performance degradation that the search protocol can have on the search system. In a normal run, a search system usually processes a very large number of transitions and states. If each state or transition is complex and uses a lot of memory then recording the entire state or transition each time would result in the system quickly running out of memory. Instead, the developers should record only the delta between the states or transitions at each step of the search. This allows each record of the search protocol to consume less memory.

If the search protocol is still using too much memory, the developers should write the information to disk to offload some of the memory usage. However, this can cause a slowdown during the disk writing since disk writes are very slow compared to memory writes. Because of this, we recommend only writing to disk when the search protocol becomes very large.
Normally, however, the protocol does not have to write both state and transition data. Instead, just recording the selected transitions is enough to reconstruct the entire search run.

2.3.5 Environment

Figure 2.8: The environment object is responsible for the external data outside of the search model. It does not expose any methods because its design is dependent on the search system and the search problem at hand.

```
<table>
<thead>
<tr>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

The **Environment** object allows the search to respond to external influences. These external influences include static information (such as input data), and run-time information (search configuration information). The internal structure of the environment (Figure 2.8) is dependent on the concrete problem and the design of the system.

So far we have discussed much about sequential search systems. However, one drawback of sequential search systems is that they tend not to make use of the all processing capabilities that are available in modern computers. In the next chapter we will look at distributing the search process to many processing units to improve the performance of the search system.
The last chapter presented knowledge based search, a concept that can improve the efficiency of a search system. However, since search systems often deal with combinatorial problems, knowledge based search alone is not enough. Of course, algorithms and heuristics can always be improved to run faster, and data structures can be improved to use less memory, but the returns on investment will eventually diminish.

In the past, developers would turn to the hardware for speed-ups by upgrading to faster processors and adding more memory. This method required little effort from the developer. Unfortunately, this method is no longer viable, as processors are moving towards having many cores rather than increasing clock rate. Developers now need to make use of the additional CPU cores to further improve their search systems, and one way of achieving this is to distribute the search process.

A simple distribution approach would be to take an existing sequential system and modify its search control to generate additional search systems. These search systems can then execute on their own processing unit (i.e. another processor or CPU core). This approach is called the competition approach. Through the competition approach, the best solution would come from the search process with the best search control or running on the fastest processing unit. A drawback of this approach is that the work of the other search processes do not contribute to the final solution.

Instead of having the search processes compete against one another, it would be better if they were to cooperate with one another to reduce redundancy and increase the effectiveness of the search. Getting the search processes to cooperate, however, is complex and requires the developer to clearly understand the advantages and disadvantages of the many different
methods of distributing search processes. This is not an easy task because search systems
tend to not lend themselves to be naturally distributed as other types of software systems.
However, researchers have developed different distribution paradigms that provide developers
with different guidelines for distributing the search among multiple processing units.

This chapter will discuss three cooperative distribution paradigms but the main focus
will be on the improving on the competition approach paradigm that our library is intended
to support. In the next section we will describe the formal definitions of a distributed search
system built upon the definitions in [Den99]. After the formal definitions we will discuss some
of the different distributed search paradigms as described in [Den02] and provide definitions
of the extension to the improving on the competition approach paradigm. Finally, the
chapter will end with a discussion of the changes to the programmatic interface that are
required for distribution.

3.1 Basic Definitions

In a distributed system, instead of focusing on each individual search process in the search
system, the main focus is on the interaction between the search processes, which are com-
monly referred to as search agents.

In order for the search agents to communicate, the system developer must decide on a
communication model between the agents. One such communication model is the shared
memory model, i.e. a blackboard. In this model each agent has access to the same memory
area. Agents only need to update their own internal area and this change can be seen by
all the other agents. The big advantage of this model is that there is no need for network
communication which can greatly speed up the communication process. A disadvantage of
this model is that it needs to be operated in an environment where each processing unit
shares memory with one another. Of course it can also be simulated by having an agent
broadcast its changes but this method requires network communication which adds latency
On the other end of the spectrum is the communication model where agents communicate directly with each other. The benefit of this model is that instead of sending each agent in the system a message every time a change occurs, messages are only sent to agents that need to be updated. The disadvantage of this model is the extra communication involved if an agent needs to broadcast a message to all the other agents. Another disadvantage of this model is that it requires agents to know how to find and connect to each other.

Definition 3.1.1. Communication Structure

A communication structure $Kom$ is a structured set of data areas, i.e. $Kom = (D_1, ..., D_l)$ for sets $D_i$ of objects. The actual value $val$ of $Kom$ is defined as the tuple $(d_1, ..., d_l)$ with $d_i \in D_i$.

The function $dat : Kom \times \{1, ..., l\} \rightarrow D_1 \cup ... \cup D_l$ is defined as $dat(val, i) = d_i$.

We can think of $Kom$ as a group of communication channels that allow agents to communicate with each other. $Kom$, however, does not specify how these data areas can be modified. It is the responsibility of the search agent to find the respective channel(s) of the other agent it wishes to communicate with.

3.1.1 Search Agent

When an agent accesses $Kom$ it needs to know what it can and cannot write to $Kom$. This is achieved with a communication function called $mes$ as defined below.

Definition 3.1.2. (Search Agent)

Let $Kom$ be a communication structure and $P = (A, Env, K)$ a search process. Further let $mes : S \times Kom \rightarrow Kom$ be a so-called communication function. Then the triple $Ag = (P, Kom, mes)$ is called a search agent.

The $mes$ function can change the values of $Kom$ but does not have to change each of the data areas in $Kom$. When an agent should call its $mes$ function is largely determined
by the concrete problem the search is trying to solve. For certain problems \textit{mes} should only be called once the agent has enough information to share. For problems that require synchronization between the agents, \textit{mes} should be called at the agreed upon time between all the agents. In this case, \textit{mes} will write to multiple data areas and the search will halt to enable all the agents to synchronize and share information at the same time. Sharing information between the agents is beneficial if the information from other agents will enable the search to perform better than it would have on its own. What to write into the data areas of \textit{Kom} is largely determined by the concrete problem.

3.1.2 Distributed Search

While most of the agents in a distributed search system are search agents there are agents that are also required by the distributed system for administrative purposes. These agents include the start agent (\textit{Ag}_s) and the end agent (\textit{Ag}_e). In a distributed search system, however, not all agents start at the initial problem instance. Some distributed search systems might switch agents half way through the search which means that the new agent requires the latest information about the problem instance and solutions. These tasks are handled by the start agent.

At the end of a distributed search run, many agents might have a solution to the problem (or can contribute to it). In these cases an end agent is required in order to determine which agent has the best solution (if any), or to piece together the solution from multiple agents. The definition of a distributed search system along with \textit{Ag}_s and \textit{Ag}_e is given below.

\textbf{Definition 3.1.3.} Distributed search system

Let \textit{SP} be a search problem, \textit{Kom} a communication structure and let \textit{Ag}_s and \textit{Ag}_e be agents, \textit{Ag}_1, ..., \textit{Ag}_n a set of search agents, \textit{Ag}_i = (P_i, \textit{Kom}, \textit{mes}_i). Further, let \textit{PU} =\{pu_1, ..., pu_z\}, \(z \leq n\), be a set of processing units. Then the tuple \(\textit{DS} = (\textit{Ag}_s, \textit{Ag}_e, \textit{Ag}_1, ..., \textit{Ag}_n, \textit{Kom}, \textit{PU})\) is a \textit{distributed search system to \textit{SP}}, if
• for each \( \mathcal{P}_i = (A_i, \mathcal{E}nv_i, \mathcal{K}_i) \) there are \( D_{j_1}, \ldots, D_{j_p} \) in \( K_{om} \) such that \( \mathcal{E}nv_i = (D_{j_1}, \ldots, D_{j_p}) \).

• the agents can use the processing units to run their search processes and their \( \text{mes-} \)functions.

• the agent \( A_{gS} \) produces for each instance \( \text{Inst} \) of \( SP \) \( n \) pairs \((s_0, G_1), \ldots, (s_0, G_n)\) such that \((s_0, G_t)\) is a search instance for \( \mathcal{P}_i \).

• If \( A_1, \ldots, A_n \) are the search derivations produced by \( \mathcal{P}_1, \ldots, \mathcal{P}_n \) to \((s_0, G_1), \ldots, (s_0, G_n)\) then \( A_{ge} \) produces a solution to the instance \( \text{Inst} \) of \( SP \), if
  
  - \( \text{Inst} \) is solvable and
  
  - all \( A_{gi} \) terminated their search in a regular fashion.

The definition of the distributed search system also defines which data area an agent can access. In addition, the definition also specifies that the agent running on a processing unit must be known. This is in case a search system require that all the processing units have the same capabilities or that the processing units must be homogeneous.

3.1.3 Time Frame and search course protocol

In a sequential system, time is only an important factor if the problem instance needs to be solved within a specific time frame. In a distributed environment, time is a significant concern for the agents and the communication function. With regards to the communication function, time dictates how many messages can be sent and the order in which the messages are received. Since agents are running concurrently, they can be processing many solutions and sharing the results with many agents. The order in which an agent receives messages is dependent on many variables, such as the time at which the message is sent, the size of the message, and the network latency at the time. If an agent receives a message with a solution and it believes the solution to be good and follows that path, and the next message
is from another agent which has followed that same path and found it to be a dead end, then if the agent had received the second message first then it could have ignored that path altogether. This behaviour suggests that time is a critical factor with respect to the quality and efficiency of a distributed search system.

**Definition 3.1.4. Time Frame**

Let $SP$ be a search problem, $\mathcal{CS} = (Ag_s, Ag_e, Ag_1, ..., Ag_n, Kom, PU)$ be a distributed search system to $SP$ and let $Inst$ be an instance of $SP$. Further, let $A_1, ..., A_n$ be the search derivations produced by the search agents $Ag_1, ..., Ag_n$, given the search instances generated by $Ag_S$ to $Inst$. A set $\mathcal{TF} = \{t_1, ..., t_m\}$ of points in time is called a *timeframe* to $\mathcal{CS}$ and $Inst$ if the following holds:

- $t_i < t_j$, if $i < j$.
- For each state $s_{i,j}$ in $A_i$ there is a $t_k \in \mathcal{TF}$, such that $s_{i,j}$ holds in $t_k$ but not in $t_{k-\epsilon}$, for some $\epsilon \in \mathbb{R}^+$. $t_k$ is called the point in time of $s_{i,j}$: $t_k = \text{time}(s_{i,j})$.
- If $s_{i,j}$ is a state in $A_i$ and $val$ the value of $Kom$ at $\text{time}(s_{ji})$, then there are $t_{r_1}, ..., t_{r_l}$; $t_{r_k} \geq \text{time}(s_{i,j})$ for all $k$, such that for all $d_k$ in $val$ it holds: The value of data area $k$ of $Kom$ at the point in time $t_{r_k}$ is $\text{dat}(\text{mes}_{i}(s_{i,j}, val), k)$.

The communication time of $\text{mes}_{i}(s_{i,j}, val)$ is defined as the maximum of $\{t_{r_1}, ..., t_{r_l}\}$ minus $\text{time}(s_{i,j+1})$.

A timeframe is a set of points in time where each point reflects the change of an agent’s state or when an agent sends and receives messages. Knowing the point in time of each agent’s actions, it is now possible to order the actions of each agent with respect to time allowing users to observe the system as a whole. This is achieved through the search course protocol.

**Definition 3.1.5. Search Course Protocol**
Let $SP$ be a search problem, $CS = (Ag_S, Ag_E, Ag_1, ..., Ag_n, Kom, PU)$ a distributed search system to $SP$ and let $Inst$ be an instance of $SP$. Further, let $A_1, ..., A_n$ be the search derivations produced by the search agents $Ag_1, ..., Ag_n$ given the search instances generated by $Ag_s$ to $Inst$. Also, let $\mathcal{T}\mathcal{F} = \{t_1, ..., t_m\}$ be a time frame to $CS$ and $Inst$ and let $\text{runs} : PU \times \mathcal{T}\mathcal{F} \rightarrow \{Ag_S, Ag_E, Ag_1, ..., Ag_n, none\}$ be a function which assigns to each processing unit at each time at most one agent. Then the sequence

$$SCP = (s_{10}, ..., s_{n0}, val_0), ..., (s_{1m}, ..., s_{nm}, val_m)$$

is called the protocol of a cooperative search run to $CS$, $Inst$ and $\mathcal{T}\mathcal{F}$ if the following holds:

- $s_{ik} \in S_i$
- For $s_{ik}$ and $s_{i,k+1}$ it holds that either $s_{ik} = s_{i,k+1}$ or $(s_{ik}, s_{i,k+1}) \in T_i$.
- $val_k \in Kom$

For an agent $Ag_i$ the set $act_i \subseteq \mathcal{T}\mathcal{F}$ of its active points in time is defined by

$$act_i = \{t \in \mathcal{T}\mathcal{F} | \text{there is a } k \text{ such that } \text{runs}(pu_k, t) = Ag_i\}.$$

The protocol is a sequence of snapshots of the entire system at specific points in time. Since it requires the entire time frame of each agent in the system the protocol can only be created after the search run is completed. With the search course protocol users can now analyse the effectiveness of the distributed system after a search run has been completed.

In the case that there are more agents than processing units, the information can be used to identify when each agent gets their share of the processing units. Knowing this can then help users identify whether an agent is effectively making use of those processing unit shares or if the agent should be replaced with another.
3.2 Distributed Search Paradigms

There are many ways to distribute a search problem which makes it difficult to decide on the best approach. Developers must answer many questions such as how the problem is to be distributed, which communication structure should be used, what are the responsibilities of each agents, and what data structures are best suited for the problem. Distribution paradigms provide a general guidance to how search problems can be distributed. This chapter will look at three methods of distributing a search problem:

- Using a common search state.
- Dividing the problem instance into sub-problems.
- Improving on the competition approach by allowing agents to share selected information.

The following sections will discuss these distribution paradigms along with the advantages and disadvantages of each paradigm. We will also briefly take a look at how each of these ideas can be improved.

3.2.1 Using a Common Search State

A challenge with distributing a search problem is trying to prevent redundant work. One method of reducing redundancy is to partition the search space so that agents will not search through the same parts. The partitioning of the search space can be done statically, partitioning the agents beforehand, or dynamically, partitioning the agents while the search is running.

With static partitioning no communication is necessary between the agents, as each agent knows the area for which it is responsible. One disadvantage of the static partitioning, however, is that some agents can finish their work faster than others, thereby leaving them
with no work to do. While dynamic partitioning requires communication between the agents it allows for more interesting areas to be looked at by more agents.

The central common search state paradigm follows the dynamic partition concept. In this paradigm there is a central agent that provides work for the search agents. The central agent sends to each search agent different sets of transitions that are to be processed. The agents will then process these transitions in their local state and send that information back to the central agent to be integrated into its search state. Once integrated, the central agent continues to determine which areas in its state look most promising and have the agents process those areas. This process is continued until no further transitions are available or a solutions has been found.

In this paradigm, \( K\) is accessible by all the search agents, since each agent must be able to retrieve work from the central agent. The start agent gets the initial start state from the search instance and writes it to \( K\). The end agent determines whether the search is finished by looking into the common search state.

All search agents contribute to the final solution, but the number of agents can scale to as many processing units as are available. Unlike other paradigms where the agents have to change their control or model to find different derivations for the search to be successful, distribution via a central common state does not have such a requirement. Even with the same search model and control, each agent can still contribute to the final solution.

A disadvantage of the central common state is the potential communication overhead required for a non-shared-memory architecture. In this paradigm each agent must write and read from \( K\) the next set of transitions that it should process. This constant communication can result in the agents spending more time communicating than doing actual work. Also, when implementing this paradigm, the developer has to balance out the number of transitions an agent should process before integrating the result back into the central state. Agents that only process a single transition before integrating it back into the central state
tend to overwhelm the system with communication. However, if the agent processes too many transitions before integrating its solutions, it could be spending time in uninteresting parts of the search space.

Another disadvantage of the central common state paradigm is that the agents need to be homogeneous agents. In order for the paradigm to work each agent must be able to understand and process the states provided by the central agent. This requires that each of the agents is using the same search model as the central agent. Having homogeneous agents is not necessarily a bad thing, since it allows for scaling, however, this paradigm cannot benefit from the positive synergy exhibited in other paradigms with heterogeneous agents.

3.2.2 Dividing the Problem into Sub-problems

Using the dividing the problem instance into sub-problem paradigm, the start agent starts by dividing the problem instance into sub-problems and gives each search agent a sub-problem to work on. This paradigm also allows the agents to divide their instance into sub-problem instances if they are capable of doing so.

When dividing a problem instance into sub-problem instances it is not always possible to do so without dependencies between the sub-problem instances. To satisfy the dependencies, communication between the agents is necessary. The best communication structure for this paradigm is the blackboard where the data is accessible by all agents. When an agent finds a solution to a sub-problem instance it will write it to the blackboard, allowing other agents to check for any inconsistency in the dependencies. Sub-problem instances that can be divided without being dependent on other sub-problems would require no communication between the agents.

The biggest advantage of dividing a problem instance is the fact that there are no restrictions on the search agents. Unlike the central common search state paradigm where the agents have to use the same search model, agents in this paradigm do not have such a requirement. Each sub-problem instance can be solved using the ideal search model and search
process. This allows for agents that can optimally solve certain sub-problem instances. Of course, the onerous task is on the developer to find such an agent, assuming that one can be found.

One disadvantage of this paradigm is that some sub-problems instances are often not divided, or cannot be divided, evenly in computational terms. In this case, some agents would have much less work while other agents might have too much work. This places a bottle neck on the entire system as other agents will be waiting on the few agents that have much more work than they can handle.

Another disadvantage is that for sub-problem instances with dependencies, inconsistencies can result with the solutions the agents generate. When this occurs the agents need to be able to resolve these conflicts. In distributed systems the most common way to resolve conflicts is through negotiation, but this usually requires a lot of communication.

3.2.3 Improving on the Competition Approach (ICA)

Although only a single agent contributes to the final solution, nevertheless, the competition approach has good properties, such as allowing for heterogeneous agents, which makes it worthwhile to try and improve on it. The obvious improvement is to allow the search agents to share information with one another.

Figure 3.1: Agents A and B are homogenous agents. In this sample run A and B execute at the same time but no information is shared between the two agents. It takes A 4 transitions to arrive at the goal state, while B requires 6 transitions.
A major problem with search systems is the large amount of time spent processing unnecessary transitions because the correct transitions are unknown ahead of time. This problem is exacerbated in distributed systems, as there are now many agents processing unnecessary transitions and doing redundant work. Through the sharing of information between the agents, however, this redundancy can be reduced. Take for example two homogeneous agents, A and B. A and B both have the same search model but B’s search control has been modified so that it would choose the right most node while A will select left most nodes. The search run of A and B is shown in Figure 3.1. We can see that when both agents are executed at the same time with no information sharing between each other it takes agent A 4 transitions to arrive at the goal state while agent B requires 6 transitions.

Now, if the agents were allowed to share information, such as their search state, with each other, a possible search run might look like Figure 3.2. This time, after processing a transition, A and B both share their search state. After the first transition, B notices that a transition in A’s state is better and decides to process that transition instead. After processing the next transition, the two agents again share information. A now looks at B’s search state and decides to process a transition from B’s state instead. After processing the second transition both agents arrive at the goal state. So, instead of agents A and B processing 4 transitions and 6 transitions respectively, it now takes both agents 3 transitions to arrive at the goal state.

Although this is a contrived example, it does show that sharing information can result in less work for each agent. With information sharing, however, there is also a possibility that the information sent can lead an agent to taking longer than it would have otherwise. This can happen if, as a result of the information, the agent processes transitions that would lead it to a dead end resulting in wasted CPU cycles. However, even if the agent has processed unnecessary transitions, it would not have to process redundant transitions thanks to the search state sent by the other agent.
Figure 3.2: After each transition is process the search state is shared between the two agents. Having additional information allows each agent to select better transitions at each step of the search.

From the previous example, it can be seen that agents can send positive information to one another, that is information that suggests good areas in the search space to explore. There is another type of information that agents can send called negative information. This type of information does not identify the good solutions to explore but instead identifies characteristics that are definitely not part of the solution. The agent can use this negative information to avoid parts of the search space that exhibits these characteristics. While it is normal for a search model to be able to integrate only positive information (since a search state is usually intended for positive information), there are search models that can also integrate negative information.

In addition to sending positive and negative information to be included into the search state, it is also helpful for the search agents to send positive and negative information to be included into the search control. While the search state is interested in the different areas of the search space, the control is interested in information that will further improve it so it can select better transitions. In homogeneous systems, a search agent can send control parameters that either it has deemed to work well or that it knows not to work. Sharing control parameters do not apply in heterogeneous systems. Instead, agents can use information about the solution space to influence where it chooses to explore.

Sharing information in this way can easily cause an agent to spend much of its time
communicating instead of searching. Also, an agent might already have the information that is being sent or it is already in a better search area than the information that it is receiving. Referees are introduced to limit such wasted communication. The referee’s task is to filter out any unnecessary information before it is presented to the other agents. Since an agent both sends and receives information, it needs to have two types of referees: send and receive referees. When an agent wants to send information to another agent the information is first filtered by the agent’s send referee to ensure that only information of interest, from the sending agent’s point of view, is sent. This suggests that the send referees should keep track of what information has been sent to each of the other agents. The suggested way to manage this is for each agent to have one send referee for each of the other agents in the system.

Even though the send referee has already filtered out the information the remaining information might still not be necessary because the receiving agent might have found better information than the one that is sent. It is up to the receiving agent’s receive referee to filter out the remaining information that it receives. Only after being filtered by the receive referee is the information then presented to the agent.

One problem with sharing of information between heterogeneous agents is that the information being sent is not one that the receiving agent might recognize because of the different search models used by the agents. In this case the agent must be able to convert the information into its own search model. Once that is done the information can be integrated into the agent’s search state.

The biggest advantage of improving on the competition approach through information sharing is that it can produce positive synergy. That is, the results produced are better than the sum of all its parts. Another advantage is that existing sequential systems do not have to make much of an adjustment to be adapted into this paradigm. The work required by the developer would be to add the send and receive referees and conversion of the information
into something the agent understands.

A disadvantage can be the additional communication overhead in the system if the referees are not designed carefully. Having send referees that are too optimistic tends to send too much redundant information and overload the system with unnecessary communication. Another disadvantage is that it can be difficult to identify necessary information to be sent to each agent since each agent might require different things. Also, the conversion process might require another subsystem to be implemented.

3.3 Improving on the Competition Approach Paradigm: Formal Description

The definition of a distributed search system presented earlier was intended to be general, as to be able to describe a wide range of different paradigms for distributed search. In this section, the definitions will be refined to reflect the improving on the competition approach paradigm (ICA) to take into account such things as the referees and the particular way of sharing information.

3.3.1 Communication Structure

In the ICA paradigm, agents can send both negative and positive information to either the state or the control. Instead of a generic data area as defined in the last definition, the definition of a communication structure for the improving on the competition approach can be instantiated as follows.

Definition 3.3.1. Communication Structure ICA

Let $K_{ICA} = (D_{ps}^1, D_{pc}^1, D_{ns}^1, D_{nc}^1, ..., D_{ps}^n, D_{pc}^n, D_{ns}^n, D_{nc}^n)$ be a communication structure. Each agent in $K_{ICA}$ has at minimum the following data areas: positive state (ps), positive control (pc), negative state (ns), and negative control (nc).

The communication structure of the improving on the competition approach paradigm is very similar to the general definition of $K_{ICA}$. The difference is that each agent needs at
minimum 4 different data areas. These different data areas allow each agent to receive the different types of information required by the paradigm.

3.3.2 Search Agent

A search agent in the ICA paradigm functions similar to that of an agent of the competition paradigm with the addition that at certain points during the search the agent will pause the search and the send referee will take over. At other points in the search process the receive referee might receive information from other agents. Again, the search is paused and the receive referee will filter the incoming information and integrate it into the search state or search control. This change is reflected in the modified definition of the search agent.

**Definition 3.3.2. Search Agent ICA**

\[ AgICA = (PICA, KomICA, mesICA) \]

extends the search agent as follows:

- \( mesICA \) needs to filter the information before it can be sent to an agent.
- The send referee for every agent \( Ag_{j,i} \) realized within \( mesICA \), can write to the data areas \( D_{jpc}, D_{jpu}, D_{jne}, D_{jns}, KomICA \).
- the search control of \( AgICA \) is responsible for handling the information received from other agents. It needs to first filter the information (using a receive referee) before it can be integrated.

Each ICA search agent has a single receive referee that is responsible for filtering messages received. Each agent, however, may have as many send referees as there are other agents in the system. An agent’s send referee may only access the data area of the specific agent that it is responsible for sending data to.

3.3.3 Distributed Search System ICA

The definition of a distributed search system for the ICA paradigm does not differ from that of the original definition. However, since the agents can be heterogeneous, the start
agent needs to be able to provide an initial start state and goal state in a format that is understandable to the receiving agent. Similarly the end agent needs to be able to understand the derivations of each of the agents participating in the search to be able to produce a final solution.

3.3.4 Time Frame and Search Course Protocol

Like the definition of the distributed search system, there is no need to extend the definition of the time frame and search course protocol. The time frames still record each time the agents change their states. Along with the local changes, it might be a good idea for the search course protocol to keep track of the information that has been received and sent, so as to allow the developers to have a better understanding of the search run. However, this can potentially be dangerous as it would increase the size of the log (in terms of memory and disk space) if there are many agents sharing information very frequently.

3.4 Changes to the Programmatic Interface of ICA

Figure 3.3: The Information interface represents the data that is sent between the agents. This interface is bare of methods and properties because it is dependent on the concrete problem. The information interface is further divided into the four types of information: PositiveControlInformation, NegativeControlInformation, PositiveStateInformation, and NegativeStateInformation. A search system does not need to support all types of information, only the ones that are supported by the agent.

The improving on the competition approach paradigm has added the concept of shar-
ing information through the send and receive referees. Our library now needs to support this information sharing concept. One new interface that was added to our library is the `Information` interface. However, there is not a single type of information but instead four different types. This required us to create additional information interfaces and have them inherit from the `Information` interface, as shown in Figure 3.3. Having more specific information allows us to determine the type of information we are dealing with, as we are required to handle different information types differently. The `Information` interface defines no methods since the information that is sent between agents is dependent on the concrete problem.

The most important distinction between ICA and other paradigms are the send and receive referees. Based on the extended definitions, the send referee is a part of the communication function while the receive referee is a part of the search control. In our library, we have similarly separated the roles of the send and receive referees. The responsibility of the send referee has been separated into its own interface (Figure 3.4), while the responsibilities of the receive referee have been added to the search control (Figure 3.5) interface.

Figure 3.4: The `SendReferee` looks at the current search state and control to determine whether or not new information should be sent to its respective agent. The information that is sent over should be information that it believes to be of benefit to the receiving agent.

<table>
<thead>
<tr>
<th><code>SendReferee</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>getPositiveControlInformation(control: SearchControl); PositiveControlInformation</td>
</tr>
<tr>
<td>getNegativeControlInformation(control: SearchControl); NegativeControlInformation</td>
</tr>
<tr>
<td>getPositiveStateInformation(state: SearchState); PositiveStateInformation</td>
</tr>
<tr>
<td>getNegativeStateInformation(state: SearchState); NegativeStateInformation</td>
</tr>
</tbody>
</table>

The send referee is responsible for filtering what it deems relevant to the receiving agent, along with converting the information to something the agent would understand. This is handled by the four `getInformation` methods. The other responsibility of the send referee is to send the information to the receiving agent. This process is taken care of internally by our library (the components that make up our library are discussed in the next chapter), such that the developer does not need to deal with it. Although not explicitly defined, to
Figure 3.5: The **SearchControl** is now responsible for handling information (both positive and negative state and control information) received from other agent’s send referee. In order to understand the information sent the receive referee must first filter the information through the corresponding filter methods.

To reduce the amount of duplicate information being sent, the concrete send referee should be recording which information has been sent.

The programmatic interface of the search control (Figure 3.5) can filter both positive and negative control and state information through its respective filter methods. For control information, the search control can use its own internal information to filter out redundant and unnecessary information. Once the control information has been filtered, that information will be integrated into the search control through the search controls `integratePositiveControlResult` and `integratedNegativeControlResult` methods.

Figure 3.6: The programmatic interface of the **SearchState** has been modified from its sequential form to allow for integration of positive and negative information filtered by the **SearchControl**. Since the integration of positive and negative results needs to be handled differently the integration is divided into two methods `integratePositiveResult` and `integrateNegativeResult`.

The search control is also responsible for filtering received search state information. To filter this information, the search control requires both the information and the search state. Once the control has finished filtering the positive and negative search state information,
the FilterResult will be passed to the search state to be integrated. Figure 3.6 shows the new extended search state interface. In order for the search state to integrate the filtered information, two methods have been added to the interface of the search state. The integratePositiveResult method allows for the integration of positive information and the integrateNegativeResult method allows for integration of the negative information.
The previous chapters have focused much on the search components of both sequential and distributed search systems. However, although the search component is the core of the distributed system, there are also many sub-systems that are required for a distributed search system to function. Figure 4.1 shows the different components that make up DisSLib:ICA. The green bars identify how much effort is required of the developer with regards to each component. Since the search component only provides skeleton interfaces that a developer needs to extend from in order to fit the library to his problem, most of the effort required of a developer using DisSLib:ICA will be towards the search component.

Figure 4.1: The 6 components that make up DisSLib:ICA. The bars represent how much effort is required of the developer for each of the components.

<table>
<thead>
<tr>
<th>Search Component</th>
<th>Communication Component</th>
<th>Dispatcher Component</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging Component</td>
<td>Event Component</td>
<td>Configuration Component</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first step of an ICA distributed search is the handshake process. This is where the search agents establish connections to each other with the help of the start agent ($A_{gs}$).
The search agents first connect to $A_g$ to ask for the list of all active agents. Once the agents receive the list they will connect to the other agents on the list. Once connected, the agents will need to wait for $A_g$ to inform them of when to start the search process, since other agents might not have connected yet. In order for $A_g$ to know how many agents will be connecting the developer must provide this information to the system. This can be done through configuration files. The agents participating in the search can be written in the configuration file that $A_g$ will read when starting up. This would allow developers to change the agents without having to recompile the search system.

The agents can begin their search processes once they have connected to all other search agents and $A_g$ has sent them the search problem instance. The search processes of the agents are similar to that of a sequential system. The search control selects a transition from the search state to process and integrates the result back into the search state. What is different is that at any point during the search process the search can be interrupted for the send referees to select and filter information to be sent to other agents. A send referee will execute at specific intervals set by the user. If, after filtering, the send referee has no information to send then the search process will continue from where it left off.

Also, at any time during the search process a search agent can receive information from another search agent’s send referee. Similar to a send referee, if the receive referee determines after filtering that there is no information worth integrating then the search process will continue where it has left off. However, if there is information, then the search process must first stop so that the information can be integrated into the agent’s search state or used to modify the search control. Once integrated, the search can continue processing where it has left off, but taking the new information into account.

When the search agent has identified that it has finished the search process (either by finding a solution, identifying that there is no solution, or the time has run out) it will let the end agent ($A_{gE}$) know that it has finished. The end agent then notifies all the other
search agents so that they can stop their search processes. Finally, the search run can be recorded to allow the users to later review the search.

From the sequence of events that has just been described, we can identify that a configuration component is required. This component will allow the users to configure which agents are participating in the search and where the start and end agents can be found.

There must also be a communication component in place to facilitate communication between the agents along the physical network. It should have support for both synchronous and asynchronous communication. With synchronous communication, the sender would wait for an acknowledged or response message from the receiver before it can continue. The handshake process requires a synchronous communication because all the agents need to start at the same time. The synchronous communication model, however, does not work for the sharing of information between agents because we do not want the sending agent to have to wait for a response. Similarly, we do not want the receiving agent to have to interrupt its search process and deal with the information if they do not need to. With asynchronous messaging the agents do not have to stop their search. Instead the sender will send the message without waiting for a response and continue with its search. The receiving agent also does not have to respond to the message, instead, it can take note that it has received a message, queues it, and continue on with its search. When it is free, it can then handle all the messages that it has received.

In a large system like a distributed search system many things can go wrong. To quickly find out why, the system needs to have a robust logging component in place. The logging system needs to be able to log different types of messages including debugging messages, warnings, errors, etc. To prevent too many messages overloading the console the messages should only be visible depending on the filters set by the developer or user. For example, it should show debug messages only when debugging, but hides them when running in production.
Building all these components in order to support the search process requires a lot of work from the developer. Luckily these components are not dependent on the problem instance but are required in all distributed search systems. This means that they can be generalized and built into our Distributed Search Library based on Improving on the Competition Approach (DisSLib:ICA) so that developers will not have to be concerned with developing them.

In the following sections we will take a look at all the different system components that the library has built-in to support the search operation. This includes looking at the core search components of DisSLib:ICA, how the system can be configured, the underlying network protocol used to support the communication between the agents, the event system involved to allow for asynchronous and synchronous message passing, the dispatcher system that prevents multi-threaded problems, and the logging of the messages and errors.

4.1 Search Component

The core search component consists of classes and interfaces that allow a search agent to function. The search control, search state, search transition, search observer, search problem instance, and search course protocol that were introduced in the previous chapters make up part of the core search components. These interfaces, however, are bare-bone and do not contain any logging or configuration components that are now commonly expected in a search system. This can prove to be a lot of redundant work for the developer to implement. Instead, the library provides abstract classes that have already implemented these search interfaces and provides features most developers would want. Listing 4.1 shows the AbstractSearchControl class that extends the SearchComponent class and implements the SearchControl interface. The Component class has references to the configuration and logging component (discussed in detail in later sections). The configuration object allows the search component to read its configuration files. For example, one might place the total time a search should run in a configuration file which then the search observer can retrieve through
the configuration object. The logging object allows the developers to log information about
the search that they might find useful.

**Listing 4.1: The AbstractSearchControl class**

```java
public interface SearchControl {
    SearchTransition getNextTransition(SearchState s);
    SearchResult process(SearchTransition t);
    ...
}

public class Component {
    protected Configuration config;
    protected Logger logger;
    public Component (){
        logger = Logger.getLogger(this.getClass());
        String configFolderPath = SearchApplication.sharedInstance().
            getConfigurationFolderPath();
        this.config = new ConfigurationLoader(configFolderPath).
            loadConfiguration(this.getClass());
    }
    ...
}

public abstract class AbstractSearchControl extends Component implements
    SearchControl{
    public abstract SearchTransition getNextTransition(SearchState s);
    public abstract SearchResult process(SearchTransition t);
    ...
}
```

In a sequential search, a search agent’s time is mostly spent in a loop (we refer to this as
the search loop) that consists of the following steps: select a transition, process the selected
transition, integrate the result back into the search state, and determine (from the search
state) whether the search is finished. The sequential search loop is depicted in Figure 4.2.

Within DisSLib:ICA we refer to these steps that a search agent performs as a task and it
is represented by a Task object, an object that holds actions to be performed. Each task is
atomic, that is, the agent will not stop and perform some other task until the current task is
finished. The entire search process is handled by the library through the SearchPerformer
class. In most cases, the tasks are created and added to the loop as the result of the previous
task. For example, the result of a search control performing its `selectTransition`
method is a **Transition** object. When the search performer sees this it will automatically create a new
task that will have the search control process said transaction.
Figure 4.2: A sequential search run is made up of a loop that consists of 4 main actions.

To support information sharing, as specified by the ICA paradigm, additional actions need to be added to the search agent. The first action is the sending of the four types of information by the send referee(s). At a specified time (this value can be set in a configuration file), the search performer will add a task that will execute the `filterInformation` methods on the send referee object (depicted in Figure 4.3). At this point, the current action must first finish before the send action can start.

The other action that an ICA system needs is the filtering of received messages by the search control. This action is scheduled by the messaging component (discussed in Section 4.3) when it receives information from other agents. This action can only be performed once the search round is finished (i.e., after step 4).

If, instead, the agent is not interested in processing the received action yet, it can reschedule it to be processed later by setting the Handled property of the action to false, as shown in Listing 4.2. When the handled property is set to false, the search performer will reschedule the action to the end of the action queue. Normally, this means that it will be scheduled after the next scheduled task. If instead the developer wants to guarantee that the action will be rescheduled after a search round has finished he can set the specific delay amount...
Figure 4.3: The send and receive referee actions can occur at any point during the search loop.

with the `setDelay` method. All of these interactions between the search components are handled by the library through the `SearchPerformer`. The developer only needs to focus on the implementation of the abstract search components.

Listing 4.2: Actions can be delayed by setting the handled property to false.

```java
public void handleReceivedPositiveEvent (PositiveInfoReceivedEventArgs e) {
    // tell the searchPerformer that this action hasn’t been dealt with
    e.setHandled(false);
    // reschedule to the end of the search round
    e.setDelay(EventDelay.EndOfSearchRound);
}
```

The start agent is an abstract class that is required in order to allow search agents to discover one another and to initiate the search. Most of the functionality of the start agent has already been implemented for the developer with the exception of the `getProblemInstance` method.

At the beginning of a search run, $Ags$ waits for search agents to connect to it. When a search agent connects, the start agent will establish a two-way channel with the search agent. After the connection has been established, $Ags$ will send the newly connected search agent the connection information of all existing search agents. When a search agent receives this connection list, it will try to establish a two-way channel with each of the agents on the list. In the end, the connection diagram of a three agent system will look similar to Figure 4.4. Once all the agents have connected to the start agent and to each other, the start agent will get the problem instance through the `getProblemInstance` method that is required to be
implemented by the developer.

Figure 4.4: The communication channel of a three agent system.

The *getProblemInstance* method is specific to the search problem since the problem instance can come from any source, such as database, text file, etc... Once the problem instance has been retrieved, \( A_g_S \) will send the problem instance to all connected search agents. The search agents will then use their search problem instance object to create the initial start state. \( A_g_S \) does not create the initial start state because of the fact that agents can be heterogeneous. If \( A_g_S \) was to create the start state, it would be required to know how to create the start state of each search agent in the system. This would add unnecessary complexity to the start agent. Instead, since the agents themselves already know how to generate their own start state (through their search problem instance), the start agent is only responsible for retrieving the problem instance and passing it to the search agent. After the agents have received the OK message to start the search they can safely disconnect their communication channel with \( A_g_S \).

The last piece of the core search component is the end agent (\( A_g_E \)). Like \( A_g_S \), \( A_g_E \) is also an abstract class that has all of the communication methods already implemented except one. The single method that needs to be implemented by the developer is the *getFinalSolution* method with all the states from the search agents passed in as a parameter. It is up to the developer to figure out how to determine the final solution from the search agent’s search state since it is problem specific. Once an agent is finished, it will inform \( A_g_E \) which will
then inform all the other agents asking them to report back to it with their final state. At this point if an agent is not finished yet, it can continue working until it is finished. It is up to the developer on how to implement the behavior of the agent. However, $A_{gE}$ will only start processing the final solution once all the agents have reported back to it.

### 4.2 Configuration Component

The configuration component allows the developer to easily configure the system through the use of configuration files. The main advantage of using configuration files is that changes can be made without having to recompile the system. This allows the developer and users to test different settings without having to change the code. Listing 4.3 shows the entire app.config configuration file for a two agent search system. The format of a setting in the configuration file is a key-value pair. The name of the setting is on the left and the value on the right separated by an equal sign.

```
standalone=false
Ags.port=20010
Ags.host=192.168.4.5
agent.id=BB
agent.id=GA
BB.searchProblemInstance=branchBound.BBSearchProblemInstance
BB.searchObserver=branchBound.BBSearchObserver
BB.searchControl=branchBound.BBDeepSearchControl
GA.searchProblemInstance=set.domain.GASearchProblemInstance
GA.searchObserver=set.domain.GASearchObserver
GA.searchControl=set.domain.GASearchControl
BB.numSendReferee=1
BB.sendReferee1=branchBound.BBSendReferee
```

Listing 4.3: The app.config file of a two agent search system.
The first line in the file describes whether or not the search should run as a standalone search. This setting effectively makes the search a sequential search. It might seem odd at first to want to run a search sequentially in a distributed search system but there are many benefits to having this capability. A distributed search is quite complex and communication between agents can add a lot of noise to the search run thereby making it hard for developers to identify what is going on. Having the ability to run each search agent as a sequential search allows developers to identify that the agent is producing the expected result sequentially. Developers with an existing sequential search system can also use this feature to verify that their converted distributed system produces the same sequential results as their old sequential system.

Lines 3 and 4 describe how the agents can connect to $\mathcal{A}_G$. When an agent first starts, it reads the configuration file and looks for the network information of $\mathcal{A}_G$. It will then try to connect to the start agent to get the list of agents currently participating in the system and also the problem instance. When the agent gets a list of all the participating agents it will also try to connect to the agents to ensure that a connection can be opened.

Following the $\mathcal{A}_G$ connection information are the settings describing the search agents involved in the search run and their unique identifiers (lines 6 and 7). Each agent requires a unique identifier in order for the configuration system to know which agent the settings are referring to. The identifiers also allow the agents to know which agent they are communicating with. The identifier is initialized with the format agent.id=$[agent\_name]$ where $agent\_name$ is the unique name of the agent. The setting $agent.id = BB$ lets the configu-
ration system know that there is an agent called \textit{BB}. The settings that follow will be based on the agent id with the format \textit{[agent id].[setting]=[value]}. In addition, \textit{Ag}s will also use these identifiers to know how many agents are involved in the search run.

The settings that need to be defined next are the ones that describe where the agent can find its search components. When a developer builds a search system using the library, the search components (i.e., the search problem instance, search control, search observer, etc.) can be named anything as long as it implements the predefined interface. Lines 9-15 in Listing 4.3 show an example of configuring the search components of two agents, one a branch and bound agent and the other a genetic algorithm agent. It might seem odd that the search state and other search components are not defined but remember that the \textit{SearchProblemInstance} is responsible for generating the initial start state based on the problem instance given.

The remaining settings involve the send and receive referees. In the improving on the competition approach paradigm, each agent might have as many referees as there are other agents. This allows the send referees to be tailored to specific agents. The first setting (lines 18 and 22) therefore is the number of send referees the agent has. The next setting is the class that is associated with that send referee (lines 19 and 23). The value of the send referee that the system expects here is the full Java package name followed by the class name. If there are more than 1 send referee, the next send referee settings are named \textit{sendReferee2}, \textit{sendReferee3}, and so on. The \textit{to} setting (lines 20 and 24) in \textit{Ag1.sendReferee1.to} is used to refer to which agent this send referee will be sending to.

The last two configuration settings configure the intervals of the send referees. The interval values are in terms of milliseconds. These settings can be set for all send referees or fine tuned for a specific send referee as shown in lines 29 and 30. \textit{GA.sendRefereeInterval} = 10000 configures all the send referees of the GA agent to run every 10,000 milliseconds. \textit{BB.sendReferee1.Interval} = 5000 on the other hand sets only the first send referee to
have an interval of 5000. If there are other send referees then their intervals would have to be individually configured. If the developer forgets to set an interval for a referee the system will report an error when it tries to run.

Having the send referee share information at an opportune interval can play an important role in the overall speed of the search. In DisSLib:CC [Ken10], it has been shown that the number of transitions processed before integrating back into the central common search state can affect the speed of the search. Integrating only after a few transitions have been processed can result in a slower search because the agents spend much of the time communicating with the central search state rather than working. Yet, having the search state process too many transitions can likewise result in a poor search run because the information the agent is working with might be too stale. It is up to the developer to experiment with the settings to determine the optimal results since each problem may require a different number of transitions. As we shall see in later chapters, the time interval for having the send referees filter and send information similarly plays an important role in how fast the search performs.

Listing 4.4: A configuration file and the corresponding class file for the GASearchObserver.

The class file can read the values in the configuration by calling the getConfig() method.

```java
// GASearchObserver.config
MaxTime=600

// GASearchObserver.java
package com.ga.search;
public class GASearchObserver extends Component implements SearchObserver {
    private int maxTime;
    public GASearchObserver() {
        maxTime=this.getConfig().getInt("MaxTime");
    }
}
```

So far, we have only discussed the configurations of the system but not of the search components. Sometimes, the search components need to be able to be configured without
having the values hard coded. For example, sometimes we have criteria of finding a solution within an allotted amount of time. In order to do so, the SearchObserver (the SearchObserver determines whether the agent can continue the search or not) needs to know the allotted time it has. Again the developer can provide this information in the code itself but a better way is to store it as a setting parameter in a configuration file.

To do so, the developer can create a configuration file for just that one particular search component that needs to be configured. In Listing 4.4 we see that in the SearchObserver’s configuration file (GASearchObserver.config) there is a setting called MaxTime with the value of 600. This setting describes the maximum time the search should run in terms of seconds. The GASearchObserver can then read the value in the configuration by calling the getConfig() method which gets the underlying configuration object and call getInt(“MaxTime”) to retrieve the value. The GASearchObserver is able to call the getConfig() method because it extends the Component class.

The Component class is doing the actual work of loading the configuration file and providing the getConfig() method to its implementer. In order for the Component to find the right configuration file, it first looks for the file with the same name as its implementer class. In Listing 4.4 the GASearchObservers’s configuration file is specifically named GASearchObserver.config because the Component is expecting a file with the same name as the class GASearchObserver. When there are multiple classes with the same name, the configuration file must be named with the full qualifying class name, that is, the GASearchObserver’s configuration file must be named com.ga.search.GASearchObserver.config if there are multiple GASearchObserver classes.

Not only can the Component be used by the classes in the library but also any classes that the developer has written as long as it extends the Component class. This effectively allows any object to have its own configuration file and to be able to read settings from it without having to write any additional code.
4.3 Network Communication Component

On the physical level, the network communication component of DisSLib:ICA uses the TCP/IP network protocol for communicating across the physical network. TCP/IP is chosen for its guaranteed delivery of information as well as its wide adoption. This communication is done through sending serialized Java objects. If the library were to send plain text we would require that the developer be able to take the information that he wishes to send and convert it in such a way as to be able to describe that information in plain text only. Since the information being sent is quite large and can vary greatly, the task would be quite onerous on the developer. Instead, messages are first serialized into byte form and then sent across the network. The recipient then de-serialize the byte information back into a Java object.

The communication model in DisSLib:ICA is based on the real world mail delivery system. For example, each agent has its own mailbox where it picks up mail from. When agents want to send a message, they create an envelope that contains the names of the agents they want to send to along with the contents of the message. All contents of the message are based on some Event (events will be discussed in the next section). Once the envelope has been created, the agent can give that envelope to the mail man which will then deliver the envelope to the addressee’s mailbox. When delivered the recipients are notified by the mail man that a new message has arrived for them. At this point the agent can choose to go and pick up the mail or continue on with its own task and let the mail pile up.

When calling for the mail man to deliver messages, the agents can tell the mail man to deliver the message asynchronously or synchronously. Asynchronous communication means that the agent will send a message and continue on with its search. We can think of it as having the agent drop a letter into the post office box and continue with its own task. Synchronous means that the agent will wait (not perform any search task) until a response is received or until a time-out has occurred. This is similar to having the agent go into a postal
Figure 4.5: A class diagram showing the relationships of the classes that make up the mail delivery system.

office, to send a package and waits there until a response is received. When the mail man delivers a synchronous message it no longer treats it as a mail to be dropped off but instead a package to be delivered straight to the recipients door. This ensures that the sender will not have to wait long as this message cannot be ignored by the recipient. When receiving a synchronous message, the agent must stop its current activity and respond accordingly to the message. The mail man then delivers the response to the agent that is waiting. Synchronous messages help agents to coordinate group activities since it ensures that no other task can be performed until a response is received.

An example of when synchronous communication between the agents is required is during the handshake process. When an agent first starts up, it will connect to the start agent (\(A_{GS}\)) and sends its unique identifier. This is a synchronous message since the agent must wait for a response from \(A_{GS}\) before it can begin the search. When \(A_{GS}\) receives a connect message it will verify that the identifier is indeed from an expected agent based on its list of agents (read from configuration file). Once verified, \(A_{GS}\) will reply to the agent with the problem instance that it needs to generate the initial start state from, along with the list of all the existing agents. The search agent then tries to connect to all the agents in the list and sends
them its unique identifier and connection information. Once connected, the search agent will send the start agent a synchronous message notifying that it has received the data and is ready to start the search. When all the search agents have notified the start agent, the start agent will reply to all the agents (which are still waiting for a response) notifying them that they can begin the search. It is only when the agents receive this start message that they can begin the search.

It should be noted that once a connection has been made, it remains open at all times throughout the search. If an agent fails for any reason then all the other agents that it has an active connection with will know right away and can remove the agent from any further consideration.

4.4 Event Component

The library can be said to be event based. That is, every action that occurs in the system has a corresponding event that can be listened in by an observer. For example, when an agent receives information from another agent it will notify that a received information event has occurred, thus alerting all the listeners that are listening on that event. This event system is a well known software design pattern described in [GHJV95], known as the Observer Pattern.

In this section, we will look at how events are implemented and how developers can create their own events. Figure 4.6 shows the class diagram for the event component. Each event in the system has a unique event name and associated event arguments. An argument contains relevant information regarding the event. For example, a SystemEventArgument contains information regarding the current state of the system.

The EventManager is the main object responsible for both registering anEventListener to listen for specific events and notifying the listeners that the event has occurred. In order for an object to register an event with the event manager, the event listener interface needs to be implemented by the object. The EventListener interface has only a single method
with the signature of \texttt{handleEvent(sender: Object, event: Event)}. The implementer of the event listener contract is expected to know how to handle the specific event that it is listening to when that event is triggered. The sender parameter allows the event listener to identify the object that has triggered the event.

Once an object has implemented the \texttt{EventListener} interface, it can register itself to listen to that event by calling the static method \texttt{registerListener(type: Event, listener: EventListener)} from the event manager. When the \texttt{registerListener} method is called, the event manager first looks at the event’s unique name to determine if a collection of listeners has already been created. If it has, it will add the object registering to its collection of listeners to notify. Otherwise it will create a new collection of listeners for that specific event and add the listener to that collection. This allows multiple listeners to listen in on the same event.

When an object wants to notify listeners that an event has occurred it can call on the method \texttt{notifyEvent(sender: object, event: Event)} from the event manager. It will pass itself in as the sender and create an event for which it wants to notify the listeners. The event manager will then find all the listeners listening on the event in its events collection, notify each listener by calling their \texttt{handleEvent} method and passing in the sender and the
event as parameters for the method.

Since the `handleEvent` method can be called from any thread there can be potential multi-threading issues if the event manager calls the listeners `handleEvent` method from a thread other than the main thread. To solve this problem, the default behaviour of the event manager is to call the listener’s `handleEvent` method by dispatching it to the main search thread to ensure that there is no problem with thread safety.

In the previous section it was said that each message sent contains an event as the content of the message. In fact two events are triggered when a message is sent. When the mail man delivers a message it is sent to the mailbox. Whenever a message is added to the mailbox it triggers a `MessageReceivedEvent`. The envelope is stored in the event argument. The listener for the `MessageReceivedEvent` event is the agent. When this event is fired the agent looks at the content of the message and determines whether or not it wants to deal with the content of the message at that moment. If it does want to deal with the event, it will notify the respective event listener that the event has occurred and the corresponding event listener will handle the message.

The library comes with many built-in events and event listeners. These include the `SystemEvent`, `SendRefereeEvent`, `ReceiveRefereeEvent`, `TimerStartEvent`, `TimerFinishedEvent` and their corresponding listeners. The `SystemEvent` occurs whenever the state of the system changes, i.e., when the system first starts up and when agents connect. The `SendRefereeEvent` is when agents receive information from another agent’s send referee. The information itself is contained in the event arguments of the send referee event. The `ReceiveRefereeEvent` occurs when the receive referee filters information received from other agents’ send referees.

The `TimerStart` and `TimerFinished` events are a bit different from the other events. These two events are used when users needs to be notified after a certain time has passed. When the search system first starts up, the respective timer event listeners are created and registered with the event manager to listen in on `TimerStartEvents`. To begin a countdown
timer, an object must notify that a **TimerStartEvent** has occurred and pass in the time value as part of the argument. When the library’s built-in timer start listener object sees that a start event has occurred it will trigger the count-down timer to count down from the time that has been passed in as the argument. When that time has been reached, the listener will notify that the **TimerFinishedEvent** has occurred and pass in the object that first started the event as part of the argument. Although there may be multiple listeners listening on the timer finished event, the listeners can tell whether their timer is finished by checking against sender of the event. If the agent is the sender then it means that it its event is finished.

In addition to the built-in events and event listeners, developers can also create their own events. To create an event, a class must extend the abstract Event class and implement the **getName** method which is used by the event manager to identify the events between one and another. Along with the event, the **EventArgs** associated with that event also have to be defined. Finally, the developer will need to implement the **EventListener** to listen in on the specific event.

### 4.5 Dispatcher Component

Agents can send messages at any time during the search process. In a synchronous system, the entire search would have to stop in order for the agents to be able to receive the message and process it. Also, if multiple agents send information at the same time, the system would have to stop and process each message in the order received before the search can continue. This can often be a bottleneck in the search system since receiving information and processing it can take quite a while and it may be the case that the information received is not relevant thereby wasting time that would have otherwise be better spent on performing the search.

Instead, what is preferred, and what the library uses, is an asynchronous communication
system where the messages received can be dealt with without interrupting the search process. This is achieved by having the communication between the agents run on a separate thread apart from the main search process. Each agent then has two threads at any given time, the main search thread and the communication thread. The communication thread, however, spends most of its time blocked (that is, it will not use any CPU time unless there is actual communication to send).

The communication thread is managed by the message delivery system described in previous sections. Thus the search process of the agent runs on the main search thread and all the communication is performed on a separate thread allowing the system to receive and process messages without having to interrupt the main search. However, a multi-threaded search system can cause problems like race conditions and deadlocks that can be difficult to debug.

To solve the problem of thread-safety the library introduces the *Dispatcher*. Most problems with multi-threaded applications lie in the fact that multiple threads are accessing shared, mutable data. If these were to be accessed on a single thread then no problems would arise. This is the purpose of the dispatcher: to dispatch tasks on to the main search thread. This method of solving thread-safety issues is used in many graphical frameworks such as Java’s Swing [ELW98], Windows Forms [GA03], and WPF [Nat06]. The search process is executed sequentially on the main thread (managed by the dispatcher) thus guaranteeing that each action performed on the search process will run on the same thread. However, problems can arise if another thread accesses any information from the main search thread.

For example, a search agent sends positive information regarding its search state. The mail delivery system receives this information (in its own thread running concurrently with the main search thread on the same CPU) and sends it to the event manager to notify the listeners that positive information has been received. The event manager then calls the `handleEvent` method of the listener and passes along the positive information. This
information gets passed to the receive referee (which is the listener of events sent from send referees) who then filters the information. To determine whether or not the information is relevant to the search the receive referee would need to look at the search state (while still in the messaging thread) thus potentially causing thread issues. To combat this, the dispatcher should be used to access any information from the search process.

The dispatcher contains two different methods for executing a task: synchronous and asynchronous. In a synchronous execution, the calling thread will wait until the task it has given to the dispatcher to execute is finished. Executing a task synchronously also allows the calling thread to be able to retrieve the results of the task when the task is finished. In an asynchronous execution, the calling thread will give the dispatcher the task to execute and continue on with its own execution not caring to wait for the task to finish. This is particularly useful when the calling thread is not interested in the result or the order of the task but only that the task is executed. Not having to wait for the task to finish increases the efficiency of the calling thread since it can execute in parallel to the main search thread.

Since the system makes use of at least two threads at any given time to support both the communication between the agents and the search process we suggest to reserve one core for the communication process between the search. For example, if we wish to distribute a search to a quad-core CPU we suggest to only distribute three agents and leave the extra core to handle the communication process. Another possible option is to reserve it for the start and end agent. These agents are only consume CPU resources at the start and end of the search, thus making them idea to run on the extra core.

4.6 Logging Component

Logging, along with the search derivation, is one of the key tools to allow the developer and users of the search system to understand what is going on during a search run. The Logging system that the library uses is a wrapper around the open source Apache Log4J project.
Log4J was chosen for its flexibility and customization. It allows logging to be done to the console, file, or both. In addition, it also provides a broad severity system where the developer can designate the log entry as an error, warning, etc.

Each component in the library has a logger built in. This includes all the search components, mail delivery system, and referees. This enables each one to be able to create a log file and log messages without having to write any code. The default logging configuration of the component is to log to both the console and a file. The file is named after the class name and stored in the logging folder named with the time-stamp of the search run.

Listing 4.5: A TestLogger class showing how to override the default logger settings.

```java
public class TestLogger extends Component{
    @Override
    protected void initializeLogger()
    {
        this.logger=createLogger("testfile.log",true);
    }
}
```

The developer is able to override the default logging behaviour by overriding the method `initializeLogger()` as seen in Listing 4.5. The developer may wish to override this method to be able to have each component logged to a single file. This will better enable the developer to see the interaction between all components as the search progresses. As we can see, the logger can be created by calling the `createLogger()` method of the Component class and specifying the file name and a boolean value specifying whether or not the file should be appended or created anew. To create a log entry, the `debug()`, `warn()`, `error()`, and `fatal()` methods can be called (each is called based on the severity) along with the message for that entry. Each log entry records the time-stamp of when the entry was created along with the name of the agent that created that entry and the message.

One advantage of using a popular open source library such as Log4J is that there is large community support already in place for that project. This means that developers coming
to use the library will most likely already be familiar with the Log4J API and will have no trouble using it. On the other hand, if the developer has not used it before, there are a lot of documentations and tutorials available along with a large community to provide support if the developer has any questions.

4.7 Agent Failure

Up to this point our assumption has been that agents do not fail. But in reality a well built, fault tolerant system is a rarity rather than the norm. Even well built systems can sometimes fail due to unforeseen external disruptions such as power or hardware failure. When this occurs we have to ask what happens to the remaining agents? If all the agents fail, then the system fails but what happens if only some of the agents fail? Does the search continue on or should the agents try to recover and re-join the search?

The ideal situation is to have the library to recover the agents when it has failed but, due to shortage of time, DisSLib:ICA has no built-in support to try to recover the agents. Luckily the improving on the competition approach paradigm does not require all the agents to work. When an agent fails other agents will know immediately since they maintain an open connection to all the agents. Once they see that a connection has been closed they will assume that agent is offline and remove it from their own internal agent list. As long as there are still agents remaining the search will continue.

Knowing all different components that make up the library and how they function will allow a developer to go in and change the functionality of the library if required. However, in day to day use, a developer would only concentrate on the components that are relevant to solving the search problem. The next chapter will describe how to use the library to build an ICA distributed search system. This will show all the classes and interfaces that will need to be implemented by a developer for the library to function.
Chapter 5

Using DisSLib:ICA

DisSLib:ICA was designed with the goal of reducing the complexity that is often associated with building a distributed search system for the improving on the competition approach paradigm. To that end, a lot of effort has gone into the design of DisSLib:ICA to separate the components that are relevant to the search problem from the components that are required to support a distributed search system. This separation allows the developers to focus solely on the components that are relevant to the search problem. This chapter will show how a developer can use DisSLib:ICA and which components of the library require the developer’s attention.

5.1 The Skeleton Classes

The current version of DisSLib:ICA consists of 102 classes and interfaces that make up the different components of DisSLib:ICA. These 102 classes perform the following functions for the developer:

- Handle all the communication between the agents. The developer does not need to write network code.

- Handle all multi-threaded programming for the developer. The Dispatcher allows the developer to not have to worry about handling multiple threads or managing shared resources between threads.

- Allows the developer to re-use the structure of a search system. The developer no longer needs to start from scratch. The library has already defined skeleton classes that are built from the basic definitions of search.
• Provide tools and utilities to speed up the development process. The library provides developers with tools for logging and configuring a search system.

• Build a distributed search system in a sequential manner. The developer does not need to learn to program in a distributed manner or learn new distribution paradigms to build a distributed search system.

In order for DisSLib:ICA to function, the developer needs to provide the library with the details of the problem relating to the search by extending the 12 skeleton classes. These 12 classes are: SearchProblemInstance, SearchState, SearchObserver, SearchTransition, TransitionResult, SearchProtocol, SendReferee, FilterResult, StartAgent, and EndAgent. Out of these 12 classes, 6 classes (the SearchControl, SendReferee, Information, FilterResult, StartAgent, and EndAgent) are relevant to the ICA paradigm and the rest are relevant to the search problem.

For the 12 skeleton classes, the methods that the developers need to implement are shown in Listing 5.1.

Listing 5.1: Methods that are required for a sequential search system.

```java
1 SearchState:
2   integrateResult(Transitionresult result);
3   selectTransition(SearchControl control);
4   integratePositiveStateResult(FilterResult result);
5   integrateNegativeStateResult(FilterResult result);
6
7 SearchControl:
8   getNextTransition(SearchState state);
9   processTransition(SearchTransition transition);
10  filterPositiveControlInformation(Information info);
11  filterNegativeControlInformation(Information info);
12  filterPositiveStateInformation(Information info, SearchState state);
13  filterNegativeStateInformation(Information info, SearchState state);
14  integratePositiveControlResult(FilterResult result);
15  integrateNegativeControlResult(FilterResult result);
16
17 SearchObserver:
```
As Listing 5.1 shows, for a DisSLib:ICA system to function, 24 methods are required to be implemented by the developer. Of these 24 methods, some will be empty depending on the problem and the data structure used. For example, a GA system cannot make use of negative information so the 6 methods relating to negative information (filterNegativeControlInformation, filterNegativeStateInformation, integrateNegativeControlResult, integrateNegativeStateResult, selectNegativeStateInformation, and selectNegativeControlInformation) will be empty.
5.2 Start with a Sequential Search System

In order to use DisSLib:ICA, the developer must first import the library. DisSLib:ICA is packaged with the Jar (Java ARchive) format to allow developers to easily import the library into their own project.

One of the advantages of using DisSLib:ICA to build a distributed search system is that the developers can build a distributed system in the same manner as they would build a sequential search system. A process that we found to help when building distributed systems using DisSLib:ICA is to start by implementing the classes that are relevant to the sequential search, test it for correctness and performance, and then add the classes relevant to the ICA paradigm.

When building a search system a search model must be decided on by the developer and implemented according to the skeleton structure. The search model consists of the search state and search transition along with any data structure relevant to the model. For example, a tree-based search would have a concept of a node data structure. When a model is chosen the developer needs to implement the search state and the integrateResult and selectTransition methods belonging to it. The search transition and search result data container classes must also be implemented as they are required by the integrateResult and selectTransition methods.

The next class to implement is the search control as it is referred to by the selectTransition method. The required methods for the search control consists of the getNextTransition and processTransition methods. The other methods that relate to the information sharing can be left empty at this point as we are only concerned with the sequential search.

Once the search control and search state have been implemented, the search problem instance and the search observer are the next classes that the developer needs to implement. The search problem instance consists of only the generateStartState method that creates the initial search state. Normally this problem instance is passed to the search agent from
the start agent but in a sequential system the search problem instance object must retrieve the problem instance itself. The other method that we need to define is the `canContinue` method in the search observer. This method determines whether a search state is a goal state or if the allotted time has been exceeded.

The final class that needs to be implemented is the search protocol. This class records the search state and selected transition at each round. Normally, these methods are not required as recording each transition and state can be too much information. However, the method `recordSearchEnd` should be implemented to determine the final solution. This method is called by the library after the search is finished.

Once these classes have been created the search system can be tested. To run the search system as a sequential system, we will need to create a configuration file and set the standalone flag to true. A sample configuration file is shown in Listing 5.2.

```
1 standalone=true
2
3 agent.id=Example1
4
5 Example1.searchProblemInstance/example.ExSearchProblemInstance
6 Example1.searchObserver/example.ExSearchObserver
7 Example1.searchControl/example.ExSearchControl
```

Listing 5.2: The app.config file of a sequential system.

After the sequential search system is verified to be functioning correctly, the developer should start implementing additional sequential search processes. As DisSLib:ICA is based on the competition approach the agents in DisSLib:ICA must compete against each other. To be competitive the agents should have different search controls or different search models. The simplest method to get additional agents is to modify the search control so that it explores different areas of the search space. For example, if a tree-based agent’s search control is set to always select the left most node, a simple modification of the search control can be to always select nodes on the right. This will allow the two agents to have different
views of the search space and traverse it rather differently.

We recommend creating additional sequential search systems first because not only does the distribution paradigm require it, but it is also much easier to test a sequential system rather than a distributed system. Since each search process acts as a separate entity, if we can show that the sequential system works then, if a problem arises later on with the distributed search, we can assume that it is a result of the information sharing and debug from there.

Listing 5.3: An existing search process of a sequential genetic algorithm search system.

```java
mgr = new SolutionManager();
List<Assignment> rouletteSolution = solution;
while (cpsc433.Environment.getTimeNotRunOut()) // Search Observer
{
    goodness.clear();
    for (int j = 0; j < 10; j++)
    {
        // selectTransition
        if (j > 6)
            mgr.addSolution(rouletteSolution);
        else mgr.addSolution(solution);

        HashMap<Lecture, Assignment> anIndividual = mgr.get(j);

        // processTransition
        swap(anIndividual);

        HashMap<String, Assignment> toSolMap = new HashMap<String, Assignment>();

        for (Entry<Lecture, Assignment> entry : anIndividual.entrySet())
            toSolMap.put(entry.getKey().toString(), entry.getValue());
        int goodnessValue = AppController.getInstance().getConstraintManager().getGoodness(toSolMap);
        goodness.add(new Pair<Integer, Integer>(goodnessValue, j));
    }
}
```
Wheel wheel = new Wheel(goodness, mgr);
rouletteSolution = wheel.pickSolution();
Collections.sort(goodness);
solution = new ArrayList<Assignment>(mgr.get(goodness.get(0).getValue()).values());
finalSolution = solution;
mgr.clear();

// integrateResult
mgr.addSolution(solution);
mgr.addSolution(rouletteSolution);
}

In the case that the developer already has a sequential system built and wants to use the library he will still repeat the same steps. However, instead of coding all the methods he can usually move the existing sequential system to the library by copying existing code with the same functionality to the corresponding method. For example, Listing 5.3 shows the search process of an existing sequential genetic algorithm [Mit98] search system. Looking at the code shows that the entire search process happens in the while loop. Looking at the code shows us that the search loop will continue until the time has run out (line 4) which corresponds to the canContinue method of the search observer. Lines 10 to 12 show individuals of the population (a solution is an individual in a GA search) being selected to be processed which corresponds to the selectTransition method of the search control. The swap method in line 17 corresponds to the processTransition method of the search control. The solution manager object corresponds to a search state and lines 36 and 37 are the integration of the processed result back into the search state. Knowing this, the developer can easily move these lines of code to the search state, search control, and search observer thus allowing the developer to reuse most of the code.
5.3 Sharing Information between Agents

With the sequential search processes implemented, the developer can direct his attention to implementing the classes relevant to the ICA paradigm. The first step, and often the hardest, is to decide what information should be shared between the agents. What to share is dependent on the search problem, search model, and the search control of each agent. In general, each search agent must consider the four different types of information that can be shared; positive and negative information sharing for both search control and search state. Not all the four types of information must be shared in each system. In fact, certain models cannot make use of the negative information and heterogeneous agents cannot share control information with each other, as they cannot make use of the other agent’s control information.

The next class to be implemented is the send referee. The send referee needs to get from the search state any information that it thinks might be of use for the other agents. This is handled by the `selectPositiveStateInformation`, `selectNegativeStateInformation`. The control information can also be shared by implementing the `selectPositiveControlInformation`, and `selectNegativeControlInformation` methods of the send referee.

To filter and integrate the information received the methods relating to the ICA paradigm of the search state and search control that were left empty previously must now be implemented. The search control contains the four filter methods (corresponding to the four types of information) that allows it to filter information that has been received. The `filterPositiveStateInformation` and `filterNegativeStateInformation` has the search state as an argument to allow the control to check the latest state information to see whether the information received is still relevant. If information is still relevant after it passes through the control’s filter, then it can be integrated into either the search state or the search control depending whether it is state information or control information.

To integrate the state information into the search state the `integratePositiveStateInfor-
motion and integrateNegativeStateInformation must be implemented by the developer. For control information, the integrate control methods must be implemented on the search control.

Before we can run the search we need to implement the start and end agents classes. The start agent is responsible for connecting the agents and distributing the problem instances. These have already been implemented by the library. The only component of the start agent that needs to be implemented by the developer is how to retrieve the problem instance through the getProblemInstance method. The problem instance can come in different forms, so it is up to the developer to implement the methods to get the problem instance.

The developer also needs to implement the getSolution method of the end agent class. This method will determine the solution from all the search agents. Once the search is finished the search agents will send the end agent their state so that the end agent can determine from all the search state the final solution to the search problem if one exists.

Listing 5.4: The app.config file of a distributed system.

```xml
<create>
    <standalone>false</standalone>

    <Aggs>
        <host>ares</host>
        <port>20010</port>
        <agent>example.SimpleStartAgent</agent>
    </Aggs>

    <Agent>
        <agent>example.SimpleEndAgent</agent>
        <host>ares</host>
        <port>20010</port>
    </Agent>

    <agent id=Example1_A1>
        <searchProblemInstance>example.Ag1SearchProblemInstance</searchProblemInstance>
        <searchObserver>example.Ag1SearchObserver</searchObserver>
        <searchControl>example.Ag1SearchControl</searchControl>
        <logging>true</logging>
        <numSendReferee>1</numSendReferee>
        <sendReferee>example.Ag1SendReferee</sendReferee>
    </agent>
</create>
```
Once these methods have been implemented, the configuration file will need to be updated to include the start and end agents along with the send referees and the send intervals, shown in Listing 5.4. After setting the correct configuration the developer can start the distributed search system. No further effort is required of the developer as all the other components required for the distributed search system to function are handled by the library.

### 5.4 Running the Distributed Search

When starting the search agents, the users will have the opportunity to designate one as a start agent or end agent. Of course, we can have the start and end agents run on their own processing unit but that would be a waste of resource as the two agents perform minimal computation. The user can designate which agent is to be the start and end agents through the command line -s and -e switches (for start agent and end agent respectively), as shown in Listing 5.5. The user must also ensure that the host and port of the start agent in the configuration file matches that with the host and port of the designated start and end agents so the other agents can find the start and end agents.

---

```plaintext
Example1_A1.sendReferee1.to=Example1_A2
Example1_A1.sendRefereeInterval=10000

Example1_A2.searchProblemInstance=example.Ag2SearchProblemInstance
Example1_A2.searchObserver=example.Ag2SearchObserver
Example1_A2.searchControl=example.Ag2SearchControl

Example1_A2.numSendReferee=1
Example1_A2.sendReferee1=example.Ag2SendReferee
Example1_A2.sendReferee1.to=Example1_A1
Example1_A2.sendRefereeInterval=10000
```

Figure 5.1: Class diagram of the start agent.
Listing 5.5: Executing agent with id of Example_A1 and designating this agent as both the start and end agent (\(-s\) and \(-e\) switch respectively). The port that the agent is using can also be specified with the \(-p\) switch.

```
java -classpath /path/to/search/bin:/path/to/DisSLib:ICA.jar com.ica.
    SearchRunner -a Example1_A1 -s -e -p 20010
```

This chapter has shown what is required of the developer to use DisSLib:ICA. The next chapter will take a look at three distributed search systems that have been built using DisSLib:ICA. It will also go into further detail about the amount of effort that is required in building distributed systems with the library.
Chapter 6

Implementation of Distributed Systems using

DisSlib:ICA: Case Studies

This chapter takes a look at how DisSLib:ICA was used to develop three different distributed search systems: an exam scheduling system, a course scheduling system, and a package delivery planning system. Each of these systems exhibits different scenarios a developer might approach the library with. In the case of the exam scheduling system, the developer has an existing sequential system that he wants to convert into a distributed search system. This case study shows that with relatively small amount of effort, we can convert an existing sequential system to a distributed system. The course scheduling system looks at how much effort is required when a developer wants to implement a distributed system from scratch using DisSLib:ICA. This case study is used to look at how developers can quickly get a distributed system up and running with the library. Finally, the package delivery planning system is a case study where a developer already has an existing distributed system (in this case one developed with the DisSlib:CC library), and wants to adapt the existing distributed system to another distributed system using a different paradigm. The package delivery planning system showcases the relative ease with which this can be accomplished with the library.

6.1 Exam Scheduling

At the end of each semester universities across the nation need to schedule time slots for students to write their exams in. The problem is difficult since there are very many different possible combinations of rooms and exam sessions. In addition, instructors often make
requests, such as having their exams at a certain time, that universities have to try to accommodate. The concrete specification of the exam scheduling problem is as follows:

- Each course can have multiple lectures and each lecture can be assigned to only one exam session (a course can have different instructors and different exams), and each exam session has a specific number of students that will write the exam.

- Each room has a specific number of students that can fit in the room and also is only available at certain time slots.

- The following hard constraints cannot be violated:
  - Every lecture is assigned an exam session.
  - No lecture is assigned more than one exam session.
  - The number of students writing an exam in a particular exam session may not exceed the capacity of the room.
  - Every lecture’s required time must be less than the session length.

Along with the hard constraints are soft constraints that determine the quality of a solution. For example, a 5 point penalty will be added to a possible solution for every professor who does not get his desired exam slot. The goal is to find the optimal solution where all the hard constraints are met and the number of soft-constraint penalties is minimal. This can be expressed as an objective function shown in Equation 6.1. The function minimizes the number of soft-constraint violations. Here, $n$ is the number of soft constraints and the function $f_i$ calculates the soft-constraint violation value of solution $s$ for soft constraint $i$.

$$F(s) = \min\left(\sum_{i=1}^{n} f_i(s)\right)$$  \hspace{1cm} (6.1)
As noted earlier, the exam scheduling system already exists as a sequential system. This system was developed by Hoang Dang and Jeremy Opalach in the course of a month as a final project for CPSC433, the Artificial Intelligence course at the University of Calgary taken in the Fall Semester of 2007. It has been fully tested and verified that it functions as expected by the course instructor.

The existing sequential system uses a genetic algorithm to solve the exam scheduling problem. Genetic algorithms [Mit98] are in the class of evolutionary algorithms and borrow from biology the idea of chromosomes and evolving the chromosomes until a satisfactory solution has been reached or a certain number of generations have been created. In order to apply genetic algorithms the solution needs to be fitted into a genetic representation. This is usually done as a binary string. Initially, a random set of solutions, known as individuals, are generated. The initial solution set, the population, is then processed either through mutation or crossover to create new solutions, as shown in Figure 6.1. Crossover forms new individuals by crossing the genes of two different parents. Mutation changes an individual randomly by changing some of its elements to form a new individual. If the solution is represented as a bit string the bits can be randomly reversed, 0 to 1 and 1 to 0, to create new bit strings. Each of the individuals in the population is then evaluated with a fitness function (for the exam scheduling problem the fitness function determines the total soft constraint violation value, as mentioned above). The best individuals will be selected for the next generation while poor individuals will be thrown out. This process is repeated until a certain number of generations has been evolved or the time limit has been reached.

Figure 6.1: Crossover and mutation operations in genetic algorithms
6.1.1 Existing Genetic Algorithm System

The sequential system starts with an initial population (the start state) and, through genetic operations, the population is transformed with each generation (ideally each generation should be progressively better than the last but this is often not the case). After a genetic operation is performed, each new individual produced must be checked to ensure that the hard constraints have not been violated. Individuals that violate the hard constraints can try to correct themselves by swapping exam sessions that have been violated. If an individual is still not valid it will be removed from the population. Each new individual is also given a fitness value calculated from the soft constraints that have been violated. Finally, if the population has exceeded the maximum population size, the worst performing individuals are removed from the population until the population member count is less than the maximum size. This entire process is repeated until the allotted time has run out.

The concrete problem instance is provided as a text file (Listing 6.1) that specifies the exams sessions, rooms, and time slots. The text file is read and parsed by the problem instance and the initial state is generated.


```
// Lectures ********************************
lecture (CPSC433,L01,Kremer,3)
lecture (CPSC433,L02,Kremer,2)
lecture (CPSC599.68,L01,Kremer,3)

// Students ****************************************
enrolled (Alice,[CPSC433,L02,CPSC599.68,L01])
enrolled (Bob,[CPSC433,L01,CPSC599.68,L01])
enrolled (Carol,[CPSC433,L01])

// Rooms **************************
capacity (JackSimpson,2)
capacity (RedGym ,2)
capacity (GoldGym ,3)

// Sessions ****************
```
Once the start state has been generated the genetic algorithm search process will begin its processing. Rather than having the initial population be generated randomly (which does not guarantee that the population will meet the hard constraints) an or-tree search is first used to get the initial solution population for the GA to use. The benefits of using an or-tree is that a solution is guaranteed to be found if it exists.

The sequential system has previously been fully tested for correctness so ideally we do not want to modify the existing code when converting it to a distributed system, as that can introduce new bugs. The first thing that needs to happen then is to adapt the sequential system to the core search interfaces.

6.1.2 Adapting to DisSLib:ICA

In the original sequential search system, the reading and parsing of the problem instance file was handled by the environment object while the creation of the initial start state was performed by the or-tree. In an ICA search system, these features are performed by a search problem instance object so they were moved to the new GAPProblemInstance class, shown in Listing 6.2. From the listing, we can see that the environment object is still responsible for reading and parsing the file and the or-tree is still generating the initial search state. In fact, the only thing that was added, in terms of lines of code, is the creation of the GASearchProblemInstance class that the library requires and changing how the retrieval of the file name is done (through a configuration file instead of the command-line).
The next step in the conversion process is adapting the search model of the sequential system. Unfortunately, search model and search process were not nicely separated, as they should be. Instead, everything was placed in a single class called `SetBasedSearch`. This class has a reference to the `SolutionManager`, which acts as a search state of sorts, managing all the possible solutions (called individuals in GA terms) that are in the population. Our first task was to create a new `GASearchState` class that implements the `AbstractSearchState` and moved the responsibility of the solution manager to the new search state class.

```java
public class GASearchState extends AbstractSearchState {
    private Population thePopulation;
    private Solution theBestSolution;
    public GASearchState()
    {
        MAX_POP_SIZE=getConfig().getInt("max_pop_size");
        ...
    }

    @Override
    public void integrateResults(SearchResult r) {
```
incrementGeneration();
GASearchResult result = (GASearchResult)r;
getPopulation().getSolutions().addAll(result.getNewSolutions());
List<Solution> individuals = getPopulation().getSolutions();
// sort the individuals by fitness
Collections.sort(individuals);
int fitness = individuals.get(0).getFitness();
if(fitness<this.getBestFitness())
{
    this.setBestFitness(bestFitness);
    this.setTheBestSolution(individuals.get(0));
    List<Assignment> sol = individuals.get(0).getSolution();
}
this.stateNumber++;
reducePopulationSize();
}
...

For the GASearchState class we have encapsulated the list of solutions in the Population class. The Population class did not exist in the original exam scheduling system, but was added to the new system to be consistent with the GA terminology. Essentially it replaces the old solution manager, as it is a data container that manages all the solutions (individuals) currently in the search state. The integration of the results processed by the search control behaves the same as in the original search system. The only difference is that instead of adding the new individuals to the solution manager, they now get added to the population.

Once the initial search state has been generated the main search loop will start. In the original exam scheduling system, how long the search will run is determined by a parameter passed into the system from the command line and stored within the environment. To tell whether or not a search can continue the search loop would check with the environment to see if the time has run out. Following the library’s structure, this functionality is now the responsibility of the GASearchObserver which extends from the AbstractSearchObserver class. Instead of the command line, the time limit is stored in a configuration file and is
retrieved in the constructor of the search observer (Listing 6.4).

Listing 6.4: The GASearchObserver

```java
public class GASearchObserver extends AbstractSearchObserver{
    private int timeLimit;
    public GASearchObserver(){
        timeLimit = getConfig().getInt("time");
    }

    @Override
    public boolean canContinue(SearchState s) {
        return getElapsedTime()<timeLimit;
    }
    ...
}
```

The SetBasedSearch class also includes methods that belong to the search control such as selecting the transitions to be processed and the processing of transitions through mutation and crossovers. Again, we have moved these functionalities into a separate class called the GASearchControl. For the search control we did not have to add additional code for it to work with the library. The only changes made were to refactor [Fow99] the method names to better match the GA terminologies. For example, the SetBasedSearch class from the original system had a method called swap which, in GA terminology, is equivalent to a mutation. We have subsequently changed the swap method to be called mutate, shown in Listing 6.5.

Listing 6.5: The GASearchControl

```java
@override
public SearchTransition getNextTransition(SearchState s) {
    // select a new transition based on roulette wheel
    GASearchState aState = (GASearchState)s;
    RouletteWheel theWheel = new RouletteWheel(aState.getPopulation());
    List<Solution> selectedSolutions = theWheel.spin(aState.getPopulation()).
        getSolutions().size());
    GASearchTransition transition = new GASearchTransition(selectedSolutions);
    return transition;
}
```
private Solution mutate(Solution solution) {
    // mutate the original solution by swapping exam sessions around
    List<Assignment> dup = new ArrayList<Assignment>(solution.getSolution());
    Random r = new Random();
    List<Session> sessions = AppController.getInstance().getSessionManager().getSessions();
    Solution mutated = new Solution();
    for (int i = 0; i < dup.size(); i++) {
        int randInt = r.nextInt(10);
        if (randInt < 2) {
            int position = r.nextInt(dup.size());
            Assignment selected = dup.remove(position);
            Assignment mutatedAssignment = selected.clone();
            mutatedAssignment.setSession(sessions.get(r.nextInt(sessions.size())));
            dup.add(position, mutatedAssignment);
        }
    }
    mutated.setSolution(dup);
    return mutated;
}

In addition to the core search classes we have also added the container classes GASearchResult and GASearchTransition. These classes were not required in the original system because no intermediary data container was required to share information between the classes, as everything was happening in a single class. However, they are required when using the library, as it is what the search control and the search state use to send the results of the processing to each other.

Finally, the last class from the core search component to be added is the GASearchDerivation class. The old search system did not have the concept of a search derivation because it was not interested in how the solution was derived at the time. Instead, it employed a series of
ad-hoc debug messages to allow the developers to see what is going on during the search. Having a search derivation class allows us to remove these debug messages and have a single place to record all the changes in the search state.

### 6.1.3 Lines of code needed to adapt to a DisSLib:ICA System

Having implemented all the core search components, it is now actually possible to run the system even though we only have a single search process. We can do this by creating an app.config file that will specify the key `standalone=true`, which will run the system as a sequential system. Having this ability is very helpful for testing as it allows us to verify that the new system functions the same as the old sequential system.

Before we add a new sequential system to allow for sharing of information, it would be good to compare how many additional lines of code were required by the new system.

<table>
<thead>
<tr>
<th>DisSLib:ICA Class</th>
<th>New LOC Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASearchControl</td>
<td>2</td>
</tr>
<tr>
<td>GASearchState</td>
<td>26</td>
</tr>
<tr>
<td>GASearchProblemInstance</td>
<td>2</td>
</tr>
<tr>
<td>GASearchObserver</td>
<td>7</td>
</tr>
<tr>
<td>GASearchResult</td>
<td>10</td>
</tr>
<tr>
<td>GASearchTransition</td>
<td>8</td>
</tr>
<tr>
<td>Population</td>
<td>14</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>69</strong></td>
</tr>
</tbody>
</table>

Table 6.1: This table shows the core search classes and the number of lines of code added during the conversion process.

Table 6.1 shows the classes of the GA sequential search system and the number of additional lines of code that were required. The method we used to produce the numbers consisted of using a difference tool to get the difference between the new classes and the original classes. We counted lines of code that were not in the original search system. However, we did not count lines that differed because of a simple name change due to refactoring.

The number of lines of code in the original system relating to the GA is 344, not counting comments and blank lines (we used the CLOC program to count the lines of code).
Looking at the number of lines added in Table 6.1 and the number of lines in the original system shows that we have had to add 20% more code. However, we need to keep in mind that the GASearchResult and GASearchTransition classes are data containers that contain very little logic, thus making them very easy to implement.

Even though the original GA sequential search system was developed before the library and developed by students that were new to search, we were still able to convert the sequential system to DisSLib:ICA with relatively few changes. This is because the concept of a search control and search state exists in all search systems, even though they may not be named or organized as such. Most of our effort was in the form of refactoring the code and creating the necessary classes that are required by the library.

6.1.4 Adding Additional Search Agents

A distributed system requires more than a single agent to run. Therefore, in addition to the GA agent, we were required to develop another search agent. One option we had available was to modify the GA agent’s search control; for example, instead of just mutations this new control might perform crossovers also. However, we know that such a system would not provide as much benefit as a heterogeneous system could, so we were inclined to use a different type of agent altogether. Existing literature [DO99] has shown that genetic algorithm and branch and bound agents can produce synergistic results, i.e., through the sharing of information between the two agents, the agents can find solutions much faster (or of a better quality) than had they performed the search without the additional information. So, we opted to build another search agent based on the branch and bound algorithm.

Branch and bound (BB) [LW66] is an and-tree based search optimization technique. Unlike the GA, a BB algorithm is guaranteed to find the optimal solution if one exists (and the algorithm is given enough time). The branch and bound algorithm optimizes the search by reducing the size of the space through pruning parts of the tree that it knows to be worse than the best case.
For the exam scheduling problem, the branch and bound algorithm starts with an initial start state containing the root node with all the pre-assigned assignments. At each node the solution space is reduced into smaller solution spaces by adding additional course and session assignments. This is the branching step. After branching, the lower bound of the new child nodes are calculated with a bounding function. The bounding function we used is one that determines the value of the soft constraint violations of a node. If a node’s lower bound is higher than the global upper bound then that node can be safely discarded. This is often referred to as pruning the tree. This process is repeated until an optimal solution has been found.

The sequential branch and bound system was written from scratch based on the library’s structure. The search problem instance returns an initial search state that contains an empty root node and the binding value set to infinity. The search observer does not change from the genetic algorithm search and is reused in the branch and bound agents.

The search control of the branch and bound algorithm needs to determine which node to select next and how to process the selected node. There are three different strategies in selecting a node, best-first, depth-first, and breadth-first. The selection strategy method chosen for the exam scheduling is a combination of depth first search and best first search. Depth first search is used to find the initial solution to be used as the upper bound. Once found the control switches to a best-first search so that it will choose the best node to possibly find better solutions.

Once a search transition is selected, the search control begins processing it by creating the child nodes for the selected transition. The children will be a combination of the assignments of the current node and an additional course and session assignment pair. Once branched out, each node will have its bounding value calculated and stored as a search result to be integrated into the search state.

The search state integrates the branched nodes by first looking at their lower bound
value. If the bound value is better than that of the current best solution then that node remains a possibility and is integrated into the tree. If it is a non-leaf node with a worse lower bound then it will be thrown out. For leaf nodes (which are possible solutions to the problem since they cannot be divided any further) the search state will check to see if it is a better solution than the current best solution. If it is then it will replace the current solution and its bounding value will be used as the new upper bound.

6.1.5 Branch and Bound Lines of Code

Table 6.2 (a) breaks down the lines of code per search classes of the BB search system. Unfortunately we did not have a branch and bound sequential search system to allow us to compare how much effort it would require in building a new system with the DisSLib:ICA versus one without. However, from (b) we can see that building a sequential search system with the library is comparable to that of building a sequential search system from scratch.

<table>
<thead>
<tr>
<th>BB Sequential Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBSearchControl</td>
<td>71</td>
</tr>
<tr>
<td>BBSearchState</td>
<td>140</td>
</tr>
<tr>
<td>BBSearchProblemInstance</td>
<td>27</td>
</tr>
<tr>
<td>BBSearchObserver</td>
<td>27</td>
</tr>
<tr>
<td>BBSearchResult</td>
<td>23</td>
</tr>
<tr>
<td>BBSearchTransition</td>
<td>15</td>
</tr>
<tr>
<td>BBNode</td>
<td>76</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>379</strong></td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Sequential Systems</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original GA System</td>
<td>344</td>
</tr>
<tr>
<td>New GA System</td>
<td>388</td>
</tr>
<tr>
<td>BB System</td>
<td>379</td>
</tr>
</tbody>
</table>

(b)

Table 6.2: (a) shows number of lines of code for the core search classes of the branch and bound system. (b) Compares the total lines of code for each sequential system.

6.1.6 Sharing Information between Exam Scheduling Agents

With the sequential search systems defined for both the GA and BB search agents we can now focus on the sharing of information between the two agents. To do so, we need to figure out exactly what information each agent requires. The GA agent is mostly interested in getting
new individuals that are either quite different (to allow for searching more of the search space) or fitter to produce better offspring. The BB agent is interested in finding better solutions so that it can prune more nodes and reduce the size of the tree and search space. For this distributed search system, we did not implement sharing of control information or negative information between the two agents.

Since both agents are interested in finding better solutions, they would both benefit from sharing state information among one another. Additionally, since a solution can be represented as a list of assignment pairs of sessions and courses, a solution can be understood by both agents.

Now that we have a format for exchanging information we need to implement the referees of the two agents. The send referee of the GA agent needs to keep track of the best solution that has been sent to the BB agent. Only solutions that improve upon the best recorded solution are sent to the BB agent. Similarly, the BB agent’s send referee store the best solutions the search has found so far so that it can send to the GA agent. Only a certain number of solutions, set by a parameter in the configuration file, will be sent to the GA agent.

One difference between the GA and BB send referee is that the BB’s send referee can send more than one solution at a time as each solution can potentially help the GA, while the BB agent is only interested in the best solution. Once sent, the receive referee of the GA agent checks that the solutions are within a threshold of an acceptable solution (within a certain percentage point of the best solution found). If it is, then the solution will get integrated with the search state. At this point the maximum population may have exceeded due to the extra individuals. Each individual will have a chance to be selected in the next selection round. The population size will once again be reduced during the integration step.

The receive referee of the branch and bound agent looks at the solution received from the GA agent and look at the current search state to determine if the fitness value of the
new solution is better than that of the best solution currently found by the search. If the solution is better then it is integrated into the search state. Two things occur during the integration process. The first is that the global upper bound will be updated with the fitness value to the new solution. The second is that the BB agent will remove some assignments from the solution and create a new node from it. This new node is then added to the tree to be processed (see Listing 6.6).

Listing 6.6: Integration of positive state information received.

```java
public void integratePositiveStateResult(FilterResult aResult) {
    logger.info("BB Search state integrating result from other agent");
    BBFilterResult result = (BBFilterResult)aResult;
    for(Solution sol: result.getSolutions()){
        int fitness = sol.getFitness();
        if(fitness < this.getBestUtility())
        {
            List<Assignment> solution = new ArrayList<Assignment>(sol.getSolution());
            List<Lecture> unassignedLecture = new ArrayList<Lecture>();
            // remove some assignments
            for (int j = 0; j < LEVELS_TO_EXPLORE; j++)
            {
                final int toRemove = solution.size() - 1;
                Assignment a = solution.remove(rand.nextInt(toRemove));
                Lecture toBeAssigned = AppController.getInstance().
                    getLectureManager().get(a.getLecture().getName(), a.getLecture().getCourse().getName());
                unassignedLecture.add(0, toBeAssigned);
            }
            Solution newSolution = new Solution(solution);
            newSolution.setFitness(HelperMethods.getFitness(solution));
            // create a node from the solution
            BBNode node = new BBNode();
            node.setUnassignedLectures(unassignedLecture);
            node.setValue(newSolution);
            node.setUtility(newSolution.getFitness());
            // update the global utility
```
setBestUtility(fitness);  
setToExplore(node);  
}
}
}

The motivation behind creating a new node from the received solution is that we know that the solution is good and there might be better solutions in that area. By removing a few assignments, it allows the tree to search around the solution space and possibly find better solutions.

6.1.7 Lines of Code for Information Sharing

<table>
<thead>
<tr>
<th>Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASendReferee</td>
<td>25</td>
</tr>
<tr>
<td>GASearchControl</td>
<td>35</td>
</tr>
<tr>
<td>GASearchState</td>
<td>13</td>
</tr>
<tr>
<td>BBSendReferee</td>
<td>27</td>
</tr>
<tr>
<td>BBSearchControl</td>
<td>34</td>
</tr>
<tr>
<td>BBSearchState</td>
<td>28</td>
</tr>
<tr>
<td>TOTAL</td>
<td>162</td>
</tr>
</tbody>
</table>

Table 6.3: The lines of code that is required for sharing information between the BB and GA agents.

Table 6.3 shows that 162 additional lines of code were added to implement communication between the agents. The number of lines of code to implement the referees is 22% of the 767 lines of code that were required to build both the GA and BB sequential search systems. This is due to two things. The first is that the referees we implemented for this case study are relatively simple. The second, and more significant reason, is that the library handled all the communication between the referees. We did not have to write any multi-threaded or networking code to support referee communication. There was no work involved in writing code to manage different threads and processes. The library enables the developer to solely focus on the problem and the questions relating to the search instead of spending time on
necessary but ultimately not relevant tasks such as the underlying network topography and communication protocol.

6.2 Course Scheduling

The next case study we look at shows how one can build a distributed search system from scratch using the library. For this case study, we built a distributed system to solve a course scheduling problem.

The goal of our course scheduling system is to optimally assign courses and labs to available time slots. The problem instance contains a set of courses, labs, and time slots. For each time slot there is a maximum number of courses and labs that can be scheduled within that time slot. In addition, there are hard constraints that each solution must satisfy. Some of these hard constraints are as follows:

- labs and tutorials of courses cannot be scheduled in the same time slot as the course

- partially assigned courses and labs need to be taken into account.

- some courses are not compatible with other courses so they should not be scheduled in the same time slot

In addition to the hard constraints there are also soft constraints that will be used to determine the quality of a solution. These soft constraints include the following:

- each time slot has a minimum number of courses and labs that must be scheduled in it. There is a penalty for not meeting this minimum assignments.

- certain courses have a preference for a specific time slot. There is a penalty for each course or lab with specific preference that is not met by the solution.
• there are courses that students would never take alongside one another and should therefore preferably be scheduled at the same time. There is a penalty for each of these pairings that is not met.

While the course scheduling problem is similar to the exam scheduling problem in that both are optimization problems, there are a few differences that require that we look at the problems differently.

With the exam scheduling problem there were four hard constraints and seven soft constraints. The small number of hard constraints meant that it was relatively cheap to do hard constraint violation checks compared to a soft constraint violations checks. We were able to make use of this when building the GA exam scheduling control as each time a swap was performed we could check whether or not that swap violated the hard constraints.

For the course scheduling problem there are seven hard constraints and only 3 soft constraints. This means that performing a hard constraint violation check is a lot more expensive than for the exam scheduling problem. The design of the course scheduling search components need to take this fact into account to reduce the number of times a hard constraint violation check is performed.

As stated earlier, we did not have access to an existing sequential system so we had to build a course scheduling system from scratch using the DisSLib:ICA library. The first thing when developing a search system is to decide on the search model that the system will employ. For the course scheduling system we, again, went with a heterogeneous system consisting of GA and branch and bound agents.

We went with the same type of agents as for the exam scheduling system for two reasons. First, GA and BB agents tend to work very well together and their results tend to have shown synergistic effects. Second, since we already had experiences with implementing both for the exam scheduling system, we will be able to make use of our experiences to develop a GA and BB search agent faster than with other search models.
We will follow the same strategy as we had done when building the exam scheduling system, by first developing a sequential system, test it, and then, once we are confident that the sequential system works, add the referees and information sharing components.

6.2.1 Course Scheduling GA Agent

Listing 6.7: A sample course scheduling problem instance file

<table>
<thead>
<tr>
<th>Course slots:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MO, 8:00, 3, 2</td>
</tr>
</tbody>
</table>

Lab slots:
| MO, 8:00, 4, 2          |

Courses:
| CPSC 433 LEC 01         |
| CPSC 433 LEC 02         |

Labs:
| CPSC 433 LEC 01 TUT 01  |

Not compatible:
| CPSC 433 LEC 01 TUT 01, CPSC 433 LEC 02 LAB 02 |

Unwanted:
| CPSC 433 LEC 01, MO, 8:00 |

Preferences:
| TU, 9:00, CPSC 433 LEC 01, 10 |

Pair:
| SENG 311 LEC 01, CPSC 567 LEC 01 |

Partial assignments:
| SENG 311 LEC 01, MO, 8:00 |

Our first task was to build a parser to read a course scheduling problem instance (Listing 6.7). Once the parser was built we were then able to implement the GASearchProblemInstance to generate the initial start state. The GASearchProblemInstance, shown in
Listing 6.8 is very similar to that of the search problem instance object of the exam scheduling problem. Both problem instances used an or-tree to get an initial population for the genetic algorithm.

Listing 6.8: The GAProblemInstance of a course scheduling system

```java
@Override
public SearchState generateStartState() {
    List<Identifier> identifiers = new ArrayList<Identifier>();
    String path = this.config.getString("filePath");
    EnvironmentData data = EnvironmentData.getInstance();
    data.parseFileData(path);

    Node root = new Node();
    identifiers.addAll(data.getCourses());
    identifiers.addAll(data.getLabs());
    for (Assignment partial : data.getPartialAssignments()) {
        partial.setPartial(true);
        root.getSolution().addAssignment(partial);
        if (identifiers.contains(partial.getIdentifier()))
            identifiers.remove(partial.getIdentifier());
    }
    root.setRemaining(identifiers);
    List<Solution> solList = new ArrayList<Solution>();
    OrTree.orTree(root, solList, InitialSize, maxTime);
    Population pop = new Population();

    for (Solution sol : solList)
        sol.setFitness(SCManager.eval(sol));
    pop.setIndividuals(solList);
    GAState state = new GAState(pop);
    return state;
}
```

To ensure that all the partial assignments are assigned in the final solution, the search problem instance adds the partially assigned assignments to the initial search state and removes the assigned courses and labs from the available courses and labs list, so they cannot be reassigned again. Along with the partial assignments, the list of available slots,
courses, and labs are also stored in the solutions which are then stored in the search state as the initial population.

Because the course scheduling system is a time-sensitive system like the exam scheduling system, the search observer of the course scheduling system must check that the running time of the system has not exceeded the time allotted. The allotted time is determined by reading the configuration file of the \texttt{GAObserver}.

The selection process of the \texttt{GASearchControl} makes use of a roulette wheel selection similar to the exam scheduling system. Each generation the entire population is given a chance of being selected. However, if a population exceeds the maximum count then the worst individuals are removed from the population until the population count is lower than the maximum.

Unlike the exam scheduling system, the \texttt{GASearchControl} of the course scheduling system performs both mutations and crossovers (Listing \ref{lst:process}). Also, there are no checks to ensure that each assignment produces a valid solution, as hard constraint checks are expensive. Instead, mutations and crossovers that violate the hard constraints can be performed. After a genetic operation if an individual violates the hard constraints then it set to be null and will not be added as a result of the processing. Only valid solutions will be sent to the search state to be integrated.

Listing 6.9: The processing of selected individuals by the \texttt{GASearchControl}

```java
public TransitionResult process(Transition t) {
    List<Solution> toBeProcessed = ((GATransition) t).getPopulation();
    List<Solution> processed = new ArrayList<Solution>();
    while (toBeProcessed.size() > 0) {
        float f = random.nextFloat();
        int size = toBeProcessed.size();
        int selection = random.nextInt(size);
        if (f <= mutationRate) {
            Solution mutate = mutate(toBeProcessed.remove(selection));
            if(mutate!=null){
                processed.add(mutate);
            }
        }
    }
    return new TransitionResult(processed, evolve());
}
```

104
if (size > 1) {
    int selection2 = random.nextInt(size - 1);
    Solution cross = crossOver(toBeProcessed.remove(selection),
                                toBeProcessed.remove(selection2));
    if (cross != null) {
        processed.add(cross);
    }
} else {
    Solution mutate = mutate(toBeProcessed.remove(selection));
    if (mutate != null) {
        processed.add(mutate);
    }
}
return new Result(processed);

The GATransition is a data container that holds a list of individuals to be processed by the search control. Processing a search transition can be done either by crossovers or mutation. The mutation rate is set as a configuration parameter and it controls how often an individual is mutated. There is also a mutation strength parameter that controls how many assignments in a solution get mutated. The higher the strength the more assignments will be mutated.

Once processed, the search control returns a GATransitionResult which, like the transition, is just a container that holds the new individuals to be integrated into the GASearchState (Listing 6.10). In this integration process the fitness values of the individuals are calculated and assigned to the solution. The fitness level is the sum of all the soft constraint violations penalties. If the size of the population is too large, then the worst performing individuals are removed from the population.

Listing 6.10: The integrateResults method of the GASearchState

```java
public void integrateResults(TransitionResult r) {
    Result result = (Result) r;
```
for (Solution individual : result.getProcessedIndividuals())
{
    int curFitness = SCManager.eval(individual);
    individual.setFitness(curFitness);
    population.add(individual);
    if (bestFitness > curFitness)
    {
        bestFitness = curFitness;
        best = individual;
    }
}

population.sortBestToWorst();
if (population.size() > MAX_POP_SIZE){
    population.update(population.take(0, MAX_POP_SIZE));
}

6.2.2 GA Course Scheduling Lines of Code

<table>
<thead>
<tr>
<th>GA Course Scheduling Core Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASearchControl</td>
<td>195</td>
</tr>
<tr>
<td>GASearchState</td>
<td>84</td>
</tr>
<tr>
<td>GASearchProblemInstance</td>
<td>39</td>
</tr>
<tr>
<td>GASearchObserver</td>
<td>24</td>
</tr>
<tr>
<td>GATransitionResult</td>
<td>17</td>
</tr>
<tr>
<td>GASearchTransition</td>
<td>14</td>
</tr>
<tr>
<td>Population</td>
<td>30</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>403</strong></td>
</tr>
</tbody>
</table>

Table 6.4: The total lines of code to implement a GA sequential course scheduling system using DisSLib:ICA

Table 6.4 shows that the number of lines of code required to build a GA course scheduling search system using the library is consistent with that of the exam scheduling problem. The reason that this system requires more lines of code than the exam scheduling system is that we have added crossovers.
6.2.3 Course Scheduling Branch and Bound

The branch and bound search problem instance (Listing 6.11) reads from its configuration file to get the file path of the concrete problem instance. The root node of the branch and bound starts with the partially assigned courses and time slots.

Listing 6.11: The BBSearchProblemInstance generating the initial start state.

```java
public SearchState generateStartState() {
    String path = this.config.getString("filePath");
    EnvironmentData data = EnvironmentData.getInstance();
    data.parseFileData(path);
    List<Identifier> identifiers = new ArrayList<Identifier>();
    identifiers.addAll(data.getCourses());
    identifiers.addAll(data.getLabs());
    BBNode root = new BBNode();
    Solution sol = new Solution();
    for (Assignment partial : data.getPartialAssignments()) {
        sol.addAssignment(partial);
        if (identifiers.contains(partial.getIdentifier()))
            identifiers.remove(partial.getIdentifier());
    }
    root.setSolution(sol);
    root.setRemaining(new ArrayList<Identifier>(identifiers));
    List<Slot> courseSlots = new ArrayList<Slot>(data.getSortedCourseSlots());
    Collections.reverse(courseSlots);
    List<Slot> labSlots = new ArrayList<Slot>(data.getSortedLabSlots());
    Collections.reverse(labSlots);
    BBSearchState state = new BBSearchState(identifiers, courseSlots, labSlots);
    state.getNodes().add(root);
    return state;
}
```

The search control of the BB agent selects the next transition through a depth-first search strategy. This ensures that a solution is found quickly without expanding the tree too much compared to other strategies.
The processing of a transition (Listing 6.12) is similar to that of the exam scheduling system. First the selected node, which is a partial solution, is expanded by creating child nodes from a course or lab with all possible combinations of the slots. This new child level is then stored as a BBTransitionResult to be integrated back into the search state.

Listing 6.12: The BBSearchProblemInstance generating the initial start state.

```java
public TransitionResult processTransition(Transition transition) {
    BBSearchTransition bbt = (BBSearchTransition)transition;
    BBNode original = bbt.getSelectedNode();

    Identifier next = bbt.getNext();
    List<Assignment> newAssignments = new ArrayList<Assignment>();
    if(next!=null) {
        List<Slot> slots = null;
        if(original.getRemaining().size()>2) {
            if(next.isACourse())
                slots = HCManager.filterSlots(original.getSolution(), next, new ArrayList<Slot>(bbt.getCourseSlots()));
            else if(next.isALab())
                slots = HCManager.filterSlots(original.getSolution(), next, new ArrayList<Slot>(bbt.getLabSlots()));
        } else {
            if(next.isACourse())
                slots = bbt.getCourseSlots();
            else if(next.isALab())
                slots = bbt.getLabSlots();
        }
        for(Slot s:slots)
            newAssignments.add(new Assignment(next,s));
    }
    TransitionResult result = new BBSearchResult(original,newAssignments,
        ResultSource.Normal);
    return result;
}
```
The transition result of the branch and bound system contains the parent node along with a list of children that was created by the processing step. These child nodes are themselves solutions that contain the partial assignments of their parents and the newly assigned course and slot. To reduce the memory usage, each child node only contains the newly assigned course and time slot. To get the full solution, the nodes traverse up the tree to the root node. One disadvantage of this is that even after a node is closed we cannot remove all the parent nodes from memory since they are used by other nodes. However, once all the child nodes are closed then the parent node can be removed.

Listing 6.13: Integration of the child nodes back into the BBSearchState.

```java
public void integrateResults(TransitionResult r) {
    BBTransitionResult res = (BBTransitionResult)r;
    BBNode original = res.getOriginal();
    BBNode copy = original.copy();
    for (Assignment a: res.getExpanded())
    {
        copy.addAssignment(a);
        int bound = getBoundValue(copy);
        if(bound>=bestFitness){
            continue;
        }
        if(!HCManager.violatesHC(copy))
        {
            int curFitness = SCManager.eval(copy.getSolution());
            copy.getSolution().setFitness(curFitness);
            if(copy.isLeaf())
            {
                solutions.add(copy.getSolution());
                if(curFitness<bestFitness)
                {
                    bestFitness = curFitness;
                    setBestSolution(copy.getSolution());
                }
            }
        }
    }
}
```
The integration of child nodes into the search state (Listing 6.13) starts with getting the bounding value of each child node. That bounding value is then compared to that of the upper bound. If it is worse the node will be pruned, thus preventing further exploration of the node. The remaining children then need to have their solutions checked to see whether the hard constraints have been violated. Children violating the hard constraints are also pruned. Next, if the child is a leaf node with a better fitness level than that of the upper bound it is set to be the current best solution.

6.2.4 BB Course Scheduling Lines of Code

<table>
<thead>
<tr>
<th>BB Course Scheduling Core Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBSearchControl</td>
<td>70</td>
</tr>
<tr>
<td>BBSearchState</td>
<td>162</td>
</tr>
<tr>
<td>BBSearchProblemInstance</td>
<td>39</td>
</tr>
<tr>
<td>BBSearchObserver</td>
<td>29</td>
</tr>
<tr>
<td>BBTransitionResult</td>
<td>37</td>
</tr>
<tr>
<td>BBSearchTransition</td>
<td>39</td>
</tr>
<tr>
<td>BBNode</td>
<td>35</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>411</strong></td>
</tr>
</tbody>
</table>

Table 6.5: The total lines of code to implement a BB sequential course scheduling system using DisSLib:ICA

Table 6.5 shows that the number of lines of code is consistent with that of the exam scheduling system. This shows that developing a sequential systems using the library does not require any more lines of code than compared to developing a sequential system without using a library.
6.2.5 Sharing Information between Course Scheduling Agents

Having two sequential search systems enables us to share information between them using the referees. Since the search models of the agents are the same as those of the exam scheduling system, we decided to implement the send and receive referees similar to that of the send and receive referees of the exam scheduling system.

The information we decided to share is a list of solutions. We decided to share the same information as the exam scheduling system as we have already implemented it once and we could reuse some of the code. Since the information that both GA and BB use can be described in terms of assignments of courses to time slots, it means that no conversion of information is necessary between the two agents. The BB agent will share various solutions that it deems interesting, while the GA send referee will only share the best solution that it found with the BB agent.

The send referee of the GA agent (Listing 6.14) is straight forward. At each referee interval, the send referee will find the solution with the best fitness. It then checks to make sure that the fitness level is better than the one that was previously sent. If the solution is better than the last one that was sent then the information object will be created and sent to the BB agent. The send referee only supports sending positive state information.

Listing 6.14: The GASendReferee of a course scheduling system

```java
public PositiveStateInformation selectPositiveStateInformation (SearchState state) {
    GAState gaState = (GAState)state;
    int stateBestFitness = gaState.getBestFitness();
    PositiveStateInformation info = null;
    if(stateBestFitness < currentBestFitness)
    {
        currentBestFitness = stateBestFitness;
        info = new GAPositiveInformation(gaState.getBestSolution());
    }
    return info;
}
```

111
The send referee of the BB agent (Listing 6.15) also examines the current search state to find the best solutions available. It will only look at the leaf nodes, as they are full solutions. Like the GA send referee, the BB agent also keeps track of solutions that have been sent to the GA. The send referee will randomly pick from the possible solutions in the search state a wide variety of solutions with a bias to the best performing ones. Once the solutions are chosen it will need to check against the list of solutions that have already been sent. Solutions that have been sent will be removed from the list. The remaining solutions then will be sent to the GA agent and also added to the list of solutions.

Listing 6.15: The **BBSendReferee** of a course scheduling system

```java
public PositiveStateInformation selectPositiveStateInformation(SearchState state) {
    BBSearchState bbstate = (BBSearchState) state;
    List<Solution> bestSolutions = new ArrayList<Solution>();
    for (Solution s: bbstate.getBestSolutions(NUM_SOLUTION))
    {
        bestSolutions.add(s.copy());
    }
    bbstate.removeSentSolutions(bestSolutions);
    return new BBPositiveInformation(bestSolutions);
}

public NegativeStateInformation selectNegativeStateInformation(SearchState state) {
    return null;
}
```

When the receive referee of the BB agent receives a solution from the GA agent, it will check to make sure that the solution is better than that of the current search state. If the solution is better, the BB receive referee will store the received solution as a **FilterResult** object to be integrated into the search state. Otherwise, it will return a null object which means that no integration is necessary.

After the search control can receive and filter information, the information must be integrated into the search state. The search state first checks that the fitness is better than
that of the best fitness. It will then update the best solution to the new solution and also
the lower bound value to that of the fitness of the new solution. In addition, the search state
will also create additional nodes to be processed by removing a few assignments from the
solution and creating a node consisting of the partial solution. This is the same approach
that we used for the exam scheduling system.

When the GA referee receives a set of possible solutions from the BB send referee, it will
first make sure that the solutions are within an acceptable lower-bound range. This is to
ensure that the search moves forward since too many poor solutions will not be beneficial to
the search. The range offset is based on a percentage of the best solution received so far. So
the accepted solution needs to be between certain percentages of the current best solution.
For our experiments we used -50% to as high as possible. Once the receive referee filters
out all the solutions it will return a FilterResult object that is a container for the set of
possible solutions to be integrated back into the search state.

The integration process of the filtered result is very simple. All the GA does is create
the corresponding individuals to be added back into the population from the list of filtered
solutions received. At this step it is possible to have a population that is greater than that of
the max population count but this is acceptable since we do not want to remove individuals
before letting them have a chance of being selected. After the next selection round the
integration process will automatically remove the worst performing individuals.

6.2.6 Lines of Code for Information Sharing of a Course Scheduling System

Similar to the exam scheduling system, sharing information between the agents requires a
lot less lines of code than compared to the total lines of code of the system. The total lines
of code for the course scheduling system are 3358 lines (the majority of the lines of code is
in the domain classes that models the courses, labs, and constraint calculations). Out of
the 3358 lines that were written, only 120 lines of code (see Table 6.6) deal with sharing
information between the agents. This shows that the library has successfully eliminated a
Table 6.6: The lines of code that is required for adding information sharing to the course scheduling system.

<table>
<thead>
<tr>
<th>Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASendReferee</td>
<td>24</td>
</tr>
<tr>
<td>GASearchControl</td>
<td>15</td>
</tr>
<tr>
<td>GAsearchState</td>
<td>8</td>
</tr>
<tr>
<td>BBsendReferee</td>
<td>28</td>
</tr>
<tr>
<td>BBSearchControl</td>
<td>12</td>
</tr>
<tr>
<td>BBSearchState</td>
<td>33</td>
</tr>
<tr>
<td>TOTAL</td>
<td>120</td>
</tr>
</tbody>
</table>

6.3 Package Delivery Planning Problem

So far, we have only looked at case studies that required us to build new search agents. However, there are cases where developers have existing distributed systems built with one of the other distribution paradigms. It would be beneficial for developers to compare how their existing distributed search system performs in other distribution paradigms. Unfortunately, the process of taking a working system and adapting it to a different distribution paradigms can be as much work as building a new search system. Using libraries such as DisSLib:ICA can help reduce the workload, as the package delivery case study will show.

The package delivery planning problem (a variant of the traveling salesman problem [GL85]) deals with delivering packages to different destinations. A problem instance for the package delivery planning problem consists of a list of drivers and a matrix consisting of each location the packages must be delivered to and the cost of traveling between the locations. The goal is to find the minimal cost path that would allow all packages to be delivered. The system we had to work with was originally developed using DisSLib:CC [Ken10], a distributed search library based on the central common search state paradigm. There were two different agents (GA and BB) that were developed for this system, which meant that we did not have to build any additional agents.
Since DisSLib:ICA and DisSLib:CC both based their skeleton interfaces on the definitions of search described in Chapter 2, there was very little work required to get the sequential systems running in DisSLib:ICA, as the skeleton interfaces of DisSLib:ICA match those of DisSLib:CC. Thus, to get the sequential systems running, all that was required of us was to change the project reference to use the DisSLib:ICA, update the existing configuration files (as the two libraries use different configuration systems), and remove methods in the search state that are not required in DisSLib:ICA, such as marking whether a transition has been processed or not.

The majority of the effort required to get the agents to run in DisSLib:ICA was implementing the send referees and various methods to enable sharing of information. Like the previous case studies, we only shared positive state information between the two agents so we did not have to implement the sharing of negative and control information.

At each sharing interval, the GA agent’s send referee will filter out the best solution it has found and send it to the BB agent. It will also keep track of the solutions it has sent along with travel costs of each driver to ensure that it will send only the best solutions. The BB agent’s send referee will likewise send solutions it has found to the GA agent. The solutions it sends consist of several solutions varying in fitness. This allows the GA agent to have many different types of solutions in different parts of the search space to work with.

The receive referees of both agents, again, are similar to the receive referee implementations in the previous case studies. The GA receive referee will check to ensure that solutions received are within a range of acceptable fitness. For solutions that are worse than the current best solution, it will make sure that they are within a certain percentage of fitness of the best solutions. Better solutions are automatically added to the population. The receive referee of the BB agent checks that the solution received has a fitness that is better than the current best fitness found so far.

This new information now needs to be integrated back into the search state. The BB
search state will process the solution by first checking to make sure that it is better than that of the current best solution. If it is, then the new solution’s bounding value will replace the current upper bound. We also create a new node by removing a number of driver-location assignments. Again, we used the same strategy for finding better solutions as we have done with the previous exam and course scheduling systems. Once the new solution is replaced and the upper bound is updated, the search state will look at all the existing nodes and prunes all nodes that have a higher fitness than that of the upper bound.

The GA agent will integrate its results by adding the new solutions into the existing population. These new individuals will be marked as to be processed to ensure that they will not be removed in the integration process until they are have processed.

6.3.1 Total Lines of Code Required

As Table 6.7 shows, adapting from DisSLib:CC requires significantly less code than for the other case studies. This is because we did not have to develop additional search systems. The fact that both libraries share the same skeleton interfaces makes adapting from one library to another extremely easy. The only additional code was the code necessary for information sharing between the agents.

<table>
<thead>
<tr>
<th>Classes</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASendReferee</td>
<td>32</td>
</tr>
<tr>
<td>GASearchControl</td>
<td>16</td>
</tr>
<tr>
<td>GASearchState</td>
<td>13</td>
</tr>
<tr>
<td>BBSendReferee</td>
<td>29</td>
</tr>
<tr>
<td>BBSearchControl</td>
<td>21</td>
</tr>
<tr>
<td>BBSearchState</td>
<td>36</td>
</tr>
<tr>
<td>TOTAL</td>
<td>147</td>
</tr>
</tbody>
</table>

Table 6.7: The total lines of code that are required for adapting distributed systems built in DisSLib:CC to DisSLib:ICA.

The three case studies we have looked at have shown that by using the library we can adapt existing sequential search systems or develop new sequential search systems with approximately the same amount of effort as without using the library. The major benefit of
using the library, however, is that it allows developers to take an existing sequential search system, modify the system to allow the it to run as a distributed search system based on the ICA paradigm, with significantly less effort than if he had implemented a distributed search system from scratch. However, having a distributed system is only of benefit if it provides better performance than what could be achieved with a sequential system. In the next chapter we will show the results of running our three distributed systems versus their sequential counterparts on different problem instances.
Chapter 7

Case-Study Results

The previous chapter has shown that DisSLib:ICA can be used to develop distributed search systems with nearly the same amount of effort as it would take to develop a sequential search system. However, having a distributed search system would do us no good if it did not produce better results than a sequential system. In this chapter, we will take a look at the results of running both the sequential and distributed search systems for our applications using different problem instances. We will compare the results produced to see how much of an improvement was achieved by our distributed search systems.

There are many factors that can affect the results a search system produces. An obvious factor is the hardware that the search system is executing on. Faster CPUs, faster disks, and more memory will produce better results. Other, not so obvious factors are the parameters of the search system. While a search system is usually developed to solve a particular problem, it cannot be configured to optimally solve all instances of said problem. Optimization of the search run is usually achieved by adjusting the parameters through an iterative process. For example, a GA system usually has multiple parameters that can be configured such as the mutation strength and the number of individuals selected per generation. A low mutation rate might produce optimal results for certain instances while a high mutation rate might produce better results in other instances. Distributed systems also introduce additional factors that can affect the results of the search. These factors include the network latency and transfer rate between the search agents, and the information sharing interval between the agents.
7.1 Hardware Configuration

In order to effectively compare the results between the sequential and distributed systems we have to ensure that all the factors that could affect the performance of the search system are consistent across all systems. This means that search runs need to be executed on machines with the same hardware and the search parameters should be consistent where possible. For our experiments, we had access to two identical quad core machines connected by a gigabit switch. To reduce the impact of the currently installed applications and services running on the machines we have created a virtual machine (VM) running Ubuntu 13.04 on each box using Oracle VirtualBox. The configuration of the two virtual machines are shown in Table 7.1.

<table>
<thead>
<tr>
<th>Oracle VirtualBox Machine</th>
<th>Specs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host OS</td>
<td>Windows 7 Professional 64bit</td>
</tr>
<tr>
<td>Guest OS</td>
<td>Ubuntu 13.04 64bit</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i7-2600 @ 3.40 GHz (1 core assigned to VM)</td>
</tr>
<tr>
<td>Memory</td>
<td>4096 MB</td>
</tr>
<tr>
<td>Disk</td>
<td>10 GB SSD</td>
</tr>
<tr>
<td>Java JDK</td>
<td>OpenJDK 7 (1.7.0_21)</td>
</tr>
</tbody>
</table>

Table 7.1: Hardware configuration of the virtual machine.

The only software added to our Ubuntu guest OS was ssh server, git source control management, and OpenJDK. The ssh server install is to allow us to remotely connect to the virtual machine and start the search. Git is used to get the source code, while the JDK is used to compile and run the search.

7.2 Experiment Setup

Since there are many factors that can affect a search run, for each problem instance we have performed three runs and report the average of the three runs as the result. Also, due to the number of search runs we have to perform, we have limited each run to a maximum of one hour.
For our tests there are two criteria that we will use to evaluate the improvements in our search system. These criteria are speed (solving a problem faster), quality (a better solution can be found within the same time), or a combination of both speed and quality. More specifically, we will measure the soft constraints (for the package delivery planning problem we use the travel cost of a delivery) that have been violated by a solution. The better search system will be the one that produces solutions that violate the least number of soft constraints or find the same solution in a faster amount of time.

7.3 Exam Scheduling Results

For the exam scheduling problem we had two problem instances to test. The first problem instance (Exam1) contains 60 courses, 116 lectures, 50 instructors, 800 students, 5 rooms, 40 sessions, and 0 fixed assignments, while the other problem instance (Exam2) consists of 42 courses, 88 lectures, 40 instructors, 640 students, 4 rooms, 40 sessions, and 6 fixed assignments. As previously noted, our exam scheduling system seeks to produce a solution that assigns all lectures to exam sessions.

There are a total of 7 soft constraints that vary in penalties if violated. The value of a solution is the sum of all the violation penalties. For example, every lecture for the same course should have the same exam time slot. Suppose that a solution has CPSC 233 L01 and CPSC 233 L02 in different time slots, that would be a soft constraint violation of 50 points. The best solutions to the exam scheduling problem will be the ones with a minimal number of penalties.

7.3.1 Original Sequential System vs DisSLib:ICA Sequential System

For the exam scheduling problem we had an existing sequential system that had previously been developed for an undergraduate AI course. In the previous chapter we have shown that adapting the sequential system to DisSLib:ICA required relatively little effort on the part of
the developer. We decided to evaluate if there was any performance hit when running it in standalone mode versus a system built from scratch. The average results of performing the search on both problem instances on the same virtual machine are shown in Table 7.2.

<table>
<thead>
<tr>
<th>Problem Instance</th>
<th>Original GA System</th>
<th>DisSLib:ICA GA Standalone System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exam1</td>
<td>27640</td>
<td>22705</td>
</tr>
<tr>
<td>Exam2</td>
<td>23480</td>
<td>21065</td>
</tr>
</tbody>
</table>

Table 7.2: The average total soft constraint violations of the best solution produced by the original exam scheduling system versus the adapted DisSLib:ICA exam scheduling system running in standalone mode.

The fact that the DisLib:ICA GA system shows an improvement over the original GA system is quite a surprise to us because the two systems share the same search control code. The reasons we suspect that might have slowed down the original system is the random selection factor that all GAs have. Also, the original system runs with a thread hooked to it in order to monitor when the search has finished, so that it can record the results to disk. This additional hook might have affected the processing capability of the system.

7.3.2 Sequential System vs DisSLib:ICA Distributed System

The next experiment we carried out was to compare the results between the sequential systems and the distributed system. To get the results we ran each system using the same parameters. That is, the sequential and distributed GA agents both have the same mutation strength and the branch and bound agents both use the same bounding function. The information sharing interval between the agents was set at 5 seconds. The results of our experiments are shown in Table 7.3.

<table>
<thead>
<tr>
<th>System</th>
<th>Exam1 Violations</th>
<th>Exam2 Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA Sequential System</td>
<td>22705</td>
<td>21065</td>
</tr>
<tr>
<td>BB Sequential System</td>
<td>24070</td>
<td>20405</td>
</tr>
<tr>
<td>DisSLib:ICA Distributed System</td>
<td>1200</td>
<td>1400</td>
</tr>
</tbody>
</table>

Table 7.3: Comparisons of the quality of the solution between the sequential agents versus the distributed.

From Table 7.3 we can see the synergistic effect of the improving on the competition
approach paradigm. The solutions from our DisSLib:ICA distributed system are an order of magnitude better than the sequential systems. With two agents we would normally expect a linear increase in the quality of the solution or to find a solution in half the time. Clearly the results of our distributed DisSLib:ICA agents have far exceeded this.

Figure 7.1: A line chart showing the best solutions of each search agent at a specific point in time.

Figure 7.1 shows a chart depicting the best results of our distributed system for the first 70 seconds of the search run. Looking at the sequential agents, we can see that the sequential GA agent started off at a poor solution but slowly improves itself. The sequential BB agent, while initially finding a good solution, was unable to improve upon it. This is because we used a simple bounding function that did not prune enough of the tree causing the search to explore bad areas of the search space.

Compared to the sequential agents we can see that both the distributed GA and BB
agents improved upon their initial solutions. In fact, there is often an improvement after each 5 second interval. This is because every 5 seconds the agents will share their information with one another. At the first information exchange the distributed BB agent was able to use the solution sent by the GA agent to find new areas of the search space to explore and found better solutions there. The sequential BB agent on the other hand was still stuck at the same solution space.

7.4 Course Scheduling Results

The course scheduling system was tested in a similar fashion to that of the exam scheduling system. We used a single problem instance called DeptInst1 which consisted of courses and labs from the Computer Science department at the University of Calgary. Again, because these problem instances are very large we only ran each agent for an hour per run. The average results of three runs are shown on Table 7.4.

<table>
<thead>
<tr>
<th>System</th>
<th>DeptInst1 Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Sequential System (GA)</td>
<td>3702</td>
</tr>
<tr>
<td>DisSLib:ICA Distributed System</td>
<td>3535</td>
</tr>
</tbody>
</table>

Table 7.4: Comparisons of the quality of the solution between the sequential agents versus the distributed

The improvements in the distributed system were not as significant as it was in the exam scheduling case study. This is most likely due to the fact that there is not as much room for improvement as there was in the exam scheduling problem.

Since the quality of the solution did not give us a good idea of the improvements we can turn to evaluating the speed at which the solutions were found. We looked at the time it took the sequential systems to find their best solution and compare it to how long it took the distributed system to find a better solution than the the best one found by the best sequential system. Table 7.5 shows the result of the speed test.

While we did not notice a significant improvement in the quality of the solution, Table
Table 7.5: Comparisons between the time it took to find the same solution between sequential and distributed agents.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sequential System Time</th>
<th>Distributed System Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeptInst1</td>
<td>47.5 minutes</td>
<td>9.8 minutes</td>
<td>4.8 times</td>
</tr>
</tbody>
</table>

Table 7.5 shows us that the distributed system has found, on average, an equal or better solution 4.8 times faster than the sequential system. This result again shows the synergistic behavior of the improving on the competition approach paradigm.

7.5 Package Delivery Planning Problem Results

For the package delivery problem we had to generate our own problem instances rather than use the ones provided to us. This allowed us to work on problem instances that can be solved within a reasonable amount of time, thus enabling us to find optimal solutions to the problem instances. For the package delivery problem, we generated two problem instances consisting of 24 and 25 locations respectively. Each location pair was given a random cost between 1 and 100.

We then ran the sequential branch and bound system for as long as necessary to find the optimal solution. We can only do this with the B&B because the GA system cannot guarantee to find an optimal solution. Once a optimal solution had been found, we ran the distributed system on the same problem instances and record the time it took to find an optimal solution. The results are shown in Table 7.6.

Table 7.6: Comparisons in the time it took to find the optimal solution for two instances of the package delivery problem.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Avg Sequential Agent Time</th>
<th>Avg Distr. Agent Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 Locations</td>
<td>25.30 minutes</td>
<td>2.50 minutes</td>
<td>10.12 times</td>
</tr>
<tr>
<td>25 Locations</td>
<td>7.13 minutes</td>
<td>2.67 minutes</td>
<td>2.67 times</td>
</tr>
</tbody>
</table>

In Table 7.6 we only presented the B&B sequential system results because the GA sequential agent failed to find the optimal solution within the 1 hour limit. The distributed system sharing interval was set at 20 seconds. Looking at the speedups we can see that there
is a big discrepancy between the problem instances. The reason that the second problem instance had a worse improvement was that the optimal solution was found much earlier than it had been in the first problem instance. There was simply not enough “room” for improvement for this instance. However, the distributed system still clearly shows synergy in the results produced.

Figure 7.2

Figures 7.2 and 7.3 show the times at which a better solution is found for the best run of each problem instance. In both figures we, again, see the improvements in the solution after the agents share information with one another at every 20 second interval.

7.6 Information Sharing Interval Comparison

Time plays an important role in the improving on the competition approach paradigm because the information sharing interval can have a big impact on the quality of the solution. If the information sharing interval is set too far apart then important information could be shared too late and would not be of help to other agents. However, if the interval is set too
small then the agents might not have had sufficient time to gather enough information that is relevant to the other agents. Also, small intervals force the agents to stop their search to deal with the information which can slow down the search process.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Average Time (24 Locations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 second</td>
<td>6.5 minutes</td>
</tr>
<tr>
<td>5 seconds</td>
<td>6.3 minutes</td>
</tr>
<tr>
<td>10 seconds</td>
<td>4.25 minutes</td>
</tr>
<tr>
<td>20 seconds</td>
<td>2.5 minutes</td>
</tr>
<tr>
<td>30 seconds</td>
<td>4.1 minutes</td>
</tr>
<tr>
<td>1 minute</td>
<td>7.8 minutes</td>
</tr>
</tbody>
</table>

Table 7.7: Comparisons in the time it took to find the optimal solution at different sharing intervals.

Table 7.7 shows the time it took to find the optimal solutions for the 24 location package delivery problem instances at different information exchange intervals. As we can see, there is a large discrepancy in the average time to find the optimal solution at each different interval. For this particular problem instance, sharing information at a 1 second interval seems to perform the worst. One reason for this is that the agents are sending too much
information which causes them to have to stop the search and deal with the information. Also, the information that is sent over might not be of the best search area but yet still influences where the agents search.

On the other hand, having a sharing interval that is too big can also have a negative effect on the results. Sharing at an interval of 1 minute produced an even worse result because the agents did not receive better solutions to help them move to better parts of the solution space.

The results that we have presented in this chapter have confirmed our expectations that the library and, more specifically, the improving on the competition approach paradigm will improve on the sequential search systems and even provide synergistic improvements.
Chapter 8

Related Work

Over the years researchers have developed different methods of distributing knowledge based search. *Frameworks for Cooperation in Distributed Problem Solving* [SD81] describes two general frameworks for distributing search: task sharing and result sharing. Task sharing divides the task into sub-tasks which are then distributed among processing units. The agents work on solving the sub-tasks independently to limit expensive communication. Result sharing is similar to the improving on the competition approach paradigm in that agents assist each other by sharing solutions. Unlike improving on the competition approach, however, the agents in both the task sharing and result sharing paradigms are homogeneous.

Another paper (see [LLL92]) describes a negotiation based distributed search paradigm. The agents in this paradigm are heterogeneous and communicate using a shared memory model such as a blackboard. In negotiation search, the main problem is divided into sub-problems for agents to work on. As agents find solutions to the sub-problems they share these solutions by writing it to the blackboard. Once published, the other agents critique the proposed solutions and only when all the agents consider a solution to be valid is it accepted.

Denzinger, in [Den02], describes three distributed search paradigms: namely the central common search state, the dividing problems into sub-problems, and the improving on the competition approach paradigms. Unlike other distributed search paradigms, the three paradigms presented in the paper have been generalized such that they support many different types of search including set-based and tree-based search.

Researchers have also considered different strategies for distributing specific types of search. The paper *Strategies for Parallel Implementation of Metaheuristics* [CMRR02] re-
views different strategies of distributing set-based search. The paper [BHX01] provides two different methods of distributing tabu search (an optimisation strategy belonging in the set-based search class). Methods of distributing genetic algorithms have also been discussed in [AT99], [BCS96], [BB93], and [BT92]. In addition to distributing set-based search, researchers have also developed strategies for distributing branch and bound search in [GC94], [EPH01], and [LRT01]. A more general framework for distributing tree-based search can be found in [XRLS05].

Unfortunately not many software libraries that support building distributed search systems are available for developers to use. The following sections discuss the general distributed search libraries that are available for developers to build distributed search systems.

8.1 DisSLib:CC

DisSLib:CC [Ken10] is a distributed search library for the central common search state paradigm. As with DisSLib:ICA, the main objective of DisSLib:CC is to provide developers with a tool to build distributed search systems faster. This goal is accomplished in the same manner as DisSLib:ICA, that is, handle all the communication between agents and provide tools to allow developers to configure and setup a distributed search system. DisSLib:ICA and DisSLib:CC share a common skeleton structure (since both libraries are designed from the same definitions of search). This is intentional as we want to allow developers that have systems built with DisSLib:CC to easily transition to DisSLib:ICA.

Where DisSLib:ICA and DisSLib:CC differ is in the paradigm that each library supports. Systems built with DisSLib:CC contain a central Search State Agent (SSA) that acts as the common search state and resides on its own processing unit. This central agent controls access to the search state which means that in order for search agents to do work they must first request transitions from SSA. Once the search agents finish processing the transitions that they have received from the SSA they will send the results back to integrate into the
SSA. The central common search state paradigm can only support homogeneous agents and produce, at best, a linear speedup while the improving on the competition approach paradigm can support heterogeneous agents and produce synergistic speedups. The communication channels for the two paradigms also require different implementations as the improving on the competition approach needs to support the different types of positive and negative information while the central common search state paradigm needs to provide communication channels for the central search state agent.

When configuring a DisSLib:CC system care must be taken in choosing the number of transitions each search agent receives from the SSA. Since communication between the SSA and the search agents is required whenever the search agents request transitions, if the SSA sends only a single transition at a time then the majority of an agent’s time will be spent in communication rather than doing work. However, sending too many transitions is also not ideal because the agents will be working in older areas of the search space while the SSA might have discovered more interesting areas to explore during that time.

One advantage that DisSLib:CC systems have over DisSLib:ICA systems is that it is easy for the former system to use as many processing units as there are available. For DisSLib:ICA systems to make use of additional processing capabilities the developers will need to add new agents by either modifying the search control or search models, as having two agents that are identical is not of any benefit to the search.

A disadvantage of DisSLib:CC is that unlike DisSLib:ICA systems, where agents can produce synergistic speedups over a sequential system, the best speedup a DisSLib:CC search system can hope to attain is linear relative to the number of search agents used and the processing units available to these search agents.
8.2 TWLib

TWLib [DL96] is another library that aims to help developers build distributed knowledge-based search systems. To distribute knowledge-based search TWLib uses a distribution concept called Teamwork, a more specialized application of the improving on the competition approach.

At any point in time a Teamwork system is either in one of three different phases: a competition phase, a judgment phase, or a cooperation phase. Initially the system starts in the competition phase where search agents (classified as either a specialist or an expert) work on solving the problem. After a certain amount of time has passed the agents come together in a team meeting and the system switches to the judgement phase.

In the judgment phase, the search agents switch to a referee role allowing them to judge the results produced so far. The referees have two tasks. The first task is to measure the overall progress of the search, that is, to judge how well the search of an agent is performing. The other task is to select outstanding results that are produced by their search agents. Both of these information pieces are sent to the supervisor.

In the cooperation phase, the supervisor compares all the results that it has received and selects the best agent. At the end of this phase the supervisor generates a new start state (out of the state of the best agent and the selected results of the other agents) along with a list of agents that will be participating in the next team meeting. Once all the search agents receive the new start state the search agents start searching from the new start state using their search controls.

DisSLib:ICA and TWLib are both search libraries that try to improve on the competition approach by allowing search agents to share information. However, the sharing of information among the agents is more involved in TWLib, as it requires all the agents to stop what they are doing. If one agent takes too long then the rest of the agents will have to wait thus squandering the time that could otherwise be used to work on the search problem. Also,
TWLib requires synchronous communication which forces the agents to share information all at the same time even if they do not have good information to share. DisSLib:ICA agents, however, can send to other agents at different intervals allowing for agents to share information at a more opportune time. Another important difference between the two libraries is that TWLib only supports set-based search problems. This limitation reduces the number of search methods that TWLib can support.

TWLib was designed almost two decades ago and while it still can be used today its limitations prevent it from being of benefit to modern distributed search systems.

8.3 ParadisEO

Like DisSLib:ICA, ParadisEO (Parallel and Distributed Evolving Objects) \cite{CMT04} is designed to remove the complexities of building a distributed search system and to allow developers to focus solely on the search problem. It provides classes that developers can extend to implement systems that solves their search problems. Unlike DisSLib:ICA, however, ParadisEO only supports set-based search, more specifically evolutionary algorithms and local searches.

The ParadisEO framework consists of three layers each of which are modular by design allowing them to be extended to provide the flexibility that is required to support different types of set-based search. The first layer is made up of Helpers. The Helpers are low-level classes that support the functions of a search process. These classes are divided into two categories (evolutionary and local search helpers) for the two classes of search that the framework supports. In addition, there are Helpers dedicated to the management of parallel and distributed communication services.

The next layer consists of runners which are a set of classes that implement the search. Runners are equivalent to the search control in DisSLib:ICA in that they operate on transitions to take the search state to the final state. Also, like DisSLib:ICA, runners can operate
either sequentially or distributed among different processing units.

The last layer of ParadisEO is the solvers layer. Solvers generate the start state and control the process of the search and can be either a single metaheuristic solver or multiple metaheuristic solvers. Multiple metaheuristic solvers allow the system to run runners with different metaheuristics.

Concentrating on just set-based search allows ParadisEO to provide specialized classes that help developers implement set-based search systems with less effort than that of DisSLib:ICA. For example, while DisSLib:ICA and ParadisEO both provide methods for selecting transitions and checking whether a search can continue, ParadisEO also provides specific classes to support tabu search and other heuristics that DisSLib:ICA users will have to implement themselves. However, the general nature of DisSLib:ICA does not require the library to focus on a few specific search heuristics as it can support many different types of search and agents that are more heterogeneous than heterogeneous runners.

8.4 MALLBA

MALLBA [AAB+02] is another library that developers can use to implement their distributed search systems. It provides developers with different skeleton classes to support the development process. Like DisSLib:ICA, the skeleton classes are abstractions of the search process.

The skeleton interfaces in DisSLib:ICA are generalized to be able to support many different types of search. In MALLBA, however, the skeletons are designed to support only a specific type of search. For example, building a branch and bound search system would require the developers to use the branch and bound skeleton classes while implementing a simulated annealing search system would require a different set of skeleton classes. Each skeleton includes a Problem and Solution class that developers are required to implement. The Problem class allows developers to describe features relating to the problem and the So-
olution class describes features of the feasible solutions relevant to the search problem. Along with the Solution and Problem classes, each skeleton also provides a Solver and Setup class. The Setup class allows the search to be tuned and the Solver class provides methods to run the skeleton and to change its state.

The distribution paradigm that MALLBA employs is dependent on the skeleton used. For instance, a branch and bound search uses a master-worker paradigm ([ANP03]) of distributing the search while a tabu search system can choose between either the independent runs paradigm ([CTG97]) or the master-worker paradigm.

Both DisSLibICA and MALLBA allow developers to take an existing sequential system and adapt it to a distributed search system. However, MALLBA and DisSLib:ICA have taken different approaches to accomplish this. DisSLib:ICA’s skeleton classes have been generalized to allow it to support many types of search. MALLBA also supports many different search problems but that support is explicitly built into the skeleton of the system. This limits the number of available system that the library can support unless developers take it upon themselves to extend MALLBA. However, extending MALLBA requires an understanding of how it functions and knowing the necessary classes that needs to be modified which adds more burden on the developer.
Distributed systems have always been among the harder systems to build due to their non-sequential nature. However, with more computing power available to developers than ever before, it is now necessary for developers to be able to build distributed systems if they wish to improve the speed of their search systems. For applications and systems that lend itself to being distributed there are many distribution libraries available to remove the complexity for the developer. Unfortunately, knowledge based search is not one of these applications. Because of this, there are not many distributed search libraries available to help developers build distributed knowledge-based search systems. The few libraries that are available mostly specialize in specific domains or techniques which makes them hard to use with search problems they were not designed for.

As we feel that there is a need for researchers and industry professionals to have good general distributed search libraries, we have set out to build a distributed search library that is general enough to be used by most search applications. The result of our work is DisSLib:ICA. The foundation of the library is built from generalized search components. From these components, we were able to construct generic programming interfaces that are the structural foundation of search applications that use the library.

Although there are several distributed search paradigms available we have decided to support just the improving on the competition approach paradigm. One of the benefits of the competition approach is the ability for heterogeneous agents to participate in the search. The ICA paradigm improves on the competition approach by allowing agents to share control and state information with each other. To limit irrelevant or redundant communication, both the send and receive referees are used. This sharing of information has shown in both
our experiments and those of other researchers to have the potential to produce positive synergies.

DisSLib:ICA allows developers to build new distributed search systems with relatively the same amount of effort as it does to build a sequential search system. This is possible because all the communication and multi-threaded code is handled by our library. The developer only needs to focus on writing code relating to the search. As we have shown, existing sequential search systems can be converted to a distributed search system using DisSLib:ICA with a relatively small amount of effort. Since our library is designed to be very general, search systems that do not use the same structure can be adapted to fit the library quite easily.

The improving on the competition approach paradigm works best when the agents are heterogeneous or have rather different search controls. This factor tends to have developers only using a small number of processing units available, as creating new agents and modifying search control requires work. The central common search state paradigm, however, can usually use as many processing units as available since the agents are homogeneous and they receive work through the central search agent. This paradigm works best in a shared memory model (such as on multi-core processors), as all agents read and write to a central location. These two paradigms complement each other well and it would be an interesting to see DisSLib:ICA integrate the central common search state paradigm. It would work by having the heterogeneous agents distributed onto different mutli-core processors. Each agent can then use the central common search state paradigm to distribute work to all the available cores. When the time comes to share information, the send referees would look at the central common search state for the latest information to send. This would allow the library to make use of as much processing power as is available.

In addition to adding support for more distributed search paradigms, we would also like to improve the usability of the library by providing graphical data about a search run.
Currently, each time a search is executed, the only data that developers have is textual information that they have decided to record. We would like to expand on that by providing developers with a visualization of the search that would include charts and graphs to show the statistics of the search run. It would provide information such as the number of transitions processed per second and how often a new solution is found. In addition to the statistics the extended library would provide, we would also like to include a framework to allow developers to easily plug into the visualization to allow them to present information they find relevant to their search. This tool would provide developers a better understanding of how effective their search parameters are and allow them to easily compare different configurations.
Bibliography


