THE IMPACT OF KNOWLEDGE ON ALGORITHM PERFORMANCE IN DISCRETE OPTIMIZATION

Xiaomin Zhong

M.E., Xian Jiaotong University, 1992
B.S., Xian Jiaotong University, 1989

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
at the
University of Connecticut
2004
APPROVAL PAGE

Doctor of Philosophy Dissertation

THE IMPACT OF KNOWLEDGE ON ALGORITHM PERFORMANCE IN DISCRETE OPTIMIZATION

Presented by

Xiaomin Zhong, M.E., B.E.

Major Advisor

Eugene Santos Jr.

Associate Advisor

Lester Lipsky

Associate Advisor

Robert McCartney

Associate Advisor

Swapna Gohkale

Associate Advisor

Eunice E. Santos

University of Connecticut

2004
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Eugene Santos, for his insight and guidance throughout this effort; Dr. Lester Lipsky, Dr. Robert McCartney, Dr. Swapna Gokhale and Dr. Eunice E. Santos for their advice during my doctoral research.

I thank my husband Shangli. Without his support, I do not think I can finish the course.

Finally, I want to thank Feng and Fei, Greg, Hien, Hui, Weimin and all the many other graduate students who have shared the agony and the ecstasy with me.
# TABLE OF CONTENTS

Chapter 1: Introduction and Motivation .............................................. 1

Chapter 2: Knowledge and Algorithm Performance ......................... 7

2.1 Directional Tree Model .......................................................... 10
2.2 Analysis of knowledge Impacts on Algorithm Performance ........ 17
2.3 Case Study: Deterministic Algorithms and DT model ............... 31
2.4 Generalized Directional Tree Model ......................................... 39
2.5 Summary ................................................................................. 45

Chapter 3: Structure-based Approach ............................................. 48

3.1 Background: GA and RL .......................................................... 50
  3.1.1 A Brief Introduction To GA ..................................................... 50
  3.1.2 A Brief Introduction to RL ...................................................... 57
3.2 Design of Structured-based Approach ....................................... 63
  3.2.1 Genetic algorithm with structure operations ......................... 65
    3.2.1.1 Representation .............................................................. 66
    3.2.1.2 Selection ..................................................................... 66
    3.2.1.3 Crossover .................................................................... 66
    3.2.1.4 Mutation ..................................................................... 67
  3.2.2 Reinforcement Learning systems for Grouping Identification 68
    3.2.2.1 Architecture ............................................................... 69
    3.2.2.2 Learning Algorithm ....................................................... 70
LIST OF TABLES

2.3.1 A Traveling Salesman Problem ............................................. 36
3.1.1 The Initial population .................................................. 55
3.1.2 The next generation population ....................................... 55
3.3.1 Performance Comparison on real Bayesian Networks. .......... 80
3.3.2 Performance Comparison on small Bayesian Networks with exponential distribution ................................................. 81
3.3.3 Performance Comparison on small Bayesian Networks with flat distribution ................................................................. 81
3.3.4 Performance Comparison on large Bayesian Networks with exponential distribution ...................................................... 82
3.3.5 Performance Comparison on large Bayesian Networks with flat distribution ................................................................. 83
3.3.6 Results of Standard GA on trace Networks for 500 Generations .. 84
3.3.7 Results of Structure-based GA on trace Networks Given CPU time

\[ T_2 < T_1. \] ................................................................. 85
3.3.8 Results of Structure-based GA on trace Networks Given Solution

\[ S_3 > S_1. \] ................................................................. 85
3.4.1 Summary of Set Characteristics ........................................ 90
3.4.2 Performance Comparison on Set I ..................................... 91
3.4.3 Performance Comparison on Set II ................................. 91
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.4</td>
<td>Performance Comparison on Set III</td>
<td>92</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Performance Comparison on Set IV</td>
<td>92</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Performance Comparison on Set I Considering Machine Cost</td>
<td>93</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Performance Comparison on Set II Considering Machine Cost</td>
<td>93</td>
</tr>
<tr>
<td>3.4.8</td>
<td>Performance Comparison on Set III Considering Machine Cost</td>
<td>94</td>
</tr>
<tr>
<td>3.4.9</td>
<td>Performance Comparison on Set IV Considering Machine Cost</td>
<td>94</td>
</tr>
<tr>
<td>3.4.10</td>
<td>Performance Comparison on Job Specification</td>
<td>95</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1.1 A sample DT with $X = \{1, 2, 3\}$ .......................... 13

3.1.1 Pseudo-code of the standard genetic algorithm ................... 54

3.3.1 A small example of Bayesian network .......................... 76

3.3.2 Performance on trace network .............................. 80

3.4.1 An instance of TFISP and the corresponding graph $G$. Here the number of job classes is 3, and the number of different machine classes is 2. Machine 1 can handle jobs in class 1 and 3, and machine 2 can handle jobs in class 2 and 3. ............... 88
THE IMPACT OF KNOWLEDGE ON ALGORITHM PERFORMANCE IN DISCRETE OPTIMIZATION

Xiaomin Zhong, Ph.D.

University of Connecticut, 2004

Discrete optimization problems are usually NP hard. When choosing or designing an algorithm for solving a discrete optimization problem domain, if we have some knowledge about the characteristics of that problem domain, such knowledge will help us to make a decision. Thus, it is important to understand the relationship between knowledge about problem domains and algorithm performance. The goal of my research is to explore the impact of knowledge on algorithm performance and how to extract and incorporate knowledge into algorithm, especially during problem solving in a systematic way.

To achieve our goal, by restricting knowledge about a problem domain as a distribution of optimal solutions over the solution space, we developed a Directional Tree (DT) model which concurrently describes algorithm behavior and represents knowledge explicitly, and thus provides us with a tool to analyze the impact of knowledge on algorithm performance. We then generalize our DT model to take into account acquisition of problem knowledge during execution. Our generalized directional tree (GDT) model provides a framework for incorporating knowledge
obtained online into algorithms, and thus enables us to develop online optimization algorithms that dynamically update their search decisions in response to knowledge obtained so far. Using GDT, we properly developed a structure-based approach which can adapt to problems by incorporating structural knowledge extracted online. More specifically, we used a reinforcement learning system to adaptively learn the structural characteristics of the problem, thereby grouping the decision variables of the problem into several groups. We then develop a structure-based Genetic Algorithm by introducing structural operations to work on these groups and recombine them to search for a better solution. We compared our approach against standard GA by testing on two NP-hard problems, belief revision over Bayesian Networks and the Tactical Fixed Interval Scheduling Problem. From experimental results, we see that for some problems our new online structural approach obtains better results than the standard GA.
Chapter 1

Introduction and Motivation

Discrete optimization problems are ubiquitous in real world. For example, in industrial processing and manufacturing areas, we may need to optimize use of resource, minimize waste, maximize throughput, etc. Such problems are computationaly hard. Mathematically, these problems can be formulated as

$$\min f(x_1, x_2, \ldots, x_N)$$  \hspace{1cm} (1)

where $x_1, x_2, \ldots, x_N$ are decision variables. Each decision variable has a finite number of values. A complete assignment to these decision variables form a possible solution.

Many approaches have been proposed to solve discrete optimization problems. The most common approaches that have been proposed for solving discrete optimization problems are: integer programming methods[73], branch and bound method [46], simulated annealing method [36], genetic algorithms [25],
tabu search[23, 24], or neural network [32]. However, how to efficiently solve the
general problem is still an open question.

When choosing or designing an algorithm to search for the optimal solution
of a discrete optimization problem, if we have some knowledge about the charac-
teristics of that problem domain, such knowledge will help us to make a decision.
Clearly, different problems have different characteristics and will affect the per-
formance of each algorithm. Thus, it is important to understand the relationship
between knowledge about problem domains and algorithm performance. Such an
understanding will provide us with useful hints and better prediction of algorithm
behavior when we choose or design algorithms.

Culberson [11] mentioned an interesting example about how knowledge can
affect the search decision.

In the movie UHF, there is a marvelous scene. As the camera slowly
pans across a small park setting, we hear a voice repeatedly asking “Is
this it?” followed each time by the response “Nah!” As the camera
continues to pan, it picks up two men on a park bench, one of them
blind and holding a Rubik’s cube. He randomly gives it a twist,
then holds it up to his friend to repeat the question/answer sequence
yet again. This is blind search. Now let us suppose that instead of
being blind, the searcher is only perhaps color blind, or of very poor
eyesight. In particular, notice that the search will know the general
structure of the cube and its basic operations. Also, let us suppose
that the friend is a little less miserly in the information he is willing
to impart. Suppose the response to the queries takes on the form
“you have 5 red cells on the red face, 3 blue cells on the blue face,
…” and so on. In this way the searcher has more information to
guide his search. He may not know which face is the blue one, but as
he performs more probes, he will gain more insight and so be able to
give his search some direction.
In the literature on diverse areas related to search algorithms, there are numerous papers that address the issue of no knowledge search engines. For example, Genetic algorithms (GAs) are often thought as “no prior knowledge” algorithms. This means that we expect GAs to perform without special information. However from our experience, standard GA is sensitive to the structure of the problem and may perform poorly in some cases. Wolpert and Macready in their *No Free Lunch Theorems* [72] discussed that if we have knowledge of problem characteristics but do not incorporate them into an algorithm, then there is no formal assurance that this algorithm will be effective. So it is important not only to understand the relationship between knowledge about problem domains and algorithm performance, but also how to extract and incorporate such knowledge into an algorithm.

For my PhD work, we first sought to explore the impact of such knowledge on algorithm performance/behavior. To do so, we restricted the knowledge about a problem domain as a distribution of optimal solutions over the solution space. We developed a Directional Tree (DT) model [77] which concurrently describes algorithm behavior and represents knowledge explicitly, and thus provided us with a tool to analyze the impact of knowledge on algorithm performance.

Wolpert and Macready[72] developed a framework to explore the connection between effective optimization algorithms and the problems they are solving. They focus on the point that for any algorithm any elevated performance over one class of problems is exactly paid for in performance over another class. Our
DT model focuses on the impact of knowledge on algorithm performance. Additionally, their formulation is significantly different from our model.

Our DT model assumes that knowledge about the problem is obtained prior to the execution of any probabilistic or deterministic algorithm. However, such prior knowledge may not be readily or immediately available. Hence, what about knowledge that may be obtained during execution either externally or inferred from the execution results thus far? In this thesis, we generalize our DT model to take into account acquisition of problem knowledge during execution. Our generalized directional tree (GDT) model provides a framework for incorporating knowledge obtained online into algorithms, and thus enables us to develop online optimization algorithms that dynamically updates their search decisions in response to obtained knowledge so far.

In most cases, we may not have a complete distribution of the optimal solutions. However, we may have some knowledge about the structural characteristics of discrete optimization problems. For example, the decision variables of a discrete optimization problem have some correlations among them, and can be grouped together into several groups according to these correlations. These groupings may provide us with some knowledge about the characteristics of the solution space, and indicate a good direction for an algorithm to search for optimal solutions.

We developed a structure-based approach [74, 34, 75, 76] by considering the correlations among decision variables. This structure-based approach employs an
online algorithm. The GDT model provides us with a basic idea for building a framework for this approach. More specifically, the decision variables of a discrete optimization problem have some correlations among them, and can be grouped together into several groups according to these correlations. We use a Reinforcement Learning (RL) system to learn the structural characteristics of the problem, thereby grouping the decision variables of the problem into several groups based on the structural characteristics. We then use these structural characteristics to develop a new Genetic Algorithm (GA) by introducing structural operations to work on these groupings and recombine them together to search for a better solution to the problem. During run time the system evaluates the performance of the GA and continues with reinforcement learning to further tune the search for a better grouping. We compared our approach with standard GA by testing on two problems, belief revision over Bayesian Networks and the Tactical Fixed Interval Scheduling Problem. From our experimental results, we can see that our structure-based GA can get better solutions than the simple GA for solving problems with skewed solution space. These results help demonstrate how knowledge can affect algorithm performance.

This dissertation is outlined as follows. Chapter 2 introduces the DT model and how to analyze the impact of knowledge on algorithm performance/behavior by using it. It also introduces GDT and ideas behind our online algorithm. Chapter 3 describes our structure-based approach and presents our experimental results by testing this approach on two NP hard problems, belief revision over
Bayesian Network and the Tactical Fixed Interval Scheduling Problem. Chapter 4 presents our conclusion and discusses future work.
Chapter 2

Knowledge and Algorithm Performance

In this chapter, we seek to formally model and analyze the impact of knowledge on algorithm performance or behavior. Such an analysis would benefit algorithm designers by providing a step for rigorously explaining algorithm behavior for discrete optimization problems. We propose a model which we call the Directional Tree (DT) to explore this impact. To build the DT model, we need to consider the following questions: how to describe algorithm behavior, how to represent problem knowledge, and how to define algorithm performance.

Informally, when an algorithm is used to search for an optimal solution in a problem domain, it can be viewed as specifying a search procedure over the space of possible solutions. At each search step, different algorithms will make different decisions regarding the next search step, thereby defining different search procedures. For example, the decision at each search step made by an $A^*$ [47] algorithm is different from the one made by a Genetic Algorithm [25]. Our DT model must be able to capture such differences in search procedures.
Knowledge about the characteristics of a problem domain provides us with some information about the solution space of that problem domain. For our model, we assume that such knowledge can be captured as a distribution of optimal solutions over the solution space. As we can easily see, knowledge about the distribution of optimal solutions is important in predicting algorithm behavior.

To describe algorithm behavior, we will need to define the stopping criteria for an algorithm. These stopping criteria will be our best guess as to when or where that algorithm will stop. If we assume an algorithm will stop when an optimal solution is found, the actual stopping criteria are determined by the optimal solution. Thus, knowledge about the distribution of optimal solutions will help us better capture the stopping criteria for each algorithm.

Finally, to define algorithm performance, we focus mainly on the search process taken by an algorithm. Every run of an algorithm is simply a sequence of points that are visited in the solution space. We define performance in terms of the length of the search sequence.

By capturing search procedures and stopping criteria for an algorithm, our DT model can fully describe the behavior of that algorithm. By using our DT model, we can predict the expected runtime of an algorithm, thus determining the expected performance for that algorithm.

By restricting the knowledge about a problem domain to a distribution of optimal solutions, we can use our DT model to discuss how such knowledge will affect algorithm performance. Consider a problem space $P$ which is the universal
set of discrete optimization problems. Let all problems in $P$ share the same solution space. Problems in $P$ may be classified into sets. Each of these sets forms a problem subspace $SP$ which represents a specific problem domain. First, we analyze the performance of an algorithm when that algorithm is used for solving any problems from $P$. We formally prove that no one algorithm will perform better than another. Next, we investigate the performance of an algorithm when that algorithm is used to solve problems in a given $SP$. Since each $SP$ represents a specific problem domain, this can be captured as some non-uniform distribution of optimal solutions. In this way, an algorithm’s performance will be different when solving problems for different $SP$’s. Then, we examine situations when we do not have complete knowledge about the distribution of optimal solutions and how this will affect the performance of deterministic algorithms.

Our DT model assumes that knowledge about the problem is obtained ahead of time prior to the execution of any probabilistic or deterministic algorithm. However, such prior knowledge may not be readily or immediately available. Hence, what about knowledge that may be obtained during execution either externally or inferred from the execution results thus far? In the last section of this chapter, we generalize our DT model to take into account acquisition of problem knowledge during execution. Our generalized directional tree (GDT) model will provide a framework for developing online optimization algorithms that dynamically update their searching decisions in response to knowledge obtained so far.
As we shall see, our GDT model forms the basis for our new discrete optimization algorithms in Chapter 3.

2.1 Directional Tree Model

Consider the problem space $P$ which is the universal set of discrete optimization problems. Assume all problems in $P$ share the same solution space $X$.

When an algorithm is used for solving a discrete optimization problem in $P$, it searches the solution space $X$ to find an optimal solution. Every run of that algorithm is a sequence of points that are visited in the space $X$. Here, we assume that all the points in each sequence are non-repeating. Furthermore, the search space $X$ is finite and has size $n$, i.e., $|X| = n$, and $x_i \in X$, $i = 1, \ldots, n$ are the possible solutions for problems in $P$. A Sequence Space is defined as follows:

**Definition 2.1.1.** A Sequence Space $S$ consists of all non-repeating sequences each of which is formed by points from $X$ and has length from 0 to $n$.

We model an algorithm’s behavior as a Directional Tree (DT) on $S$. A DT is a weighted tree consisting of two kinds of nodes which are sequence nodes and query nodes. The semantics of the weighted arcs are described below.

Each sequence node in the DT denotes a sequence in $S$ with length between 0 to $n$. A sequence node without any children is called a termination sequence node, otherwise it is called a continuation sequence node. Any continuation sequence node has one or more query nodes as children. Each sequence node is a sequence
representing all the points that an algorithm has explored. The outgoing arcs of a (continuation) sequence node will thus represent the possible solutions in $X$ to be visited next and extend the current sequence appropriately.

To capture the stopping criteria of an algorithm, i.e., when an optimal solution is found, we employ the second type of nodes called query nodes. Sequence nodes are indirectly connected to other sequence nodes through query nodes. Each query node represents the probability that an algorithm will stop after the next search step. Thus, there are at most two children per query node: one child representing termination and the other continuation, hence, our termination and continuation sequence nodes.

The weights on the arcs between a continuation sequence node and its children query nodes reflect the decisions made by the algorithm in its search. These weights represent the probabilities that the algorithm will proceed to the next possible sequence of solutions. The weights between a query node and its children sequence nodes correspond to knowledge about the distribution of optimal solutions and define our stopping criteria as we shall discuss in detail in the next section.

Formally, a DT can be defined as follows:

**Definition 2.1.2.** A Directional Tree built on $S$ is a weighted tree and has the following properties:

- The root node of a DT is a continuation sequence node containing the empty sequence at level 0.
• If $s$ is a continuation sequence node, it has $|X| - |s|$ children query nodes. The weights on the arcs between $s$ and its children represent the probability of proceeding from $s$ to its children.

• If a continuation sequence node $s_1$ is the parent of a query node $q$, and $q$ is the parent of a continuation sequence node or a termination sequence node $s_2$, then $s_1$ is a prefix of $s_2$ and $|s_2| = |s_1| + 1$.

• If a query node has a parent node with sequence length $n - 1$, then the query node has a single child termination sequence node with length $n$. Otherwise, it will have a child continuation sequence node and a child termination sequence node both with the same sequence. The weights on the arcs between the query node and its children represent the probability of proceeding from the query node to its children.

From our definition, we see that sequence nodes occur at the even numbered levels of the DT while query nodes appear in the odd numbered levels. Figure 2.1.1 gives an example of a DT with the solution space containing three points $X = \{1, 2, 3\}$. We use $w_s(.)$ to denote the weight on an arc connecting a continuation sequence node to a query node, where the first (.) indicates which level a continuation sequence node is located and the second (.) indicates which query node follows that continuation sequence node. And use $w_q(.)$ to denote the weight on an arc a connecting query node to a continuation sequence node or a termination sequence node, where the first (.) indicates which level a query node is located and the second (.) indicates which continuation sequence node or
Fig. 2.1.1: A sample DT with $X = \{1, 2, 3\}$

termination sequence node follows that query node. $w_q(., .)$ are defined and set later in the next section.

We also have the following lemma for DTs:

**Lemma 2.1.1.** A DT has the following properties:

- The total levels of a DT is $2n + 1$.
- At sequence level $i$ which corresponds to DT level $2i$,
  - If $i = 0$, there is only one node which is the root node.
  - If $0 < i < n$, there are $2 \prod_{j=0}^{i-1} (n - j)$ nodes. Half of nodes are continuation sequence nodes and half of nodes are termination sequence nodes.
  - Otherwise, when $i = n$, there are $n!$ nodes. All nodes are termination sequence nodes of length $n$. 
• At query level \( i \), \( 0 \leq i < n \) corresponding to DT level \( 2i + 1 \), there are 
\[
\Pi_{j=0}^{i}(n - j) \text{ query nodes}
\]

Proof.

• From the definition of a DT, we know that a DT starts with root node which is a sequence node with length 0, then follows query node, and then follows sequence node with length 1 and so on. The last level is sequence level. The length of sequence nodes in the last level is \( n \) So there are \( n + 1 \) sequence levels and \( n \) query levels. Therefore, the total levels of a DT is \( 2n + 1 \).

• We can prove the number of sequence nodes at sequence level \( i \) which corresponds to DT level \( 2i \) as follows:

  – If \( i = 0 \), it is obvious there is only one node which is the root node.

  – If \( 0 < i < n - 1 \), we use induction to prove there are \( 2 \Pi_{j=0}^{i-1}(n - j) \) nodes. Let the root node be \( s_0 \). According to the definition of a DT, the root node is at level 0 and has \( n - |s_0| = n \) children query nodes each of which has two children sequence nodes (one is termination sequence node and the other is continuation sequence). So if \( i = 1 \), there are \( 2n \) nodes which means the claim is true. We now assume that at level \( i-1 \) there are \( 2 \Pi_{j=0}^{i-2}(n - j) \) nodes. We know from the definition of a DT, for any continuation sequence node \( s_{i-1} \) in level \( i-1 \), there are \( n - |s_{i-1}| = n - (i - 1) \) children query nodes each of which has two children sequence nodes (one is termination sequence node
and the other is continuation sequence node). There are $\prod_{j=0}^{i-2}(n - j)$ continuation sequence nodes in level $i - 1$. So the total sequence nodes in level $i$ should be:

$$2(n - (i - 1)) \prod_{j=0}^{i-2}(n - j) = 2 \prod_{j=0}^{i-1}(n - j)$$

Thus, for $0 < i < n - 1$, there are $2 \prod_{j=0}^{i-1}(n - j)$ sequence nodes. Half of nodes are continuation sequence nodes and half of nodes are termination sequence nodes.

- If $i = n$, according to the definition of a DT, all sequence nodes in this level are termination sequence nodes with length $n$. The number of all possible sequence nodes with length $n$ should be $n!$.

- We can also prove the number of query nodes in each query level $i$ corresponding to DT level $2i + 1$ by induction. $i = 0$, it is obvious there are $n$ query nodes since these $n$ query nodes have the root node as parent according to the definition of a DT. We now assume there are $\prod_{j=0}^{i-1}(n - j)$ query nodes at level $i - 1$ and $i < n - 1$. We know from the definition of a DT, for any query node $q_{i-1}$ in level $i - 1$, it has two children sequence nodes (one is continuation sequence node and the other is termination sequence node). Each continuation nodes has $n - i$ children query nodes. So the total number of query nodes in level $i$ should be:

$$\left(n - i\right) \prod_{j=0}^{i-1}(n - j) = \prod_{j=0}^{i}(n - j)$$
Furthermore, the weights \( w_s(.,..) \) and \( w_q(.,..) \) have some properties. Given different \( i \) and \( j \), for weights \( w_s(i, j) \),

- if \( i = 0, j = 1, \ldots, n \), we have:

\[
\sum_{j=1}^{n} w_s(i, j) = 1 \tag{2}
\]

- if \( 1 \leq m \leq n - 1, j = 1, \ldots, \prod_{t=0}^{i-1} (n - t) \), we have:

\[
\sum_{j=1}^{\prod_{t=0}^{i-1} (n - t)} w_s(i, j) = 1, \ldots, \prod_{t=0}^{i-1} (n - t) \sum_{j=(n-i)\prod_{t=0}^{i-2} (n-t)(n-t)+1} \prod_{t=0}^{i-1} (n - t) w_s(i, j) = 1 \tag{3}
\]

If \( k \) represents the continuation sequence node index at sequence level \( i-1 \), we can rewrite the above equations as follows:

\[
\sum_{j=(k-1)(n-i)+1}^{k(n-i)} w_s(i, j) = 1 \tag{4}
\]

where \( k = 1, \ldots, \prod_{t=0}^{i-2} (n - t) \)

Given different \( u \) and \( v \), for weights \( w_q(u, v) \),

- if \( 0 \leq u \leq n - 2, v = 1, \ldots, 2 \prod_{j=0}^{u} (n - j) \), we have:

\[
w_q(u, 1) + w_q(u, 2) = 1, \quad w_q(u, 3) + w_q(u, 4) = 1, \ldots, \tag{5}
w_q(u, 2 \prod_{j=0}^{u} (n - j) - 1) + w_q(u, 2 \prod_{j=0}^{u} (n - j) - 1) = 1
\]

- if \( u = n - 1, v = 1, \ldots, n! \), we have:

\[
w_q(n - 1, 1) = w_q(n - 1, 2) = \cdots = w_q(n - 1, n!) = 1 \tag{6}
\]

Although DTs explicitly describe all the possible behaviors of an algorithm, we can provide closed-form solutions for predicting algorithm behavior as we shall see below.
Here, we use an Oracle Model which will be described in detail in the next section to capture the knowledge about the distribution of optimal solutions. The Oracle Model is our best guess as to when or where an algorithm will stop. It is obvious that different algorithms and knowledge about the distribution of optimal solutions will define different weights on the arcs. In the next section we will analyze the knowledge impacts on algorithm performance.

2.2 Analysis of knowledge Impacts on Algorithm Performance

When an algorithm is used to search for optimal solutions, it searches the points in the solution space and builds a sequence of points that are visited by that algorithm. The length of the sequence describes the runtime for that algorithm. Thus, we can define the average performance of an algorithm by using the expected length of the executed sequence. Before giving the details on how to compute the expected length by using DT, we introduce the concept of an Oracle Model (OM).

As described earlier, we restrict the knowledge about a problem domain \( SP \subset P \) to a distribution of optimal solutions. With this, we build an Oracle Model that captures the distribution for that problem domain. Intuitively, the Oracle Model is our best guess as to when or where that algorithm will stop.

**Definition 2.2.1.** An Oracle Model (OM) is a probabilistic distribution of optimal solutions over \( X \).
An Oracle Model defines the weights, $w_q(.,.)$, on the arcs between query nodes and continuation sequence nodes or query nodes and termination sequence nodes in a DT. Different OMs will define different weights $w_q(.,.)$.

The weights, $w_s(.,.)$, on the arcs between a continuation sequence node and its children query nodes are decided by an algorithm. When an algorithm is chosen, which means $w_s(.,.)$ are fixed, giving different OMs will result in different performance predictions. Thus, for each problem domain $SP$ which specifies an OM $o$, we will be able to predict the performance of different algorithms over $o$.

Given a DT $\tau(o)$ where $o$ is an OM, if set $S_T$ contains all the termination sequence nodes in DT, we can compute the expected length, denoted as $E_L(\tau(o))$, by the following:

$$\sum_{s_i \in S_T} P_i |s_i|$$  \hspace{1cm} (7)

where $s_i$ is a termination sequence node and $P_i$ is the probability of $s_i$ which can be computed by using the weights of the DT. For example, the probability of the termination sequence node $\{1,2\}$ in Figure 2.1.1 can be computed as:

$$P(1,2) = w_s(0,1)w_q(0,2)w_s(1,1)w_q(1,1)$$

When considering all discrete optimization problems in $P$, we can assume that the distribution of optimal solutions is uniform, i.e. all solutions are equally likely to be optimal. From this, we have the following theorem:
Theorem 2.2.1. If an OM $o$ is a uniform distribution and there is only one optimal solution in solution space with size $n$, then for any DT $\tau$,

$$E_L(\tau(o)) = \frac{n + 1}{2}$$

Proof. Assume that there are $n$ solutions in the search space $X$, i.e. $|X| = n$. We can build a DT $\tau$ corresponding to $X$. Let the root node at sequence level 0.

Since $o$ describes a uniform distribution of all optimal solutions, the weights $w_q(.,.)$ in $\tau$ defined by $o$ are as follows:

- for $i = 0$, if $j = 1, 3, \ldots, 2n - 1$,
  
  $$w_q(i, j) = \frac{1}{n}$$

  if $j = 2, 4, \ldots, 2n$,
  
  $$w_q(i, j) = 1 - \frac{1}{n}$$

- for $1 \leq i \leq n - 2$,
  
  if $j = 1, 3, \ldots, 2 \prod_{j=0}^{i-1}(n - j) - 1$, $w_q(i, j) = \frac{1}{n-i}$

  if $j = 2, 4, \ldots, 2 \prod_{j=0}^{i-1}(n - j)$ $w_q(i, j) = 1 - \frac{1}{n-i}$

- when $i = n - 1$, if $j = 1, 2, \ldots, n!$,
  
  $$w_q(i, j) = 1$$

The total sequence levels of $\tau$ is $n + 1$. Assume $E_{L_i}(\tau(o))$ is the expected length at sequence level $i$, where $i = 1, 2, \ldots, n$,

$$E_L(\tau(o)) = \sum_i E_{L_i}(\tau(o))$$
\[ E_{L_i}(\tau(o)) = i \sum_{\text{level } i \text{ termination nodes}} P(\text{termination nodes}) \]

According to equations (2), (3) and (4) which describe the properties for weights \( w_s(.,.) \), and equations (5) and (6) which describe the properties for weights \( w_q(.,.) \), we have:

When \( i = 1 \),

\[
E_{L1}(\tau(o)) = \sum_{k=1}^{k=n} w_s(0,k)w_q(0,2k-1) = \frac{1}{n} \sum_{k=1}^{k=n} w_s(0,k) = \frac{1}{n}
\]

When \( i = 2 \),

\[
E_{L2}(\tau(o)) = 2 \sum_{i_1=1}^{n} (w_s(0,i_1)w_q(0,2i_1-1)) \left( \sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1,i_2)w_q(1,2i_2-1) \right) 
\]

\[
= 2 \sum_{i_1=1}^{n} (w_s(0,i_1)(1 - \frac{1}{n}) \left( \sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1,i_2) \frac{1}{n-1} \right) 
\]

\[
= 2 \sum_{i_1=1}^{n} (w_s(0,i_1)(1 - \frac{1}{n}) \frac{1}{n-1} 
\]

\[
= 2 \frac{n-1}{n} \frac{1}{n-1} 
\]

\[
= \frac{1}{n}
\]

When \( i = 3 \),

\[
E_{L3}(\tau(o)) = 3 \sum_{i_1=1}^{n} (w_s(0,i_1)w_q(0,2i_1-1)) \left( \sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1,i_2)w_q(1,2i_2-1) \right) 
\]

\[
( \sum_{i_3=(i_2-1)(n-2)+1}^{i_2(n-2)} w_s(2,i_3)w_q(1,2i_3-1) ) 
\]

\[
= 3 \sum_{i_1=1}^{n} (w_s(0,i_1)(1 - \frac{1}{n}) \left( \sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1,i_2)(1 - \frac{1}{n-1}) 
\]

\[
( \sum_{i_3=(i_2-1)(n-2)+1}^{i_2(n-2)} w_s(2,i_3) \frac{1}{n-2} ) 
\]
\[
3 \sum_{i_1 = 1}^{n} (w_s(0, i_1)(1 - \frac{1}{n})(\sum_{i_2 = (i_1 - 1)(n-1)+1}^{i_1(n-1)} w_s(1, i_2)) \\
\left(1 - \frac{1}{n - 1}\right)\left(1 - \frac{1}{n - 2}\right)) \right) \\
= 3 \sum_{i_1 = 1}^{n} (w_s(0, i_1)(1 - \frac{1}{n})(1 - \frac{1}{n - 1})(\frac{1}{n - 2})) \\
= 3\left(\frac{n - 1}{n}\right)\left(\frac{n - 2}{n - 1}\right)\left(\frac{1}{n - 2}\right) \\
= 3\frac{1}{n}
\]

We can do the similar computation for \(i = m, m = 4, \ldots, n\)

\[
E_{L_{cm}}(\tau(o)) = m(1 - \frac{1}{n})(1 - \frac{1}{n - 1})(1 - \frac{1}{n - 2})\cdots \\
\left(1 - \frac{1}{n - (m - 2)}\right)\left(\frac{1}{n - (m - 1)}\right) \\
= m\frac{1}{n}
\]

From the above, the expected length:

\[
E_L(\tau(o)) = \frac{1}{n} + \frac{2}{n} + \cdots + \frac{n}{n} \\
= \frac{n + 1}{2}
\]

\[\square\]

Theorem 2.2.1 assumes that there is only one optimal solution in the solution space. What will happen if there are \(k\) (\(1 \leq k \leq n\)) optimal solutions?

Since there are \(k\) optimal solutions in the search space, the total query level will be \(n - k\). Assume an OM \(o\) still describes a uniform distribution of all optimal solutions, then the weights \(w_d(\ldots)\) in \(\tau\) defined by \(o\) are as follows:
• for \( i = 0 \), if \( j = 1, 3, \ldots, 2n - 1 \),

\[
w_q(i, j) = \frac{k}{n}
\]

if \( j = 2, 4, \ldots, 2n \),

\[
w_q(i, j) = 1 - \frac{k}{n}
\]

• for \( 1 \leq i < n - k \),

if \( j = 1, 3, \ldots, 2 \prod_{j=0}^{i-1}(n - j) - 1 \),

\[
w_q(i, j) = \frac{k}{n - i}
\]

if \( j = 2, 4, \ldots, 2 \prod_{j=0}^{i-1}(n - j) \)

\[
w_q(i, j) = 1 - \frac{k}{n - i}
\]

• when \( i = n - k \), if \( j = 1, 2, \ldots, \prod_{j=0}^{i-1}(n - j) - 1 \),

\[
w_q(i, j) = 1
\]

The total sequence levels of \( \tau \) is \( n - k + 1 \). Let \( E_L^k(\tau(o)) \) represent the expected length and \( E_{L_i}^k(\tau(o)_k) \) represent the expected length at sequence level \( i \) when there are \( k \) optimal solutions, where \( i = 1, 2, \ldots, n - k + 1 \),

\[
E_L^k(\tau(o)) = \sum_i E_{L_i}^k(\tau(o))
\]

\[
E_{L_i}^k(\tau(o)) = i \sum_{\text{level } i \text{ termination nodes}} P(\text{termination nodes})
\]

According to equations (2), (3) and (4) which describe the properties for weights \( w_s(\ldots) \), and equations (5) and (6) which describe the properties for weights
$w_q(., .)$, we have: when $i = 1$,

$$E^k_{L_1}(\tau(o)) = \sum_{k=1}^{k=n} w_s(0, k)w_q(0, 2k - 1)$$

$$= \frac{k}{n} \sum_{k=1}^{k=n} w_s(0, k) = \frac{k}{n}$$

when $i = 2$,

$$E^k_{L_2}(\tau(o)) = 2 \sum_{i_1=1}^{n} (w_s(0, i_1)w_q(0, 2i_1 - 1)(\sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1, i_2)w_q(1, 2i_2 - 1)))$$

$$= 2 \sum_{i_1=1}^{n} (w_s(0, i_1)(1 - \frac{k}{n})(\sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1, i_2)(\frac{k}{n-1})))$$

$$= 2 \sum_{i_1=1}^{n} (w_s(0, i_1)(1 - \frac{k}{n})(\frac{k}{n-1}))$$

$$= 2(1 - \frac{k}{n})(\frac{k}{n-1})$$

$$= \frac{2k(n-k)}{n(n-1)}$$

When $i = 3$,

$$E^k_{L_3}(\tau(o)) = 3 \sum_{i_1=1}^{n} (w_s(0, i_1)w_q(0, 2i_1 - 1)(\sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1, i_2)w_q(1, 2i_2 - 1))$$

$$+ (\sum_{i_3=(i_2-1)(n-2)+1}^{i_2(n-2)} w_s(2, i_3)w_q(1, 2i_3 - 1)))$$

$$= 3 \sum_{i_1=1}^{n} (w_s(0, i_1)(1 - \frac{k}{n})(\sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1, i_2)(1 - \frac{k}{n-1})))$$

$$+ (\sum_{i_3=(i_2-1)(n-2)+1}^{i_2(n-2)} w_s(2, i_3)(\frac{k}{n-2})))$$

$$= 3 \sum_{i_1=1}^{n} (w_s(0, i_1)(1 - \frac{k}{n})(\sum_{i_2=(i_1-1)(n-1)+1}^{i_1(n-1)} w_s(1, i_2)$$

$$+ (1 - \frac{k}{n-1})(\frac{k}{n-2})))$$

$$= 3 \sum_{i_1=1}^{n} (w_s(0, i_1)(1 - \frac{k}{n})(1 - \frac{k}{n-1})(\frac{k}{n-2})))$$
\[
E^k_l(\tau(o)) = 3 \left(1 - \frac{k}{n}\right) \left(1 - \frac{k}{n-1}\right) \left(\frac{k}{n-2}\right)
\]
\[
= 3 \frac{k(n-k)(n-k-1)}{n(n-1)(n-2)}
\]

We can do the similar computation for \(i = m, m = 4, \ldots, n - k\)

\[
E^k_{L,m}(\tau(o)) = m \left(1 - \frac{k}{n}\right) \left(1 - \frac{k}{n-1}\right) \left(\frac{k}{n-2}\right) \cdots
\]
\[
(1 - \frac{k}{n - (m - 2)}) \left(\frac{k}{n - (m - 1)}\right)
\]
\[
= m \frac{k \prod_{j=0}^{m-2} (n-k-j)}{(n-i) \prod_{i=0}^{n-k-1}(n-i)}
\]

At level \(n-k+1\), there are only termination nodes. The weight \(w_q(., n-k) = 1\).

\[
E^k_{L,n-k+1}(\tau(o)) = (n-k+1) \frac{\prod_{j=0}^{n-k-1} (n-k-j)}{n \prod_{i=1}^{n-k-1}(n-i)}
\]

From the above, the the expected length:

\[
E^k_L(\tau(o)) = \sum_{i=1}^{n-k+1} E^k_{L,i}
\]

\[
= \frac{k}{n} \left(1 + 2 \frac{k(n-k)}{n(n-1)} + 3 \frac{k(n-k)(n-k-1)}{n(n-1)(n-2)} + \cdots + (n-k) \frac{k \prod_{j=0}^{n-k-2} (n-k-j)}{(n-i) \prod_{i=0}^{n-k-1}(n-i)} \right)
\]

We observed that the formula (8) can be rewritten as a recursive formula as follows:

\[
Q(n) = \frac{n-k}{n} Q(n-1) + R(n)
\]

where \(R(n) = \frac{k}{n} + \frac{n-k}{n} R(n-1)\) with \(Q(k) = 1\) and \(R(k) = 1\). If we expand out \(Q(n)\), we get the following:

\[
Q(n) = \frac{n-k}{n} Q(n-1) + R(n)
\]
\[
= \frac{n - k}{n} Q(n - 1) + \frac{k}{n} + \frac{n - k}{n} R(n - 1)
\]
\[
= \frac{k}{n} + \frac{n - k}{n} (Q(n - 1) + R(n - 1))
\]
\[
= \frac{k}{n} + \frac{n - k}{n} \left( \frac{(n - 1) - k}{n - 1} Q(n - 2) + 2R(n - 1) \right)
\]
\[
= \frac{k}{n} + \frac{n - k}{n} \left( \frac{(n - 1) - k}{n - 1} Q(n - 2) + 2 \frac{k}{n - 1} + 2 \frac{(n - 1) - k}{n - 1} R(n - 2) \right)
\]
\[
= \frac{k}{n} + 2 \frac{k(n - k)}{n(n - 1)} + \frac{(n - k)(n - k - 1)}{n(n - 1)} (Q(n - 2) + 2R(n - 2))
\]
\[
= \frac{k}{n} + 2 \frac{k(n - k)}{n(n - 1)} + \frac{(n - k)(n - k - 1)}{n(n - 1)} \left( \frac{n - 2}{n - 2} Q(n - 3) + 3R(n - 2) \right)
\]
\[
= \frac{k}{n} + 2 \frac{k(n - k)}{n(n - 1)} + \frac{(n - k)(n - k - 1)}{n(n - 1)} \left( \frac{n - 2}{n - 2} Q(n - 3) \right)
\]
\[
+ 3 \frac{k}{n - 2} + 3 \frac{(n - 2) - k}{n - 2} R(n - 3)
\]
\[
= \frac{k}{n} + 2 \frac{k(n - k)}{n(n - 1)} + \frac{3k(n - k)(n - k - 1)}{n(n - 1)(n - 2)}
\]
\[
+ \frac{(n - k)(n - k - 1)(n - k - 2)}{n(n - 1)(n - 2)} (Q(n - 3) + 3R(n - 3))
\]
\[
= \ldots =
\]
\[
= \frac{k}{n} + 2 \frac{k(n - k)}{n(n - 1)} + \frac{3k(n - k)(n - k - 1)}{n(n - 1)(n - 2)} + \ldots
\]
\[
+ \frac{\Pi_{j=0}^{n-k-1}(n - k - j)}{n \Pi_{i=1}^{n-k-1}(n - i)} (Q(k) + (n - k)R(k))
\]
\[
= \frac{k}{n} + 2 \frac{k(n - k)}{n(n - 1)} + \frac{3k(n - k)(n - k - 1)}{n(n - 1)(n - 2)} + \ldots + (n - k + 1) \frac{\Pi_{j=0}^{n-k-1}(n - k - j)}{n \Pi_{i=1}^{n-k-1}(n - i)}
\]
\[
= E^k_n (\tau(\alpha))
\]

For this recursive formula we have the following theorem:

**Theorem 2.2.2.** If \( Q(n) = \frac{n-k}{n} Q(n - 1) + R(n) \) where \( R(n) = \frac{k}{n} + \frac{n-k}{n} R(n - 1) \)

with \( Q(k) = 1, \ R(k) = 1 \) and \( 1 \leq k \leq n \), then \( Q(n) = \frac{n+1}{k+1} \).

**Proof.** First, we prove \( R(n) = 1 \) by induction on \( n \).
When $n = 1$, since $1 \leq k < n$, $k = 1$. So $R(n) = R(k) = 1$.

Now we assume $R(n) = 1$ and consider the case $n + 1$.

$$R(n + 1) = \frac{k}{n + 1} + \frac{(n + 1) - k}{n + 1} R(n)$$
$$= \frac{k}{n + 1} + \frac{(n + 1) - k}{n + 1}$$
$$= 1 + \frac{k}{n + 1} - \frac{k}{n + 1} = 1$$

So $R(n) = 1$ for $n$. Second, we prove $Q(n) = \frac{n+1}{k+1}$ by induction on $n$.

When $n = k$, then $k = 1$. $Q(n) = Q(k) = 1 = \frac{n+1}{k+1} = \frac{k+1}{k+1}$

Now assume $Q(n) = \frac{n+1}{k+1}$ and consider the case $n + 1$.

$$Q(n + 1) = \frac{(n + 1) - k}{n + 1} Q(n) + R(n + 1)$$
$$= \frac{(n + 1) - k n + 1}{n + 1} + 1$$
$$= \frac{n + 1 - k}{k + 1} + 1 = \frac{n + 2}{k + 1}$$

So $Q(n) = \frac{n+1}{k+1}$ for all $n$. 

Based on Theorem 2.2.2, we have the following theorem:

**Theorem 2.2.3.** If an OM o is a uniform distribution and there are $k$ ($1 \leq k \leq n$) optimal solutions in the solution space with size $n$, then for any DT $\tau$,

$$E^k_L(\tau(o)) = \frac{n + 1}{k + 1}$$

**Proof.** If we expand out $Q(n)$, it is equivalent to $E^k_L(\tau(o))$. So

$$E^k_L(\tau(o)) = \frac{n + 1}{k + 1}$$
According to Theorem 2.2.1, we can easily see that the average performance in terms of the sequence length of algorithm $A$ will not outperform algorithm $B$ when these two algorithms are used for solving all problems in $P$. Also note that if you have multiple ($k$) optimal solutions, you would expect an upper bound reduction of the expected length to $\frac{n-k+1}{2}$. However, as Theorem 2.2.3 demonstrates, the actual closed-form reduction is $\frac{n+1}{k+2}$ which matches Theorem 2.2.1 when $k = 1$ but is a more significant reduction than $\frac{n-k+1}{2}$.

Assuming that the OM for $P$ is uniform, the above states that there is no single general purpose algorithm that is best at solving all problems in $P$. This part of our result is similar in the spirit of the “No Free Lunch Theorem” developed by Wolpert and Macready [72] (we describe this theorem in more detail in Appendix A). However, their formulation is significantly different from ours and has specific limitations and problems as discussed by many including Culberson [11].

As we mentioned earlier, $P$ may be classified into several $SP$s. Since each $SP$ represents a specific problem domain, this would likely be captured as some non-uniform distribution of optimal solutions. Now let us consider the situation when the distribution of optimal solutions is non-uniform for a $SP$. For example, let is take a look at a simple Traveling Salesman Problem (TSP) involving four cities. On any given day, the cost for a salesman traveling between any two cities can vary slightly. Thus, a collection of TSPs with slight variances in cost between cities forms a $SP$. Assume that a salesman is at city $A$ and must travel through
the remaining cities $B, C, D$ and return to $A$. The problem is to find the cheapest path while visiting each city exactly once except for the starting city. There exist 6 possible paths from city $A$ through $B, C, D$ and back to $A$: $ABCDA, ABDDA, ACBDA, ACDBA, ADBCA$, and $ADCBA$. Among these paths, $ABCDA$ is the same as $ADCBA$, $ABDDA$ is the same as $ACDBA$, and $ACBDA$ is the same as $ADBCA$. Therefore, there are 3 non-equivalent solutions in the solution space, $X = \{1, 2, 3\}$, where 1 corresponds to path $ABCDA$, 2 to $ABDDA$ and 3 to $ACBDA$. Since Figure 2.1.1 shows a DT with solution space containing three points, the DT for the SP of simple TSPs can also be represented by this figure.

TSPs are NP-hard. Practical methods for solving TSPs often use local search heuristics [42, 43]. For example, Iterated Local Search [43] is a well-known approach for solving TSP. ILS works as follows:

1. Find an initial solution $s$ by using local search.

2. Do the following for a given number of iterations

   (a) Perturb $s$ obtaining a new solution $t$.
   (b) Run local search on $t$ to obtaining $u$.
   (c) If $length(u) < length(t)$, set $s = u$.

3. Return $s$.

Different perturbation and local search strategies will define different probabilities for $s$ to $u$. In our DT model these probabilities are captured by the weights $W_s(., .)$. Thus different strategies will give different numbers to $W_s(., .)$ in Figure 2.1.1.
Assume that we have a complete distribution of optimal solutions for this set of TSPs and the distribution is non-uniform as defined by the OM $o$. Let $o$ be defined as follows: $P(\text{opt} = 1) = \alpha_1$, $P(\text{opt} = 2) = \alpha_2$, $P(\text{opt} = 3) = \alpha_3$. According to the OM $o$, the weights $w_q(.,.)$ in Figure 2.1.1 are as follows:

- at query level 0:

  \[
  w_q(0, 1) = \alpha_1, \quad w_q(0, 2) = 1 - \alpha_1, \quad w_q(0, 3) = \alpha_2, \\
  w_q(0, 4) = 1 - \alpha_2, \quad w_q(0, 5) = \alpha_3, \quad w_q(0, 6) = 1 - \alpha_3
  \]

- at query level 1:

  \[
  w_q(1, 1) = \frac{\alpha_2}{1 - \alpha_1}, \quad w_q(1, 2) = \frac{1 - \alpha_1 - \alpha_2}{1 - \alpha_1}, \\
  w_q(1, 3) = \frac{\alpha_3}{1 - \alpha_1}, \quad w_q(1, 4) = \frac{1 - \alpha_1 - \alpha_3}{1 - \alpha_1}, \\
  w_q(1, 5) = \frac{\alpha_1}{1 - \alpha_2}, \quad w_q(1, 6) = \frac{1 - \alpha_2 - \alpha_1}{1 - \alpha_2}, \\
  w_q(1, 7) = \frac{\alpha_3}{1 - \alpha_2}, \quad w_q(1, 8) = \frac{1 - \alpha_2 - \alpha_3}{1 - \alpha_2}, \\
  w_q(1, 9) = \frac{\alpha_1}{1 - \alpha_3}, \quad w_q(1, 10) = \frac{1 - \alpha_3 - \alpha_1}{1 - \alpha_3}, \\
  w_q(1, 11) = \frac{\alpha_2}{1 - \alpha_3}, \quad w_q(1, 12) = \frac{1 - \alpha_3 - \alpha_2}{1 - \alpha_3}
  \]

- at level 3:

  \[
  w_q(2, 1) = 1, \quad w_q(2, 2) = 1, \quad w_q(2, 3) = 1, \\
  w_q(2, 4) = 1, \quad w_q(2, 5) = 1, \quad w_q(2, 6) = 1
  \]

Assume $\alpha_1 = \frac{1}{3}$, $\alpha_2 = \frac{1}{3}$, $\alpha_3 = \frac{1}{3}$. Assume algorithm $A$ defines $w_s(.,.)$ as follows:
• at sequence level 0:

\[ w_s(0, 1) = \frac{3}{4}, \quad w_s(0, 2) = \frac{1}{8}, \quad w_s(0, 3) = \frac{1}{8} \]

• at sequence level 1:

\[ w_s(1, 1) = w_s(1, 2) = w_s(1, 3) = w_s(1, 4) = w_s(1, 5) = w_s(1, 6) = \frac{1}{2} \]

• at sequence level 2:

\[ w_s(2, 1) = w_s(2, 2) = w_s(2, 3) = w_s(2, 4) = w_s(2, 5) = w_s(2, 6) = 1 \]

Let \( \tau_A \) represent algorithm \( A \) and \( \tau_A(o) \) represent the DT we have constructed for \( A \) and \( o \). Thus, the expected length, denoted as \( E_L(\tau_A(o)) \), for algorithm \( A \) and OM \( o \) is, \( E_L(\tau_A(o)) = \frac{59}{32} \).

Assume algorithm \( B \) defines \( w_s(., .) \) as follows:

• at sequence level 0:

\[ w_s(0, 1) = \frac{1}{4}, \quad w_s(0, 2) = \frac{1}{2}, \quad w_s(0, 3) = \frac{1}{4} \]

• at level 2:

\[ w_s(1, 1) = w_s(1, 2) = w_s(1, 3) = w_s(1, 4) = w_s(1, 5) = w_s(1, 6) = \frac{1}{2} \]

• at sequence level 1:

\[ w_s(2, 1) = w_s(2, 2) = w_s(2, 3) = w_s(2, 4) = w_s(2, 5) = w_s(2, 6) = 1 \]

Let \( \tau_B \) represent algorithm \( B \) and \( \tau_B(o) \) represent the DT. We have: \( E_L(\tau_B(o)) = \frac{65}{32} \).
For a problem space $P$ with three points in its solution space, from Theorem 2.2.1, we know the expected length of any algorithm is 2 when that algorithm is used for solving all problems in $P$. Given a problem domain $SP$ like the one above which has a non-uniform distribution, we can clearly see that some algorithms will perform better over $SP$ while others may perform worse.

2.3 Case Study: Deterministic Algorithms and DT model

The discussion in the previous sections give us some insights on how knowledge about the problems affects algorithm performance. In this section, we analyze the impact of knowledge on the performance of deterministic algorithms by using our DT model. A deterministic algorithm is an algorithm with a fixed search path.\(^1\) The order of points visited by every run of a deterministic algorithm will not change. In the case of heuristic search algorithms like $A^*$, they can be represented by a set of deterministic algorithms for the different possible search paths. We begin by developing a general formula to compute the expected length for deterministic algorithms.

**Theorem 2.3.1.** Let $P(X_i = Opt) = \alpha_i$. For a deterministic algorithm $A_d$ with DT $\tau$ which searches the solution space in the order $x_1, x_2, \ldots, x_n$, the expected length of $A_d$ is

$$E_L(\tau_{A_d}(o)) = \sum_{i=1}^{n} i\alpha_i$$  \hspace{1cm} (10)

\(^1\)We are “overloading” the term deterministic algorithm in our discussion.
Proof. Since the OM $o$ is $P(x_i = Opt) = \alpha_i$, the weights $w_q(., .)$ in $\tau$ are as follows:

- at query level 0: $w_q(0, 1) = \alpha_1$, $w_q(0, 2) = 1 - \alpha_1$, $w_q(0, 3) = \alpha_2$, $w_q(0, 4) = 1 - \alpha_2$, ..., $w_q(0, 2n - 1) = \alpha_n$, $w_q(0, 2n) = 1 - \alpha_n$

- at query level $j$, $j = 1, 2, \ldots, n - 2$: let $Q_i$ be the $i$th query node at query level $j$. From the definition of the DT, we know $Q_i$ will be followed by a termination sequence node $S^T_{2i-1}$ and a continuation sequence node $S^C_{2i}$ at sequence level $j + 1$. Also these two sequence nodes have the same sequence points. Assume $S^T_{2i-1} = \{x_i, x_{i2}, \ldots, x_{ik}\}$, where $x_{it}$, $t = 1, \ldots, k$ can be any point in $X$, the weight on the arc connecting $Q_i$ and $S^T_{2i-1}$ is:

$$w_q(j, 2i - 1) = \frac{\alpha_{ik}}{1 - \alpha_{i1} - \alpha_{i2} - \cdots - \alpha_{ik-1}}$$

and the weight on the arc connecting $Q_i$ and $S^C_{2i}$ is:

$$w_q(j, 2i) = \frac{1 - \alpha_{i1} - \alpha_{i2} - \cdots - \alpha_{ik}}{1 - \alpha_{i1} - \alpha_{i2} - \cdots - \alpha_{ik-1}}$$

For example, if $S^T_{2i-1} = \{x_1, x_2, \ldots, x_m\}$,

$$w_q(j, 2i - 1) = \frac{\alpha_m}{1 - \alpha_1 - \alpha_2 - \cdots - \alpha_{m-1}}$$

$$w_q(j, 2i) = \frac{1 - \alpha_1 - \alpha_2 - \cdots - \alpha_m}{1 - \alpha_1 - \alpha_2 - \cdots - \alpha_{m-1}}$$

- at level $n - 1$:

$$w_q(n - 1, i) = 1$$

where $i = 1, 2, \ldots, n!$. 

Since the deterministic algorithm $A_d$ search the solution space from $x_1, x_2, \ldots$, to $x_n$, thus, defines the $w_s(.,.)$ in $\tau$ as follows:

- at sequence level 0, let $w_s(0,1) = 1$, $w_s(0,i) = 0$, where $i = 2, \ldots, n$.
- at sequence level $j$, $j = 1, 2, \ldots, n - 2$, let $w_s(j,1) = 1$. We do not care the actual values of other weights $w_s(j,i)$, where $i = 2, \ldots, \Pi_{j=0}^{j-1}(n - j)$ since these branches have been cut when computing the expected length of the algorithm.
- at level $2n - 2$,
  \[ w_s(n - 1, i) = 1 \]
  where $i = 1, 2, \ldots, n!$

Based on above description, the expected length $E_L(\tau_{A_d}(o))$ for $A_d$ can be computed as follows:

\[
E_L(\tau_{A_d}(o)) = \alpha_1 + 2(1 - \alpha_1) \frac{\alpha_2}{1 - \alpha_1} + 3(1 - \alpha_1) \frac{1 - \alpha_1 - \alpha_2}{1 - \alpha_1} \frac{\alpha_3}{1 - \alpha_1 - \alpha_2} + \cdots + \sum_{i=1}^{n} \frac{i\alpha_i}{1 - \sum_{j=1}^{n-1} \alpha_j}
\]

\[
= \alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots + i\alpha_i + \cdots + n(1 - \sum_{j=n-1}^{j=1} \alpha_j)
\]

\[
= \sum_{i=1}^{n} i\alpha_i
\]
Assuming that we have the full distribution of optimal solutions, i.e., all $\alpha_i$’s, are available, we can construct a deterministic algorithm $A_D$ as follows: Sort the $\alpha_i$’s in descending order. Have $A_D$ search the solution space in this order. We can prove the following concerning $A_D$:

**Theorem 2.3.2.** $A_D$ is the best performing algorithm, i.e., it has minimal expected length.

**Proof.** We prove this by contradiction. Assume $\alpha_1 > \alpha_2 > \cdots > \alpha_n$. $A_D$ searches the solution space according to the order of $\alpha_i$’s. Assume that there exists an algorithm $A_*$ with the minimal expected length and searches the space following the order $x_1^*, x_2^*, \ldots, x_n^*$ such that $\alpha_j^* < \alpha_k^*$ and $j < k$. According to equation (10), we get:

$$E_L(\tau_{A_*}(o)) = \alpha_1^* + 2\alpha_2^* + \cdots + j\alpha_j^* + \cdots + k\alpha_k^* + \cdots + n\alpha_n^*$$

If we switch $x_j^*$ with $x_k^*$, we get:

$$E'_L = \alpha_1^* + 2\alpha_2^* + \cdots + j\alpha_k^* + \cdots + k\alpha_j^* + \cdots + n\alpha_n^*$$

Since $E_L(\tau_{A_*}(o)) > E'_L$ contradicts to assumption, so the expected length of $A_D$, $E_L(\tau_{A_D})$ is minimum. □

Thus, the expected length of $A_D$ will be a lower bound for any algorithm, deterministic and otherwise, over this problem domain.

In most cases though, it is unlikely that we have complete knowledge of the distribution of the optimal solutions for a $SP$. But we may have some knowledge
to break the solution space into several clusters. For example, let us consider another TSP involving five cities. On any given day, the cost for a salesman traveling between any two cities can vary slightly. Assume that for the current day we are interested in, the cost are shown in Table 2.3.1. There exist 12 possible non equivalent paths from city A through B, C, D, E and back to A: \(ABCDEA, ABCEA, ABDECA, ABECDA, ABEDCA, ACBDEA, ACBEDA, ACDBEA, ACBDEA, ADBCEA, ADBCEA, ADCBEA\). Therefore, there are 12 solutions in the solution space. We observe that the distance between \(C\) and \(E\) is minimal and the distance between \(B\) and \(E\) is maximal. Based on this observation, we can break the solution space into 4 disjoint clusters as follows:

1. Cluster 1 contains any path with \(CE\) or \(EC\) and without \(BE\) and \(EB\).

   So there are 4 solutions in Cluster 1, \(ABCEDA, ABDECA, ABDECA, ADBCEA\).

2. Cluster 2 contains any path without \(CE\) or \(EC\) and without \(BE\) and \(EB\).

   So there are 2 solutions in Cluster 2, \(ABCDEA, ACBDEA\).

3. Cluster 3 contains any path with \(CE\) or \(EC\) and with \(BE\) and \(EB\). So there are 2 solutions in Cluster 3, \(ABECDA, ACBDEA\)

4. Cluster 4 contains any path without \(CE\) or \(EC\) and with \(BE\) and \(EB\).

   So there are 4 solutions in Cluster 4, \(ABEDCA, ACBEDA, ACDBEA, ADCBEA\).
Assume that based on solving earlier TSP problem instances, we observed the following:

1. The probability that the optimal solution is in Cluster 1 is between 30% and 40%.
2. The probability that the optimal solution is in Cluster 2 is between 20% and 30%.
3. The probability that the optimal solution is in Cluster 3 is between 10% and 20%.
4. The probability that the optimal solution is in Cluster 4 is between 5% and 10%.

By using these rules, we can define a bound on the distribution of optimal solutions for each cluster if we assume all points in a cluster have the same probability of being optimal. For example, there are 4 solutions in Cluster 1. From rule 1, we know the bound on the probability that the optimal solution is in Cluster 1. Assume that those 4 solutions are equally likely to be optimal, then the probability that a solution in Cluster 1 is optimal is between 7.5% and 10%.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>120</td>
<td>180</td>
<td>202</td>
<td>300</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>175</td>
<td>340</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>98</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>168</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3.1: A Traveling Salesman Problem
Formally, the above example can be described as the following scenario: The solution space has been classified into several disjoint nonempty clusters. Assume that we know the bound on the probability that the optimal solution is in a given cluster. Further assume that the probability of a point being the optimal solution is the same as every other point in its cluster. We can get a bound on the distribution of optimal solutions. For example, if the probability bound that the optimal solution is in Cluster $C_j$ is between $\gamma_j$ and $\beta_j$, and there are $N_j$ solutions in Cluster $C_j$, then the probability that a point in Cluster $C_j$ is optimal is between $\gamma_j/N_j$ and $\beta_j/N_j$. In this case, we cannot fully specify the individual numerical probabilities for $SP$ to build the appropriate OM. Only partial information in terms of the bounds on the clusters is provided to us.

In the above scenario, while we may not be able to find a deterministic algorithm that has the best performance based on the information we have, we still want to find an algorithm with reasonable performance. Furthermore, we want to determine a general formula that can compute an upper bound on performance for that deterministic algorithm.

Assume the solution space has been classified into $m$ nonempty clusters, $C_1, C_2, \ldots, C_m$. $N_j$ denotes the number of points in Cluster $C_j$. $\gamma_j$ and $\beta_j$ are the bound on the probability that optimal solutions are in Cluster $C_j$, and $\gamma_j \leq \beta_j$. We assume that for any point $x_i \in X$, if $x_i \in C_j$, the probability of $x_i$ being an
optimal solution satisfies:
\[ \frac{\gamma_j}{N_j} \leq P(x_i = \text{Opt}) \leq \frac{\beta_j}{N_j} \]

To simplify the description, we assume:
\[ \frac{\beta_1}{N_1} < \frac{\beta_2}{N_2} < \cdots < \frac{\beta_m}{N_m} \]

We can design a deterministic algorithm \( A_{D'} \) as follows: Let \( A_{D'} \) first search all
the points in Cluster \( C_1 \), then \( C_2 \), \ldots, \( C_m \). By using Equation (10), we can get
an upper bound of the expected length for \( A_{D'} \):
\[
E_L(\tau_{A_{D'}}(o)) \leq \sum_{j=1}^{N_1} j \frac{\beta_1}{N_1} + \left( \sum_{j=1}^{N_2} (N_1 + j) \frac{\beta_2}{N_2} \right)
+ \cdots + \left( \sum_{j=1}^{N_i} (N_{i-1} + j) \frac{\beta_i}{N_i} \right)
+ \cdots + \left( \sum_{j=1}^{N_m-1} (N_{m-1} + j) \frac{\beta_m}{N_m} \right)
\]

If algorithm \( A_{D'} \) is used to solve the TSP described before, by using the above
equation, we can get an upper bound of the expected length for \( A_{D'} \):
\[
E_L(\tau_{A_{D'}}(o)) \leq \sum_{j=1}^{4} j \cdot \frac{0.40}{4} + \sum_{j=1}^{2} (4 + j) \cdot \frac{0.30}{2}
+ \sum_{j=1}^{2} (4 + 2 + j) \cdot \frac{0.20}{2}
\]
\[
= 5.2
\]

By using the above general formula for an algorithm, we can quickly and easily
get an idea on how that algorithm is expected to perform. Furthermore, it gives
us a target to aim for when designing new and hopefully better algorithms.
2.4 Generalized Directional Tree Model

Our DT model assumes that knowledge about the problem is obtained ahead of time prior to the execution of any probabilistic or deterministic algorithm. As we saw in the previous section, such prior knowledge can allow us to construct effective deterministic algorithms. However, such prior knowledge may not be readily or immediately available. Hence, what about knowledge that may be obtained during execution either externally or inferred from the execution results thus far? In this section, we generalize our DT model to take into account acquisition of problem knowledge during execution. Our generalized directional tree (GDT) model provides a framework for developing online optimization algorithms that dynamically updates its searching decisions (continuation sequence weights, \( w_s(\cdot, \cdot) \)) in response to updating its oracle model (termination sequence weights, \( w_q(\cdot, \cdot) \)). As we shall see, our GDT model forms the basis for our new discrete optimization algorithms in Chapter 3.

In the GDT model, we consider a sequence of bags of points instead of just points as in DT. In this dissertation, we restrict the number of points in each bag to less than or equal to \( K, K < n \). So we call the GDT model as a \( K \)-bounded \( GDT \) model. By considering bags, we allow points to be revisited which can be important to randomized algorithms and in re-assessing the potential structural knowledge contained within each point. Thus, the termination/halting condition for our \( K \)-bounded GDT model is generalized as a non-repeating sequence of
bags. As such, the convergence condition of approaches such as evolutionary computation algorithms is naturally reflected here.

**Definition 2.4.1.** The length of a bag is the number of points in that bag.

**Definition 2.4.2.** Bags are non-repeating bags if any two bags $b_i$ and $b_j$ among them satisfy $b_i \neq b_j$.

**Definition 2.4.3.** A $K$-based Bag Space consists of all non-repeating bags each of which is formed by points from $X$.

We have the following lemma to compute the total number of bags in the Bag Space $B$.

**Lemma 2.4.1.** If $|X| = n$, the total number of bags in the Bag Space $B$, $N_B$ can be computed by:

$$N_B = \sum_{i=1}^{K} C(n + i - 1, i)$$

**Proof.** First let us see how to compute the number of bags with length $i$ ($i \leq n$). We can assume that we have $n$ balls denoted as $a_1, a_2, a_3, \ldots, a_n$. Each bag of balls are arranged with $a_1$’s first, $a_2$’s second and etc. Each bag can be identified with a sequence of $i$ zeros and $n-1$ ones, where each 0 stands for a ball is selected and each 1 indicates a point in the sequence where the ball number increases by 1. For example, if $i=3$ and $n=5$, then $a_1a_1a_3$ becomes 0011011, $a_3a_3a_3$ becomes 1100011, $a_1a_2a_3$ becomes 0101011, $a_2a_5a_5$ becomes 1011100. If
assume those \(i\) zeros are markers, it is equivalent to putting \(i\) markers onto \(n+i-1\) positions. The ways to place this is \(C(n + i - 1, i)\).

Second, since the bag space \(B\) is \(K\)-based, \(1 \leq i \leq K\). So:

\[
N_B = \sum_{i=1}^{K} C(n + i - 1, i)
\]

\(\square\)

**Definition 2.4.4.** A bag \(b_1\) is a subbag of \(b_2\) if \(b_1 \subset b_2\) and \(b_1 \neq b_2\).

Since \(b_1\) and \(b_2\) are bags, they may contain repeated members. \(b_1 \subset b_2\) means that all members are in \(b_1\) are also in \(b_2\).

**Definition 2.4.5.** A sequence of bags \(s_b = \{b_1, b_2, \ldots, b_m\}\) is said to be non-repeating if for each \(b_k\), \(b_k\) is not a subbag of any \(b_i\) for all \(i < k\).

Intuitively, if \(b_k\) is a subbag of \(b_i\), \(b_k\) is not likely to have provided any additional improvement or knowledge to the algorithm.

**Definition 2.4.6.** A Bag-Sequence Space \(S_B\) consists of all non-repeating sequences of bags formed with bags from \(B\).

Intuitively, given any bag-sequence, \(s_b\) from \(S_B\), for any nonempty subset \(A\) of \(X\), \(A\) appears at most once in \(s_b\). \(s_b\) has length from 0 to \(N_B\).

A \(K\)-bounded GDT is also a weighted tree consisting of bag-sequence nodes and bag-query nodes from \(S_B\). The semantics of the weighted arcs are similar to those of a DT.
Definition 2.4.7. A k-bounded Generalized Directional Tree built on $S_B$ is a weighted tree with the following properties:

- The root node of a k-bounded GDT is a continuation bag-sequence node containing the empty bag-sequence at level 0.

- If $s_b$ is a continuation set-sequence node, it has $|B \setminus (s_b \cup \{b \in B | \exists b_i \in s_b, b \subset b_i\})|$ children bag-query nodes. The weights on the arcs between $s_b$ and its children represent the probability of proceeding from $s_b$ to its children.

- If a continuation bag-sequence node $s_{b_1}$ is the parent of bag-query node $q_b$ and $q_b$ is the parent of a continuation bag-sequence node or a termination bag-sequence node $s_{b_2}$, then $s_{b_1}$ is a prefix of $s_{b_2}$ and $|s_{b_2}| = |s_{b_1}| + 1$.

- If a sequence node $s_b$ has length $N_B$, it satisfies the following:

$$\forall b \in B, \exists b_i \in s_b, b \subset b_i$$

then this sequence node has one parent bag-query node which has single child termination bag-sequence $s_b$. Otherwise, it has a parent bag-query node which will have a child.

- If a bag-query node has a parent node $s_b$ with sequence length $N_B$, it satisfies the following:

$$N_B - |\cup_{b \in B} \{b \in B | \exists b_i \in s_b, b \subset b_i\}| = 1$$

then the bag-query node has a single child termination bag-sequence node. Otherwise, it will have a child continuation bag-sequence node and a child termination bag-sequence node both with the same bag-sequence. The weights
on the arcs between the bag-query node and its children represent the probability of proceeding from the bag-query node to its children.

We note that like the DT, for any given point $a \in X$, $a$ must occur somewhere in a given K-bounded GDT. To simplify the description, all GDTs mentioned in this dissertation are K-bounded.

Although the size of any GDT built on $S_B$ is exponentially larger than any DT built on $S$, with the online nature of the GDT model, we will see that the GDT can actually be constructed dynamically. As such, this construction is directly reflected by the online algorithms derived from the GDT.

For algorithms that rely on fixed population sizes such as genetic algorithms [25, 31] and tabu search [23, 24] to name a few, the bags in the bag-sequences of a GDT for these algorithms are also of fixed size.

Assuming a fixed size of $m$, we can naturally model this in our GDT by setting the bag-continuation weights to 0 for transitions to bags not of size $m$.

As we mentioned earlier, additional knowledge of the given problem can be obtained as we explore the space of solutions. With such additions, we can then modify our algorithm behavior online to better exploit such derivations.

Recall that as an algorithm executes, it moves along its associated GDT beginning at the root bag-sequence node down through the descendants. Let $o_i$ be the OM obtained after processing bag-query level $i - 1$ in the GDT $\tau$ for $i > 0$ and $o_0$ is the initial OM. Basically, $o_{i+1}$ represents the additional knowledge obtained after processing the bag-sequence node at level $i$. 
Since our target algorithm is online in nature, this implies that the weights \( w_k(\cdot, \cdot) \) will change as the algorithm executes. In addition, since the GDT is constructed dynamically, we can also view the online process as a sequence of partial GDTs. In particular, we define partial GDTs as follows:

**Definition 2.4.8.** A partial GDT \( \tau_i \) of \( \tau \) is a tree such that the following holds:

- The root of \( \tau \) is in \( \tau_i \)
- \( \tau_i \) is a connected subtree of \( \tau \)
- All the leaves of \( \tau_i \) are bag-sequence nodes.
- \( \tau_i \) has at least 2 levels.

Furthermore, given two partial GDTs \( \tau_1 \) and \( \tau_2 \) of \( \tau \), we say that \( \tau_2 \) is a descendant of \( \tau_1 \) if and only if by removing the bottom-most two levels of \( \tau_2 \), we are left with \( \tau_1 \).

To represent our online algorithm, when given some OM \( o_i \) and partial GDT \( \tau_{i-1} \), we construct descendant \( \tau_i \).

**Definition 2.4.9.** An online GDT over BS is a 3-tuple \((o_0, F_\tau, F_o)\) where

- \( o_0 \) is an OM over \( X \) designated the initial OM,
- \( F_\tau \) is the partial GDT builder of the form \( F_\tau(o_i, \tau_{i-1}, bs) = \tau_i \), i.e., constructs the descendant of \( \tau_{i-1} \) from \( o_i \) and contains \( bs \) as an interior node where \( bs \) is a bag-sequence node, and,
- \( F_o \) is the OM augmentor of the form \( F_o(o_i, \tau_i, bs) = o_{i+1} \) where \( bs \) is a set-sequence node.
The construction of the online algorithm proceeds as follows: At step $i$, given $\tau_{i-1}$ and $o_{i-1}$,

1. Evaluate/execute $\tau_{i-1}$. If optimal solution found, stop. Otherwise, identify the selected continuation bag-sequence node $bs$.

2. Construct new $o_i = F_o(o_{i-1}, \tau_{i-1}, bs)$.

3. Construct new $\tau_i = F_T(o_i, \tau_{i-1}, bs)$.

4. Repeat loop.

$o_0$ is the initial OM and $\tau_0$ is constructed from $o_0$. This results in a sequence of partial GDTs $\{\tau_0, \tau_1, \ldots, \tau_k\}$ and OMs $\{o_0, o_1, \ldots, o_k\}$.

We have now presented a framework for developing online algorithms based on our generalized directional trees. In the next chapter, we will apply our ideas and construct our mapping functions $F_o$ and $F_T$ to discrete optimization problems from the real-world.

### 2.5 Summary

This chapter is concerned with how to formally model and analyze the impact of knowledge on algorithm performance. By restricting the knowledge about a problem domain to a distribution of optimal solutions, we proposed a Directional Tree (DT) model which concurrently describes algorithm behavior and represents knowledge explicitly, and thus provides us with a tool to analyze the impact of knowledge on algorithm performance.
We analyzed the performance of an algorithm when that algorithm is used for solving any problems from problem space $P$. We formally proved that no one algorithm will perform better than another when the distribution is uniform. Because the Oracle Model for $P$ is uniform, this means there is no single ideal general purpose algorithm. We also discussed the performance of an algorithm when that algorithm is used to solve problems in different sub problem spaces, $SP$s, by giving specific examples. We can see that an algorithm's performance is different when solving problems for different $SP$s. Finally, we investigated the impact of knowledge on deterministic algorithms. We introduced a general formula to compute the expected length for deterministic algorithms. Additionally, we consider the situation when we do not have the full distribution of optimal solutions for a $SP$. In particular, the solution space has been classified into several disjoint nonempty clusters each of which has a bound on the distribution of optimal solutions. In this case, we found a general formula that identifies an upper bound on performance for a deterministic algorithm. Although DTs explicitly describe all the possible behaviors of an algorithm, we demonstrated that the model can provide closed-form solutions for behavior prediction.

In most cases, we may not be able to obtain knowledge about the problem ahead of time prior to the execution of any probabilistic or deterministic algorithm. Hence, we generalize our DT model to take into account acquisition of problem knowledge during execution. Our generalized directional tree (GDT) model provides a framework for developing online optimization algorithms that
dynamically updates its searching decisions in response to updating its oracle model.
Chapter 3

Structure-based Approach

The descriptions in Chapter 2 gives us some ideas on how knowledge about the distribution of optimal solution affects algorithm performance by using DT model. It also provide us a framework for developing online optimization algorithms by using GDT. In most cases, we may not be able to find such distribution information. However, we may have some knowledge about the structural characteristics of the problems, such as the correlations among the decision variables. According to those correlations, these decision variables can be grouped together into several groups. These groupings may capture the characteristic of solution space, and indicate a good direction for an algorithm to search for better solutions. Corresponding to GDT, each grouping may define a different OM. We can modify search algorithm according to that OM dynamically to better explore the space of solutions.

As we know, the structural characteristics of discrete optimization problems can drastically affect the solution landscape. The existing algorithms seldom or
only superficially considered the structural characteristics of these problems [52, 56, 55, 66, 46, 36, 38]. However these algorithms are sensitive to the solution landscape [76, 69, 33].

In this chapter, we propose a structure-based approach [74, 34] by considering the structural characteristics of the problems based on GDT. We regard the correlations among the decision variables as the structural characteristics in our approach. Usually, the decision variables can be grouped together into several groups according to their correlations. Thus, the basic idea of our approach is to find a way to group the decision variables of a problem based on structural characteristics. These groupings will define different OMs online. A search algorithm then updates its search direction according to OMs to search for better values for these decision variables and then combines them together to get a better solution for that problem.

In our approach, we consider Genetic Algorithm (GA) as our search algorithm, and use a Reinforcement Learning (RL) system to learn the structural characteristics of the problem, thereby grouping the decision variables of the problem into several groups based on the structural characteristics. We then use these structural characteristics to develop a structure-based Genetic Algorithm (GA) by introducing structural operations to work on these groupings and recombine them together to search for a better solution to the problem. During run time the system evaluates the performance of the GA and continues with reinforcement learning to further tune the search for a better grouping. We compared
our approach with standard GA by testing on two problems, belief revision over Bayesian Networks and the Tactical Fixed Interval Scheduling Problem. From experimental results we can see, our approach can find better solutions than standard GA. Before describing our approach in detail, we will first introduce some background knowledge on GA and RL.

3.1 Background: GA and RL

3.1.1 A Brief Introduction To GA

Genetic Algorithms (GAs) are search and optimization methods based upon adaptation principles found in nature. The basic idea of GAs is designed to simulate processes in a natural system necessary for evolution, specifically to follow the principle of survival of the fittest.

Genetic algorithms, evolution strategies, evolutionary programming, and genetic programming [1, 44, 45, 57, 18, 39, 25, 31] are usually grouped under the term evolutionary algorithms or evolutionary computation [61]. Evolutionary algorithms have been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, economics, chemical engineering, operations research, ecology, population genetics, studies of evolution and learning, and social systems.

A genetic algorithm is an iterative procedure that consists of a constant-size population of individuals. A population of individuals is maintained within the search space for a GA. The search space consists of all possible solutions to the
problem at hand. Each individual which is also called as chromosome is coded as a finite length vector of components (or variables) in terms of some alphabet, usually the binary alphabet \{0, 1\}. There are some other representations such as real-valued encodings, character-based encodings and tree representations. Each individual may form a possible solution for the given problem. A fitness score is assigned to each individual representing the abilities of an individual to compete. The concept of GA is based on the following foundations:

- Individuals in a population compete with each other for survival based on their fitness score.
- Those individuals which perform successfully in each competition will produce more offspring than those individuals that perform poorly.
- Genes from good individuals propagate throughout the population so that two good parents may sometimes produce offspring that are better than either parent.
- Thus each successive generation may become more fitting to their environment.

More specifically, GAs maintain a population of \( n \) chromosomes (solutions) with associated fitness values. Parents are selected to mate based on their fitness. Thus, the individuals with high fitness are given more opportunities to reproduce, so that offspring inherit characteristics from each parent. Since the size of the population is fixed, room must be made for the new arrivals as parents mate and produce offspring. So, the parents are replaced by the children, and thus create
a new generation. In this way, it is hoped that successive generations will have better solutions.

On average, the individuals in the new generations may contain more good genes than a typical chromosome in a previous generation. Each successive generation will contain more good partial solutions than previous generations. Eventually, once the population has converged and is not producing offspring noticeably different from those in previous generations, the algorithm itself is said to have converged to a set of solutions to the problem at hand.

The standard genetic algorithm proceeds as follows: After an initial population is randomly generated, the algorithm evolves through three operators:

- selection
- crossover
- mutation

The key idea of the selection operator is to give preference to better individuals, and allow them to pass on their genes to the next generation. The goodness of each individual depends on its fitness. Many selection procedures are currently in use, such as ranked-based fitness assignment, roulette wheel selection, stochastic universal sampling, local section, truncation selection, tournament selection and etc.[3, 67, 65, 2, 26]. One of the simplest is Holland’s original fitness-proportionate selection. In his method, individuals are selected with a probability proportional to their relative fitness. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the
population. Thus, high-fitness ("good") individuals has a better chance of "reproducing", while low-fitness ones are more likely to disappear. Selection alone cannot introduce any new individuals into the population. That means it cannot find new points in the search space.

The crossover operator is a prime distinguishing factor of GAs from other optimization techniques. Given two selected individuals (called parents), crossover is performed with some probability (the "crossover probability" or "crossover rate") by exchanging parts of their genes to form two new individuals, called offspring. The simplest way to do this is to exchange substrings using a randomly selected crossover point. The two new offspring are put into the next generation of the population. By recombining portions of good individuals, this process is likely to create even better individuals. Many crossover approaches are currently in use, such as single-point crossover, multi-point crossover, uniform crossover, and shuffle crossover, etc. [5, 62, 64, 6].

The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by randomly flipping bits with some (small) probability.

Using selection alone will tend to fill the population with copies of the best individual from the population. Using selection and crossover operators will tend to cause the algorithms to converge on a good but sub-optimal solution. Using mutation alone induces a random walk through the search space. Mutation
Fig. 3.1.1: Pseudo-code of the standard genetic algorithm

and selection (without crossover) create a parallel, noise-tolerant, hill-climbing algorithms[45, 25, 39].

Genetic algorithms are stochastic iterative processes that are not guaranteed to converge; the termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level. Figure 3.1.1 presents the standard genetic algorithm in pseudo-code format.

Let us consider the following simple example, due to [45, 61], demonstrating how the genetic algorithm works. The population consists of 4 individuals. Each of them is binary-encoded strings of length 8. The fitness value is the number of ones in the bit string. Assume the crossover rate $P_{\sigma_{\text{cess}}} = 0.7$, and mutation rate $P_{\text{mutation}} = 0.0001$. Table 3.1.1 shows a possible initial (randomly generated) population.

If we choose to use fitness-proportionate selection, we need to choose 4 individuals with probabilities proportional to their relative fitness values. The four selected individuals are divided into two pairs. Suppose that the two parent pairs
are B,D and B,C (note that A did not get selected because the procedure is probabilistic). Once a pair of parents is selected, crossover is performed between them with probability $P_{\text{cross}}$, resulting in two offspring. If no crossover is performed (with probability $1 - P_{\text{cross}}$), then the offspring are exact copies of each parent. In this example, for parent B and D, assume that crossover takes place at the first bit position. Then after exchanging corresponding parts, we get offspring $E = 10110100$ and $F = 01101110$. For parent B and C, assume that no crossover is performed, then we get exact copies of B and C. Next, each offspring is subject to mutation with probability $P_{\text{mutation}}$ per bit. Assume that offspring E is mutated at the sixth position to form $E' = 10110000$, offspring B is mutated at the first bit position to form $B' = 01101110$, and offspring F and C are not mutated at all. Table 3.1.2 shows the next generation population, created by the above operators of selection, crossover, and mutation.

<table>
<thead>
<tr>
<th>Label</th>
<th>Chromosome</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>E'</td>
<td>10110000</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>01101110</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>00100000</td>
<td>1</td>
</tr>
<tr>
<td>B'</td>
<td>01101110</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3.1.2: The next generation population
Note that in the new population, although the best individual with fitness 6 has been lost, the average fitness has increased. Iterating this procedure, the genetic algorithm will eventually find a perfect string, i.e., with maximal fitness value of 8.

Genetic algorithms dynamically balance exploration with exploitation through selection, crossover and mutation. A high probability of crossover and mutation increases exploration. Different selection methods have different degrees of exploitation. The balance of exploration and exploitation needs to be considered carefully for two reasons:

1. If too much exploration is conducted in unpromising regions, the speed and efficiency of a GA will suffer.

2. If too much exploitation is conducted in unpromising areas, GA may lose critical information on correct solutions and converge on a local optimum. It is important to choose appropriate crossover and mutation rates in order to obtain good performance for genetic algorithms.

The fact that GAs work on a population of individuals, rather than a single individual, is an essential part of the algorithm. The population size determines how many initial candidate solutions are available for GAs and affects performance and efficiency. A small population size may provide insufficient sample size. A larger population size provides more candidate solutions; however, it requires more evaluations per generation and results in slower convergence.
In GAs, many researchers have been working on ways to improve GAs performance. To name a few, Fogel and Ghozeil [19] use fitness distribution to improve GAs; Clearwater and Hogg [9] use the parameters which characterizing the structure of problem as heuristics to improve the GA; Pelikan, Goldberg and Cantu-Paz [49] use an estimation of the joint distribution of promising solutions to improve GA. Many researchers have modified their implementation of GAs either by using non-stand chromosome representations and/or by designing problem-specific genetic operators to accommodate the problem to be solved[44]. Our structure-based approach provide a framework to improve GAs by incorporating characteristic knowledge of problems.

3.1.2 A Brief Introduction to RL

Reinforcement Learning [35, 63] is a learning technique which has attracted many researchers in the machine learning and artificial intelligence communities. Reinforcement Learning is learning what to do, how to map situations to actions so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must find which actions yield the most reward by trying them. Actions may affect not only the immediate reward, but also the next situation and, through that, all subsequent rewards. These two characteristics, trial and error search and delayed reward, are the two most important distinguishing features of reinforcement learning.
Reinforcement learning is different from supervised learning. The most important difference is that in supervised learning, there is presentation of input/output pairs. But in reinforcement learning such pairs do not exist. Instead, after choosing an action the agent is told the immediate reward and the subsequent state, but is not told which action would have been in its best long-term interests. Another difference from supervised learning is that the evaluation of the system is often concurrent with learning.

An intuitive way to understand the relation between the learner and its environment is with the following example dialogue [35].

**Environment:** You are in state 65. You have 4 possible actions.
**learner:** I will take action 2.
**Environment:** You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.
**learner:** I will take action 1.
**Environment:** You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.
**learner:** I will take action 2.
**environment:** You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions.

\[ \vdots \]

The learner’s job is to find a policy \( \pi \), mapping states to actions, that maximizes some long-run measure of reinforcement.

Usually, there are three models that have been considered in Reinforcement Learning area.
1. The finite-horizon model: in this model the agent should optimize its expected reward for the next \( h \) steps at a given moment:

\[
E(\sum_{t=0}^{h} r_t)
\]  

(12)

where \( r_t \) represent the scalar reward received \( t \) steps into the future. This model can be used in two ways. In the first way, the agent will have a non-stationary policy. On its first step, agent will take a \( h \)-step optimal action. This is defined to be the best action available given that it has \( h \) steps remaining in which to act and gain reinforcement. On the next step it will take a \((h-1)\)-step optimal action, and so on, until it finally takes a 1-step optimal action and terminates. In the second way, the agent always takes the \( h \)-step optimal action and acts according to the same policy. The value of \( h \) limits how far ahead it looks in choosing its actions. The finite-horizon model is not always appropriate. In many cases we may not be able to know the precise length of the agent’s life in advance.

2. The infinite-horizon discounted model: in this model the long-run reward of the agent is taken into account, but rewards that are received in the future are geometrically discounted according to discount factor \( \gamma \), where \( 0 \leq \gamma < 1 \):

\[
E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right)
\]  

(13)
$\gamma$ can be interpreted in several ways. It can be viewed as an interest rate, a probability of living another step, or as a mathematical trick to bound the infinite sum.

3. the average-reward model: in this model, the agent is supposed to take actions that optimize its long-run average reward:

$$\lim_{h \to \infty} E\left(\frac{1}{h} \sum_{t=0}^{h} r_t\right)$$  \hspace{1cm} (14)

Such a policy is referred to as a gain optimal policy. One problem with this model is that there is no way to distinguish between two policies if one of which gains a large amount of reward in the initial phases and the other of which does not. Reward gained on any initial phases of the agent’s life is overshadowed by the long-run average performance. It is possible to overcome this problem by generalizing this model to take into account both the long run average and the amount of initial reward than can be gained. To do so, bias optimal model, a policy is preferred if it maximizes the long-run average and ties are broken by the initial extra reward.

Now we will introduce techniques for determining the optimal policy given a correct model. Here, we only consider finding optimal policies for the infinite-horizon discounted model, but most of these algorithms have analogs for the finite-horizon and average-case models as well. The optimal value of state is defined as the expected infinite discounted sum of reward that the agent will gain if it starts in that state and executes the optimal policy. Let $\pi$ be a complete
decision policy, it is written:

$$V^*(s) = \max_{\pi} E \left( \sum_{t=0}^{\infty} \gamma^t r_t \right)$$

This optimal value function is unique and can be defined as the solution to the simultaneous equations:

$$V^*(s) = \max_a (R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s')) , \forall s \in S$$

where $e$ of a state $s$ is the expected instantaneous reward plus the expected discounted value of the next state, using the best available action. Given the optimal value function, we can specify the optimal policy as:

$$\pi^*(s) = \text{argmax}_a (R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V^*(s'))$$

One way to find an optimal policy is to find the optimal value function. It can be determined by a simple iterative algorithm called value iteration that can be shown to converge to the correct $V^*$ values. The algorithm is as follows:

1. initialize $V(s)$ randomly

2. repeat the following until policy good enough

   (a) for all $s \in S$ do the following:

   i. for all $a \in A$:

   $$Q(s, a) := R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V(s')$$

   ii. $V(s) := \max_a Q(s, a)$
It is not obvious when to stop the value iteration algorithm. One important result is to use a function of the Bellman residual of the current value function to bound the performance of the current greedy policy\cite{71}. It says that if the maximum difference between two successive value functions is less than $\epsilon$, then the value of the greedy policy, differs from the value function of the optimal policy by no more than $\frac{2\epsilon\gamma}{(1-\gamma)}$ at any state. This provides an effective stopping criterion for the algorithm. Puterman \cite{51} discusses another stopping criterion, based on the span semi-norm. This method may result in earlier termination. Another important result is that the greedy policy is guaranteed to be optimal in some finite number of steps even though the value function may not have converged\cite{4}. In practice, the greedy policy is often optimal long before the value function has converged.

Another way to find an optimal policy is to manipulate the policy directly. It operates as follows:

1. choose a policy $\pi'$ arbitrarily
2. repeat the following until $\pi = \pi'$
   (a) $\pi := \pi'$
   (b) compute the value function of policy $\pi$: solve the linear equations  
   \[
   V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi(s')
   \]
   (c) improve the policy at each state:
   \[
   \pi'(s) = \arg\max_a (R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_\pi(s'))
   \]
The value function of a policy is the expected infinite discounted reward gained by executing that policy at each state. It can be specified by solving a set of linear equations. Once we know the value of each state under the current policy, we check whether the value could be improved by changing the first action taken. If it can, whenever it is in that situation we change the policy to take the new action. This step is guaranteed to strictly improve the performance of the policy. When there is no improvements anymore, then the policy is guaranteed to be optimal. Since there are at most \(|A|^k\) distinct policies, and the sequence of policies improves at each step, this algorithm terminates in at most an exponential number of iterations [51].

3.2 Design of Structured-based Approach

When we use GA, we find that it is very sensitive to the structural characteristic of the problem [74, 34]. If we can design a new GA to consider the structural knowledge, we expect that this new GA may be less sensitive to the structural characteristic and can get better solutions. Sometimes, such structural knowledge may not be readily or immediately available. But, we may obtain such knowledge through online learning techniques such as RL. Based on above consideration, we propose a structure-based approach. Our structure-based approach is an online algorithm. Recall that in Section 2.4 we describe the construction of the online algorithm proceeds based on GDT as follows: At step \(i\), given \(\tau_{i-1}\) and \(o_{i-1}\),
1. Evaluate/execute $\tau_{i-1}$. If optimal solution found, stop. Otherwise, identify the selected continuation bag-sequence node $bs$.

2. Construct new $o_i = F_o(o_{i-1}, \tau_{i-1}, bs)$.

3. Construct new $\tau_i = F_\tau(o_i, \tau_{i-1}, bs)$.

4. Repeat loop.

$\phi_0$ is the initial OM and $\tau_0$ is constructed from $o_0$. This results in a sequence of partial GDTs $\{\tau_0, \tau_1, \ldots, \tau_k\}$ and OMs $\{o_0, o_1, \ldots, o_k\}$.

In our approach, we develop a new GA by using structure operations which will be introduced in Section 3.2.1. In this thesis, we consider the structural knowledge as the correlations among the decision variables. We use RL to identify these correlations, thereby grouping the decision variables of the problem into several groups. Intuitively, above construction gives us basic idea on building the framework of our approach. More specifically, each grouping has an evaluation function related to fitness, thus may defining an OM. Hence, at step $i$, $\tau_{i-1}$ describes our approach’s search procedure and $o_{i-1}$ reflects grouping information obtained so far. Each bag in $bs$ indicates populations evolved by. The next grouping is updated online by RL considering the current parameters, and thus construct new $o_i$ and $\tau_i$.

The framework of our structure-based approach is as follows: At step $i$, given $P_{i-1}$ and $G_{i-1}$,

1. Evaluate $P_{i-1}$. If termination condition satisfied, stop. Otherwise, update the grouping evaluation function $E_{G_{i-1}}$. 
2. Construct new $P_i = F_{GA}(G_{i-1}, P_{i-1})$

3. Construct new $G_i = F_{RL}(E_{G_{i-1}}, P_{i-1})$.

4. Repeat loop.

$P_0$ is the initial population of GA and $G_0$ is initial grouping. $F_{GA}(., .)$ is the GA to generate a new population based on current population and grouping. $F_{RL}(., .)$ is the RL to generate a new grouping.

### 3.2.1 Genetic algorithm with structure operations

As we described before, to apply GAs, one generally expresses the problem in such a way that potential solutions can be coded in a gene-like bit sequence and a population of those sequences is prepared. An optimal solution is searched for by evolutionary operations (selection, crossover and mutation). Most researchers modify their implementation of GA either by using non-standard chromosome representations or by designing problem specific genetic operations [44, 49, 41, 68] to accommodate the problem to be solved in order to build efficient evolution programs.

Unfortunately, we can not determine if the optimal solution has been found if we only use GA. However, even the identification of a near optimal solution can be used in cooperation with other algorithms to efficiently find the optimal solution [69]. In the following, we will describe the representation and operations employed in our GA.
3.2.1.1 Representation

When mapping a problem domain into a problem that is solvable by GA, we need to consider how to represent a solution to the problem as a chromosome containing a set of genes that can be genetically manipulated.

For the discrete problem (1), we can represent a complete assignment to all decision variables as a chromosome. Each gene represents the possible value of single decision variable. The genetic operations manipulate each chromosome/individual by changing the value of each gene in the chromosome. The fitness evaluation is simply our solution quality calculation. In this case, it is the value $f(x)$.

3.2.1.2 Selection

The selection operation is the standard fitness-proportionate selection approach with keeping the best individual in the population.

3.2.1.3 Crossover

The crossover operation performs a “group” crossover. We partition the decision variables into several groups based on correlations of decision variables. Each group is a subset of decision variables that forms a sub-solution. In our approach, a RL system is used to identify grouping (described later).
The mechanism of the “group” crossover is explained as follows: assume the problem (1) has $N$ decision variables. The $N$ decision variables have been appropriately divided into $m$ groups represented as $G_i$ ($i = 1, \ldots, m$). Assume, there are two selected parents $p_1$ and $p_2$. Each parent is a chromosome and represents a feasible solution. For group $i$, we can use the value of sub-solution $E_i$ as evaluation for group $i$. Of course it is possible to use other evaluations.

For $p_1$, we get $E_{i1}$ ($i = 1, \ldots, m$). For $p_2$, we get $E_{i2}$ ($i = 1, \ldots, m$). For all groups, we then compare $E_{i1}$ with the corresponding $E_{i2}$. We get one child $c_1$ by combining the groups with best values of $E_{ik}$ together. That means, for group $i$, if $E_{i1} \geq E_{i2}$, $k = 1$ and the corresponding alleles in $G_{i1}$ are added to the chromosome $c_1$, otherwise $k = 2$ and the corresponding alleles in $G_{i2}$ are added to $c_1$. We get the other child by combining the groups with next best values of $E_{ik}$ together.

### 3.2.1.4 Mutation

There are two kinds of mutation operations employed in our method. One is standard mutation and the other is called “group” mutation. Standard mutation randomly selects an allele of a chromosome to modify and then randomly chooses a new value for that allele. “Group” mutation randomly selects an allele to change the group it belongs to. These mutation operations help the GA to maintain diversity in the population to avoid premature convergence.
3.2.2 Reinforcement Learning systems for Grouping Identification

In our approach, we use a RL system to do group identification. Group identification is to identify which decision variables should be in the same group based on a policy. A policy determines which action should be performed in each state; a policy is a mapping from states to actions. The value of a state is defined as the expected total discounted payoff over the infinite horizon when starting in that state and following an optimal policy thereafter. The optimal policy would be the mapping from states to actions that maximizes the expected total discounted payoff over the infinite horizon.

Assume that $G = 1, 2, \ldots, m$ is the finite group set. At instant $n$, decision variable $x_k$ is assigned to group $i \in G$ with the probability $p_{ki}(n)$. For the above assignment a payoff $r(n)$ is obtained. We use the infinite-horizon discounted model to compute the expected total payoff for for decision variable $x_k$ in group $i \in G$. Equation (13) describes the infinite-horizon discounted model. So under a policy $\sigma$, the expected total payoff for for decision variable $x_k$ in group $i \in G$ is defined as:

$$V_{x_k}^\sigma(i) = E_\sigma[\sum_{n=0}^{\infty} \gamma_n r(n)|S_{x_k}(n) = i]$$

(16)

where $\gamma_n \in (0, 1)$ is the discount factor. $E_\sigma$ denotes expectation with respect to the policy $\sigma$; $S_{x_k}(n)$ is the state of decision variable $x_k$ and $S_{x_k}(n) = i$ means that decision variable $x_k$ belongs to group $i$. Equation 15 provides us with the way to define the optimal payoff value. So the optimal payoff value in any state
for decision variable \( x_k \) is defined as:

\[
V_{x_k}(i) = \max_{\sigma} V_{x_k}^\sigma(i)
\]  

(17)

where \( i \in G \). For decision variable \( x_k \), any policy \( \sigma \) such that \( V_{x_k}^\sigma = V_{x_k}(i) \), \((\forall i \in G)\) is an optimal policy.

For decision variable \( x_k \), we defined a group value \( q_{ki} \) as the expected total discounted payoff over the infinite horizon, if \( x_k \) is assigned to group \( i \in G \) and an optimal policy is followed thereafter. So we have,

\[
V_{x_k}(i) = \max_{i \in G} q_{ki}
\]  

(18)

In the following, we will introduce how to use the reinforcement learning to implement the group identification. The reinforcement learning algorithm employed in our system is similar to the method proposed by G. Santharam [53].

### 3.2.2.1 Architecture

The system has two major parts: the controller network and the adaptive critic network. For the controller network, it has one unit corresponding to each decision variable. The \( k \)th unit of the controller has a weight vector \( w_k = (w_{k1}, \ldots, w_{km}) \) and \( g_{ki} \in G = \{1, 2, \ldots, m\} \) indicating the corresponding group index. The unit generates a group output \( y_k \in G \) stochastically according to the law: \( y_k = i \) with probability \( p_{ki} \) and

\[
p_{ki} = \frac{f(w_{ki}g_{ki})}{\sum_{j=1}^{m} f(w_{kj}g_{kj})}
\]  

(19)
where \( i \in G, f : R \rightarrow R \) is the activation function given by \( f(x) = \frac{1}{1+e^{-x}} \). From the above formula it follows that each value of parameter \( w_k \) determines a specific stationary random grouping. The weight vector is updated at each instant using a scalar reinforcement signal supplied by the adaptive critic network.

The adaptive critic network has one unit corresponding to each decision variable. The function of the adaptive critic network is to generate a reinforcement and send this reinforcement to the corresponding controller unit, and update the estimates of group values after observing the state and payoff.

### 3.2.2.2 Learning Algorithm

At instance \( n \), for any decision variable \( x_k \) the control network uses equation (19) to compute \( p_{ki} \) \((i = 1, \ldots, m)\) which is the probability of decision variable \( x_k \) being in group \( i \). These probabilities determine a specific stationary random grouping.

The system gets a payoff for the grouping. The adaptive critic network generates a reinforcement based on group values and updates the group values according to the payoff. Then the parameters of the controller network, controller weights which are used to compute \( p_{ki} \), are updated using the reinforcement. At instance \( n = 0 \), controller weights and group values can be arbitrary. The modified learning algorithm is as follows:

1. For \( k = 1 \) to \( n \) (\( n \) is the number of decision variables) do

   (a) for each \( i \in G \) the \( k \)th controller unit computes \( p_{ki} \) based on weight vector.
(b) for i=1 to m (m is the number of groups), sort the $p_{ki}$ in ascending order.

(c) randomly generate a value $u \in [0, 1]$. For $i = 1$ to $m$

i. if $p_{ki} < u$, $i +=$ and continue.

ii. else the group output $y_k = i$ and jump to step 2.

2. For k=1 to n

(a) the reinforcement $r'$ to the $k$th controller unit from the $k$th adaptive critic will be:

$$r' = \begin{cases} 
1 & q_{ki} = \max_{m \in G} q_{km} \\
0 & \text{otherwise}
\end{cases}$$

(b) update the $k$th controller weights:

$$\Delta w_{ki} = \beta (r' - f(w_{ki})) - \delta w_{ki}$$  \hfill (20)

where $\beta$ is the learning rate, $\delta$ is the weight decay rate. The weight decay is to prevent converging prematurely.

(c) update the group value:

$$\Delta q_{ki} = \beta (r_i(n) - q_{ki} + \alpha \max_{m \in G} q_{km})$$  \hfill (21)

where $\beta$ is the learning rate, $\gamma$ is the discount factor, and $r_i(n)$ is the payoff at instant $n$.

The above steps are repeated at each time instant.
3.2.3 Combining GA and RL

As stated above, the GA is used to search solutions, and the RL is used to identify groupings for the GA. During run-time, the GA interacts with the RL in order to tune the RL to find a better grouping and that results in improved GA performance. The RL system sends its grouping results to the GA. The GA implements its operations (selection, crossover and mutation) to find a solution. A payoff is produced to tune the RL system. In Section 3.2.1, for any group i we use a variable $E_i$ to evaluate the group i. Here these variables are used to determine the performance of GA. Based on the grouping, for group i we compute the corresponding evaluation $E_i (i = 1, \ldots, g )$ of the best individual of the population. At instant n, we use the following to compute the payoff for group i:

$$r_i(n) = \frac{E_i(n - 1)}{E_i(n)}$$ \hspace{1cm} (22)

We give an example to describe how the RL and the GA interacts with the each other to produce the solution. Assume there is a discrete optimization problem with 12 decision variables, $X_1, X_2, \ldots, X_{12}$. Each decision variable $X_k (k = 1, \ldots, 12)$ has two states (T and F). After initial grouping selection, the 12 decision variables have been appropriately divided into 3 groups: $G_1 = \{X_1, X_2, X_5, X_8, X_{12}\}$, $G_2 = \{X_3, X_6, X_7, X_9\}$, and $G_3 = \{X_4, X_{10}, X_{11}\}$. The RL controller sends these groups to the GA. Then the GA implements its operations: selection, mutation and crossover. As stated before, there are
two kinds of mutation operations employed in our GA: standard mutation and "group" mutation. Both mutation rates are very small. To simplify the example, we assume that "group" mutation operation does not change the above groupings. After finishing mutation, the GA implements "group" crossover. Assume that there are two selected parents \( p_1 = \{TTFTFTFFFTFF\} \) and \( p_2 = \{FTTFGTFFFTFF\} \). For \( p_1 \), we get \( E_{11}' \) for \( G_1 \), \( E_{21}' \) for \( G_2 \) and \( E_{31}' \) for \( G_3 \). For \( p_2 \), we get \( E_{12}' \) for \( G_1 \), \( E_{22}' \) for \( G_2 \) and \( E_{32}' \) for \( G_3 \). We then compare \( E_{11}' \) with \( E_{21}' \), \( E_{21}' \) with \( E_{22}' \), and \( E_{31}' \) with \( E_{32}' \). Let’s assume \( E_{11}' > E_{12}', E_{21}' > E_{22}' \) and \( E_{31}' < E_{32}' \). We get one child \( c_1 \) by combining the groups with best values of \( E_{ik}' \) (\( i = 1, 2, 3 \) and \( k = 1, 2 \)). So \( G_1 \) of \( p_1 \), \( G_2 \) of \( p_1 \), and \( G_3 \) of \( p_2 \) forms \( c_1 \). And \( c_1 = \{TTFTFTFFFTFF\} \). We get the other child \( c_2 \) by combining the groups with next best values of \( E_{ik}' \). Assume \( E_{12}', E_{21}' \) and \( E_{32}' \) are next best values, we get \( c_2 \) by combining \( G_1 \) of \( p_2 \), \( G_2 \) of \( p_1 \) and \( G_3 \) of \( p_2 \) together, and \( c_2 = \{FTTFGTFFFTFF\} \). Crossover continues until a new population is formed. Then the system compute the payoff \( r_i \) (\( i = 1, 2, 3 \)) by using equation (22). The payoff is sent back to the RL. Then the RL will do the following:

For \( k = 1 \) to 12

1. the \( k \)th unit of critic network will do the following:

   (a) check which group decision variable \( X_k \) belongs to. (For example, from above we know \( X_1 \in G_1 \)). Assume \( X_k \in G_i \). Find the maximum group value \( q_{k\text{max}} \) for \( X_k \). \( q_{k\text{max}} = \max_j q_{kj}, j = 1, 2, 3 \); Compare \( q_{ki} \) with
\text{if } q_{ki} = q_{kmax}, \text{ send a reinforcement } r_k' = 1 \text{ to the } k \text{th controller unit. Otherwise send } r_k' = 0 \text{ to the } k \text{th controller unit.}

(b) update its group value \( q_{ki} \) by using (21) and

\[ q_{ki} = q_{ki} + \Delta q_{ki} \]

2. the \( k \)th unit of controller network will do the following:

(a) update weight \( w_{ki} \) by using (20).

(b) follow the step 1 in learning algorithm to decide which group decision variable \( X_k \) will belong to.

After implementing the above steps, the RL gets a new grouping result and sends it to the GA again.

3.3 Problem I: Bayesian Networks and Belief Revision

Bayesian Networks [48] are one of the most popular models for probabilistic reasoning. They have been applied to various domains such as story comprehension [8, 7], circuit fault detection [14, 48], medical diagnosis [60] and planning systems [37]. However, computing with these networks have been proven to be NP-hard [59, 10]. One type of computation is called belief revision which concerns finding the maximally probable global assignment. There are different methods exist for belief revision. Pearl [48] proposed an exact algorithm which can find the two best assignments for singly connected networks, but its growth is exponential for multiply connected networks. Shimony and Charniak [58] obtain
the maximum aposterior assignment by using a best first search on a modified belief network. The algorithm naturally extends to find next-best assignment, is linear in the size of polytree but exponential in the general case. Peng and Reggia[50] formalize causal and probabilistic associative knowledge in a two level network which associates disorders and manifestations. The structure is a special case of a belief network and calculations are computationally complex if multiple simultaneous disorders may occur. In this section, we use our structure-based approach to solve this problem.

3.3.1 Problem description

The structure of the Bayesian Networks contains representations of both the conditional dependencies and independences between elements of the problem domain. The knowledge is represented by Random Variables (RVs) and the relationships are described using conditional probability tables; this knowledge representation is based on probability theory. An example of a small Bayesian network is shown in Figure 3.3.1. This network contains five RVs, each of which has two states (T and F). The conditional probability tables for the five RVs are shown to the right of the graph.

To calculate the joint probabilities we use the chain rule as follows:

\[
P(A, B, C, D, E) = P(D, E, C, B, A) =
\]

\[
P(D|E, C, B, A)P(E|C, B, A)P(C|A, B)P(B|A)P(A)
\]

(23)
Fig. 3.3.1: A small example of Bayesian network

Bayesian networks take this process further by making the important observation that certain RV(s) pairs may become uncorrelated once information concerning some other RV(s) is known. More precisely, considering the Bayesian Network in Figure 3.3.1, we may have the following conditional independence:

\[ P(D|E, C, B, A) = P(D|C, A), \quad P(E|C, B, A) = P(E|C), \quad P(C|A, B) = P(C|B) \]

and \[ P(B|A) = P(B). \]

Combined with the chain rule, these conditional independences allow us to replace the terms in (23) with smaller conditions. We get:

\[
P(A, B, C, D, E) = P(D, E, C, B, A) = P(D|C, A)P(E|C)P(C|B)P(B)P(A)
\]

Thus, instead of explicitly keeping the joint probabilities, all we need are smaller conditional probability tables which can then be used to compute the joint probabilities. What we have is a directed acyclic graph of RV relationships. Directed arcs between RVs represent conditional dependencies. When all the parents of
a given RV are instantiated, that RV is said to be conditionally independent of any ancestors given its parents.

Belief revision is best used for modeling explanatory/diagnostic tasks. Basically, some evidence or observation is given to us, and our task is to come up with a set of hypothesis that together constitute the most satisfactory explanation/interpretation of the evidence at hand. More formally, if \( W \) is the set of all RVs in the given Bayesian network and \( e \) is the evidence (that is, \( e \) represents a set of instantiations made on subset of \( W \)), any complete instantiations to all the RVs in \( W \) that is consistent with \( e \) is called an explanation on interpretation of \( e \). The problem is to find an explanation \( w^* \) such that:

\[
p(w^*|e) = \max_{w \in W} p(w|e) \tag{25}
\]

Intuitively, we can think of the non-evidence RVs in \( W \) as a possible hypothesis for \( e \).

As an example, reconsider the small Bayesian network in Figure 3.3.1. If we are given the observation that node D is true, it is our task under belief revision to determine the most probable complete instantiation over all the nodes in the network which is consistent with D being true and maximizes (25). This results in the solution \( \{A = F, B = T, C = F, D = T, E = F\} \) which has the maximal joint probability 0.21168.

With a small network, a valid method is to simply tabulate all the possible values of the RVs and then calculate the joint probabilities. Once the network
gets larger and more complex, this method is obviously unacceptable. Researchers have developed many methods to solve this problem [15, 28, 54, 27, 13, 20].

3.3.2 Experiments

This section describes experimental results evaluating the performance of our approach on a variety of Bayesian networks, including real Bayesian networks and randomly generated Bayesian networks. We ran both the standard GA and our structure-based GA. All the programs written in C++ ran on a Sun Ultra5 270MHz. The GA used in our method is based on the GENESIS[29] framework. Each population has 100 individuals. The crossover rate is 0.55 and the mutation rate is 0.001 in our experiments. These values were selected from pre-trial runs conducted to determine good parameter settings for the standard GA.

3.3.2.1 Results on solution comparision

For small networks which contain no more than 200 random variables, we ran both standard GA and our structure-based GA through 500 iterations. From our experiments, we observed that after 500 iterations, both GAs would converge to local optimal solutions. Running through 500 iterations for these small networks is enough to capture the performance of both GAs. For large networks containing 500,1500 and 2000 random variables, running through 1000 iterations is sufficient to capture the performance of both GAs.
Table 3.3.1 shows performance comparison on real-world networks. In this table, “SGA” represents for standard GA, and “SB-GA” denotes our structure-based GA. The numbers listed in this table are maximal joint probabilities for corresponding networks. The first four trace networks were generated by the Andes [21] system. Among these trace networks, trace2 and trace4 both have zero problems (which means there are many entries in the RV conditional tables that have probability value 0). The other two GENIE networks were provided by Marek Druzdzel of the GENIE group at the University of Pittsburgh. Figure 3.3.2 shows the performance profile of our GA and standard GA for trace files. In this figure, we see that our GA achieves a better solution in fewer generations than the standard GA. Especially for trace2 and trace4, our GA performance is better than standard GA for these two networks. The distribution of the conditional probabilities for the random variables in these two networks is extreme with some conditional probabilities that are extremely high and the remaining conditional probabilities very low. In this case the solution landscape is characterized by a very low plateau and high isolated peaks. The standard GA is sensitive to the probability distribution and does not work well for the extreme networks. However, our GA learns this phenomenon through its reinforcement learning and actually “restructures” the landscape through its groupings. Thus, this results in better solutions for the extreme networks. We get similar results for the other two networks.
<table>
<thead>
<tr>
<th>Name of BN</th>
<th>Number of RVs</th>
<th>Optimal Solution</th>
<th>SGA</th>
<th>SB-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>103</td>
<td>4.172e-11</td>
<td>2.313e-15</td>
<td>6.914e-13</td>
</tr>
<tr>
<td>trace2</td>
<td>158</td>
<td>9.301e-12</td>
<td>5.794e-22</td>
<td>4.635e-13</td>
</tr>
<tr>
<td>trace3</td>
<td>186</td>
<td>7.004e-18</td>
<td>3.213e-26</td>
<td>1.749e-23</td>
</tr>
<tr>
<td>trace4</td>
<td>196</td>
<td>N/A</td>
<td>4.868e-32</td>
<td>1.143e-13</td>
</tr>
<tr>
<td>genet1</td>
<td>179</td>
<td>6.754e-03</td>
<td>3.635e-35</td>
<td>4.151e-09</td>
</tr>
<tr>
<td>genet2</td>
<td>104</td>
<td>5.794e-22</td>
<td>3.204e-25</td>
<td>6.420e-23</td>
</tr>
</tbody>
</table>

Table 3.3.1: Performance Comparison on real Bayesian Networks.

![Graph](image)

Fig. 3.3.2: Performance on trace network

We also tested our method on 60 randomly generated Bayesian networks with different sizes, probability distributions and connective densities. Connective density is computed by

\[
Conn.\text{Density} = \frac{\text{number of total edges}}{\text{max possible edges}}
\]  

(26)

\[
\text{max possible edges} = \frac{(\text{number of nodes}) \times (\text{number of nodes} - 1)}{2}
\]  

(27)

Tables 3.3.2 through 3.3.5 show the results of testing on these randomly generated Bayesian networks. The networks in Table 3.3.2 and Table 3.3.4 were
generated with exponential probability distribution (some conditional probabilities are extremely high and some very low. The networks in Table 3.3.3 and Table 3.3.5 were generated with flat probability distribution (conditional probabilities are uniform).

<table>
<thead>
<tr>
<th>Number of RVs</th>
<th>Conn. Desity</th>
<th>Opt.Solution</th>
<th>SGA</th>
<th>SB-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.006337</td>
<td>7.372e-03</td>
<td>1.378e-08</td>
<td>7.372e-03</td>
</tr>
<tr>
<td>100</td>
<td>0.017030</td>
<td>3.404e-02</td>
<td>1.731e-19</td>
<td>6.473e-03</td>
</tr>
<tr>
<td>100</td>
<td>0.025743</td>
<td>2.547e-02</td>
<td>5.034e-45</td>
<td>2.547e-02</td>
</tr>
<tr>
<td>100</td>
<td>0.032079</td>
<td>8.291e-02</td>
<td>1.198e-70</td>
<td>8.291e-02</td>
</tr>
<tr>
<td>100</td>
<td>0.041386</td>
<td>1.720e-01</td>
<td>2.054e-85</td>
<td>6.491e-04</td>
</tr>
<tr>
<td>200</td>
<td>0.002836</td>
<td>2.843e-02</td>
<td>1.042e-248</td>
<td>2.843e-02</td>
</tr>
<tr>
<td>200</td>
<td>0.005771</td>
<td>1.015e-02</td>
<td>4.199e-322</td>
<td>1.949e-03</td>
</tr>
<tr>
<td>200</td>
<td>0.007910</td>
<td>5.917e-03</td>
<td>8.198e-311</td>
<td>5.917e-03</td>
</tr>
<tr>
<td>200</td>
<td>0.010100</td>
<td>3.543e-02</td>
<td>3.233e-298</td>
<td>4.575e-08</td>
</tr>
<tr>
<td>200</td>
<td>0.013632</td>
<td>N/A</td>
<td>2.750e-314</td>
<td>2.826e-15</td>
</tr>
</tbody>
</table>

Table 3.3.2: Performance Comparison on small Bayesian Networks with exponential distribution

<table>
<thead>
<tr>
<th>Number of RVs</th>
<th>Conn. Desity</th>
<th>Opt. Solution</th>
<th>SGA</th>
<th>SB-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.004950</td>
<td>1.306e-26</td>
<td>2.159e-27</td>
<td>8.757e-27</td>
</tr>
<tr>
<td>100</td>
<td>0.010891</td>
<td>1.949e-26</td>
<td>5.869e-27</td>
<td>1.596e-26</td>
</tr>
<tr>
<td>100</td>
<td>0.020594</td>
<td>6.470e-26</td>
<td>1.768e-27</td>
<td>1.182e-26</td>
</tr>
<tr>
<td>100</td>
<td>0.026139</td>
<td>2.380e-26</td>
<td>5.900e-29</td>
<td>7.923e-27</td>
</tr>
<tr>
<td>100</td>
<td>0.035842</td>
<td>1.182e-26</td>
<td>1.782e-30</td>
<td>1.447e-27</td>
</tr>
<tr>
<td>200</td>
<td>0.003333</td>
<td>N/A</td>
<td>3.481e-57</td>
<td>1.540e-51</td>
</tr>
<tr>
<td>200</td>
<td>0.005721</td>
<td>N/A</td>
<td>1.168e-60</td>
<td>5.166e-55</td>
</tr>
<tr>
<td>200</td>
<td>0.009751</td>
<td>N/A</td>
<td>1.441e-64</td>
<td>2.572e-56</td>
</tr>
<tr>
<td>200</td>
<td>0.012090</td>
<td>N/A</td>
<td>5.301e-65</td>
<td>9.463e-57</td>
</tr>
<tr>
<td>200</td>
<td>0.016716</td>
<td>N/A</td>
<td>1.778e-68</td>
<td>1.733e-58</td>
</tr>
</tbody>
</table>

Table 3.3.3: Performance Comparison on small Bayesian Networks with flat distribution
<table>
<thead>
<tr>
<th>Number of RVs</th>
<th>Conn. Density</th>
<th>SGA (log)</th>
<th>SB-GA (log)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000878</td>
<td>-2.78e+03</td>
<td>-8.03e+01</td>
</tr>
<tr>
<td>500</td>
<td>0.001357</td>
<td>-3.09e+03</td>
<td>-1.62e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.001677</td>
<td>-3.48e+03</td>
<td>-3.24e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.002387</td>
<td>-3.29e+03</td>
<td>-5.68e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.004950</td>
<td>-4.22e+03</td>
<td>-8.13e+02</td>
</tr>
<tr>
<td>1000</td>
<td>0.000232</td>
<td>-1.05e+04</td>
<td>-2.37e+03</td>
</tr>
<tr>
<td>1000</td>
<td>0.000987</td>
<td>-1.16e+04</td>
<td>-3.64e+03</td>
</tr>
<tr>
<td>1000</td>
<td>0.001552</td>
<td>-1.19e+04</td>
<td>-4.61e+03</td>
</tr>
<tr>
<td>1000</td>
<td>0.002470</td>
<td>-1.27e+04</td>
<td>-6.54e+03</td>
</tr>
<tr>
<td>1000</td>
<td>0.002933</td>
<td>-1.28e+04</td>
<td>-7.24e+03</td>
</tr>
<tr>
<td>1500</td>
<td>0.000525</td>
<td>-1.91e+04</td>
<td>-8.56e+03</td>
</tr>
<tr>
<td>1500</td>
<td>0.000920</td>
<td>-1.94e+04</td>
<td>-1.13e+04</td>
</tr>
<tr>
<td>1500</td>
<td>0.001336</td>
<td>-2.20e+04</td>
<td>-1.30e+04</td>
</tr>
<tr>
<td>1500</td>
<td>0.001861</td>
<td>-2.12e+04</td>
<td>-1.48e+04</td>
</tr>
<tr>
<td>1500</td>
<td>0.002334</td>
<td>-2.19e+04</td>
<td>-1.57e+04</td>
</tr>
<tr>
<td>2000</td>
<td>0.000405</td>
<td>-2.88e+04</td>
<td>-1.64e+04</td>
</tr>
<tr>
<td>2000</td>
<td>0.000472</td>
<td>-2.91e+04</td>
<td>-1.67e+04</td>
</tr>
<tr>
<td>2000</td>
<td>0.000622</td>
<td>-2.94e+04</td>
<td>-1.77e+04</td>
</tr>
<tr>
<td>2000</td>
<td>0.000833</td>
<td>-3.02e+04</td>
<td>-1.95e+04</td>
</tr>
<tr>
<td>2000</td>
<td>0.001124</td>
<td>-2.98e+04</td>
<td>-2.22e+04</td>
</tr>
</tbody>
</table>

Table 3.3.4: Performance Comparison on large Bayesian Networks with exponential distribution

In the above tables, the solutions listed in the columns of SGA and the columns of SB-GA were obtained by running both GAs 20 runs and recording the best values. The optimal solutions in Table 3.3.2 and Table 3.3.3 were computed by an exact method called barrier method [70]. Using this barrier method we can not get solutions for large Bayesian Networks (for example trace4 in Table 3.3.1, all 200-RV networks in Table 3.3.3, and all networks in Tables 3.3.4 and 3.3.5). Using our new GA though, we can get good solutions.
<table>
<thead>
<tr>
<th>Number of RVs</th>
<th>Conn. Density</th>
<th>SGA (log)</th>
<th>SB-GA (log)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.000735</td>
<td>-3.82e+02</td>
<td>-3.23e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.001948</td>
<td>-3.95e+02</td>
<td>-3.26e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.002411</td>
<td>-4.01e+02</td>
<td>-3.35e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.002938</td>
<td>-4.21e+02</td>
<td>-3.45e+02</td>
</tr>
<tr>
<td>500</td>
<td>0.004447</td>
<td>-4.10e+02</td>
<td>-3.45e+02</td>
</tr>
<tr>
<td>1000</td>
<td>0.000464</td>
<td>-8.90e+02</td>
<td>-7.24e+02</td>
</tr>
<tr>
<td>1000</td>
<td>0.000891</td>
<td>-9.22e+02</td>
<td>-7.52e+02</td>
</tr>
<tr>
<td>1000</td>
<td>0.001794</td>
<td>-9.10e+02</td>
<td>-7.55e+02</td>
</tr>
<tr>
<td>1000</td>
<td>0.002025</td>
<td>-9.46e+02</td>
<td>-7.90e+02</td>
</tr>
<tr>
<td>1000</td>
<td>0.003095</td>
<td>-9.38e+02</td>
<td>-8.13e+02</td>
</tr>
<tr>
<td>1500</td>
<td>0.000238</td>
<td>-1.42e+03</td>
<td>-1.16e+03</td>
</tr>
<tr>
<td>1500</td>
<td>0.000770</td>
<td>-1.46e+03</td>
<td>-1.21e+03</td>
</tr>
<tr>
<td>1500</td>
<td>0.001145</td>
<td>-1.48e+03</td>
<td>-1.23e+03</td>
</tr>
<tr>
<td>1500</td>
<td>0.001404</td>
<td>-1.50e+03</td>
<td>-1.28e+03</td>
</tr>
<tr>
<td>1500</td>
<td>0.002649</td>
<td>-1.54e+03</td>
<td>-1.39e+03</td>
</tr>
<tr>
<td>2000</td>
<td>0.000391</td>
<td>-2.01e+03</td>
<td>-1.66e+03</td>
</tr>
<tr>
<td>2000</td>
<td>0.000537</td>
<td>-2.04e+03</td>
<td>-1.70e+03</td>
</tr>
<tr>
<td>2000</td>
<td>0.000776</td>
<td>-2.01e+03</td>
<td>-1.72e+03</td>
</tr>
<tr>
<td>2000</td>
<td>0.000878</td>
<td>-2.06e+03</td>
<td>-1.75e+03</td>
</tr>
<tr>
<td>2000</td>
<td>0.001142</td>
<td>-2.06e+03</td>
<td>-1.80e+03</td>
</tr>
</tbody>
</table>

Table 3.3.5: Performance Comparison on large Bayesian Networks with flat distribution

### 3.3.2.2 Results on CPU time comparision

To further compare the performance of our structure-based GA with standard GA, we took the four real-world trace networks and conducted the following tests\(^1\):

- Run the Standard GA once for 500 generations on each network. Table 3.3.6 shows the results. Column “Best \((S_1)\)” represents the best solution obtained by the Standard GA in 500 generations. Column “CPU \((T_1)\)” represents

---

\(^1\)Due to technical problems (hard disc failure), the two GENIE networks were unrecoverable.
the CPU time required to obtain these best solutions. Column “Generation ($G_1$)” represents the corresponding generations to CPU time $T_1$.

- Run our structure-based GA once for at most CPU time $T_2$ ($T_2 < T_1$). Table 3.3.7 shows the results. Column “Best ($S_2$)” represents the best solution obtained by the structure-based GA after running for CPU time $T_2$. Column “CPU ($T_3$)” represents the CPU time required to obtain these best solutions. Column “Generation ($G_1$)” represents the corresponding generations to CPU time $T_3$.

- Given solutions $S_3 > S_1$, let the structure-based GA run until the solutions obtained by it is better than $S_3$. Table 3.3.8 shows the results. Column “Best ($S_4$)” represents the best solution obtained by the structure-based GA running CPU time $T_4$. Column “CPU ($T_3$)” represents the CPU time required to obtain the solutions $S_3$. Column “Generation ($G_2$)” represents the corresponding generations to CPU time $T_4$.

<table>
<thead>
<tr>
<th>BN</th>
<th>RV Num</th>
<th>Best ($S_1$)</th>
<th>CPU ($T_1$)</th>
<th>Generation ($G_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>103</td>
<td>4.044e-19</td>
<td>20.49</td>
<td>352</td>
</tr>
<tr>
<td>trace2</td>
<td>158</td>
<td>4.065e-36</td>
<td>49.27</td>
<td>495</td>
</tr>
<tr>
<td>trace3</td>
<td>186</td>
<td>3.825e-42</td>
<td>59.05</td>
<td>477</td>
</tr>
<tr>
<td>trace4</td>
<td>196</td>
<td>1.098e-44</td>
<td>61.58</td>
<td>459</td>
</tr>
</tbody>
</table>

Table 3.3.6: Results of Standard GA on trace Networks for 500 Generations.

From these tables, we see that our structure-based GA achieves a better solution in fewer generations (or CPU time) than the standard GA.
<table>
<thead>
<tr>
<th>BN</th>
<th>Given CPU($T_2$)</th>
<th>Best ($S_2$)</th>
<th>CPU ($T_3$)</th>
<th>Generation $G_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>19.00</td>
<td>2.089e-13</td>
<td>1.90</td>
<td>18</td>
</tr>
<tr>
<td>trace2</td>
<td>48.00</td>
<td>1.433e-16</td>
<td>7.04</td>
<td>38</td>
</tr>
<tr>
<td>trace3</td>
<td>58.00</td>
<td>2.387e-23</td>
<td>19.49</td>
<td>79</td>
</tr>
<tr>
<td>trace4</td>
<td>60.00</td>
<td>1.095e-13</td>
<td>16.91</td>
<td>71</td>
</tr>
</tbody>
</table>

Table 3.3.7: Results of Structure-based GA on *trace* Networks Given CPU time $T_2 < T_1$.

<table>
<thead>
<tr>
<th>BN</th>
<th>Given Sol($S_3$)</th>
<th>Best ($S_4$)</th>
<th>CPU ($T_4$)</th>
<th>Generation $G_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>4.000e-18</td>
<td>6.135e-16</td>
<td>1.23</td>
<td>11</td>
</tr>
<tr>
<td>trace2</td>
<td>4.000e-35</td>
<td>1.805e-32</td>
<td>2.33</td>
<td>13</td>
</tr>
<tr>
<td>trace3</td>
<td>3.000e-41</td>
<td>2.828e-35</td>
<td>2.43</td>
<td>11</td>
</tr>
<tr>
<td>trace4</td>
<td>1.000e-41</td>
<td>6.245e-38</td>
<td>3.09</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 3.3.8: Results of Structure-based GA on *trace* Networks Given Solution $S_3 > S_1$.

From the results presented above, we can see that our new GA by employing “group” crossover and mutation improves significantly over the standard GA approach. Intuitively, reinforcement learning captures groups of highly correlated RVs, and uses these groups to direct GA to climb a hill. By adaptively identifying the structural characteristics of the Bayesian Network, our approach is less sensitive to the structure and probability distributions of the Bayesian network than standard GA. It works well for Bayesian networks containing many *zero probabilities* or having exponential probability distributions. The standard GA mainly performs well only for the Bayesian network with flat probability distributions. From the above results, we could see that our GA still performs better than standard GA for Bayesian networks with flat probability distributions.
Since we know that changing the evidence in Bayesian networks can drastically alter the search space, our method can automatically adapt to such changes and learn accordingly. When the evidence is changed, the evaluation $E_i^t$ $(i = 1, \ldots, m)$ for group $i$ may be updated. According to equation (22), the payoff may be updated. From the learning algorithm (Section 4.2.5) we know that updating the payoff may change the group values. Changing the group values will update the reinforcement and thus updating the weights of the controller. Updating the weights of the controller is likely to cause the controller to generate a new grouping result and so automatically directs the GA to search in this new search space.

### 3.4 Problem II: The Tactical Fixed Interval Scheduling Problem

TFISP is the problem of determining the minimum number of parallel non-identical machines, such that a feasible schedule exists for a given set of jobs. It is a NP-hard problem. Kroon, Salomon and Wassenhove [40] described two practical situations in which this problem occur. One is for Schiphol Amsterdam Airport. The objective was to obtain insight into the future required gate capacity at the terminal for different scenarios relative to flight intensities. For each flight intensity, a number of timetables were generated. TFISP was used to calculate the number of required gates. The other is for tactical capacity planning of aircraft maintenance personnel for KLM Royal Dutch Airlines. TFISP is used as the core model for this situation. The goal is to determine the most efficient size
and composition of the teams in which the engineers operate, i.e. to determine
the minimum number of engineers per team and the licenses they should have
under different scenarios with respect to flight intensities and composition of the
fleet. Exact algorithms are intractable to solve the TFISP. Researchers have
developed some methods to solve this kind of problems[12, 22, 30, 17, 16]. In this
section, we use our structure-based approach to solve this problem.

3.4.1 Problem description

TFISP is the problem of determining the minimum number of parallel non-
identical machines, such that a feasible schedule exists for a given set of jobs.
Each job belongs to a specific job class, and has a fixed start time, a fixed finish
time, and a processing time that equals the length of the time interval between
the job’s start and finish time. Each machine is allowed to process jobs only from
a prespecified subset of job classes and can process, at most, only one job at a
time.

We may describe TFISP as a network flow model with side constraints [40].
The underlying directed graph $G$ contains $C$ subgraphs $G_c$, each one correspond-
ing to one of the machine classes. The node set $N_c$ of $G_c$ has a one-to-one
 correspondence with the set of start and finish times of the jobs that can be han-
dled by machine class $c$. The node set $N_c$ is also denoted as $\{n_{c,r}^s| r = 1, \ldots, p_c\}$,
where $p_c = |N_c|$. Here, it is assumed that for $r = 2, \ldots, p_c$ the nodes $n_{c,r-1}$ and
$n_{c,r}$ correspond to subsequent time instant. A particular job $j$ with $j \in M_c$, is
represented in $G_c$ by an arc from the node corresponding to start time $s_j$ to the
node corresponding to finish time $f_j$, where $M_c$ is the set of all jobs that can be
carried out by machines in machine class $c$. This arc has upper capacity one. In
$G_c$ there is for $r = 2, \ldots, p_c$ an arc from $n_{c,r-1}$ to $n_{c,r}$ with unlimited capacity.
The graphs $G_c$ are linked together by a super source $P$ and a super sink $Q$. There
is an arc from $P$ to each of the node $n_{c,1}$ and there is an arc from each of he nodes
$n_{c,p_c}$ to the super sink $Q$. Both arcs have unlimited capacity. Figure 3.4.1 shows
an example of a set of jobs and the corresponding graph $G$.

Fig. 3.4.1: An instance of TFISP and the corresponding graph $G$. Here the
number of job classes is 3, and the number of different machine classes is 2.
Machine 1 can handle jobs in class 1 and 3, and machine 2 can handle jobs in
class 2 and 3.

The side constraints that must be satisfied specify that all flows must be
integer, and that for each job the total amount of flow in the corresponding arcs
must be exactly one unit. Now, in TFISP the objective is to send an amount
of flow from the super source $P$ to the super sink $Q$ in such a way that: (1) all
capacity constraints are satisfied, (2) all side constraints are satisfied, and (3) the total amount of flow is minimal. As for the problem considering the machine costs in TFISP, the structure of network flow model will not change, but the side constraints will change to consider the machine costs. As we mentioned before, we further extend the problem by considering that each specific job is a job class. At this time the structure of network flow model still fits this problem. Instead of being allowed to process jobs from a prespecified subset of job classes, each machine is allowed to process a prespecified subset of jobs. Needless to say, exact algorithms are intractable to solve the TFISP.

3.4.2 Experiments

This section describes experimental results evaluating the performance of our method on various instances of TFISP. We ran both the standard GA and our new GA through 500 generations. We also ran an Integer Program (IP) method to compare the results. All the programs written in C++ ran on a Sun Ultra5 270MHz. The GA used in our method is based on the GENESIS[29] framework. Each population has 100 individuals. The crossover rate is 0.55, the standard mutation rate is 0.01, and “group” mutation rate is 0.001 in our experiments.

First we randomly generate the instances following the testing design proposed by Kroon [40]. We need to mention that Kroon’s paper focuses on finding some lower bounds and upper bounds for the feasible solution (not necessary tight). Here, we are interested in searching for the optimal or near optimal solution.
In their design, they considered four sets of problem instances summarized in Table 3.4.1. In these four sets, they considered two kinds of job overlap over time (uniform distribution and peak distribution), and two kinds of the available machine classes (all combinations and limited number of combination).

Table 3.4.1: Summary of Set Characteristics

<table>
<thead>
<tr>
<th>Job Overlap</th>
<th>Machine Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>Set I</td>
</tr>
<tr>
<td>Peaks</td>
<td>Set II</td>
</tr>
<tr>
<td></td>
<td>Limited</td>
</tr>
<tr>
<td></td>
<td>Set III</td>
</tr>
<tr>
<td></td>
<td>Set IV</td>
</tr>
</tbody>
</table>

Tables 3.4.2 through Table 3.4.5 show the results on these four sets. In these tables, “A” is the number of job classes, “J” is the number of jobs, “D” is the maximum job duration, “SB-GA” stands for our structure-based GA, “SGA” stands for the standard GA, “Gen” is the generations when GAs get the solution, “CPU” is the cpu time in seconds, and “N/A” means not available. However, it turns out that the solution landscape of the instances in these cases are really flat. Thus the standard GA and our new GA can find a near optimal solution easily, especially for Set III and Set IV. But IP can not solve all the instances in these tables, especially the ones with larger number of jobs.

In more practical situations, we usually need to consider the cost of each machine class. This will skew the solution landscape and increase the complexity of the problem. In this case, the fitness function becomes:

\[
    f = \sum_{m=1}^{m=M} \sum_{j=1}^{K} C_{m}Y_{j}
\]  

(28)
Table 3.4.2: Performance Comparison on Set I

<table>
<thead>
<tr>
<th>A</th>
<th>J</th>
<th>D</th>
<th>IP</th>
<th>SB-GA Solution</th>
<th>CPU</th>
<th>Gen.</th>
<th>CPU</th>
<th>Solution</th>
<th>Gen.</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>12</td>
<td>70.71</td>
<td>12</td>
<td>17.93</td>
<td>9.91</td>
<td>15.96</td>
<td>18</td>
<td>127</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>200</td>
<td>17</td>
<td>508.75</td>
<td>19</td>
<td>7.92</td>
<td>18</td>
<td>127</td>
<td>13.99</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>300</td>
<td>25</td>
<td>494.86</td>
<td>25</td>
<td>23.97</td>
<td>25</td>
<td>103</td>
<td>11.72</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>100</td>
<td>11</td>
<td>313.82</td>
<td>17</td>
<td>11.24</td>
<td>17</td>
<td>77</td>
<td>8.35</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>200</td>
<td>18</td>
<td>825.13</td>
<td>23</td>
<td>12.01</td>
<td>22</td>
<td>282</td>
<td>31.38</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>35</td>
<td>11.51</td>
<td>33</td>
<td>171</td>
<td>19.21</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>100</td>
<td>19</td>
<td>1344.76</td>
<td>21</td>
<td>54.49</td>
<td>21</td>
<td>124</td>
<td>43.05</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>200</td>
<td>34</td>
<td>N/A</td>
<td>34</td>
<td>52.24</td>
<td>34</td>
<td>71</td>
<td>25.61</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>47</td>
<td>57.76</td>
<td>46</td>
<td>218</td>
<td>79.38</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>100</td>
<td>25</td>
<td>1873.29</td>
<td>28</td>
<td>92.97</td>
<td>29</td>
<td>196</td>
<td>67.67</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>42</td>
<td>71.17</td>
<td>41</td>
<td>272</td>
<td>95.96</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>53</td>
<td>113.23</td>
<td>52</td>
<td>253</td>
<td>89.87</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>76</td>
<td>493.97</td>
<td>77</td>
<td>85</td>
<td>165.6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>79</td>
<td>386.86</td>
<td>79</td>
<td>283</td>
<td>516.98</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>144</td>
<td>1022.04</td>
<td>144</td>
<td>227</td>
<td>1669.95</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>153</td>
<td>1185.62</td>
<td>152</td>
<td>211</td>
<td>1446.93</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4.3: Performance Comparison on Set II

<table>
<thead>
<tr>
<th>A</th>
<th>J</th>
<th>D</th>
<th>IP</th>
<th>SB-GA Solution</th>
<th>CPU</th>
<th>Gen.</th>
<th>CPU</th>
<th>Solution</th>
<th>Gen.</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>16</td>
<td>78.77</td>
<td>18</td>
<td>10.61</td>
<td>16</td>
<td>106</td>
<td>13.94</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>200</td>
<td>25</td>
<td>208</td>
<td>27</td>
<td>21.9</td>
<td>26</td>
<td>180</td>
<td>20.27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>300</td>
<td>37</td>
<td>61.22</td>
<td>37</td>
<td>8.82</td>
<td>37</td>
<td>65</td>
<td>7.47</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>100</td>
<td>18</td>
<td>60.41</td>
<td>22</td>
<td>7.86</td>
<td>22</td>
<td>26</td>
<td>2.96</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>200</td>
<td>26</td>
<td>147.44</td>
<td>32</td>
<td>6.18</td>
<td>30</td>
<td>183</td>
<td>19.89</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>300</td>
<td>31</td>
<td>249.83</td>
<td>38</td>
<td>40.44</td>
<td>37</td>
<td>156</td>
<td>17.29</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>100</td>
<td>30</td>
<td>N/A</td>
<td>32</td>
<td>76.43</td>
<td>32</td>
<td>164</td>
<td>58.81</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>58</td>
<td>41.68</td>
<td>58</td>
<td>32</td>
<td>12.05</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>77</td>
<td>77.01</td>
<td>77</td>
<td>125</td>
<td>48.4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>40</td>
<td>64.07</td>
<td>38</td>
<td>260</td>
<td>89.8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>61</td>
<td>80.32</td>
<td>60</td>
<td>233</td>
<td>82.78</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>79</td>
<td>60.87</td>
<td>77</td>
<td>253</td>
<td>89.97</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>115</td>
<td>251.93</td>
<td>115</td>
<td>200</td>
<td>404.53</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>117</td>
<td>494.1</td>
<td>120</td>
<td>271</td>
<td>496.22</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>266</td>
<td>1948.45</td>
<td>267</td>
<td>240</td>
<td>1867.77</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>251</td>
<td>1376.61</td>
<td>243</td>
<td>293</td>
<td>1989.22</td>
<td></td>
</tr>
</tbody>
</table>

where $M$ is the total number of the machine classes, $C_m$ is the cost of machine class $m$, $K$ is the number of machines in machine class $m$, and $Y_m$ is the number of jobs that can be handle on same machine during the planning horizon $T$. For instances in Set I and II, $A = 4, M = 6$, and $A = 5, M = 10$. For instances in Set III and IV, $A = 4, M = 3$, and $A = 5, M = 4$. The cost of each machine class is randomly chosen as follows: $C_0 = 10.3865, C_1 = 90.866, C_2 = 123.659, C_3 = 193.744, C_4 = 251.711, C_5 = 300.279, C_6 = 43.1259, C_7 = 145.276, C_8 = 202.385,$
and $C_g = 170.894$. Table 3.4.6 through 3.4.9 show the results of the above four sets by considering the cost of each machine class.

We use

$$\left( S_{SGA} - S_{SB-GA} \right) / S_{SB-GA} \quad (29)$$

to compute the improvement, where $S_{SB-GA}$ is the solution using our new GA method and $S_{SGA}$ is the solution using standard GA. From the results we can see:

The instances in Set III and IV generated by using limited machine combination
are easier to solve. For the instances in these two sets, we find that their paths are too specific, and do not have many choices. Considering the machine cost unfortunately did not skew the solution landscape very much. So both GAs perform well. The instances in Set I and II were generated by using all machine combinations. When considering the machine cost to these instances, the solution landscape of these instances are skewed. In this case, our new GA performed better than the standard GA. We can also see that the standard GA is more sensitive to the solution landscape than our new GA.

Table 3.4.6: Performance Comparison on Set I Considering Machine Cost

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>660</td>
<td>31.91</td>
<td>1600</td>
<td>20.43</td>
<td>2576</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>200</td>
<td>3064</td>
<td>24.94</td>
<td>3064</td>
<td>21.93</td>
<td>3806</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>300</td>
<td>3091</td>
<td>54.49</td>
<td>3091</td>
<td>15.81</td>
<td>3591</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>100</td>
<td>1158</td>
<td>62.21</td>
<td>1158</td>
<td>481</td>
<td>71.68</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>200</td>
<td>1944</td>
<td>31.71</td>
<td>1944</td>
<td>348</td>
<td>52.94</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>300</td>
<td>2143</td>
<td>93.24</td>
<td>2143</td>
<td>132</td>
<td>20.39</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>100</td>
<td>1949</td>
<td>1030.89</td>
<td>3256</td>
<td>312</td>
<td>139.51</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>203</td>
<td>125.61</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>7326</td>
<td>213.53</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>2521</td>
<td>78.88</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1428</td>
<td>84.21</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>5069</td>
<td>227.87</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>13970</td>
<td>420</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>9544</td>
<td>223</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>36090</td>
<td>458</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>30440</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 3.4.7: Performance Comparison on Set II Considering Machine Cost

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>660</td>
<td>31.91</td>
<td>1600</td>
<td>20.43</td>
<td>2576</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>200</td>
<td>3064</td>
<td>24.94</td>
<td>3064</td>
<td>21.93</td>
<td>3806</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>300</td>
<td>3091</td>
<td>54.49</td>
<td>3091</td>
<td>15.81</td>
<td>3591</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>100</td>
<td>1158</td>
<td>62.21</td>
<td>1158</td>
<td>481</td>
<td>71.68</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>200</td>
<td>1944</td>
<td>31.71</td>
<td>1944</td>
<td>348</td>
<td>52.94</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>300</td>
<td>2143</td>
<td>93.24</td>
<td>2143</td>
<td>132</td>
<td>20.39</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>100</td>
<td>1949</td>
<td>1030.89</td>
<td>3256</td>
<td>312</td>
<td>139.51</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>203</td>
<td>125.61</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>7326</td>
<td>213.53</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>2521</td>
<td>78.88</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1428</td>
<td>84.21</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>5069</td>
<td>227.87</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>13970</td>
<td>420</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>9544</td>
<td>223</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>36090</td>
<td>458</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>30440</td>
<td>500</td>
</tr>
</tbody>
</table>

Finally, we further extended the problems by considering that each specific job belongs to a unique job class and can be processed on several machine classes. The
problem becomes that of determining the minimum cost of parallel non-identical machines, such that a feasible non-preemptive schedule exists for a given set of jobs. This increases the complexity of the problem. We generate the new instances using the following design: We set the planning horizon \(T\) to 1, 000 and the number of machine classes to 10. We consider four maximum job duration: 100, 200, 300 and 400. Each machine class can run several jobs. Each job can run on several machine classes. Table 3.4.10 shows the results on instances based on this design. We can see that our new GA performs better than standard GA.
Table 3.4.10: Performance Comparison on Job Specification

<table>
<thead>
<tr>
<th>J</th>
<th>D</th>
<th>LP</th>
<th>Deterministic</th>
<th>Solution</th>
<th>CPU</th>
<th>Solution</th>
<th>SGA</th>
<th>Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>910.4</td>
<td>484</td>
<td>15,54</td>
<td>1936</td>
<td>400</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>1489</td>
<td>230</td>
<td>35.09</td>
<td>2790</td>
<td>382</td>
</tr>
<tr>
<td>100</td>
<td>300</td>
<td>1389</td>
<td>146.65</td>
<td>1564</td>
<td>260</td>
<td>40.24</td>
<td>3475</td>
<td>423</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
<td>N/A</td>
<td>N/A</td>
<td>1744</td>
<td>433</td>
<td>75.31</td>
<td>3699</td>
<td>364</td>
</tr>
<tr>
<td>200</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>1804</td>
<td>135</td>
<td>72.35</td>
<td>3194</td>
<td>296</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>2815</td>
<td>398</td>
<td>150.23</td>
<td>4142</td>
<td>487</td>
</tr>
<tr>
<td>200</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>3192</td>
<td>294</td>
<td>145.54</td>
<td>5535</td>
<td>448</td>
</tr>
<tr>
<td>200</td>
<td>400</td>
<td>N/A</td>
<td>N/A</td>
<td>3913</td>
<td>455</td>
<td>239.31</td>
<td>7134</td>
<td>438</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>4913</td>
<td>346</td>
<td>788.59</td>
<td>6842</td>
<td>370</td>
</tr>
<tr>
<td>500</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>7456</td>
<td>456</td>
<td>1062.03</td>
<td>10630</td>
<td>2358</td>
</tr>
<tr>
<td>500</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>10210</td>
<td>268</td>
<td>881.48</td>
<td>15230</td>
<td>449</td>
</tr>
<tr>
<td>500</td>
<td>400</td>
<td>N/A</td>
<td>N/A</td>
<td>12810</td>
<td>357</td>
<td>878.34</td>
<td>18650</td>
<td>471</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>8061</td>
<td>289</td>
<td>2968.47</td>
<td>12040</td>
<td>427</td>
</tr>
<tr>
<td>1000</td>
<td>200</td>
<td>N/A</td>
<td>N/A</td>
<td>14810</td>
<td>471</td>
<td>2957.3</td>
<td>20470</td>
<td>403</td>
</tr>
<tr>
<td>1000</td>
<td>300</td>
<td>N/A</td>
<td>N/A</td>
<td>21800</td>
<td>465</td>
<td>4031.36</td>
<td>23740</td>
<td>417</td>
</tr>
<tr>
<td>1000</td>
<td>400</td>
<td>N/A</td>
<td>N/A</td>
<td>27500</td>
<td>329</td>
<td>2864.05</td>
<td>38530</td>
<td>325</td>
</tr>
</tbody>
</table>

From our experiments, we can see that our new structure-based GA with "group" crossover and "group" mutation is quite promising. Intuitively, the reinforcement learning system captures groups of correlated jobs and uses these groups to direct the GA. By adaptively identifying the structural characteristics of the problem, our approach is "less" sensitive to the structure.

3.5 Summary

This chapter introduced our structure-based approach by considering the structural characteristics of the problems based on GDT. In our approach, knowledge of problem characteristics was obtained online and incorporated into the search algorithm. Thus, the search algorithm dynamically and automatically changes its search direction according to the knowledge obtained so far.

We regarded the correlation among the decision variables as the structural characteristics in our approach. Usually, the decision variables can be grouped together into several groups according to their correlations. In our approach, we use a Reinforcement Learning (RL) system to learn the structural characteristics
of the problem, thereby grouping the decision variables of the problem into several
groups based on the structural characteristics. We then use these structural
characteristics to develop a new Genetic Algorithm by introducing structural
operations to work on these groupings and recombine them together to search
for a better solution to the problem. During run time the system evaluates the
performance of the GA and continues with reinforcement learning to further tune
the search for a better grouping.

We compared our approach with standard GA by testing on two problems,
belief revision over Bayesian Networks and the Tactical Fixed Interval Scheduling
Problem. From experimental results we can see, our approach can get better
results than standard GA for solving problems with skewed solution space. For
example, when solving belief revision over Bayesian Networks, our approach can
get better solutions than standard GA, especially for networks with extreme
distribution of probabilities. The solution spaces of these cases are skewed. For
these cases, the structural knowledge may capture the search space and direct
the new GA to search. When solving original TFISP and original TFISP by
considering the cost, since the solution spaces of these problems are flat, our
approach may not be able to get better solutions than standard GA. But when
solving the extend TFISP, job specification problems, since the solution spaces of
these problems are skewed, our approach can get better solutions than standard
GA. Intuitively, when the solution space of a problem is skewed, the structural
knowledge may capture the search space and direct the new GA to search. By
adaptively changing the search direction, our approach is less sensitive to the problem and can get better solutions.
Chapter 4

Conclusion

The focus of this research has been to explore the impact of knowledge on algorithm performance and how to extract and incorporate knowledge into a search algorithm, especially during problem solving in a systematic way. As far as we know, no one has systematically discussed this before. To do so, in Chapter 2 we defined the knowledge about a problem domain as a distribution of optimal solutions over the solution space. We developed a Directional Tree (DT) model which concurrently describes algorithm behavior and represents knowledge explicitly, and thus provides us a tool to analyze the impact of knowledge on algorithm performance.

We analyzed the performance of an algorithm when that algorithm is used for solving any problems from problem space $P$. We formally proved that no one algorithm will perform better than another when the distribution is uniform. Because the Oracle Model for $P$ is uniform, this means there is no single ideal general purpose algorithm. We also discussed the performance of an algorithm
when that algorithm is used to solve problems in different sub problem spaces, \(SPs\), by giving specific examples. We can see that an algorithm’s performance is different when solving problems for different \(SPs\). Finally, we investigated the impact of knowledge on deterministic algorithms. We introduced a general formula to compute the expected length for deterministic algorithms. Additionally, we consider the situation when we do not have the full distribution of optimal solutions for a \(SP\). In particular, the solution space has been classified into several disjoint nonempty clusters each of which has a bound on the distribution of optimal solutions. In this case, we found a general formula that identifies an upper bound on performance for a deterministic algorithm. Although DTs explicitly describe all the possible behaviors of an algorithm, we demonstrated that the model can provide closed-form solutions for behavior prediction.

Our DT model assumes that knowledge about the problem is obtained ahead of time prior to the execution of any probabilistic or deterministic algorithm. However, such prior knowledge may not be readily or immediately available. Hence, we generalize our DT model to take into account acquisition of problem knowledge during execution. Our generalized directional tree (GDT) model considers a sequence of bags of points instead of just points as in DT, and the number of points in each bag is restricted to less than or equal to \(K, K < n\). The GDT can be constructed dynamically. This online construction can be viewed as a sequence of partial GDTs and directly reflected by the online algorithms derived
from the GDT. The GDT provides a framework for developing online optimization algorithms that dynamically updates its searching decisions in response to knowledge obtained so far.

In most cases, we may not have a complete distribution of the optimal solutions. However, we may have some knowledge about the structural characteristics of discrete optimization problems. In Chapter 3, we proposed a structure-based approach by considering the knowledge of structural characteristics. The GDT model forms the basis for our approach. More specifically, the decision variables of a discrete optimization problem have some correlations among them, and can be grouped together into several groups according to these correlations. We use a Reinforcement Learning (RL) system to learn the structural characteristics of the problem, thereby grouping the decision variables of the problem into several groups based on the structural characteristics. We then use these structural characteristics to develop a Genetic Algorithm (GA) by introducing structural operations to work on these groupings and recombine them together to search for a better solution to the problem. During run time the system evaluates the performance of the GA and continues with reinforcement learning to further tune the search for a better grouping.

From experimental results we can see, our approach can get better results than standard GA for solving problems with skewed solution space. For example, when solving belief revision over Bayesian Networks, our approach can get better solutions than standard GA, especially for networks with extreme distribution of
probabilities. The solution spaces of these cases are skewed. For these cases, the structural knowledge may capture the search space and direct the new GA to search. When solving original TFISP and original TFISP by considering the cost, since the solution spaces of these problems are flat, our approach may not be able to get better solutions than standard GA. But when solving the extend TFISP, job specification problems, since the solution spaces of these problems are skewed, our approach can get better solutions than standard GA. Intuitively, when the solution space of a problem is skewed, the structural knowledge may capture the search space and direct the new GA to search. By adaptively changing the search direction, our approach is less sensitive to the problem and can get better solutions.

4.1 Future Work

Based on our GDT framework, we designed a structure-based approach using GA and RL, we may use this framework to build other new algorithms.

In Chapter 3 we proposed a new GA by using group operations. We know that each grouping provides a direction for that GA to search. In the future, we may seek to explore what is a good grouping and how to make a grouping better. Currently, we use the quality of subsolution as the group evaluation. So one possible way is to find relationships between group evaluation and fitness. We believe that the fitness function may have properties that can help in group evaluation. For example, a natural property is that the optimal value obtained
from the fitness function can be naturally decomposed into subfunctions whose
global optimas are highly likely to be put together to form the global optima of
the fitness function.

Also, from our experience we know that number of groups affects the perfor-
mane of our structure based GA. Hence, we would like to determine the optimal
number of group for efficient searching.

4.2 Contribution

The contributions in this dissertation are as follows:

- Developed a Directional Tree (DT) model which concurrently describes al-
gorithm behavior and represents knowledge explicitly, and thus provides us
with a tool to analyze the impact of knowledge on algorithm performance.

- Generalized our DT model to take into account acquisition of problem
knowledge during execution. Our generalized directional tree (GDT) model
provides a framework for incorporating knowledge obtained online into al-
gorithms, and thus enables us to develop online optimization algorithms
that dynamically updates their search decisions in response to knowledge
obtained so far.

- Using GDT, we properly developed a structure-based approach which can
adapt to problems by incorporating structural knowledge extracted online.
We compared our approach against standard GA by testing on two NP-
hard problems, belief revision over Bayesian Networks and the Tactical
Fixed Interval Scheduling Problem. From experimental results, we see that our new online structural approach obtains better results than the standard GA when the solution space of the problem is skewed.
Appendix A

No Free Lunch Theorems

In this appendix, we introduce the No Free Lunch Theorems. The following description is excerpted from Wolpert and Macready [72].

Usually, one might expect that there are pairs of search algorithms $A$ and $B$ such that $A$ performs better than $B$ on average, even if $B$ sometimes outperforms $A$. As an example, one might expect that hill-climbing usually outperforms hill-descending if one’s goal is to find a maximum of the cost function. One might also expect it would outperform a random search in such a context. The No Free Lunch (NFL) theorems of Wolpert and Macready [72] point out that this is not the case. There are no “free lunches” for effective optimization if we do not take into account any particular biases or properties of our cost function. This appendix introduce the basic idea of NFL.

Assume $X$ is the search space, $Y$ is associated cost values space. Both $X$ and $Y$ are finite. Define $d_m = d_m(i) \equiv a(i), d_m(i)$ for $i = 1, \ldots, m$ to be a set of $m$ distinct search points and associated cost values ordered in some way (usually,
according to the time at which they are generated) with the ordering index given by \( i \). Let us call this a population of size \( m \). We denote the set of all populations of size \( m \) by \( D_m \).

Let \( f \) indicate a single-valued function from \( X \) to \( Y \), \( f \in Y_X \). Note that there are a finite number of \( f \) if \( |X| \) and \( |Y| \) are finite. At each stage of a search algorithm, a new point \( x' \in X \) is chosen based on the members of the current population \( d \); the pair \( \{x', f(x')\} \) is added to \( d \); and the procedure repeats.

Any search algorithm beginning with a population of one or more complete solution \( x \in X \) and the associated \( Y \) values is a mapping taking any population to a new point in the search space. Assume that the new search point has not already been visited. So in particular a deterministic search algorithm is a mapping \( a : d \in D \to \{x| x \not\in d^x \}, \) where \( D \equiv \cup_m D_m \), and in particular contains the empty set.

Note that the population contains all points sampled so far. In particular, in a conventional hill-climber that works by moving from \( x \) to that neighbors \( x \). All those evaluated points are contained in the population, not only \( x \) and the neighbor of \( x \) with highest fitness.

We are interested in the histogram, \( \bar{c} \), of cost values that an algorithm, \( a \), obtains on a particular cost function, \( f \), given \( m \) distinct cost evaluations. Note that \( \bar{c} \) is given by the \( y \) values of the population, \( d^y_m \), and is a vector of length \( |Y| \) whose \( \bar{a}h \) component is the number of members in the population \( d_m \) having cost \( f_i \). Once we have \( \bar{c} \) we can use it to assess the quality of the search in any
way we choose. Consequently, we are interested in the conditional probability that histogram $\hat{c}$ will be obtained under $m$ iterations of algorithm $a$ on $f$. This quantity is given by the conditional probability $P(\hat{c}|f, m, a)$.

A natural question concerning this scenario is how $F_1$, the set of $f$ for which some algorithm $a_1$ outperforms another algorithm $a_2$, compares to $F_2$, the set of $f$ for which the reverse is true. We have the following theorem:

**Theorem A.0.1.** For any pair of algorithm $a_1$ and $a_2$,

$$\sum_f P(\hat{c}|f, m, a_1) = \sum_f P(\hat{c}|f, m, a_2)$$

An immediate corollary is that for any performance measure $\Phi(\hat{c})$, the average over all $f$ of $P(\Phi(\hat{c})|f, m, a)$ is independent of $a$. So the precise way that the histogram is mapped to a performance measure is irrelevant.

Note that the no free lunch result implies that if we know nothing about $f$, then $P(\hat{c}|m, a)$, which is the probability we obtain histogram $c$ after $m$ distinct cost evaluations of algorithm $a$, is independent of $a$. This follows from:

$$P(\hat{c}|m, a) = P(\hat{c}|f, m, a)P(f|m, a) = \sum_f P(\hat{c}|f, m, a)P(f)$$

If we know nothing about $f$ then all $f$ are equally likely, which means that for all $f$, $P(f) = \frac{1}{|Y|}$. Accordingly, for this ”no knowledge” scenario, $P(\hat{c}|m, a) = |Y|^{-|x|}\sum_f P(c|f, m, a)$, which is independent of $a$ by the no free lunch theorem.

Similarly, you can derive an NFL result for averaging over all priors. In this, the uniform $P(f)$ case is not some ”pathological case”, on the edge of the space. Rather it is the typical case.
Another immediate consequence of the NFL result is that the expected histogram \( E(\bar{c}|f, m, a) = \sum_c \bar{c} P(\bar{c}|f, m, a) \) is, on average, the same for all algorithm. More generally, for any two algorithms, at the point in their search where they have both created a population of size \( m \), if algorithm \( a_1 \) has better performance than algorithm \( a_2 \) over some subset \( \phi \subset F \) of functions, then \( a_2 \) must perform better on the set of remaining functions \( F \setminus \phi \). So for example if simulated annealing outperforms genetic algorithms on some set \( \phi \), genetic algorithms must outperform simulated annealing on \( F \setminus \phi \). As another example, even if one’s goal is find a maximum of the cost function, hill-climbing and hill-descending are equivalent, on average.

A particularly striking example of this last point is the case where \( a_2 \) is the algorithm of random search. The NFL result says that there are as many \( f \) for which the random algorithm outperforms your favorite search algorithm as vice-versa. There are as many \( f \) for which your algorithm’s guesses for where to search are worse than random as for which they are better. The risk you take in choosing an algorithm is not that it may perform randomly on the \( f \) at hand, but that it may very well perform even worse.

Often in the real world one has some a priori knowledge concerning \( f \). However only very rarely is that knowledge explicitly used to help set the algorithm. The unreasonableness of this is demonstrated by the NFL theorem, which illustrates that even if we do know something about \( f \), if we fail to explicitly incorporate
that knowledge into $a$ then we have no assurances the $a$ will be effective; we are simply relying on a fortuitous matching between $f$ and $a$. 
Bibliography


decision support tool for complex uncertain systems modeled with bayesian
Intelligence, pages 368–375, 1993.

for adaptive control of markov chains. IEEE Transactions on System, Man

[54] Eugene Santos, Jr. On the generation of alternative explanations with im-
lications for belief revision. In Proceedings of the Seventh Conference on
Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers, Inc.,

[55] Eugene Santos, Jr. and Solomon E. Shimony. Deterministic approximation
of marginal probabilities in bayes nets. IEEE Transactions on Systems,

[56] Eugene Santos, Jr., Solomon Eyal Shimony, and Edward Williams. Hy-
brid algorithms for approximate belief updating in bayes nets. International


[58] S. E. Shimony and E. Charniak. A new algorithm for finding map assign-
ments to belief networks. In Proc. 6th conference on Uncertainty in Artificial
Intelligence, 1990.


