Honours Year Project Report

A Toolkit for

Constraint Based Tree Search

By

Chew Tee Yong

School of Computing

National University of Singapore

1998/99

Project Number:  0892
Advisor:        Dr Martin Henz
Deliverables:  Report:  1 Volume
                Program:  1 Diskette
Abstract

Existing constraint programming systems offer a fixed set of inference engines for search. A search engine encompasses several dimensions that are independent of one another and different implementations of the dimensions form engines with different properties. In order to modify a particular dimension or to add a new dimension, the user usually had to code the entire search engine from scratch, which can be non-trivial. The toolkit developed in this project is designed to minimize the programming effort needed for designing and modifying search engines.

To support rapid-prototyping, the search engine is broken down into several modular dimensions in a toolkit so that a particular implementation of a dimension can be coded or modified and added to the toolkit as a module without changing the other existing modules.

The toolkit is coded in Oz and makes use of computational space -- a first-class citizen in Oz. The toolkit is geared towards solving of constraint problems.

Subject Descriptors: D.2.10 Design
I.2.8 Problem Solving, Control Methods, and Search

Keywords: Search Engine
Constraint Programming

Implementation Software: Oz Programming System Mozart 1.0.0
Acknowledgement

I would like to extend my heartfelt gratitude to my project supervisor Dr Martin Henz for his guidance, invaluable advice, patience and support throughout the course of this project.
Contents

Title .............................................................................................................. i
Abstract ...................................................................................................... ii
Acknowledgement .................................................................................... iii
Contents ...................................................................................................... iv

Chapter 1: Introduction ............................................................................. 1
  1.1 What are Constraints? ................................................................. 1
  1.2 Constraints Problems ................................................................. 2
  1.3 Constraint Inference Methods ...................................................... 4
  1.4 Oz 3 (Mozart) ............................................................................. 9

Chapter 2: Why is the Toolkit needed? .................................................... 16

Chapter 3: Aims of the Project ................................................................. 17

Chapter 4: The Toolkit ............................................................................. 18
  4.1 Structure of the Toolkit ................................................................. 18
  4.2 Nodes ......................................................................................... 19
  4.3 The Dimensions ......................................................................... 20

Chapter 5: Toolkit Usage ......................................................................... 26
  5.1 Levels of Usage .......................................................................... 26
  5.2 Using the Toolkit ........................................................................... 27
  5.3 Organization of the Search Engine .............................................. 43
Contents

5.4 Developing a Module ..................................... 44
5.5 Developing a Dimension ................................. 54
Chapter 6: Performance of the Toolkit ....................... 56
  6.1 Space Performance .................................... 57
  6.2 Time Performance .................................... 59
Chapter 7: Possible Improvements .......................... 61
Chapter 8: Comparison and Conclusion ...................... 62
  8.1 Search Engines in Oz vs ToolKit Engine ............... 62
  8.2 Oz Explorer vs ToolKit Engine ....................... 64
  8.3 Conclusion ........................................... 66
References ..................................................... 67

Appendix A : Scripts ......................................... A-1
Chapter 1: Introduction

1.1 What are Constraints?

Constraints are a mathematical formalization of relationships that can hold between objects. For example, “bigger-than” or “mother-of” are constraints which hold between objects in the real world. Real world constraints can be modeled by idealized mathematical constraints. Idealized mathematical constraints refer to constraints between mathematical objects such as numbers.

The statement $A+B=3$ is an example of a constraint. It stipulates a relation that must hold between any values chosen to replace the variables $A$ and $B$, namely the addition of the two chosen values must be equal to 3. The statement is not simply true or false; its status depends on the substituted values. For example, substituting $A$ by 1 and $B$ by 2 results in true, substituting $A$ by 1 and $B$ by 1 is false and substituting $A$ by a range of $[1, 3]$ and $B$ by $[1, 3]$ can be either true or false.

Thus a more meaningful question would be -- Is there any substitution that will yield truth? The answer is clearly yes for $A+B=3$ and no for $A=A+2$. There are some cases where the answer is don’t know.
1.2 Constraint Problems

We restrict ourselves to finite domain constraint problems. All variables in such a problem have a finite domain. Thus the problem has at most finitely many solutions.

The following are some examples of constraint problems.

**Send More Money**

The Send More Money Problem consists in finding distinct digits for the letters D, E, M, N, O, R, S, Y such that S; M \neq 0 (no leading zeros) and the equation

\[ \text{SEND} + \text{MORE} = \text{MONEY} \]

is satisfied. The unique solution of the problem is \(9567 + 1085 = 10652\).

**Safe**

The code of Professor Smart’s safe is a sequence of 9 distinct nonzero digits \(C_1, \ldots, C_9\) such that the following equations and inequalities are satisfied:

\[ C_4 - C_6 = C_7 \]

\[ C_1 \times C_2 \times C_3 = C_8 + C_9 \]

\[ C_2 + C_3 + C_6 < C_8 \]

\[ C_9 < C_8 \]

\[ C_1 \neq 1, \ldots, C_9 \neq 9 \]

Can you determine the code?
Coloring

Given a map showing the West European countries Netherlands, Belgium, France, Spain, Portugal, Germany, Luxembourg, Switzerland, Austria, and Italy, find a coloring such that neighboring countries have different color and a minimal number of colors is used.

Scheduling Problems

Scheduling problems refers to computation of a timetable for tasks competing for a given set of resources.
1.3 Constraint Inference Methods

There are many ways to solve constraint problems such as mathematical programming and local search in artificial intelligence. Constraint programming is used here to solve constraint problems.

The easiest way to understand how constraint problems are solved using finite-domain constraint programming is via an example.

**Example**

Let's look at an example involving linear equalities.

$X, Y$ and $Z$ are integer variables with domain between 1 to 4, that is

$X \in [1, 4], \quad Y \in [1, 4], \quad Z \in [1, 4].$

$X, Y$ and $Z$ are further constrained by the following equations.

$X \neq Y \neq Z$

$X < Y < Z$

$X + Y = Z$

**Constraint propagation**

Constraint propagation is an inference rule for finite domain problems that narrows the domain of variables. For instance, given the inequality

$A < B,$

and the domain (constraints)

$A \in [1, 4],$

$B \in [1, 4],$
Constraint propagation can narrow the domains of $A$ and $B$ to

\[
A \in [1, 3], \\
B \in [2, 4].
\]

Typically, constraint propagation alone is not enough to solve a constraint problem. Constraint propagation can result in a stable state that neither finds a solution nor fails.

Using our example, constraint propagation would stop when

\[
X \in [1, 2], \\
Y \in [2, 3], \\
Z \in [3, 4].
\]

This happens because there is no communication between the constraints in finite domain constraint programming. Propagators only look at the domains, and not at other propagators. That is why 2 is not excluded from the domain of $X$. The problem is thus neither solved nor failed.

Constraint propagation can also not result in failure even though no solution can be found. A straightforward example is as follows

\[
A \in [0, 1] \\
B \in [0, 1] \\
C \in [0, 1] \\
A \neq B \neq C
\]
This problem has no solution. Yet, constraint propagation will not be able to proceed beyond this stable state. The same reasons above apply.

For more complex types of reasoning, complex propagators are used. They are typically called "global constraints".

To proceed beyond the stable state, we need to do distribution.

**Distribution**

Distribution may involves selecting a not yet determined variable $w$ and an integer $n$ in the domain of $w$ so that constraint propagation can be jump-start by fixing $w = n$.

This choosing of a not-yet-determined variable and a value in the domain of the variable is determined by the search strategy. Common strategy of choosing the variable includes choosing the variable with the smallest domain, choosing the first variable defined or choosing the variable with the smallest value in its domain. Common strategy of selecting the values of a variable includes selecting the smallest value, selecting the largest value or even splitting the domain into half.

In our example, we can fixed $X = 1$ so that constraint propagation can continue.

In general, distribution involves selecting a constraint $C$ and adding it to the store. If this results in failure, $\neg C$ is added to the store to advance the propagation in another direction.
The Search Tree

The combination of constraint propagation and distribution yields a complete solution method for finite domain problems. Given a problem, do constraint propagation until no further propagation is possible. Stop if the problem is solved or failed. Otherwise, choose a constraint $C$ (in accordance with the search strategy) and add $C$ to the constraint store. If the resulting propagation fails, backtrack and add $\neg C$ to the store to continue the propagation in another direction.

By proceeding this way a search tree is obtained.

Each internal node of the tree corresponds to a stable state where distribution occurs. Each leaf of the tree corresponds to either a solution or a failure. The search tree is always finite since there are only finitely many variables all constrained to finite domains.

![Figure 1.1]

Figure 1.1
For our example, the search tree in Figure 1.1 is formed. The green diamond represents a solved space, the red square represents a failed space and the blue circle represents a stable space that is neither failed nor solved.
1.4 Oz 3 (Mozart)

Oz is the programming language used in the development of this toolkit. Oz is a multi-paradigm programming language. It is a high level programming language that is designed for modern advanced, concurrent, intelligent, networked, soft real-time, parallel, interactive and pro-active applications.

- Oz combines the salient features of object-oriented programming, by providing state, abstract data types, classes, objects and inheritance.
- It provides the salient features of functional programming by providing a compositional syntax, first class procedures, and lexical scoping. In fact, every Oz entity is first class, including procedures, threads, classes, methods, objects and computational spaces.
- It provides the salient features of logic programming and constraint programming by providing logic variables, disjunctive constructs, and programmable search strategies.
- It is also a concurrent language where users can create dynamically any number of sequential threads that can interact with each other. However, in contrast to conventional concurrent languages, each Oz thread is a data-flow thread. Executing a statement in Oz proceeds only when all real data-flow dependencies on the variables involved are resolved.
- The Mozart system supports network-transparent distribution of Oz computations. Multiple Oz sites can connect together and automatically behave like a single Oz computation, sharing variables, objects, classes, and procedures. Sites disconnect automatically when references between entities on different sites cease to exist.
- In a distributed environment Oz provides language security. That is, all language entities are created and passed explicitly. An application cannot forge references
nor access references that have not been explicitly given to it. The underlying representation of the language entities is inaccessible to the programmer. This is a consequence of having an abstract store and lexical scoping. Along with first-class procedures, these concepts are essential to implement a capability-based security policy, which is important in open distributed computing.

### Constraints in Oz

In Oz, constraints are divided into two groups (which is also the case for all finite domain constraint programming systems), basic and non-basic.

Basic constraints are logic formulae interpreted in a fixed first order structure. Restricting to only finite domain constraints, basic constraints thus have one of the following forms

- \( x \in D \) where \( D \) is a finite subset of positive integers
  
  (\( x \in D \) is also known as domain constraint)

- \( x = y \) where \( y \) is a variable

- \( x = n \) where \( n \) is a positive integer

Non-basic constraints are more expressive than basic constraints. Some examples of non-basic constraints are

- \( x + y = z \)
- \( x > 3 \)
- \( z - w = v^2 \)
In general, constraints that do not fall into any of the three forms of basic constraints are non-basic.

- Computational Spaces

Central to the constraint programming in Oz is the notion of computational space. Computational spaces host threads, constraints stores and propagators.

A thread is a stack of statements. It runs by reducing its topmost statement and synchronizes on them. If the topmost statement cannot be reduced the thread suspends. A thread that contains a choice as its topmost statement suspends. A unary choice has only one alternative while an $n$-ary choice has $n$ alternatives. For example, a binary choice statement looks like

```
choice $S_1$ [] $S_2$ end
```

$S_1$ and $S_2$ are the two alternatives of the binary choice statement. We restrict ourselves to binary choice points to simplify discussion.

A space also consists of propagators connected to constraint stores (see Figure 1.2). These constraint stores hold information about variables expressed by conjunction of basic constraints. Propagators impose non-basic constraints. A propagator is a concurrent computational agent that tries to amplify the store by constraint propagation.
Suppose a constraint store hosting a constraint $C$ and a propagator imposing a constraint $P$. The propagator can tell a basic constraint $B$ to the store, if $C \land P$ entails $B$ and $B$ adds new and consistent information to $C$. Thus telling $B$ updates the store to host $C \land B$.

A propagator imposing $P$ disappears as soon as $P$ is entailed by the store constraints. By this, it means that every variable assignment that satisfies the constraint store also satisfies the constraints impose by the propagator.

A propagator becomes failed is it detects that $P$ is inconsistent with the constraints hosted by the store.

A propagator is stable if it is either failed or its operational semantics cannot tell new information to the constraint store.

A space is stable, if no further constraint propagation in the space is possible, that is, all propagators in the space are stable and it contains no threads that can reduce further.
A space that contains a failed propagator is failed.

A stable space not containing a propagator is solved. If no threads are suspended in the space, the space is solved and entailed. Otherwise, it is solved but suspended.

If a space becomes stable and contains a thread that suspends on a unary choice, the unary choice is replaced by its (only) alternative. If it contains a thread suspended on a binary choice, the space is distributable. A distributable space is stable but is neither solved nor failed. When a space becomes distributable one thread containing a binary choice as its topmost statement is selected. A distributable space is allowed to commit to one alternative of the selected choice.

Distributing a distributable space $S$ with respect to an arbitrary chosen constraint $C$ (assuming $C$ is one alternative of the binary choice as determined by the search strategy) results in two spaces. One is obtained by adding $C$ to $S$ and the other is obtained by adding $\neg C$ to $S$.

Computational spaces provide a powerful abstraction for constraint inference methods. In the search tree, internal nodes can be represented by distributable spaces while leaves nodes are represented by failed or solved spaces.

- **Operations of Computational Spaces**

  Computational spaces are first class citizens of Oz.

  The operations of spaces are detailed as follows:
\[ S = \{\text{Space.new } P\} \]

\( S \) yields a reference to the newly created space and has a fresh variable known as the root variable. \( P \) is a unary procedure that usually represents the problem. A new thread is created in \( S \) that applies the procedure \( P \) to the root variable.

\[ A = \{\text{Space.ask } S\} \]

\( A \) gets bound when \( S \) becomes stable.

\( A \) is bound to failed if \( S \) fails, alternatives if \( S \) is distributable and succeeded if \( S \) is succeeded.

\[ A = \{\text{Space.askVerbose } S\} \]

This command returns the status \( S \) in verbose form. It reduces only when \( S \) becomes merged or stable.

\( A \) gets bound to merged if \( S \) is merged.

\( A \) gets bound to blocked(\( T \)) if \( S \) becomes blocked but not stable. \( T \) gets bound to the status of \( S \) when \( S \) becomes unblocked again.

If \( S \) is stable, \( A \) gets bound to failed if \( S \) fails, succeeded(suspended) if \( S \) is succeeded but still contains threads, succeeded(entailed) if \( S \) is succeeded but does not contain threads and alternatives if \( S \) is distributable.

\[ \{\text{Space.commit } S I\} \]

The choice point in \( S \) is replaced by its \( I \)-th alternative.

\[ C = \{\text{Space.clone } S\} \]

\( C \) is a new space that is a copy of \( S \)
Similar to space creation, Inject creates a new thread that applies \( P \), a unary procedure, to the root variable of \( S \).

Merge binds \( X \) to the root variable of \( S \) and discards \( S \). This operation basically allows the constraints of a space to be retrieved.
Chapter 2: Why is the toolkit needed?

Existing constraint programming systems offer a fixed set of inference engines for search. System’s designer can only implement new search engine at a low level. It is thus difficult to integrate other functionality by the user. For instance, to change a search engine using depth first search to breath-first search, the user will probably have to rewrite the entire search engine from scratch. The problem becomes non-trivial when the search engine needs to be integrated together with other related aspects like re-computation, optimization technique and visualization capability.

These different aspects or dimensions are in a considerable degree independent of one another so why should a user code everything from scratch if he is interested in only one particular aspect? This is where the toolkit will come in handy. The user should be able to do rapid prototyping with the toolkit to test out ideas and not have to spend additional time coding an entire engine. The user’s can then concentrate more on the aspect of interest instead of wasting time to reproduce a large part of something, which already exist in one form or another. The user should be able make use of sophisticated tools such as visualization of search trees while developing new engines, without having to worry about the internals of visualization. That is why the toolkit is needed.
Chapter 3: Aims of the project

The aim of this project is to design a toolkit for tree search with the following characteristics.

The toolkit must be modular with respect to the different independent dimensions of the search engine. Thus, the user can mixed and matched any combination of the different dimensions of a search engine with ease.

It must support easy integration by the user. Therefore the user do not have to create the search engine from scratch whenever they want to test something new (like a new optimization technique or a new search strategy) but could instead concentrate on the dimension of interest.

The toolkit must also allow a user to plug in any dimension that they might have identified which does not falls into any of the existing dimensions without difficulty.
Chapter 4: The Toolkit

4.1 Structure of the Toolkit

The Layered Approach

The toolkit is built in a layered fashion as shown in Figure 4.1.

At the lowest level is the language Oz, in which the whole toolkit is programmed. In particular, the notion of computational spaces is central to the toolkit.

The node is an abstraction of computational spaces. This enables a user to make use of the toolkit without knowing about computational spaces. Furthermore, the notion of nodes is closer to the idea of a search tree. However, the abstraction limits and concentrates the capabilities of spaces to the needs of tree search. (There is no cloning of nodes so saturation for example is not easily expressible in our toolkit.)

Associated with a search engine are several dimensions. Specific implementations of these dimensions are called modules.

A search engine is made up of the various (but not all) modules.
4.2 Nodes

Nodes implement the search tree. A node references to its parent node and its children’s node (if any). It encapsulates a space. Thus the functionality of computational spaces should also be provided through the nodes. As mentioned before, the capability of the spaces are concentrated to the needs of the search tree. An internal node holds a distributable space while the leave nodes encapsulate a failed or succeeded space.

Furthermore, the nodes are optimized in two ways by default. Firstly, searched nodes are discarded. A node is searched if it is a leaf node or when all its children nodes are searched. In this case, it is not necessary to clone a space for the last child. For binary trees, the number of cloning is reduced by half. Secondly, if searched nodes are not discarded then spaces of searched nodes may be discarded. Re-computation is performed to get back the discarded space if necessary. Both these optimizations may be “switched on” and “switched off”. This is necessary because they may not be applicable in all cases. For example, if there is a need to reference searched portion of the tree then those nodes cannot be discarded. Re-computation will not work if the search strategy is non-deterministic.

The above describes the basic nodes or base nodes. The various dimensional modules can enhance the nodes. This becomes more apparent when the organization of the search engine is discussed.
4.3 The Dimensions

A search engine can be broken down into several orthogonal dimensions that are independent of one another. We have identified these several dimensions – Re-computation, Links, Explorer, Tracer, Display Unit and Optimizer. Each dimension can have many different implementations referred to as modules. Every dimension has a module Null. When Null is used, it means that the dimension will not be a part of the search engine. Modules may be built on other modules that are not necessary from the same dimension. The rest of the modules are described below.

**Re-computation**

It can be memory intensive if every node stores a computational space, especially if the space contains a large number of variables and constraints. Re-computation avoids this problem by re-computing space on demand. This dimension has one predefined module (discounting Null) – standard re-computation.

- **Standard Re-computation**

  Rather than storing the space at every node in the search tree, store a space once in a while, trading space for time. The frequency of storing a space is defined by \( n \), an integer referred to as the maximum re-computation distance. This means that the maximum number of nodes to be searched before finding one that stores a computational space is at most \( n \). This module allows the maximum re-computational distance to be changed dynamically during search.
Links

This dimension is involved in storing the decision-making information at choice point during the distribution phase of constraint problem solving. It has only one predefined module, standard links.

- **Standard Links**

  At every choice point, the variable and value chosen for distribution are stored. It is essential that information be passed in from the distributor (distribution strategy) correctly. Otherwise, no useful information will be stored.

  In order to communicate with this module correctly, the distributor had to pass in the relevant information at every choice point.

Explorer

The explorer dimension describes how the search tree is explored. There are four predefined modules, namely, depth first search, depth first search with iterative deepening, breadth first search and limited discrepancy search.

- **Depth First Search**

  The children and descendants of a node are examined first before any of its siblings. Depth-first search goes deeper into the search tree whenever possible. When all descendants nodes are examined, the sibling nodes are then examined.

![Figure 4.2](image)
Hence, with respect to the search tree shown in Figure 4.2, the order of exploring the search tree would be 1-2-4-8-9-5-10-11-3-6-12-13-7-14-15.

- **Breadth First Search**
  The siblings nodes are examined first before the descendents of a node are examined. Thus, the tree is searched in a level by level fashion.

  With respect to the same search tree shown previously (Figure 4.2), the order of search would be 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15.

- **Depth First Search with Iterative Deepening**
  This is depth first search with depth bounds. The search performs like a normal depth first search until it hits the depth bounds. Then the search backtracks and continues in other branches of the tree, which are within the depth bounds. This causes a breadth like sweep of the search tree at that depth level. The depth bound is initially one. When all nodes are examined, the search restarts with a depth bound of two. This continues, increasing the depth bound at each iteration, during which a complete depth first search to the current depth bound is performed.

  With respect to the search tree shown in Figure 4.2, the order of exploring the search tree is 1-2-3 with depth bound of one, 1-2-4-5-3-6-7 with depth bound of two and 1-2-4-8-9-5-10-11-3-6-12-13-7-14-15 with depth bound of three.
• **Limited Discrepancy Search**

In this context, discrepancy refers to a decision made during exploration of the search tree against heuristics. The rational behind limited discrepancy search is as follows. At the root of the search tree, it is more likely for a heuristic to make a wrong suggestion since less information is available. Thus making discrepancies at the top preempts this mistake. If this does not lead to a solution, discrepancies are then made further down the tree.

In our case, discrepancy means searching the second alternative at a choice point first. The tree is probed iteratively with an increasing number of discrepancies staring with zero.

Applying limited discrepancy search to the same tree shown previously (Figure 4.2), the following series of sketches (in Figure 4.3) show how the tree is explored. Discrepancies are shown as blue arcs.
This dimension tracks the exploration of the search tree. It defines operations that traverse the search tree and provides references to the root node and the current node. It has only one predefined module, standard tracer.

- **Standard Tracer**

  The standard tracer provides functionality such as one step, find next solution and find all solution. In addition, it has a graphical interface and an embedded browser. This module will work properly only when an Explorer module and a Links module are used.
Display

This dimension allows visualization of the search tree. It has only two predefined modules namely, simple display and standard display

- Simple Display
  
  This display shows the search tree and allows the user to select a node and retrieve its information.

- Standard Display
  
  The standard display has the same capabilities of the simple display. In addition, it interacts with the tracer and thus will only work if a tracer module is used.

Optimizer

This dimension optimizes the tree search. There are two predefined modules, namely branch and bound optimization and restart optimization.

- Branch and Bound
  
  Branch and bound optimization remembers the best solution found so far and prune the search tree by eliminating partially searched sub-trees when the partial solution obtained is worse than the current best solution. Hence, each new solution found is better than the previous solution.

- Restart
  
  Restart optimization restarts the search from the root whenever a solution is found. Every time, the search restarts, the problem is constraint to yield a better solution than the current solution. A tracer module needs to be used for this module to function properly.
Chapter 5: Toolkit Usage

5.1 Levels of Usage

There are three levels of using the toolkit. They are termed as using the toolkit, developing a module and developing a dimension. The depth of understanding needed is different for each level of usage.
5.2 Using the Toolkit

The following sections describe how the toolkit can be used together with the various predefined modules. The depth of knowledge required at this level amounts to just knowing how to create the search engine and what functionality are available depending on the modules used.

**Define the problem**

Define the problem as a script appropriately just like any script in Oz.

**Load the toolkit**

Load the toolkit via

```oz
declare
[<Name>]={Module.apply [{Pickle.load <WebLink>}]}
```

<Name> refers to the functor name. All references to the toolkit have to be preceded by <Name>. For example, if the functor is named STK, then to use the EngineGenerator object, use STK.engineGenerator. Note that the first letter is changed to lower-case. All references in this report will not include the functor’s name.

<Website Link> is where the toolkit functor can be found. It is currently available at 'http://www.comp.nus.edu.sg/~henz/projects/SearchToolkit/searchtoolkit.ozf'

**EngineGenerator** is an object that is created that will generate the search engine.
Create the search engine

Create the search engine by making the following method call to

EngineGenerator.

\[
\text{makeEngine(<Options> <SearchEngine>)}
\]

This method returns a search engine in \(<\text{SearchEngine}>\) created according to the

options specified in \(<\text{Options}>\).

\(<\text{Options}>\) is of the form

\[
<\text{Dimension}_1> : <\text{ModuleName}_1> \ldots \\
<\text{Dimension}_i> : <\text{ModuleName}_i> \ldots \\
<\text{Dimension}_n> : <\text{ModuleName}_n>
\]

A particular dimension can be specified together with the module name in the form

shown above. If a dimension is not specified, the default \textbf{Null} will be used. As

mentioned in section 4.3, using \textbf{Null} means that the search engine will not have this

particular dimension.
<Dimension> and its associated <ModuleName> are shown in the table below.

<table>
<thead>
<tr>
<th>&lt;Dimension&gt;</th>
<th>&lt;ModuleName&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recomputation</td>
<td>StdRecom</td>
<td>Standard re-computation.</td>
</tr>
<tr>
<td>Links</td>
<td>StdLinks</td>
<td>Standard links.</td>
</tr>
<tr>
<td>Explorer</td>
<td>DepthFirst</td>
<td>Depth-first search.</td>
</tr>
<tr>
<td></td>
<td>BreathFirst</td>
<td>Breadth-first search.</td>
</tr>
<tr>
<td></td>
<td>IterativeDeepening</td>
<td>Depth-first search with iterative deepening.</td>
</tr>
<tr>
<td></td>
<td>LimitedDiscrepency</td>
<td>Limited discrepancy search.</td>
</tr>
<tr>
<td>Tracer</td>
<td>StdTracer</td>
<td>Standard tracer.</td>
</tr>
<tr>
<td>Display</td>
<td>SimpleDisplay</td>
<td>Simple display.</td>
</tr>
<tr>
<td></td>
<td>StdDisplay</td>
<td>Standard display.</td>
</tr>
<tr>
<td>Optimizer</td>
<td>BranchBound</td>
<td>Employ branch and bound optimization during tree search</td>
</tr>
<tr>
<td></td>
<td>Restart</td>
<td>Employ restart optimization during tree exploration</td>
</tr>
</tbody>
</table>

**Using the search engine**

Depending on the modules used to create the search engine, the search engine will have different functionality. However, since the search tree is the invariant in the engine regardless of the modules used, functionality of nodes will always be available. The functionality of the nodes will be discussed first followed by the modules in each dimension.
Nodes

The following are public methods of nodes.

\[ \text{init}(<\text{Engine}>) \]

Initializes the node. Called during the instantiation of the node object. Note that the method takes in the engine object as a parameter.

\[ \text{createRoot}(<\text{Engine}>, <\text{Script}>) \]

Creates the root node of the search tree. <Script> represents the problem to be solved. <Engine> represents the engine object. This method is called during the instantiation of the root node object.

\[ \text{getType} \]

Returns the type of nodes. Nodes can be of type success(entailed), success(suspended), choices or failure.

\[ \text{getConstraints} \]

Returns the constraints of the space held by the node.

\[ \text{addConstraints}(<\text{Script}>) \]

Add constraints to the space held by the node.

\[ \text{getSearched} \]
Toolkit Usage

Check if node had been searched. If a node had been searched (return value is true), its descendents nodes will also be searched. Otherwise (return value is false), at least one of its descendents node is still not explored.

getSpace

Returns the space held at the node.

setSpace(<Space>)

Set the space held at the node to be <Space>. <Space> can be nil, which means that the node is not holding a space.

createChild(<Which>)

Creates a child node. <Which> can be either left or right. The method returns the newly created child node.

getChild(<Which>)

Returns the correct child node depending on <Which>. The method call returns nil when the child node does not exist.

getMom

Returns the node’s parent node. This method returns nil if the parent node does not exist.

setMom(<Node>)
Set the parent node to `<Node>`. `<Node>` can be `nil` which means that the node does not have a parent node or it loses its reference to its parent node.

    getRecompute

Returns a boolean value indicating the need to re-compute the space at the node. If it is `true`, then the node is not holding a space and re-computation is necessary. On the other hand, if the value returned is `false`, the node is space carrying and no re-computation is necessary.

    setRecompute(<Bool>)

Sets the re-compute value to the boolean value `<Bool>`.

**Search Engine**

Similar to the nodes, this method is always available regardless of the modules used.

    reset

This method resets the engine.

If a method is passed to the search engine but it is not available, an error dialog would appear (Figure 5.1).

Figure 5.1
Recomputation: StdRecom

Public method of this module includes

\[ \text{setRecomDist(<integer>)} \]

This method sets the maximum re-computation distance, \( n \), for the search engine, passed in via the single parameter \(<integer>\). The default value of \( n \) is 0.

Links module: StdLinks

Public methods for this module are

\[ \text{passLinkInfo(<Link Information>)} \]

Called by the distribution strategy to pass the link info into the search engine. Insert the following command at every choice point in the distribution strategy.

\{self.engine passLinkInfo(<Link Information>)}

\(<Link Information>\) refers to the information pass to the nodes.

\[ \text{getLinkValue} \]

Returns the node’s link value. This is actually an enhancement to the basic nodes. Thus the nodes, when this module is used have this additional method.

Note that the search strategy used in the problem script must communicate with this module. A predefined search strategy had been included that is just an enhancement of the search strategy defined in Oz \( \text{FD.distribute} \). All options available in \( \text{FD.distribute} \) can be used in this predefined search strategy.

\{<SearchEngine>.distribute ...\}
This search strategy is only available when this module is used. To include a user-defined search strategy, simply replace the strategy in the file `distribute.oz` and recompile the engine.

**Explorer modules: DepthFirst, BreadthFirst, IterativeDeepening and LimitedDiscrepency**

All the modules have this public method.

```plaintext
oneStep
```

This method continues the search and returns either `nil` when the search is exhausted or the newly created node.

In addition, IterativeDeepening has these other methods.

```plaintext
setMaxDepth(<Depth>)
```

This method sets the maximum depth bound for the search. `<Depth>` is an integer specifying the maximum depth bound of the search. It can be `-1` (the default) which means that there is no maximum depth bound.

```plaintext
setStep(<Step>)
```

This method sets the increment size of the depth bound. `<Step>` is an integer specifying the step size. The default value is `1`. 
**Tracer module: StdTracer**

As noted previously, this module will work properly only when an Explorer module and a Links module are used. The public methods of this module are described below.

\[
\text{solve}(\text{<Script>})
\]

\text{<Script>} refers to the problem. The tracer needs to be initialized with this method call. A window (Figure 5.2) will appear. Most functions of the tracer can be invoked by pressing the buttons on the left side of the window.

![Figure 5.2](image)

Pressing this button will cause the search engine to distribute once resulting in the creation of at most one new node. If the search is completed, nothing will happen. This performs the same action as the method \text{step}.

Pressing this button will cause the search engine to find the next solution. The engine will stop when a solution node is found or when the search is exhausted with no more new solution. This performs the same action as the method \text{next}. 

35
The engine will explore the entire tree and find all solutions when this button is pressed. If the tree is partially searched, the rest of the tree will be explored. The same action can be invoked by the following method:

```
all
```

The current node’s information is shown when this button is pressed. The same action can be invoked by the following method:

```
displayCurrNodeInfo
```

The information, by default, will be displayed in the embedded browser as shown in Figure 5.3.

```
<script>
...define a procedure that takes in the current node’s information and display it as required.
```

The current node’s link information will be displayed in the embedded browser. The link information refers to the variable selected and the value...
of the variable chosen during distribution. The same information could be displayed by the following method.

\[ \text{displayCurrentLinkInfo} \]

The current node mentioned above is always the newly created node. When the `StdDisplay` module is used, the current node is the node with shadow. See Figure 5.4 for clarity. Note that the same nodes shown in Figure 5.4 when `SimpleDisplay` is used need not be the current node since the module is independent of the `Tracer` module.

![Figure 5.4](image)

When this button is pressed, the engine is reset.

In addition to the functionality that can be invoked through graphical interface, there are a few other methods, which are public that do not have a graphical interface counterpart.

\[ \text{getRootNode} \]

This method returns the root node.

\[ \text{setCurrNode(<Node>)} \]

This method sets the root node to `<Node>`.

\[ \text{getCurrNode} \]

This method returns the current node.
**Display: SimpleDisplay**

When this module is used, a window shown in Figure 5.5 will appear when the search engine is created.

![Figure 5.5](image)

This module has only one public method.

\[
\text{drawTree}(\text{<Root Node>})
\]

This method draws the tree in the display.

The node with the shadow is the selected node that is usually the newly created node.

Note that there are two different types of success nodes. (See Figure 5.6)

To select a node, click left mouse button within the node. To retrieve a node’s information, double-click left mouse button within the node. The information will be displayed in a browser and it will become the selected node.
Display: StdDisplay

As mentioned before, this module needs to be used with a Tracer module for it to function properly. This module has all the capability of the SimpleDisplay module and similarly has no public methods. In addition, the tracer will recognize the selected node as the current node. Double-clicking within a node with the left mouse button will select the node and show its information. Double-clicking with the right mouse button within a node will also select the node and its link information will be shown.

![Image of Display: StdDisplay](image-url)

**Figure 5.6**
Optimizer: BranchBound and Restart

Public methods of these modules are

```
s COSTPROC(<Script>)
```

This method set the cost procedure that specifies how the next solution is better than the current solution with respect to a cost. `<Script>` specifies the cost. If the cost procedure is not specified, the search behaves as if it is not optimized.

```
getSolution
```

This method returns the best solution found so far.

```
getAllSolutions
```

This method returns all the solutions found so far in a list. Note that the later solutions found are better than the earlier solutions due to the mature of the search.

Note that Restart needs to be used in conjunction with a Tracer module.

Example

The following is a complete example to show how the toolkit may be used.

**Problem:** Find solutions for *send most money* problem, a variation of *send more money* problem (See chapter 1.2 for description), with a search engine using limited discrepancy search and branch and bound optimization. The engine has display capability and provides link information.
The problem script is defined as

```tcl
proc {Money Root}
    :
    <Problem definition>
    :
    {SearchEngine.distribute naive Root}
end
```

Load the toolkit via

```
[STK]={Module.apply
        SearchToolkit/
        searchtoolkit.ozf'}]}
```

Create the search engine by

```
SearchEngine= {EngineGenerator
    makeEngine(links: StdLinks
        explorer: LDS
        tracer: StdTracer
        display: StdDisplay
        optimizer: BranchBound
    )}
```

The StdTracer interface window (Figure 5.2) and the StdDisplay window (Figure 5.5) would appear.
Initialized the engine with the problem via

\{\text{SearchEngine solve(Money)}\}

A choice node should appear in the \text{StdDisplay} window.

Since \text{BranchBound} optimization is used, a cost procedure should be defined.

Assume the cost procedure is \text{More}. Let the engine know via

\{\text{SearchEngine setCostProc(More)}\}

At this stage, the search engine is properly initialized and can be used. Figure 5.6 shows a snapshot of the engine in used. The browser shows the current node’s information and links information.

![Standard Display](image1)

![Standard Tracer](image2)

Figure 5.6
5.3 Organization of the Search Engine

Before working at the other two levels, the user needs to know how the modules (of the various dimensions) are organized within the search engine.

As shown in Figure 5.7, the search engine is actually an instance of a class that inherits from the various classes each representing a particular module of a specific dimension. Parallel to the search engine class hierarchy is the node class hierarchy. Each class has an associated feature, which may hold a node class. The user works within the blue region when developing a new module in an existing dimension or when developing a new dimension.
5.4 Developing a Module

A module is a specific implementation of a dimension. Modules generally fall into two categories.

First Category

A module may be independent of the other dimensional-modules and works fine when it is used alone or in conjunction with other dimensional-modules. Examples of such pre-defined modules are StdRecomp, StdLinks, DepthFirst, BreadthFirst, IterativeDeepening, LimitedDiscrepency, SimpleDisplay and BranchBound. These modules fall into the first category.

A module under development that falls into the first category does not need to interact with the other modules therefore the developer need not worry about the other modules. But to make it compatible to other modules that may make use of it, there is a set of guidelines to follow.

Second Category

Modules can also be built upon other dimensional modules. For example, the pre-defined module StdTracer is built on an Explorer module and a Links module. In other words, for StdTracer to work perfectly, an Explorer module and a Links module needs to be used. Similarly, Restart and StdDisplay requires a Tracer module to be used. These fall into the second category.
Dependencies of predefined modules

Figure 5.8 shows the dependency graph of the predefined modules. Technically, Nodes is not a module but since the modules all make use of it (directly or indirectly), it is included in the diagram. Modules at the lower level are dependent upon a module at a higher level if a line joins them.

The main mode of communication between modules of the different dimensions is through public methods and the overriding of the super class method. Since a module from a dimension inherits from a super class that is another module from another dimension, the location of the class in the class hierarchy determines how much the developer needs to know before constructing a module. This represents the maximum
amount of knowledge that the developer needs to know. Similarly, some guidelines need to be followed to make it compatible to other modules that might make use of it.

**Skeleton Code**

The module is not simply a class definition. Instead it is a function with a single parameter `Super` – the super class in the class hierarchy – and it returns a class that inherits from this super class. This is required since the specific super class is not known when the module is written.

Thus the skeleton code has the following structure.

```plaintext
fun {<Module Name> Super}
    class $ from Super
        feat
            node: Super.node
                :
        end
    end
end
```

Notice that the class has a feature, node, which is simply the super class node feature (shaded region). This is the case for a module that does not enhance the nodes that form the search tree. By nature of inheritance, the shaded region could of course be left out. However, if the module need to enhance the nodes, simply create a new node class that inherits from the super class’s node and has the required enhancements. Then initialized the feature node to this node class (shaded region).
The skeleton code would look like

```plaintext
fun {<Module Name> Super}
    class <Node Class Name> from Super.node
    feat
        discardSearchedNode? : <Boolean>
        discardSearchedSpace?: <Boolean>
    :
    end
in
    class $ from Super
        feat
            node: <Node Class Name>
        :
    end
end
```

As mentioned before, the (basic) nodes has two built-in optimizations that can be
“switched on” and “switched off”. To do that, simply add the correct feature in the
new node class and initialized to the required boolean value. Both are “switched on”
by default.

To “switch off” discarding of searched nodes, add

```plaintext
discardSearchedNode?: false
```

To “switch off” discarding of searched space (that is re-computation), add

```plaintext
discardSearchedSpace?: false
```
A module may also be built on an existing module in the same dimension. The skeleton code is as follows

```fortran
fun <New Module> Super
    class $ from <Existing Module> Super
        ...
    end
end
```

**Guidelines**

To develop a module in an existing dimension, the developer needs to know the general entry points of the modules in each dimension. For some dimension, there may be some additional guidelines that need to be adhered. Two entry points common to all modules and are the initialization method, `init` and the reset method, `reset`.

The `init` method has to call the initialization of its super class. The skeleton code is as follows.

```fortran
meth init
    : <own initialisation>
    :
    Super,init
    : <own initialisation>
    :
end
```
Due to the nature of inheritance, this method could be left out if it does nothing else other than just call the initialization of its super class. The same goes for reset method whose skeleton code is shown below.

```ruby
meth reset
  :
  <own reset>
  :
  Super, reset
  :
  <own reset>
  :
end
```

The entry points and guidelines of each dimension are described below. Entry points are normally some specific methods.

- **Recomputation**

  Modules in this dimension do not have any other specific entry points or guidelines.

- **Links**

  Modules in this dimension have two additional entry points.

  `passLinkInfo(<Info>)`: This method is called by the distribution strategy to pass `<Info>` into the node.
The nodes should also be enhanced to provide this method

GetLinkInfo: It returns the link information of the node.

- **Explorer**

 Modules in this dimension have this entry point.

oneStep: This method continues the search and returns either a newly created node or nil when the search is exhausted.

- **Tracer**

 Modules in this dimension need to maintain a reference to the current node and the root node. Generally, modules in this dimension do not need to have specific entry points. However, to make sure that the StdDisplay and Restart modules can use any Tracer modules, some additional entry points are needed.

For StdDisplay, the following methods are needed.

step: performs one step of the search

getCurrNode: returns the current node

setCurrNode(<Node>): set the current node as <Node>

getRootNode: returns the root node

displayCurrNodeInfo: display the current node’s information

displayCurrLinkInfo: display the current node’s link information
For Restart, the following methods are needed

- \texttt{step}: performs one step of the search
- \texttt{getCurrNode}: returns the current node
- \texttt{getRootNode}: returns the root node

### Display

Modules in this dimension have only one entry point.

- \texttt{drawTree(<Root Node>): draws the tree in the display}

### Optimization

Modules in this dimension do not have any other specific entry points or guidelines.

### Example

For illustrating purposes, assume that the \texttt{Explorer} module \texttt{IterativeDeepening} is not defined and we want to implement the module.

From Figure 5.7, we see that \texttt{Explorer} is below \texttt{Links} and \texttt{Recomputation} in the class structure. However, since \texttt{IterativeDeepening} is not going to be dependent upon any modules in the two dimensions, they can be ignored.

There are two ways we can build this module. Since \texttt{IterativeDeepening} is essentially depth-first search, it is possible to build on \texttt{DepthFirst}. Alternatively, the module can be written from scratch. The first method is preferable since there is very likely to be code duplication if the second method is used. We shall use the first method. Thus the code would look like
fun {IterativeDeepening Super}
    class $ from {DepthFirst Super}
    :
end
end

Following the guidelines, Explorer modules have one entry point, oneStep. Therefore, this method must be present. Note that within this method, oneStep of DepthFirst will be called.

fun {IterativeDeepening Super}
    InheritedClass={DepthFirst Super}
in
    class $ from InheritedClass
    :
        meth oneStep($)
            :
                Node=InheritedClass, oneStep($)
            :
end
end
end

If there are no other public methods (private methods are not shown here), and if the nodes need not be enhanced, the module is basically done. In this case, however, the
nodes are enhanced and there are two other public methods, `setMaxDepth` and `setStep`. Thus the code would look like

```plaintext
fun {IterativeDeepening Super}
    InheritedClass={DepthFirst Super}
    class ItrDeepNodes from InheritedClass.node
        :
    end
in
    class $ from InheritedClass
        feat
            node: ItrDeepNode
                :
            meth setMaxDepth ...... end
            meth setStep ...... end
            meth oneStep($)
                :
                Node=InheritedClass, oneStep($)
                :
            end
        end
    end
end
```
5.5 Developing a Dimension

To develop a dimension, the user needs to decide where the new dimension should be in the class hierarchy. Then add the dimension via the following method calls.

```
insertDimension(<new dimension>)
```

This will insert the dimension at the top of the hierarchy (see Figure 5.7) between Base and Recomputation.

```
insertDimension(<new dimension>
    before:<existing dimension>)
```

This will insert <new dimension> so that it becomes the direct super class of <existing dimension>.

```
insertDimension(<new dimension>
    after:<existing dimension>)
```

This will insert <new dimension> so that its direct super class is <existing dimension>.

If both before and after are specified, then the dimension specified for before has priority. Note that if <existing dimension> does not exist, it will be ignored. Also, if <new dimension> is already an existing dimension, it’ll also be ignored.
The module of the new dimension can be developed as per normal and the new dimension can be added as a parameter in the method call `makeEngine to EngineGenerator` in the same way as existing dimensions.

**Example**

Assume we want to implement a new dimension `InformationAction`. This dimension would include modules that work on information of the search engine. The current implementation is included in the `Tracer` dimension.

Firstly, consider where this dimension should be situated in the class hierarchy shown in Figure 5.7. It would have to be below `Links` since `Links` dimension provides one type of information of the search engine. The question now becomes where would be a logical place to insert the dimension? There is no definite answer to this question. This dimension could be placed directly after `Links` or lower down in the hierarchy.

Since it is unlikely that a module in `InformationAction` will need to override a method call in any of the dimensions after `Links`, it is a logical choice to insert the dimension directly after `Links`.

Thus, modules of the dimension could be created as described in the previous chapter and the dimension is added via

```
{Engine insertDimension(InformationAction after:Links)}
```
or

```
{Engine insertDimension(InformationAction
    before:Explorer)}
```
Chapter 6: Performance of the Toolkit

The tests are performed on a PC running at 133MHz with 64Mb of ram. The performance of the toolkit is tested against Oz in built SearchOne, SearchAll and Oz Explorer.

Only the StdTracer module is used when the performance is compared to SearchOne and SearchAll. StdTracer and StdDisplay modules are used when compared to the Oz Explorer. In both cases, the DFS module is used.

The problems used in the performance tests are n-Queens problem and Magic Square. The n-Queens problem used is not optimized in anyway and uses the naive search strategy. However, Magic Square is optimized. The scripts of the two problems are included in Appendix A.

All values tabulated are the average of five measurements.
6.1 Space Performance of the toolkit

From the values gathered and shown in the above two charts, the space utilization of an engine created by the toolkit is comparable to the Oz built-in \texttt{SearchOne} and \texttt{SearchAll} even though the toolkit uses slightly more space in all cases.
It is obvious that the space utilization of the Oz Explorer is vastly superior to the search engine created by the modules StdTracer and StdDisplay. The reason is likely that the algorithm used in the module StdDisplay is not efficient.
6.2 Time Performance of the toolkit

**Graph 1:**
- **n-Queen size** vs **Time in seconds (log)**
- **Std_Tracer (All)**: 1.0, 20.0, 423.0
- **SearchAll**: 1.0, 17.0, 373.0
- **Std_Tracer (Next)**: - , - , -
- **SearchOne**: - , - , -

**Graph 2:**
- **Magic Square Size** vs **Time in seconds**
- **Std_Tracer (All)**: 100.0
- **SearchAll**: 96.0
- **Std_Tracer (Next)**: 1.0, 323.0
- **SearchOne**: 1.0, 298.0
Engine | Timing for 8-Queens
---|---
ExploreAll | 2.31s
StdTracer + StdDisplay | 35s

The results of the performance time is similar to that of space utilization; the search engine created by the toolkit is comparable to the built-in SearchOne and SearchAll but is unfavorable when compared to Oz Explorer.

In general, some inefficiency is expected from the search engine created by the toolkit since certain optimization will inevitably be loss by breaking the search engine into modular parts. However, the gain is in flexibility. That said, the performance of the StdDisplay module is undesirable and below expectation.
Chapter 7: Possible Improvements

Currently, the engine only supports search trees that are binary tree. Hence
distribution strategy that splits the domain of the selected variable into more than two
sub-domains is not supported. Therefore, a logical improvement is to extent the
capability of the toolkit to support distribution strategy that results in non-binary trees.

To change a particular module in the search engine such as changing the explorer
from DFS to BFS, another engine had to be created. One possible enhancement is to
allow the modules to be changed without the need to create another engine.

The space and time performance of the toolkit with respect to the graphical display is
not good. This should be improved.
Chapter 8: Comparison and Conclusion

Search Engines in Oz vs ToolKit Engine

The basic search engines in Oz can do one, all and best solution search. The general-purpose search engines in Oz support re-computation, the possibility to stop their execution and various kinds of output. In these engines, bounded depth first search, iterative deepening depth first search, branch and bound best solution search or restart best solution search can be performed. The search object in Oz implements a demand driven search engine which supports re-computation, single, all, and best solution search and different kinds of output in the same way as the general-purpose search engines.

Most of the functions provided by the various Oz search engines and object are available in the toolkit engine. The toolkit engine created with StdTracer can do one and all solution search. Using the StdTracer module also allows demand driven search. Together with optimization modules like BranchBound, the toolkit engine can do the same best solution search. IterativeDeepening does bounded search in addition to iterative deepening depth first search. BranchBound and Restart does the required optimization of the tree search.

However, the execution of the toolkit engine cannot be stop before it completes and it only returns the solutions in a list.

On the other hand, the toolkit engine can combine various modules together to create an engine that has functionality of two or more Oz search engine. For example,
IterativeDeepening together with BranchBound creates a toolkit engine that
does iterative deepening depth first search with branch and bound optimization. The
search engines of Oz cannot do that. Furthermore, the engine cannot do step-wise
exploration while the toolkit engine allows that with the use of StdTracer.
Visualization is also not possible in these Oz engines. By using one of the display
modules, visualization is possible in the toolkit engine.
Oz Explorer vs ToolKit Engine

In the Oz Explorer, the search tree can be explored in a user-guided manner. Starting from any node in the search tree further parts can be explored. Explored parts of the search tree are drawn as they are explored. Entire sub-trees as well as single nodes can be explored. Nodes in the tree representing choices and solutions carry as information their corresponding computation spaces. The Explorer gives first-class access to them by predefined or user-defined procedures. Nodes in the search tree can be selected directly or by convenient short cuts. Exploration and/or drawing can be stopped and resumed at any time. Statistical information is available in a status bar during exploration. The display of large search trees can be kept economic by scaling the tree and by hiding sub-trees. Support for automatically hiding complete sub-trees not containing solutions is provided. Hidden sub-trees can be unhidden on demand. The Explorer can run scripts with a large number of constraints and propagators and with large search trees efficiently with respect to both space and time. User-configurable re-computation can be employed to trade space for time for scripts which would otherwise require too much memory. The search tree of a script can be dumped in postscript format.

The toolkit also provide a limited user-guided search in that it allows the user to do step-wise exploration if StdTracer module is used. Explored part of the tree is also drawn as they are explored if display modules are used. Nodes in the toolkit engine can also be selected directly, carry solutions and allow access to them via predefined or user-defined procedures. Re-computation are provided if StdRecom is used. The nodes in the toolkit engine has in-built information re-computation.
However, the exploration and drawing of the tree cannot be stopped and resumed at any time and no statistical information is available. The toolkit engine provides does not provide capability for hiding failed sub-tree. The search tree cannot be dumped in postscript format. The performance of the Oz Explorer is also superior to the toolkit engine.

On the other hand, all nodes in the trees carry links information if Links is used. This is not available in the Oz Explorer. In addition, the Oz Explorer is hard-wired in depth-first search. Therefore to produce a full-fledged graphical traceable engine for limited discrepancy search with re-computation, hundreds of lines of code (and hours to understand the Oz Explorer) is needed to extend it even if the code for all components is available. The toolkit needs just this line

```
{EngineGenerator createEngine(
    explorer: LimitedDiscrepency
    recompute: StdRecom
    tracer: StdTracer
    display: StdDisplay)}
```
Conclusion

In conclusion, the toolkit created satisfied the initial goals of the project.

The toolkit is modular with respect to the different independent dimensions of the search engine. The user can mixed and matched any combination of the different dimensions of a search engine with ease.

It supports easy integration by the user. Therefore the user do not have to create the search engine from scratch whenever they want to test something new.

It allows a user to plug in any dimension that they might have identified which does not falls into any of the existing dimensions without difficulty.

However, the space and time utilization of the graphical display is below expectation and should be improved.
References


References


Appendix A: Scripts

**n-Queens script**
declare
fun {Queens N}
proc {$ Row}
    L1N = {MakeTuple c N}
    LM1N = {MakeTuple c N}
in
    {FD.tuple queens N 1#N Row}
    {For 1 N 1 proc {$ I}
        L1N.I=I LM1N.I=~I
    end}
    {FD.distinct Row}
    {FD.distinctOffset Row LM1N}
    {FD.distinctOffset Row L1N}
    {FD.distribute naive Row}
end
end

**Magic Square Script**
fun {MagicSquare N}
    NN = N*N
    L1N = {List.number 1 N 1} % [1 2 3 ... N]
in
    proc {$ Square}
        fun {Field I J}
            Square.((I-1)*N + J)
        end
        proc {Assert F}
            % {F 1} + {F 2} + ... + {F N} =: Sum
            {FD.sum {Map L1N F} '=:' Sum}
        end
        Sum = {FD.decl}
in
        {FD.tuple square NN 1#NN Square}
        {FD.distinct Square}
        %%% Diagonals
        {Assert fun {$ I} {Field I I} end}
        {Assert fun {$ I} {Field I N+1-I} end}
        %%% Columns
        {For 1 N 1
            proc {$ I} {Assert fun {$ J} {Field I J} end} end
        end
        %%% Rows
        {For 1 N 1
            proc {$ J} {Assert fun {$ I} {Field I J} end} end
        end
        %%% Eliminate symmetries
        {Field 1 1} <: {Field N N}
        {Field N 1} <: {Field 1 N}
        {Field 1 1} <: {Field N 1}
        %%% Redundant: sum of all fields = (number rows) * Sum
        NN*(NN+1) div 2 =: N*Sum
        %%%
        {FD.distribute split Square}
end
end