Honours Year Project Report

Component-based Visualization Of Tree Search

By

Gao Chaowei

Department of Computer Science

School of Computing

National University of Singapore

2002/2003

Honours Year Project Report

Component-based Visualization Of Tree Search

By

Gao Chaowei

Department of Computer Science

School of Computing

National University of Singapore

2002/2003

Project No: H41040
Advisor: Dr Martin Henz

Deliverables:
    Report: 1 Volume

Abstract

Information-rich search tree visualization is very useful in constraint programming. The entire process of solving a constraint problem is captured by the search tree. We will develop a component based visual tool to visualize the search trees in Figaro (a finite domain constraints solver) independent of search strategies. Users can interact with the visual tool to achieve various objectives. The report will describe how to draw trees that suit aesthetic rules, the design criteria and the implementation details of the visual tool.

Subject Descriptors:

D.2.2	User interfaces
G.2.2	Graph Theory
I .2.8	Problem Solving, Control Methods, and Search
F.4.1	Logic and constraint programming

Keywords:

Constraint programming, search tree, tree drawing, visualization

Implementation Software and Hardware:

Software: RedHat7.2 GNU/Linux (kernel 2.4.9-34), gcc (GCC) 3.0.4, Figaro-1.1.0, X-win32 5.4

Hardware: Intel Pentium 4 CPU 1500 MHz, 899644K main memory

Acknowledgement

When doing a project, I often encounter problems or difficulties like other people. Fortunately, I achieved the objectives with the help from many people. Due to the limitation of words, I do not mention all of them. Anyway, thanks them!

I am deeply indebted to my wonderful supervisor, Dr. Martin Henz for his guidance throughout my honours year study and the enlightments he gave me in the discussions. He also provided valuable advices and ensured that I was on the right track. I am pride of him for his kindness and responsibility.

I want to thank Mr. Yi Junkai, the maintainer of the Figaro system. He helped me a lot in understanding Figaro system.

List of Figures

List of Tables

Table of Contents

Chapter 1: Introduction

1.1 Constraint Programming

In the last two decades, constraint programming, especially finite domain constraint programming, has become popular in many application areas. A problem can be solved using constraint programming as long as it can be modeled as a constraint satisfaction problem. Now let us know about constraint programming by introducing important terms.

1.1.1 Constraints

The central notion in constraint programming is that of a constraint. Informally, a constraint on a sequence of variables is a relation on their domains (Krzysztof 2002). It can be viewed as a requirement imposed on the variables as it explains which combinations of values from the variable domains are admitted.

For example, $A < B, A, B \in [1...3]$ is a constraint. It stipulates a relation that must hold between any values chosen to replace the variables A and B, namely the chosen value of A must be smaller than the chosen value of B. A substitution of variables by values from their domains is a solution of the constraint if the substitution yields truth. For example, the substitution $\{A/1, B/2\}$ is a solution to the constraint since $1 < 2$ yields truth while the substitution $\{A/2, B/1\}$ is not a solution to the constraint since $2 < 1$ yields failure.

1.1.2 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) consists of a finite set of constraints. Modeling a problem as a CSP is very the first step to solve the problem using constraint programming. A substitution of all the variables by values in their domains is called a solution of the CSP if the substitution offers the solutions to all constraints. Two CSP's are equivalent regarding to a set of variables if they have the same solutions for these variables.

Example: SEND MORE MONEY. In the problem under consideration we are asked to replace each letter by a different digit so that the sum

```
   SEND
 + MORE
  MONEY
```

is correct.

CSP representation:

Variables and Domains: $S, M \in [1 \cdots 9], E, N, D, O, R, Y \in [0 \cdots 9]$.

Constraints: An equality constraint

$$\begin{aligned} & 1000\times S + 100\times E + 10\times N + D \\ + \ & 1000\times M + 100\times O + 10\times R + E \\ = \ & 10000\times M + 1000\times O + 100\times N + 10\times E + Y \end{aligned}$$

and 28 disequality constraints x ≠ y for x, y$\in \{S,E,N,D,M,O,R,Y\}$ with x preceding y in the presented order.

The unique solution of the CSP is 9567 + 1085 = 10652. It corresponds to the substitution$\{S/9, E/5, N/6, D/7, M/1, O/0, R/8, Y/2\}$.

Example: SIMPLE ENEQUALITY.

Variables and Domains: $X, Y, Z \in [1 \cdots 3]$.

Constraints: $X < Y, Y < Z$.

We express a CSP in form of $\langle Constra\,\mathrm{int}\,s\,; Variables\ and\ domains\rangle$. Thus the examples SIMPLE EQUALITY can be represent as

$$\langle X < Y, Y < Z \ ; \ X \in [1 \cdots 4], Y \in [1 \cdots 4], Z \in [1 \cdots 4]\rangle$$

Once we formalized a problem as a CSP, we can solve it using either domain specific methods or general method. The domain specific methods are usually provided in form of specific purpose algorithms. In turn, the general methods are concerned with the ways of reducing the searching space (constraint propagation) and with specific search methods. And this is how a CSP is solved using constraint programming.

1.1.3 Constraint Propagation

Constraint propagation replaces a given CSP by a “simpler” one, yet equivalent (Krzysztof 2002). The idea is that such a replace, if efficient, is profitable, since the search space becomes smaller. Typically, “simpler” means that the domains and/or constraints become smaller. The constraint propagation is performed by repeatedly reducing domains and/or reducing constraints using some rules while maintaining equivalence.

Now consider the CSP SIMPLE ENEQUALITY. Suppose we have the constraint propagation rule LINEAR INTEGER INEQUALITY 1:

$$\frac{\left\langle x < y \,;\, x \in [l_x \cdots h_x],\ y \in [l_y \cdots h_y] \right\rangle}{\left\langle x < y \,;\, x \in [l_x \cdots \min(h_x, h_y - 1)],\ y \in [\max(l_y, l_x + 1) \cdots h_y] \right\rangle}$$

Repeatedly applying this rule, we can finally transform the CSP SIMPLE ENEQUALITY as follows:

$$\left\langle X < Y, Y < Z \,;\, X \in [1 \cdots 4], Y \in [1 \cdots 4], Z \in [1 \cdots 4] \right\rangle$$

$$\overset{x<y}{\Rightarrow} \quad \left\langle X < Y, Y < Z \,;\, X \in [1 \cdots 3], Y \in [2 \cdots 4], Z \in [1 \cdots 4]\} \right\rangle$$

$$\overset{y<z}{\Rightarrow} \quad \left\langle X < Y, Y < Z \,;\, X \in [1 \cdots 3], Y \in [2 \cdots 3], Z \in [3 \cdots 4]\} \right\rangle$$

$$\overset{x<y}{\Rightarrow} \quad \left\langle X < Y, Y < Z \,;\, X \in [1 \cdots 2], Y \in [2 \cdots 3], Z \in [3 \cdots 4]\} \right\rangle.$$

So we reduced all three variable domains.

1.1.4 Search and Search Trees

Search and search trees are the central notions of the project so we must know them in the beginning. In general, constraint propagation gives us a simpler CSP but the resulting CSP is not solved yet. SIMPLE ENEQUALITY is such an example. In such a situation, a progress can be achieved only by splitting the current CSP *P* into two or more CSP’s the union of which is equivalent to *P*. So the general pattern consists of an alternating use of constraint propagation and splitting. This leads to what we call search trees. Conceptually, it is helpful to have in mind the following slogan:

Search Algorithm = Search Trees + Traversal Algorithm.

We explain this part using the example SIMPLE ENEQUALITY. Figure 1.1 shows the complete search tree for SIMPLE ENEQUALITY. Usually we combine enumeration and constraint propagation as one step so that the tree becomes smaller.

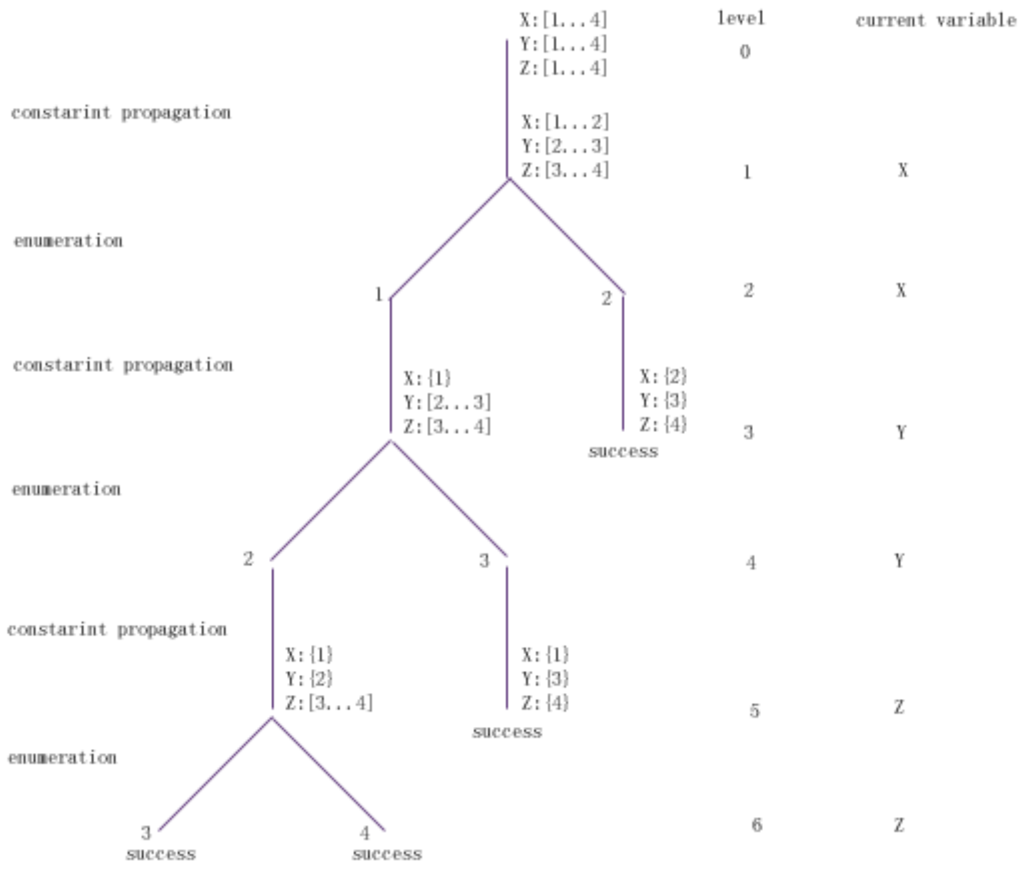


Figure 1.1: Complete search tree for SIMPLE ENEQUALITY

1.2 Figaro

Figaro is developed to provide researchers and software practitioners with a library for solving discrete constraint satisfaction and optimization problems using tree search and local search in a unified software architecture (Martin, Tobias and Ka Boon, 1999).

The implementation of problem solving algorithms is a challenging task. Sources of complexity are the problem models to be used, the need for heuristics and the need for experimentation and performance tuning. Another major source of complexity is the emerging need to interleave and combine different problem solving algorithms, such as tree search and local search, at runtime. Figaro project is a software architecture designed for mastering this complexity. This architecture will serve as the base for the Figaro library for problem solving. The Figaro library is public domain software that supports finite domain constraint programming and local search and is open to extension to other techniques for problem solving.

1.3 Tree search Visualization

The development of constraint programming solutions requires extensive experimentation which is ideally supported with a tool for visualization of tree search. A successful example of such a tool is the Oz Explorer (Christian, 1997). This tool is limited to depth-first search, but allows us to visualize any single-pass tree search. The aim of this project is to use Figaro's component-based architecture to provide a search tree visualization that is independent of the search engines (such as depth-first-search) and that can be used easily in complex solutions where search engines are combined hierarchically. This task will likely require a redesign of parts of the current Figaro library and will lead to a significant contribution to the constraint programming community.

1.4 Report Overview

The report will talk about the visualization of search tree in Figaro. In chapter 2, the Figaro system will be explained in more details. After that, in chapter 3, general topics about tree search visualization in Figaro will be talked about. Then we talk about the algorithms and implementations in the following 2 chapters, namely chapter 4, and 5. More specifically, chapter 4 talks about the tree position algorithm used in drawing search trees; chapter 5 talks about main implementation issues. Near the end, chapter 6 gives some information about testing and performance. We will conclude the report in chapter 7.

Chapter 2: Figaro

2.1 Figaro Description

Figaro contains many components (Choi Chiu, Martin and Ka Boon, 2001). And the components can be combined to solve a CSP. Figaro system has some basic solutions to finite domains, constraints and constraint propagation algorithms. The variables and domains of a CSP are represented as stores. And each node object represents a CSP together with branching methods. Besides these basic components, Figaro system has various engines and problem models. The overall structure of Figaro system is shown in figure 2.1.

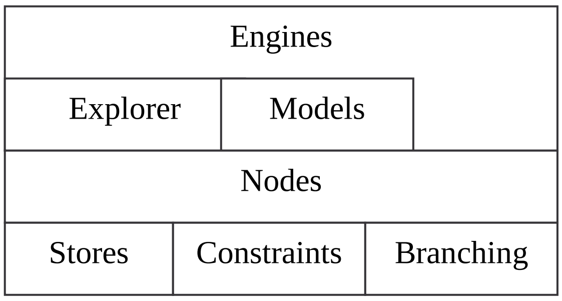


Figure 2.1: Overall structure of Figaro system

I will talk more about some important components, namely models, engines and branching, separately.

2.1.1 Models

To use Figaro system, users must describe their CSP's in a desired way. Figaro provides problem models to let the users describe their problems. All the problems must be specified following the model. Users must define the store within the model, post constraints into the model. The model must be able to output its store and generating branching.

2.1.2 Engines

Figaro has many types of engines. Once an engine is created and initialized, it can find its next solution. There are some types of engines each of which perform different task. Here are the engines provided in Figaro with short comments.

- **First Engine**: find the first solution of the CSP.
- **Last Engine**: find the last solution of the CSP.
- **Compose Engine**: compose two engines and form a new engine.
- **Solution Engine**: help to output solutions and related information.
- **Tree Search Engine**: use search methods to find solutions.
- **Model Engine**: modify the store and constrains while finding the CSP's solutions.

Since the report is about the tree search visualization and the tree search engine performs the search, it is necessary to know more about tree search engine. Tree search engines are the most import or basic type of engine in solving CSP's. Each tree search engine uses an explorer, which specifies the search strategy. There are currently three types of explorers available so far: depth-first explorer, breadth-first explorer and branch-and-bound explorer.

Figaro Engines can be combined in many ways base on the users' requirements (Choi Chiu et al, 2001). Figure 2.2 is a sample engine combination. This combination is to find the first solution of the CSP represented by the model engine using depth-first search.

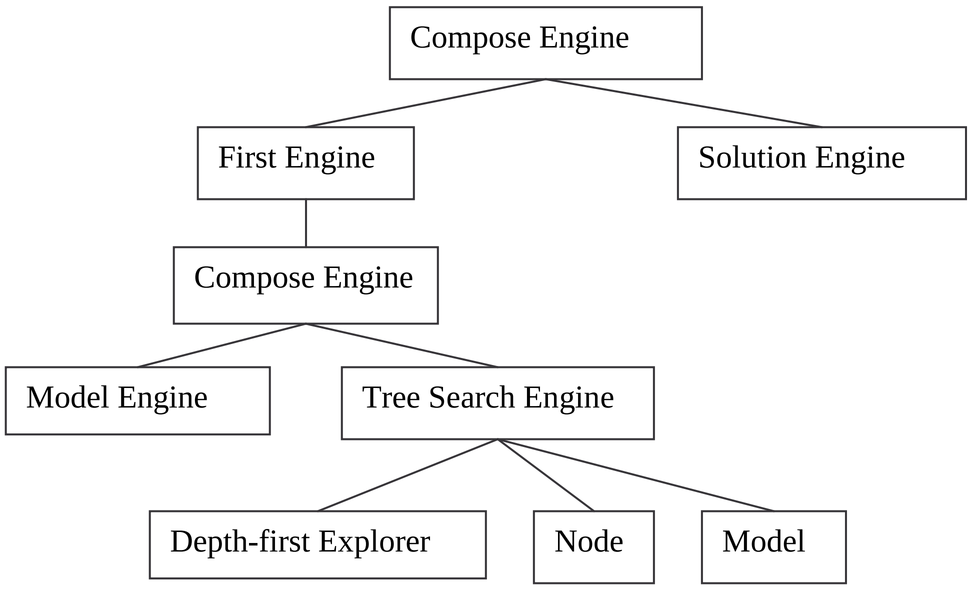


Figure 2.2 Combine Figaro engines

2.1.3 Branching

Figaro provides three branching strategies used in search. They are naïve, first-fail and split. Users can select whichever they want. Branching algorithms define the size and the shape of the search trees (Martin and Tobias, 2000).

2.2 Use Figaro to Solve Problems

We can use Figaro to solve our problem. First we construct the problem model. In the model we specify the variables and their domains, constrains, branching method, output methods. After define the model, we create an engine combination that can solve the problem. Then we just solve the engine combination to solve the problem. Figure 2.3 is a sample program solving n queen problem using Figaro.

```
#include "figaro.h"
using namespace figaro;

class NQueen : public Model {
private:
  int n;
  vector<varId> Row;
public:
  NQueen(int x) : n(x) { }
  ~NQueen() { }
  StoreState impose(store* s) {
    int i;
    for (i=0; i<n; i++)
      Row.push_back(s->newvar(1,n));
    vector<int> L1N(n);
    vector<int> LM1N(n);
    for (i=1; i<=n; i++) {
      L1N[i-1] = i;
      LM1N[i-1] = -i;
    }
    RETONFAIL(s->addcon(new Distinct(Row)));
    RETONFAIL(s->addcon(new DistinctOffset(Row,LM1N)));
    RETONFAIL(s->addcon(new DistinctOffset(Row,L1N)));
    return SLEEP;
  }
  branching* generateBranching() {
    return (new firstfail(Row.begin(),Row.end()));
  }
  void output(store* s) {
    vector<varId>::const_iterator idx = Row.begin();
    for (; idx != Row.end(); ++idx) {
      cout << (*(*idx))+1 << "=>" << s->getMin(*idx) << endl;
    }
  }
};

int main (int argc, char** argv) {

  int n = (argc > 1) ? atoi(argv[1]) : 100;

  NQueen* nqueen = new NQueen(n);
  solve(
     EngineCompose(
        Last(
45             EngineCompose(
46                new ModelEngine(nqueen),
47                new VisualTreeSearchEngine(
48                   new DepthFirstExplore(),
49                   new CopyNode(),
50                   nqueen
51                )
52             )
53          ),
54          new SolutionEngine(nqueen)
55       )
56    );
57    delete(nqueen);
58    return 0;
59  }
```

Figure 2.3. Program solving n queen CSP using Figaro

In the example, the n queen problem is specified use a Figaro Model. In the n queen model class, lines 12-14 specify the variables and their domains, lines 15-23 post the constrains, lines 26-28 state the branching method and lines 29-34 give the output method for the store. After the model is created, an engine combination is created (lines 43-55) and solved (lines 42). Figure 2.2 shows the structure of the engine combination.

Chapter 3: Tree Search Visualization in Figaro

3.1 Concepts

The visualization of tree search in Figaro has several concepts. They are visual tree search engine, solve interactive, stepping engine.

- **Visual tree search engine.** The visual tree search engine is an alternative of existing tree search engine from which it is derived. Users can use visual tree search engine if they want to see the tree search visually. This means users just need to replace tree search engines by visual tree search engines in the structure of the engine combination.
- **Solve interactive.** Solve interactive is an alternative of solve. It is used with visual tree search engine. It will keep the window which visualizes the tree search when the search job finishes so that users can do more interaction with the tree search. If solve instead of solve interactive is used, the program will quit when visual tree search engine finishes its job.
- **Stepping engine.** Sometimes, users want to control from the search starts. Stepping engine will stop the engine once the engine is initialized and let the user interact with the visual tree.

3.2 Features

The features are the functionalities that the visual window can provide to users. After this section, the readers will have the idea what the visual tool works. Now I will describe the features by dividing them into groups.

The first group concern about the search engine. The search engine uses explorer which specifies the search strategies. When visualizing the search, users can modify the search tree in three ways.

- **Show all nodes.** All the nodes in the search tree will be displayed. The user can interact with any part of the search tree.
- **Delete fail subtrees.** All the failed subtrees are deleted and the user can not view them any more. Deleting failed subtrees can save memory space since the trees constructed are smaller. Rendering a smaller tree is faster and the tree will takes less space in screen. If users are only interested in the solutions and

solution paths or want to find the solutions faster, deleting fail subtrees will be a good choice.

- **Hide fail subtrees.** It is similar with deleting fail subtrees except the failed subtrees are kept in memory and user can expand them. It has the advantages of deleting fail subtrees but it save less memory space comparing deleting fail subtrees since the tree structure must be kept.

During search, the tree will be refreshed frequently to show search process. Since the positions of all the nodes will be affected when new nodes are added to the tree, the whole tree will be considered to be redrawn once a single node is added. If we refresh the tree once a new node is added, or each search step, much time will be spent on the redrawing. So the user can specify how frequently the search tree is refreshed. There are two choices:

- **Refresh every n nodes**. Refresh every n nodes. This is often used with showing all ndoes.
- **Refresh every n solutions.** Refresh every n solutions are found. This is often used with deleting or hiding failed subtrees.

Users can also always see the last node searched by selecting

- **Trace last node**

Users can also control the search engine using the visual tool:

- **Reset/pause/resume/stop search.** Reset the tree search engine so that users can search from beginning. Stopping the engine search will also keep the window for interactions.

Once the search engine is stopped, user can do the following search:

- **One step.** Let the engine search one more step.
- **Find next solution.** Let the engine search for next solution.
- **Find all solutions.** Let the engine search for all solutions.

The second group concerns about viewing and traversing the search tree. When users view the search tree, user can

- **Hide/Unhide/Delete a subtree.**

The visual tool also provides following functionalities to facilitate users in finding node and traverse the tree.

- **Find top node/last node/up node/left child/right child.**
- **Zoom in/out the tree.**

The third group is concern about the search that is driven by users instead of the search engine. Only one such feature is provided.

- **Make child.** The user can search one step from any node that node finish searching. And users can search in the way they like instead of use the strategy of the explorer.

The final group is the general features.

- **Close/Exit/Help/About**. Close is different from exit. Close is just close the window and exit is exit the program.

3.3. Design Ideas

The design of the visual tool focuses mainly on two criteria: compatibility and search strategy independence.

Compatibility

The visual tool is just a component or a layer built on Figaro library. The existing Figaro system should not be affected by the visualization. In another word, the underline Figaro library should not care about the visualization.

Search strategy independence

The tree search visualization should not care about the search strategies. Existing Oz explore is a successful tool to visualize tree search in constraint programming, but it can only visualize depth-first tree search. Our visual tool can visualize any kinds of tree search such as depth-first, breadth-first, branch-and-bound search. The visual tool need not know the search strategies used when visualize the search.

3.4 Overall Design

The overall design structure of the visual tool is can be viewed as figure 3.1. There are mainly 4 parts in the structure: visual tree search engine, tree, window and explorer. Since the explorer has already exist in Figaro library. So it need not do this part of job any more.

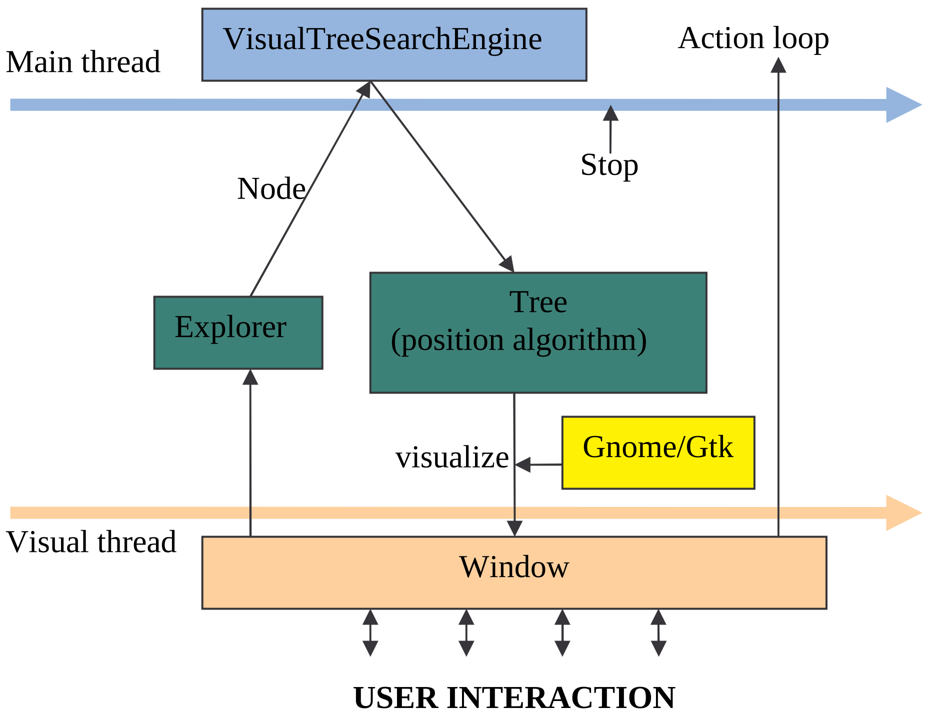


Figure 3.1: Visual tool design structure

Before visualize the search tree, the tree must be constructed first. The tree is constructed by visual tree search engine. More precisely, every visual tree search engine corresponds to a search tree. The visual tree search engine obtains nodes from the explorer it uses and adds nodes to its search tree. The tree object has the position algorithms to determine how to draw itself. Then the visual tree search engine draws the tree on the corresponding window using Gtk/Gnome graphic library. During search, the window will handle the user's actions. After the engine is stopped, an action loop will be run in the main thread to response the user's interactions that are permitted to run in this phase. Some response needs to use the explorer and tree draw methods as well. The program runs as visualization by using two threads.

Chapter 4: Node-Positioning Algorithm of General Trees

4.1 Drawing Trees

Drawing a tree consists of two stages: determine the position of each node, and actually rendering the individuals nodes and interconnecting branches. Figure 4.1 shows a well-drawn general tree.

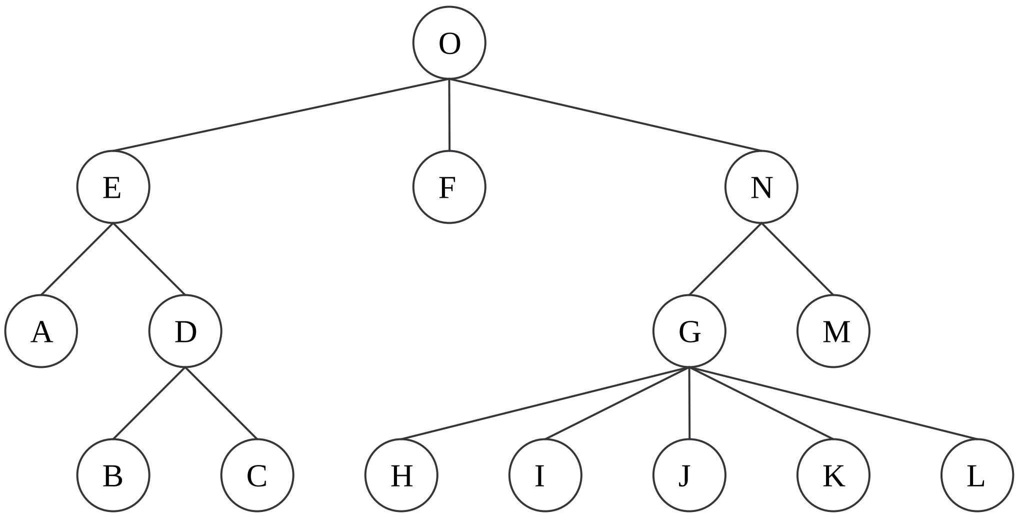


Figure 4.1. A well drawn general tree

Wetherell and Shannon first describe a set of aesthetic rules against which a good positioning algorithm must be judged (Wetherell and Shannon, 1979).

Tidy drawings of trees occupy as little space as possible while satisfying certain aesthetics:

1. Nodes at the same level of the tree should lie among a straight line, and the straight line defining the levels should be parallel.
2. A parent should be centered over its offspring.
3. A tree and its mirror image should be drawn the same way regardless of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree. In some application, one wishes to examine

large trees to find repeated patterns; the search for patterns is facilitated by having isomorphic subtree drawn isomorphically.

This implies that small subtrees should not appear arbitrarily among larger subtrees.

(a) Small, interior subtrees should be spaced out evenly among larger subtrees (where the larger subtrees are adjacent at one or more levels) (node F in figure 4.1).

(b) Small subtrees at the far left or far right should be adjacent to larger subtrees.

4.2 Node-Positioning Algorithm of Fixed General Trees

4.2.1 Walker's algorithm

John Q. Walker II has already given a node-positioning algorithm for fixed general trees that satisfies all the aesthetic rules (John, 1990). In general tree, there is no limit on the number of offspring per node. I will explain the algorithm and give some comments.

Walker's algorithms can calculate the position of the nodes of any general trees in *O(n)* time, where n is the number of nodes or the size of the tree (John, 1990). This algorithm initially assumes the common practice among computer scientists of drawing trees with the root at the top of the drawing. Node-positioning algorithms are concerned only with determining the x-coordinates of the nodes; the y-coordinate of a node can easily be determined from its level in the tree, owing to the aesthetics rule 1 and the natural convention of a uniform vertical separation between consecutive levels.

The algorithm uses two concepts. First is the concept of building subtrees as rigid units. When a node is moved, all of its descendants are also moved – the entire subtree being thus treated as a rigid unit. A general tree is positioned by building it up recursively from its leaves towards its root.

Second is the concept of using two fields for the positioning of each node. These two fields are

i. a preliminary x-coordinate, and
ii. a modifier field.

Two tree traversals are used to produce the final x-coordinate of a node. The first traversal assigns the preliminary x-coordinate and modifier fields for each node; the second traversal computes the final x-coordinate with the modify fields of all its ancestors.

In the following, walker's algorithm will be given with small modification. The algorithm is involved by calling the procedure POSITION. The tree position need traverse the tree twice (figure 4.2).

```
Procedure Tree::POSITION (Node)
    Begin
        /* Do the preliminary positioning with a postorder walk */
        FIRSTWALK (0);

        /* Do the final positioning with a preorder walk */
        SECONDWALK (0, 0)
    End;
End;
```

Figure 4.2 Procedure POSITION.

The first tree traversal (figure 4.3) is a post order traversal, positioning the smallest subtrees (the leaves) first and recursively proceeding from left to right to build up the position of the large subtrees. Sibling nodes are always separated from one another by at least a predefined SUBTREE_SEPARATION. Subtrees of a node are formed independently and placed as close together as these separation values allow.

```
Procedure Tree:: FIRSTWALK(level)
Begin
    SET_NEIGHBORS
    If (ISLEAF) then
        Begin
            If  HAS_LEFT_SIBLING then
                PRELIM ← LEFT_SIBLING → PRELIM +
                         SIBLING_SEPARATION+
                         NODE_SIZE;
            Else
                PRELIM ← 0;
            End;
        Else
            Begin
                For each OFFSPEING from left to right do
                    OFFSPRING $\rightarrow$ FIRSTWALK(leve+1);
            End;
            MidPoint $\leftarrow$ (LEFT_OFFSPRING $\rightarrow$ PRELIM+RIGHT_OFFSPRING $\rightarrow$ PRELIM)/2;
            If HAS_LEFT_SIBLING then
                Begin
                    PRELIM $\leftarrow$ LEFT_SIBLING $\rightarrow$ PRELIM +
                              SIBLING_SEPARATION +
                              NODE_SIZE;
                    MODIFIER $\leftarrow$ PRELIM – MidPoint;
                    APPORTION (level);
                End;
            Else
                PRELIM $\leftarrow$ MidPoint;
        End;
    End;
```

Figure 4.3. Procedure FIRSTWALK.

As the tree walk moves from the leaves to the apex, it combines smaller subtrees and their root to form a large subtree. For a given node, its subtrees are positioned one by one, moving from left to right following a second traversal (Figure 4.4). Imagine that its newest subtree has been drawn and cut out of paper along its contour. Superimpose the new subtree atop its neighbor to the left, and move them apart until no two points are touching. Initially their root their roots are separated by the sibling separation value; then at next low level, they are pushed apart until the subtree separation value is established between the adjacent subtrees at the low level. This process continues at success at successively lower levels

```
Procedure Tree::SECONDWALK (level, ModSum)
Begin
    XCOORD = PRELIM + ModSum;
    YCOORD = level * LEVEL_SEPARATION;
    For each OFFSPRING from left to right do
        OFFSPRING $\rightarrow$ SECONDWALK(level+1, ModSum+MODIFIER);
    End
End
```

Figure 4.4. Procedure SECONDWALK.

The second walk us a preorder walk (Figure 4.4). During the walk, each node is given a final x-coordinate by summing its preliminary x-coordinate and the modifiers of all the node’s ancestors. The y-coordinate depends on the height of the tree. If the actual position of an interior node is right of its preliminary place, the subtree must be moved right to center the sons around the father. Rather than immediately readjust all the nodes in the subtree, each node remembers the distance to the provisional place in a modifier field. In this second pass down the tree, modifiers are accumulated and applied to every node.

When pushing a new, large subtree further and further to the right, a gap may open between the large subtree and the smaller subtree that had been previously positioned correctly, but now appears to be bunched on the left with an empty area to their right (figure 4.5). This produces an undesirable appearance; this characteristic of *left-to-right gluing* will be removed by APPORTION procedure. APPORTION will shift the smaller subtrees to uniformly distributed between the larger subtrees (Figaro 4.6). For more information of the procedure, please refer to Walker's paper (John, 1990).

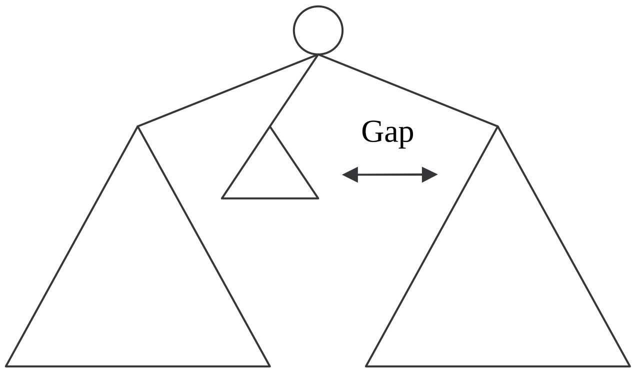
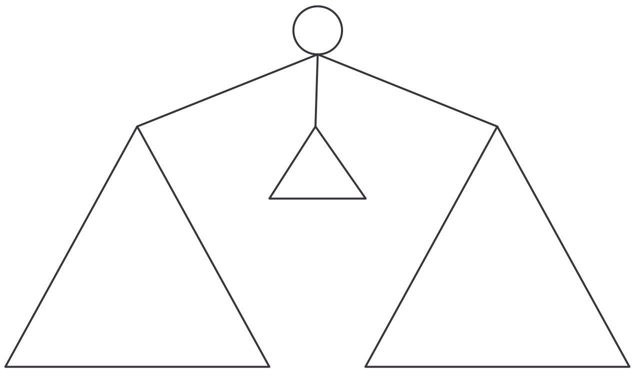


Figure 4.5: left-to-right gluing



Figaro 4.6. left-to-right gluing eliminated using APPORTION procedure.

4.1.2 An example

The tree shown in figure 4.1 is a tree positioned using Walk's algorithm. Let's see how Walker's algorithm computes the position of each node using two walks.

Suppose the node-size is 4, both sibling-separation and subtree-separation is 4. After the first walk, the preliminary x-coordinate and modifier value of each node are computed (table 4.1). Then the final x-coordinate value is computed in second walk (table 4.2).

Nodes	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**
Prelim	0	0	6	6	3	13.5	0	0	6	12	18	24	6	24	13.5
Modifier	0	0	0	3	0	4.5	0	0	0	0	0	0	-6	21	0

Table 4.1. Preliminary x-coordinate and modifier value for each node after first walk.

Node	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**
Xcoord	0	3	9	6	3	13.5	21	15	21	27	33	39	27	24	13.5

Table 4.2. Final x-coordinates of the nodes computed after second walk.

4.3 Incremental Node-Positioning Algorithm for General Trees

4.3.1 Analysis on Walker's Algorithm

Walker's algorithm positions general trees well. But when the tree are changed, if we applies walker's algorithms again, the calculation will starts from very beginning. This causes a lot of extra work since the preliminary x-coordinate, modifier or x-coordinates of some nodes will not change and it is better to just use the values has calculated previously. Thus an incremental version of Walker's algorithm is needed.

Before we develop such an algorithm, we must get insight Walker's algorithm and give some observations and findings. Walk did not explain the underline meanings of the preliminary x-coordinates, modifiers or first walk although he said this is to determine the preliminary position of the nodes. Here I will give some interpretations on these ideas.

Preliminary x-coordinates

The preliminary x-coordinate of a node only ensures that the subtree rooted by the node and subtrees rooted by siblings are well separated (figure 4.6). It dose not care about nodes not belonging the subtrees. Since the preliminary x-coordinates are about relative positions and do not care about the entire tree structure, the nodes without left siblings always have 0-value preliminary x-coordinates for simplicity (reference line show in figure 4.7). Assuming a virtual parent of the siblings, then the parent and the subtrees will form a well positioned tree.

It is easy to see that if the relative position regarding to its siblings dose not change, the preliminary x-coordinate will not change. Consequently, if a subtree's structure stays unchanged, the preliminary x-coordinates of all the nodes except the root node of the subtree also stays the same. If we want to shift a subtree *d* unit right, only the preliminary x-coordinates of the root node of the subtree changes (increased by *d*).

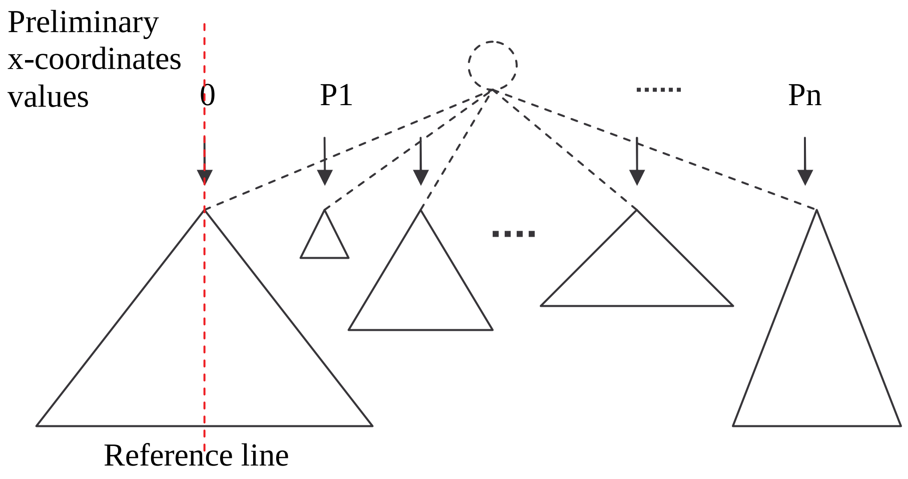


Figure 4.7: preliminary x-coordinates separate the subtrees rooted by sibling nodes.

Modifier

The modifier field of a node represents how the subtrees rooted by its offspring are shifted regarding to it (figure 4.8). In another word, modifier is to how to shift the subtree rooted by the node so that the it goes to the right position regarding its parent.

Similar to preliminary x-coordinates, it is easy to see that if the relative position regarding to its parent dose not change, the modifier will not change. Consequently, if a subtree's structure stays unchanged, the modifier of all the nodes except the root node of the subtree also stays the same. If we want to shift a subtree *d* unit right, only the modifier x-coordinates of the root node of the subtree changes (increased by *d*).

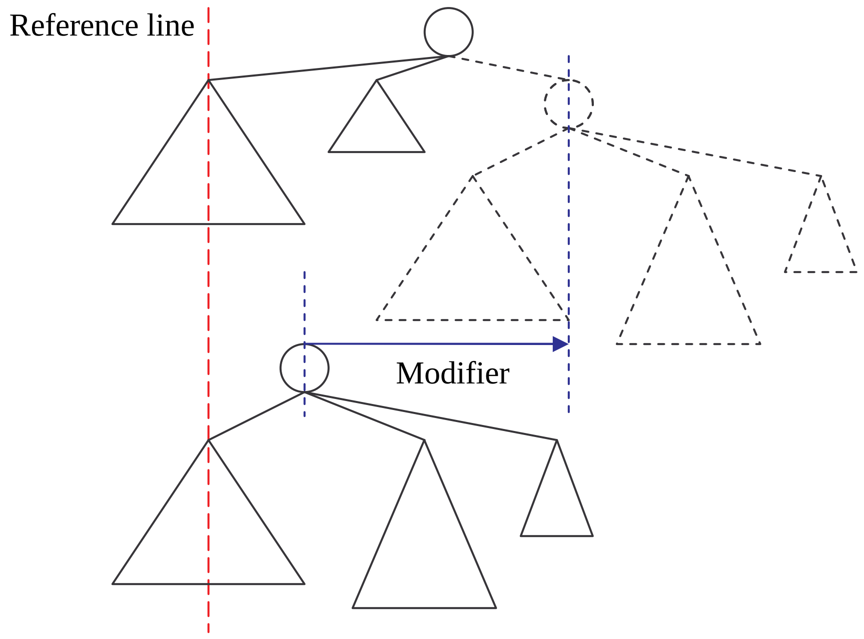


Figure 4.8: Modifier shifts the subtrees roots by the nodes offspring to right positions under it.

Similar to preliminary x-coordinates, it is easy to see that if the relative position regarding to its parent dose not change, the modifier will not change. Consequently, if a subtree's structure stays unchanged, the modifier of all the nodes except the root node of the subtree also stays the same. If we want to shift a subtree *d* unit right, only the modifier x-coordinates of the root node of the subtree changes (increased by *d*).

First walk

Combine the analysis of preliminary x-coordinates and modifiers, we know

- ➢ If a subtree is shifted (*d* unit right), only the root node of the subtree changes its preliminary x-coordinate and modifier value (both increase by *d*).
- ➢ Since first walk compute these two values, so we need only first walk the nodes which may be shifted.

Base on the analysis, we developed the incremental node-positioning algorithm for general trees.

4.3.2 Incremental node-positioning algorithm for general trees

Since the only the root nodes of the shifted subtrees need first walk. We can add SHIFT variable to indicate where a node will shift or not. Thus the modified first walk procedure comes out as figure 4.9.

```
Procedure Tree:: FIRSTWALK(level)
Begin
      If  not SHIFT then
            Return;
      SET_NEIGHBORS
      If  (ISLEAF) then
            Begin
                  If  HAS_LEFT_SIBLING then
                        PRELIM $\leftarrow$ LEFT_SIBLING $\rightarrow$ PRELIM +
                                    SIBLING_SEPARATION+
                                    NODE_SIZE;
                  Else
                        PRELIM $\leftarrow$ 0;
            End;
      Else
            Begin
                  For each OFFSPEING from left to right do
                        OFFSPRING $\rightarrow$ FIRSTWALK(leve+1);
                        OFFSPRING $\rightarrow$ SHIFT $\leftarrow$ true;
            End;
            MidPoint $\leftarrow$ (LEFT_OFFSPRING $\rightarrow$ PRELIM+RIGHT_OFFSPRING $\rightarrow$ PRELIM)/2;
            If HAS_LEFT_SIBLING then
                  Begin
                        PRELIM $\leftarrow$ LEFT_SIBLING $\rightarrow$ PRELIM +
                                    SIBLING_SEPARATION +
                                    NODE_SIZE;
                        MODIFIER $\leftarrow$ PRELIM – MidPoint;
                        APPORTION (level);
                  End;
            Else
                  PRELIM $\leftarrow$ MidPoint;
      End;
End;
```

Figure 4.9. procedure FIRSTWALK *in incremental node-positioning algorithm.*

The tree structure is different with that used by Walker in the sense SHIFT field, left and right neighbor are new in the figure. The neighbors should be stored because we should not be recomputed the neighbors some times. In The tree node structure used is shown in figure 4.10.

We must set some nodes' SHIF state to be tree is the tree is changed. Since trees' structure can be changed by adding or deleting nods to/from the tree. Some actions will change the structure of the tree

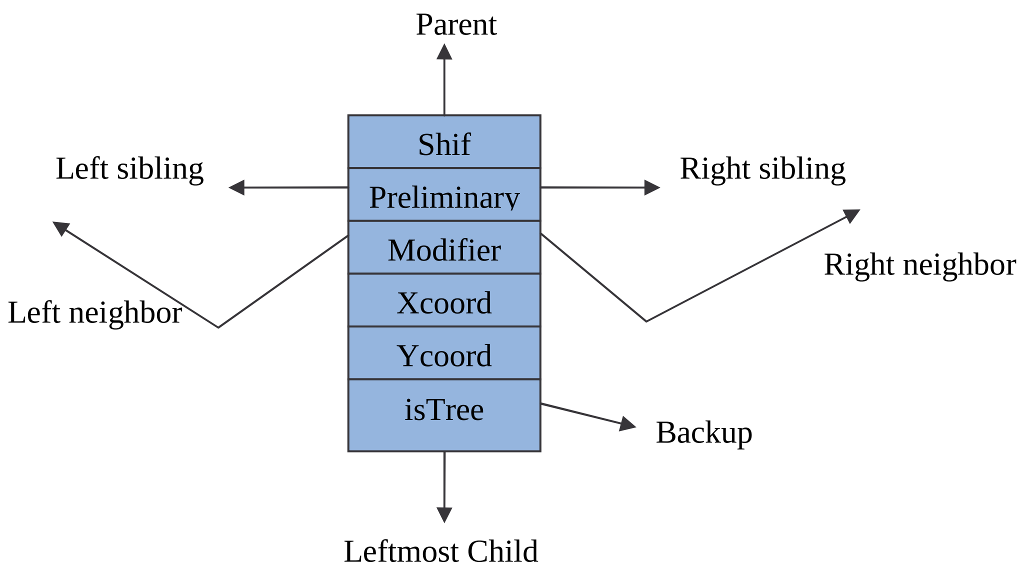


Figure 4.10. Tree node structure

.**Adding/deleting nodes**

The path of a node consists of the node itself and all its ancestors. We say a node A is adjacent to node B's path if the node is on or adjacent to the node B's path. When add a new node to a tree, only the node, its siblings, its ancestors and the subtrees rooted by the siblings its node's ancestors may shift. Thus only the nodes adjacent to the new node's path need to set SHIF to be true. Figure 4.11 shows the nodes that need first walk (in gray color) when a node is added as a child of node D.

Thus the first walk procedure run in O(logr(N)) time instead of O(N) time, where r is the order of the tree and N is the size of the tree. Adding more than one nodes is just doing the same work.

Deleting a node is similar to adding a node: just enable SHIFT state of the ancestors and their siblings.

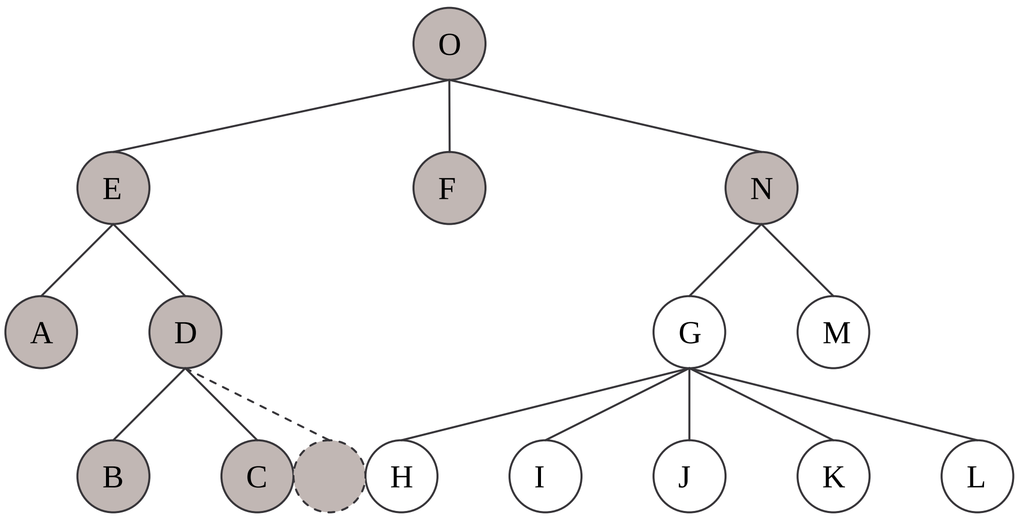


Figure 4.11. Mark nodes' SHIFT *state when adding a node.*

Hiding/unhiding subtrees

Hiding a subtree is just a trick. We just backup the subtree (the internal structure is not destroyed) and add two dummy nodes as the children of the subtrees's root. Then indicate apex node is a tree. When we draw a node which represents a subtree, we just draw a rectangle and omit its two dummy children (Figure 4.12).

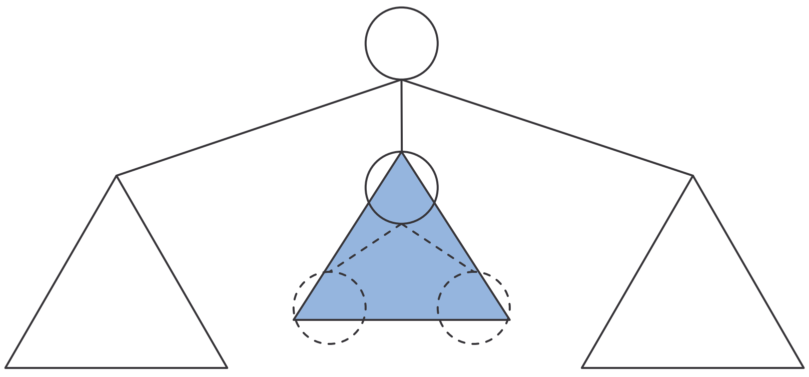


Figure 4.12. Hiding a subtree

A hidden subtree can be unhidden by restoring its original offspring previous backed up and indicate the node should be drawn as a node instead of a subtree.

Finding the node using an x-y coordinate

Finding a node by the position is useful when nodes are visualized and users want to access it by mouse clicking. When the tree are large, if we traversal the tree and check the positions of each nodes, it will waste time which is undesirable since we need quick response when clicking. Our algorithms can find the desired node quickly in O(r*$\log_r$N) time where r is the order of the tree and N is the tree size.

First we check the y-coordinate, if it is not near multiples of level separations, it is impossible to exist there. If the y-coordinate is reasonable, we check the x-coordinate as following:

We search from the root and go down to the desired level. In order to visit fewer nodes, in every level, we go down from the node whose x-coordinate is closest to the given coordinate. From figure 4.13, we know the path (path 1) is much shorter than other path (path 2). And this greedy algorithm will reach the desired level at the position near the desired node.

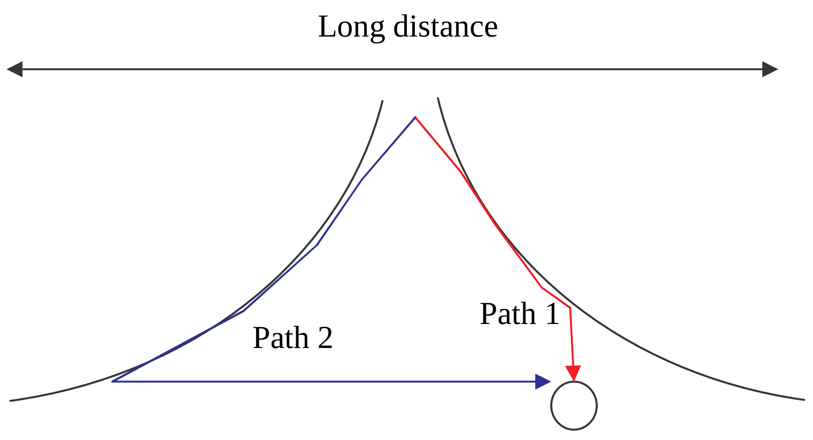


Figure 4.13: Finding node by position

Chapter 5: Implementation Issues

5.1 Search Visualization

In order to show how to visualize Figaro search trees, we should know what Figaro search trees are and what the resulting visualized trees look like. Figaro search trees are all binary search trees since Figaro always do binary search. There are 4 types of nodes in Figaro search tree:

- **Open nodes**. The nodes that have not finish branching. We can search from this type of nodes.
- **Close nodes**. The nodes that have finished branching. We can not search from this type of nodes. Both open nodes and close nodes are choice nodes.
- **Fail nodes**. The nodes that represent fail CSP's.
- **Solution nodes**. The nodes that represent the solution of the CSP. Both fail nodes and solution nodes are leaves of the search tree.

As we have talked in chapter 3, People want to hide or delete subtrees. We call a subtree failed if it contains no solution nodes or consistent otherwise.

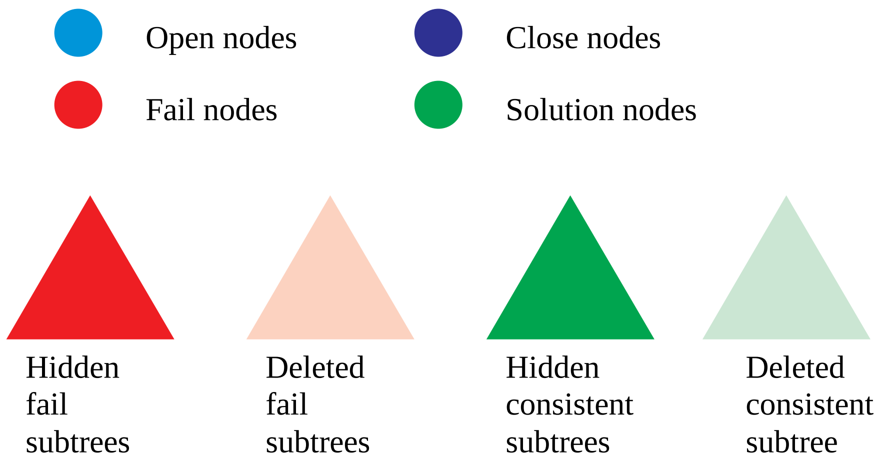


Figure 5.1. Display nodes and subtrees

When drawing the search trees, we use circles and triangles to represent nodes and subtrees respectively. The red color represents failure; green color represents solutions or consistency. Figure 5.1 shows how the nodes and subtrees are displayed in the search trees. Figure 5.2 is a sample search tree drawn by the visual tool.

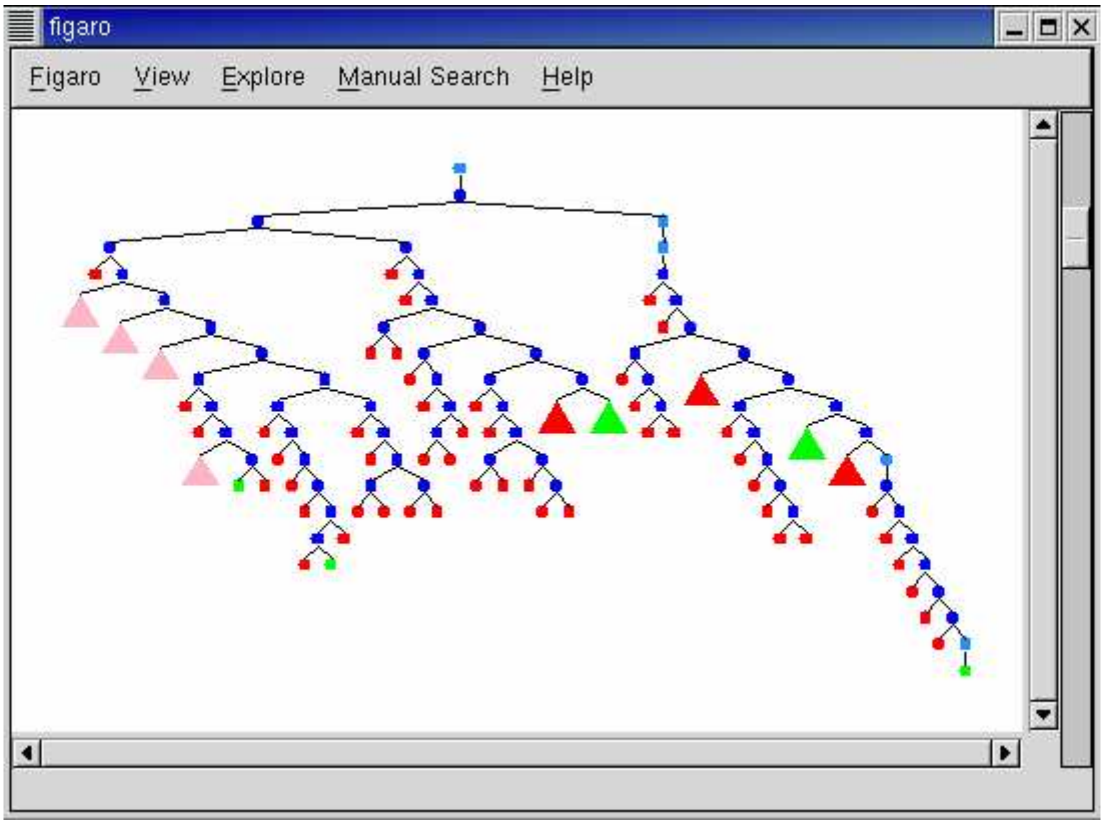


Figure 5.2. The Figaro search tree for magic square problem of size 4.

Now how to draw such trees? This consists of two problems.

- How to construct the search tree?
- How to rendering the search tree on the screen?

Tree construction

Since one design criteria search strategy independency of search visualization, the nodes must give the information where to put themselves onto the search tree. During searching process, the visual tree search engine uses an explorer which specifies the search strategy. When a node is explored and return to the engine, the engine can find the node’s position in the tree by find the path of the node from the root. Each Figaro node has a pointer to its parent and a position. The position is 0 or 1 which represent the nodes is the left or right child of its parent. With this structure, we can find the path of the node from root by tracking its parent recursively. Once the path is founded, we can add the node to the tree as figure 5.3. It is easily to see that the tree is constructed independent of its search strategy.

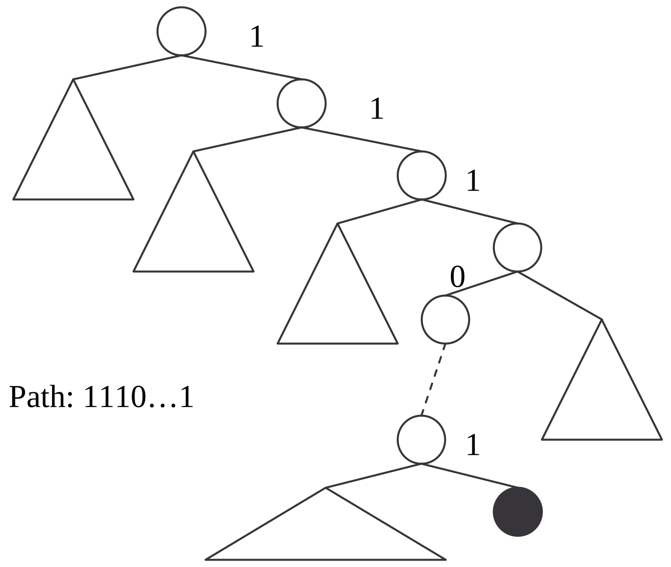


Figure 5.3. Add nodes by position paths.

The tree to be visualized is being constructed when searching solutions. Notes that during search, some nodes will changes its state, for examples open nodes become close nodes. And to distinguish the different types of nodes and subtrees, we must add some attributes in the tree nodes. This attributes are:

- ISFIAL. True if the subtree rooted by the node is contains no solutions.
- ISLEAF. True if the node are leaves.
- CHANGECOLOR: True if the state of the nodes are changed (open to close for example).
- ISNODE: True if the node should be drawn as a node instead of a subtree.
- ISOPEN: True if it is an open node.

When construct the search tree, users may want to hide the fail or delete fail subtrees. initially we assume all the node are consistent, then after the engine visits an fail node, trace from the fail node up and set the ancestors' state. For a node, if both of its children are in fail sate, then the node is in fail state. So we can compute the state bottom up. Each time we add a node, we must check whether its parent is open or close. If we want to delete the subtree, we do the same as above except that we need not do the backup.

Tree rendering

Once the search tree is constructed, we can render it on the screen. We use GTK/Gnome graphic library to do the rendering and handle the events (Havoc,1999).

We use a gnome canvas for each search tree, and draw the nodes using gnome items. The canvas has the transformation function. When second walk the tree and the x-y coordinate of a node is obtained, we put a canvas item on the position. We can count the number of nodes or solutions have been visited since last tree drawing and thus refresh (redraw) the tree for a certain period.

If we want to always see tree growing, we just scroll the scrolled window contains the tree and move position of the last node. We will not adjust scroll the scrolled window every time a node is displayed because it is two frequent and the tree will keep jumping. We scroll the scrolled window only when the last nodes go beyond the window.

As the window handles events and we also do the search. We need to run the event loop in another thread so that both search and event handling work well.

5.2 Tree View

After the tree is rendered on the screen, users need to view and explore the tree. Some general actions are zooming in/out the tree, hiding/unhiding or deleting a subtree, locating some nodes of the tree.

Since the canvas can zoom in/out its content, zooming in/out of a tree is simple. When gnome's event loop detects zooming actions, we set the zoom factor in the called response function.

A subtree is hidden, unhidden and deleted using the incremental algorithm described in chapter 4. To save memory space, we clear the subtree when hide the subtree. When unhide the subtree, we relocate the Gnome resources to draw the subtree. When delete a subtree, we delete the tree permanently. We *clear* a subtree by destroying the Gnome resources used by the subtree (canvas items, lines for example) and the subtree's structure stays unchanged. We *delete* a subtree by destroy the Gnome resources and the tree object.

We give each engine a target node representing the node that currently selected by users, root node and last node representing last node the explorer visited. If the user want to select a node (root node, last node, parent, left child, right child), just set the target node to be the node want to selected and highlight the node. Then adjust the scrolled window to center the target node.

5.3 Search Engine Control

5.2.1 Pause, search and stop engine search

The search done by a visual tree search engine can be paused, resumed and stopped. Each visual tree search engine keep monitoring its state variable during search. The state variable has three possible values: SEARCH, PAUSE or STOP.

The visual tree search engine takes different actions according the state. If the state is SEARCH, it continues searching as usual. If the state is PAUSE, it will wait on a condition variable associated with some mutex. Then the search thread will wait until the conditional variable will signaled. If the state is STOP, then the search is stopped and an action loop will start to run. The visual thread set the state according to users' input. The user pauses the search engine by changing SEARCH state to PAUSE, resumes the search engine by changing SEARCH state to SEARCH together with signaling the conditional variable the search is waiting on and stops the search engine by changing the state to STOP. In summary, we say the visual thread passes the users' events to the search thread through the state variable (table 5.1).

	Pause	Resume	Stop
Visual thread *Get user command* *Set state value*	State = PAUSE	State = SEARCH Signal conditional variable	State = STOP
Search thread *Read state value*	Waiting on condition variable		Stop search and start action loop

Table 5.1. Actions of the two threads in search control

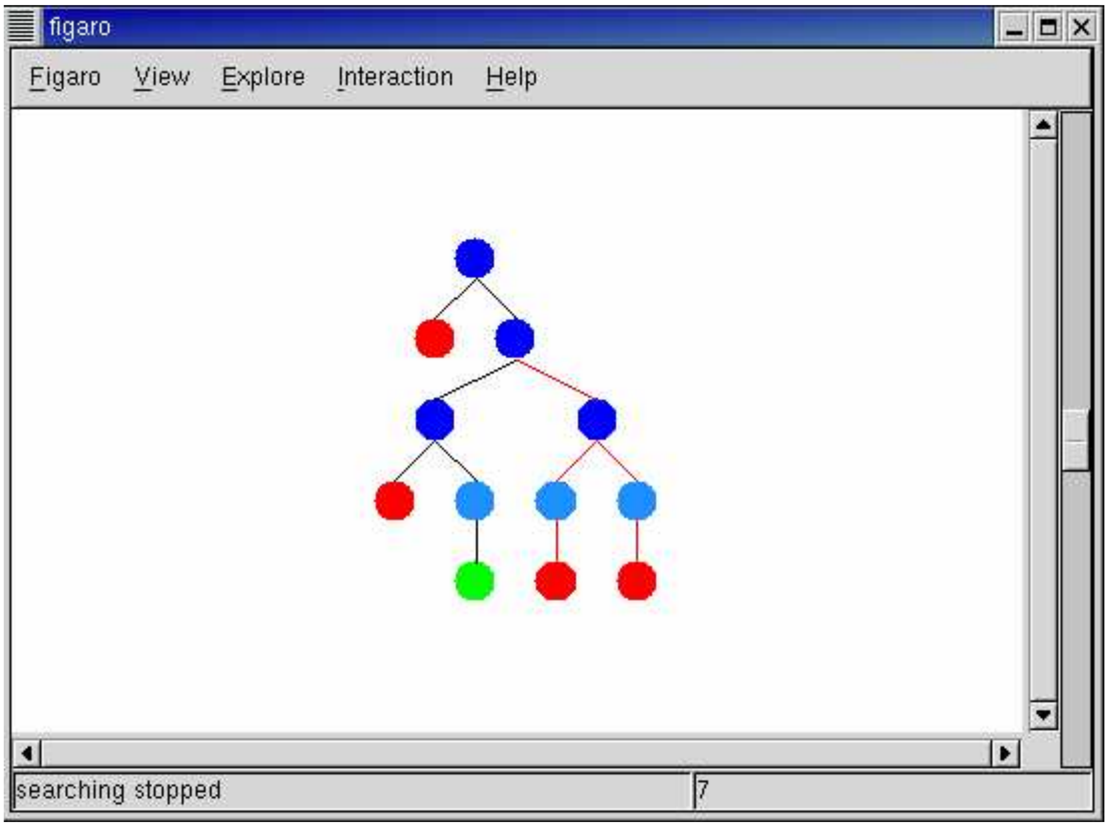
5.2.2 One step, next solution, all solutions and reset

When the search engine stopped, if the user wants to search one more step, we can tell the explorer used by the engine go one step and add the nodes to the tree. If the user wants to find next solution, we just let the visual engine find next solution. If the user wants to find all the solutions, we keep letting the engine find next solutions until finish the search. If the user wants to reset the search engine, we delete the search tree and init the engine using the root Figaro node stored.

5.4 Store Recomputation

In order to avoid storing unnecessary information, we do not keep the stores of each node in the tree. We only keep the root Figaro node for each engine and compute all the nodes' stores when needed. Every tree node has a path from the root, which correspond the path of the Figaro node it represents. A Figaro node can generate its left child Figaro node and right child Figaro node using MAKE_CHILD method. With the root Figaro node and the path of the selected tree node we can computer its Figaro node.

5.5 Manual search



searching stopped 7

Figure 5.4. Manual search

What the search talked above is done by engine where the search strategy is determined by the explorer the engine uses and can not be changed after program starts running. What we control is the visual tree search engine. Besides using search engine, users can also search manually. Manual search give users the search flexibility and it is independent of the search engine. In figure 5.4, the nodes follows red lines are the nodes searched by users while the engine use depth first search strategy.

When a node is selected, if it is not a leaf, the Figaro node is computed as we describe in section 5.3. Then we make child from the Figaro node obtained and added a new node representing the Figaro node to the tree. We also denote that the node is generated manually instead of by engine so that its branch line is colored in red.

5.6 Deal with Time-consuming Events

The Gnome has event handling system (figure 5.5). The Gnome keep running event loop to receive and handling the events such as clicking keys, pressing mouse button, resizing windows. The event loop consists of infinite iterations. The actions (such as rendering) regarding to the event are performed in a single Gnome iteration. The iteration ends and another iteration starts when the action is finished. The Gnome also has an event queue, which buffers events. If an actions that handle an event need long time to finish, then no more response to other events. This situation is like the Gnome is dead and cannot response to any events. This causes a big problem. For examples, finding next solution usually takes a long time (more than several second), and the search cannot be visualized since the Gnome iteration is doing its job (finding next solution). In the time finding next solution, the visual toll is dead an dwe do not what is going on by observation. Besides searching next solution, searching all solutions and unhiding trees are also such time-consuming events.

To deal with this problem, some events cannot be simply handled by Gnome iterations because a long time is needed. We should handle these events in a new way. Figure 5.6 illustrate out new event handling system used for some events. From the diagram, we know the actions Gnome event loop perform is only telling the action loop what event occurs and occurs to which object and let the action loop handle the event. This task is easy and can be done quickly.

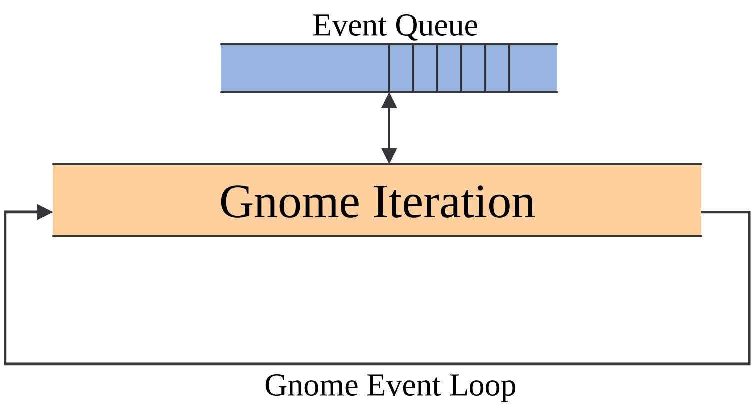


Figure 5.5. Event handling system in Gnome

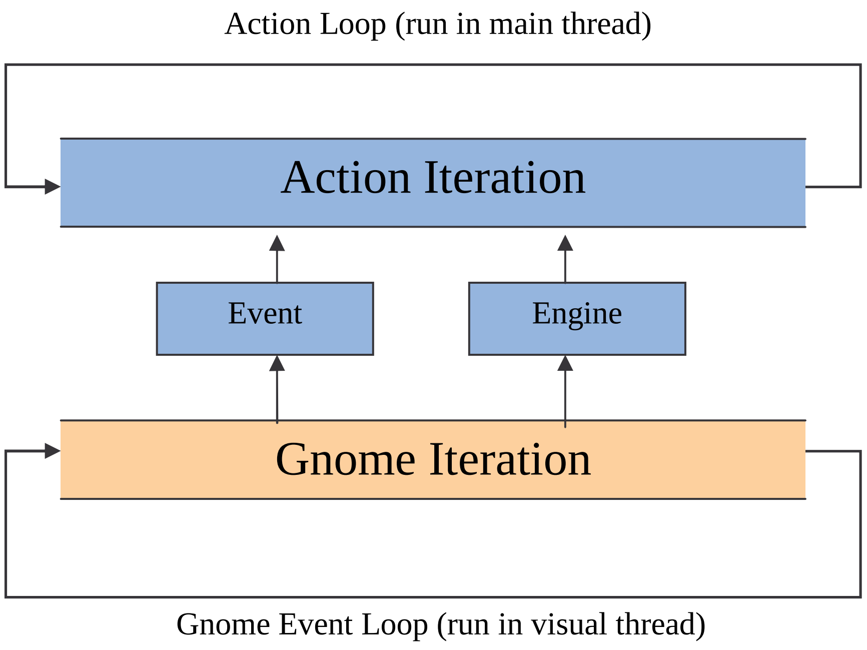

Action Loop (run in main thread)
Action Iteration
Event
Engine
Gnome Iteration
Gnome Event Loop (run in visual thread)

Figure 5.6. Handle big event in action loop

The action loop is just a loop that waiting for events and response to the events. Figure 5.7 describes how action loop acts. Two global variables are EVENT and ENGINE, MUTEX and CONDITION. When the event occurs, the Gnome loop only set the active tree search engine and the event type, and signal the conditional variable to let the action loop start performing the time-consuming actions.

```
Procedure ACTIONLOOP
Begin
    while (true) do
        thread_wait_on(CONDITION);
```

```
            **swich**(EVENT)
                **case** UNHIDE:
                      unhide the subtree;
                **case** NEXT_SOLUTION:
                      find next solution;
                **case** ALL_SOLUTION:
                      find all solutions;
                **default:**
            **End**
      **End**
  **End**
```

Figure 5.7. Procedure ACTIONLOOP

5.7 Thread synchronization

The visual thread runs a Gnome event loop, which consists of Gnome iterations. All the Gnome objects and settings must be ready before being rendered on the screen in an iteration. This means we cannot create or modifier the objects in another thread when an iteration is running. We say the iteration action in visual thread and objects creating actions in main thread are critical sections, which cannot run at the same time. So we must protect the iterations by using a mutex to lock iterations (figure 5.8). And any creating or modification of Gnome objects must require to the mutex.

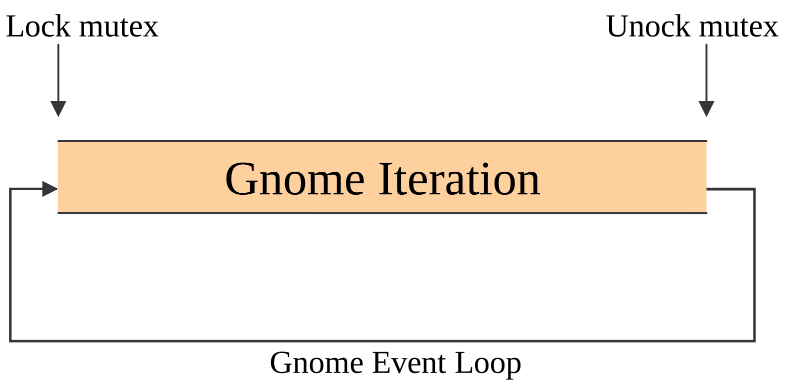


Figure 5.10. Protect Gnome iteration using a mutex

5.8 Solve Interactive, Stepping engine

Solve interactive is an alternative of solve in Figaro when using visual tree search engines. If use solve interactive, we can still interact with the search tree when the program finish searching. We just continue the program by starting an action loop instead of exit the program. The stepping engine will stop the search engine once the engine finishes initialization. If more the search engine is composed engine, then stop search when both the engine finish their initialization.

5.9 Search Tree Hierarchy

In Figaro, two search tree engines can be composed to search for solutions. Suppose they are engine *A* and *B*. Usually each step *A* searches, if the step is correct, then it initializes engine *B* in the step. Then engine *B* searches for solutions of the new CSP it represents. If engine *B* is also has solutions, then the whole CSP find a solution. To visualize such a search tree hierarchy, both the search engines need to be visual tree search engines. Two windows will be displayed to visualize these two search trees. And window *A* visualizes one search trees and window *B* will keep refreshing to visualize different search trees.

Chapter 6: Testing and Performance

6.1 System requirements

Figaro and the visual tool are implemented using C++. The operation system the visual tool used is Linux as Figaro. The visual tool also needs Gtkmm (C++ binding to GTK) of version 1.2 and above, Gnomemm (C++ binding to Gnome) of version 1.2 and above to be installed.

6.2 Performance

The total overhead includes three parts: visual tool initialization, tree visualization and tree rendering. Initialization consists of creating and popping up windows, which costs constant time. Visualization consists of constructing and positioning search trees. It takes different amount of time according to the sizes of the search trees. Tree rendering time also related with the size of search trees. We use a formula representing this (where *T* represents total time. *S* represents the basic search time, *I* represents the initialization overhead, *V* represents visualization overhead and *R* represents tree rendering overhead):

$$T = S + (I + V + R)$$

We only focus on the overhead percentage. Since overhead for certain number of nodes is about the same, the overhead percentages are various due to the propagation complexity. The overhead percentages are lower for simpler problems that propagate faster. Here we use two examples to testing the performance. One is *larry*, which is a simpler problem and the other is *robin* which is a more complex problem. For the larry problem, For the robin problem with 22 teams and 2 seasons, we search for its first solution. One Figure 6.1 is the testing results. All the time data is average value after testing more than 5 times and all the nodes of the tree are displayed.

Example	**Nodes**	**S(s)**	**I(s)**	**V(s)**	**R(s)**	**T(s)**
Larry	731	13.0	1.3	11	3.5	29.0
Robin	2059	570	1.3	168	36	775

Table 6.1 Performance testing results

When analyze the data (figure 6.2), the initialization overhead *(I)* is omitted. For larry example, visualization adds 85% overhead and tree rendering adds another 27% overhead. The total overhead is 112%. For robin examples, visualization adds 29% overhead and tree rendering adds another 6% overhead. The total overhead is 35%.

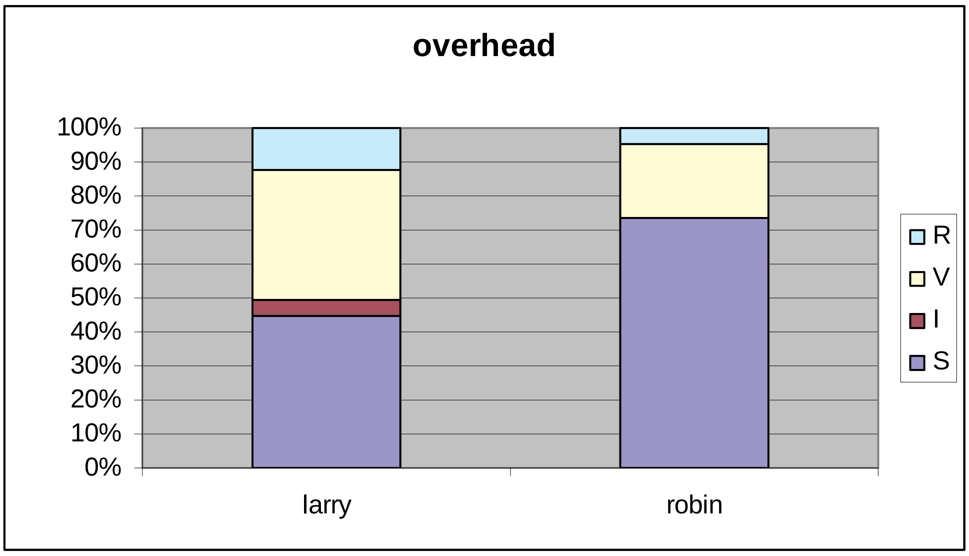


Figure 6.1 visual tool overhead

Base on the performance testing we know the overhead of the tool is reasonable. And the more complexly the CSP propagates, the lower percentage the overhead takes. Search tree construction makes up the majority of the overhead since the since we add a node to the tree by its path, which takes $O(\lg N)$ time where N is the tree size. This means when search tree is large, it takes more time to add a node while the Figaro nodes generating speed dose not increase.

Chapter 7: Conclusion

7.1 Summary

With the visual tool, users can get insight of the search visually, and can investigate on the search tree and do their own search. It is designed as component of the Figaro and independent of search strategies. The visual tool can be involved simply use visual tree search engines.

7.2 Limitations

- **Induced by the design**
    - No environment available set for the visual tool. We pass the search options as parameters of visual tree search engines instead of configure it in the system environment.
- **Induced by GTK/GNOME**
    - The tool cannot save or print the search trees displayed. The reason is the Gnome canvas dose not have print or save methods so far.
- **Induced by Figaro system**
    - We cannot manually init the second engine and start search in engine hierarchies. The reason is that we do not know the whole engine structure.

References

Andrew J. Kennedy. (2002). Functional Pearls: Drawing Trees. Journal of Functional Programming, 6(3), Cambridge University Press, May 1996, pp.527-534.

C. S. Wetherell and A. Shannon. (1979). Tidy drawing of Trees, IEEE trans. Software Engineering, SE-5, (5), 1979, pp.514-520.

Choi Chiu Wo. (2002). Advanced Components for Finite Domain Constraint Programming. M.Sc Thesis, National University of Singapore, Singapore.;2002.

Choi Chiu Wo, Martin Henz and Ka Boon Ng. (2001). A Compositional Framework for Search. In Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems, (Paphos, Cyprus, Dec 2001) CICLOPS 2001.

Choi Choiu Wo, Martin Henz, Ka Boon Ng.(2001). Components for State Restoration in Tree Search. In Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming,(Cyprus, Nov/Dec 2001) CP2001.

Christian Schulte. (1997). Oz Explorer: A Visual Constraint Programming Tool. In Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, pp.286-300. The MIT express, July 1997.

Christoph Buchheim, Michael Jünger and Sebastian Leipert (2002). Improving Walker's Algorithm to Run in Linear Time. 10th International Symposium, GD 2002 Irv, California, August 26-28, 2002, 2002.

E. M Reigold and J. S Tilford. (1981). Tidy drawings of Trees, IEEE Trans. Software Engineering, SE-7, (2), 1981, pp.223-228.

Havoc Pennington. (1999).GTK+/Gnome Application Development. New Riders Publishing, Indianapolis, 1999.

John. Q. Walker II. (1990). A Node-positioning Algorithm for General Trees. Software – Practice and Experience. Vol.20, No.7, July 1990, pp.685-705.

Ka Boon Ng. (2001). A General Software Framework for Finite Domain Constraint Programming. M.Sc Thesis, National University of Singapore, Singapore.;2001.

Krzysztof R. Apt. (2002) Principles of Constraint Programming. Notoes, National University of Singapore, Singapore.;2002.

Martin Henz and Tobias Müller. (2000). An Overview of Finite Domain Constraint Programming. In Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies, APORS 2000.

Martin Henz, Tobias Müller and Ka Boon Ng. (1999). Figaro: Yet Another Constraint Programming Library. Workshop on Parallelism and Implementation Technology for Constraint Logic Programming at ICLP 1999.