

# State Space Problem Solver

Corrin Lakeland  
Supervisor: Dr Geoff Wyvill

1996

## **Abstract**

A large class of problems can be described by a discrete set of states states. The report describes the Macintosh application *Game Solver* which was developed for manipulating and solving state space problems.

A state space problem is any problem which can be represented as current state with operations that allow moving between different states. A simple example is games where blocks slide around a board, or any board game.

Game Solver has two advantages over other applications. Firstly, it is simple to incorporate a new problem and secondly Game Solver provides a graphical representation of the state space, an *extent tree*.

# Contents

<b>1 Introduction</b>	<b>2</b>
1.1 A program called Game Solver . . . . .	2
1.2 Project plan . . . . .	2
<b>2 The Game Solver application</b>	<b>4</b>
2.1 Supporting arbitrary game descriptions . . . . .	4
2.2 Functions for a user . . . . .	4
2.3 Functions for a programmer . . . . .	4
2.3.1 Benefits of the shell . . . . .	5
2.3.2 Library functions . . . . .	5
2.4 Implementation . . . . .	5
2.4.1 Project Structure . . . . .	5
2.4.2 Interfaces and libraries . . . . .	5
<b>3 Sample Games</b>	<b>7</b>
3.1 Generalised games . . . . .	7
3.2 Board games . . . . .	7
3.2.1 Who wrote it . . . . .	7
3.2.2 Functions provided . . . . .	7
3.2.3 Restrictions . . . . .	8
3.2.4 Go Moku . . . . .	8
3.2.5 Othello . . . . .	9
3.2.6 Wei Qi . . . . .	9
3.3 Board games with sliding pieces . . . . .	10
3.3.1 Nuftafl . . . . .	10
<b>4 The undo tree</b>	<b>11</b>
4.1 Implementation . . . . .	11
4.1.1 Graphical display . . . . .	11
4.2 State space expansion . . . . .	12
4.3 Choice of a tree over a rooted graph . . . . .	12
<b>5 Searching</b>	<b>14</b>
5.1 Evaluation function . . . . .	14
5.2 Using a minimax searching algorithm . . . . .	15
5.2.1 Alpha beta pruning . . . . .	16
5.3 Metagame . . . . .	16
5.4 Implementation . . . . .	16

<b>6 Conclusion</b>	<b>18</b>
6.1 The extent tree . . . . .	18
6.2 Searching . . . . .	19
<b>A Glossary of terms</b>	<b>20</b>
A.1 Branching Factor . . . . .	20
A.2 GetChildren . . . . .	20
A.3 PutState . . . . .	20
A.4 GetState . . . . .	20
A.5 High Level Events . . . . .	20

## List of Figures

1	Project class structure . . . . .	6
2	Go Moku ‘Get Winner’ algorithm . . . . .	8
3	Othello piece flipping algorithm . . . . .	9
4	Example from Dr Wyvill’s algorithm . . . . .	11
5	Example of the left to right gluing problem . . . . .	11
6	Tree insertion algorithm . . . . .	12
7	Tree drawing algorithm . . . . .	13
8	Sample Othello evaluation algorithm . . . . .	14
9	Sample Minimax search . . . . .	15
10	$\alpha \beta$ searching algorithm . . . . .	17

# 1 Introduction

The purpose of this group project was to produce a complete Macintosh application which allowed users to manipulate games. In this project, the term game has become a synonym for a state space problem. This is because the term game is easier to relate to and the mathematical branch of game theory defines the terms game and state space equivalently [Ras94]. Therefore, using games instead of state spaces does not decrease the scope of problems that can be handled.

Included with the application are a collection of sample games which the user can play and several tools for manipulating the state space. The tools provided are an extent tree [Ras94] which shows the states that have been found and a 'computer player' where the computer will attempt to play the game.

It is expected that the application would be used by people interested in 'playing' with a game as well as by programmers who wish to produce a game playing program very quickly and easily.

The remainder of the report is structured as follows: Section 2 describes the Macintosh application that was produced and what it provides to users and programmers. Section 3 describes the implementation details relating to the games provided with the application. Section 4 describes the extent tree which provides the support for the user to explore the state space. Section 6 describes what the project achieved and what could be improved.

## 1.1 A program called Game Solver

The initial task of the project was to determine which features were appropriate for the application. We decided that to be useful for a non programmer, an intuitive user interface was necessary. This should also support normal user operations such as saving and quitting.

Additionally the application supports user interaction with the game. This involves:

- Reading a game description.
- Displaying the current state of the game on the screen.
- Obtaining user input and updating the internal state of the game.

As the application is intended for the user to research the state space rather than simply providing enjoyment, an 'impressive' user interface was not considered a high priority. This is an area for possible future development.

## 1.2 Project plan

The project group consists of David Burrell, Greg Scott, Stephen Stuart and myself. It was supervised by Dr Geoff Wyvill, Associate Professor of Computer Science at Otago.

Initially it was intended to subdivide the group into David and Greg who would work on the Macintosh interface while Stephen and I would work on developing an internal game 'engine' similar to that used in the general problem solver [NE69]. Additionally I would develop tools for searching a problems

state space, and provide a tree based mechanism of returning to states that have already been visited.

Under this system game files would be implemented in a rule description language such as Prolog [Row88], or a constraint language [CDPSV92]. Using this approach, a language interpreter could be used to determine how to respond to user interaction.

However it was later found that separating the game engine from the user interface required too much work and so David and Greg extended the user interface to support games thereby avoiding requiring a game engine and I concentrated on developing tools for examining the state space. This included:

- Providing an ‘undo tree’ into which every move is inserted so the tree provides a graphical representation of how to reach a state.
- Expanding the explored state space by inserting the ‘reachable’ set of states into the ‘undo tree’.
- Developing a function to search for a goal state in a state space problem.

These tools allow a large number of states to be explored before the best option is chosen.

## 2 The Game Solver application

The application is a complete Macintosh application. It is designed to work on all Macintosh computers and conform to the Macintosh interface guidelines given by Apple computer [RH91].

As well as being a useful application for a user, a large number of utility functions have been provided so that it is very easy for a programmer to write a new game for the application.

### 2.1 Supporting arbitrary game descriptions

As mentioned in the introduction, the game engine idea was dropped which means that the rules for every game must be defined inside the source of Game Solver, requiring the application to be modified and recompiled in order to support a new game. METAGAME is a similar project to Game Solver in which a simple *metarule* language was developed to support the definition of different games [Pel92].

This technique has significant benefits in that it makes it much faster and easier to define new games, however it does decrease the class of problems that can be solved [Pel92]. In order to give Game Solver some generality, as much as possible is defined in external game files. These include such information as the current board position as well as other game dependent data such as the rules of a boatgame.

### 2.2 Functions for a user

The following functions are provided for users

- Loading any number of different games.
- Explore the game using the extent tree
- Saving the current game along with the extent tree.
- Providing computer generated suggestions.

My assistance in developing the user interaction was minor, involving writing the functions to save the game, assisting Greg in developing a ‘Modal dialog’ and providing some technical assistance and examples when David and Greg first started the user interface.

### 2.3 Functions for a programmer

The application provides ‘library’ like functions such as string to number conversions. Additionally it provides a complete game shell so that the only work that has to be made is game specific. This is especially true for board games where all that needs to be done is write a function to set up the board and a function to make moves.

### 2.3.1 Benefits of the shell

By providing a shell, a game can be implemented by only writing domain specific code as all of the user interface is already written. This is in comparison to writing a game without using Game Solver where only a small proportion of the code is domain specific. For example, I implemented the game Go Moku in approximately twenty minutes using Game Solver compared to the several days it would have taken without using Game Solver .

### 2.3.2 Library functions

All library functions were implemented by David. These functions include file utilities for tokenising an input stream and graphics functions for dragging objects efficiently.

## 2.4 Implementation

The application was written in C++ using Metrowerks Code Warrior 9. This initially presented a difficulty for the group as I was the only group member with significant prior C++ experience . However it was decided the object orientated approach (C++) would provide significant benefits over the procedurally orientated alternative Pascal - a language which we all knew.

Games naturally lead themselves to *isa* relationships and this can be taken advantage of in object orientated languages. For example Chess **is a** board game and so any function which works for board games, such as saving to disk, will also work on Chess. C++ supports *isa* relationships using inheritance while standard pascal does not.

Throughout the remainder of the report, several properties of Object orientated programming and particular C++ will be used. These terms are defined fully in Stroustrup [Str91].

### 2.4.1 Project Structure

Figure 1 was produced by Greg and describes the class structure used in the application.

While this figure is technically correct, it does not represent how the code is actually used. In practise, several backwards loops were added to the class structure due to deficiencies which were not detected during the design phase. For example, all of the games access the manager through a backwards loop in order to insert new moves into the undo tree as games do not have an undo tree.

### 2.4.2 Interfaces and libraries

It was decided not to use any form of user interface builder or class library such as MAC APP as the group wished to learn how to program the Macintosh itself rather than how to program it indirectly through a programming utility. We decided to use the new "Universal Interfaces" so that we were learning up to date techniques rather than older techniques which will soon become redundant. Universal Interfaces are Apple Computer's new interface to the Macintosh toolbox of library routines.

### CLASS HEIRARCHY FOR GAME SOLVER

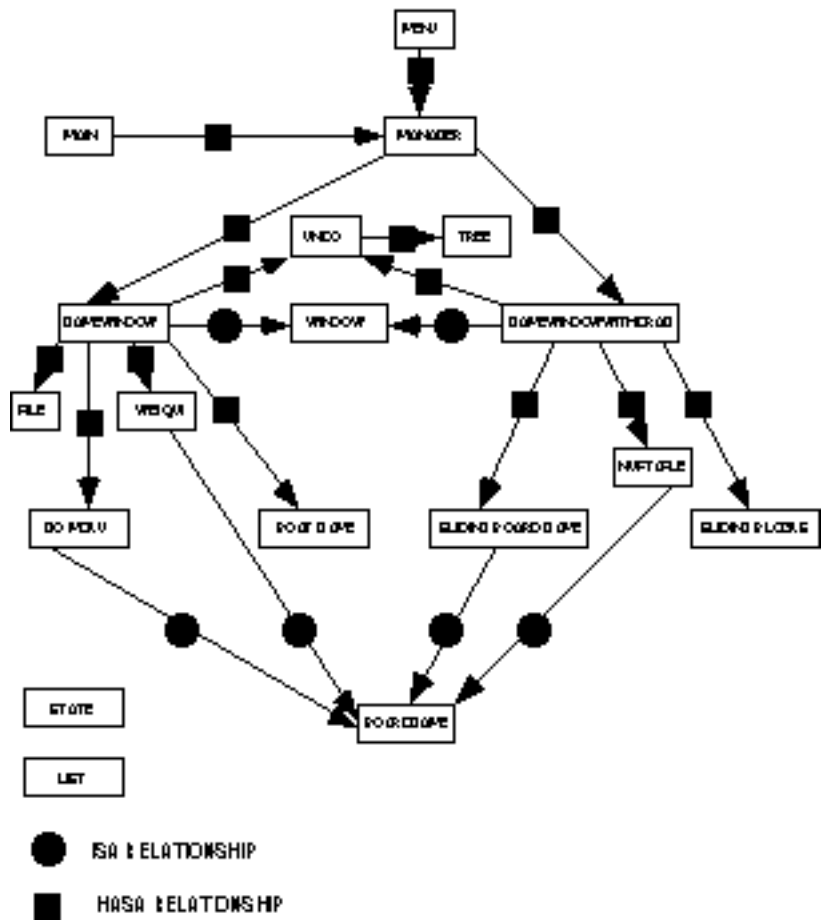


Figure 1: Project class structure

In order to support the extent tree and searching, I produced generic tree and list class's. The advantage of producing generic list and tree classes over non generic classes is that they can now be reused in later projects. I was initially going to take these classes from a book of C++ data structures [FT96], however I decided it would be easier to write them myself.

I also rewrote the ANSI library `(string.c)` so that no libraries had to be included in the application. This decreased the size of the final application by several hundred kilobytes. There were two reasons for this. Firstly, the waste of space seemed unacceptable and secondly its removal will greatly decrease the time required to download the application. Note that this should not have been necessary as the linker should have automatically removed the dead code, however the Metrowerks linker was not performing this operation properly.

## 3 Sample Games

To make the application more useful we included a collection of sample games. The sample games are intended to be as general as possible so that a non programmer can still explore different ‘games’ by modifying the game files. For example, they could try playing Othello on a different sized board.

A complete description of how to play all the games is provided in the users manual [BLSS96].

### 3.1 Generalised games

These games are general in that the one source file supports a very large number of different game files. The two generalised games that have been written are Sliding block games and boat games.

In sliding block games the user moves complex blocks around a board of blocks attempting to move the block that starts in the top left hand corner to the bottom right corner. This class was implemented by David.

In boatgames the user attempts to get the objects on one side of the river to the other according to some restrictions such as the weight of the objects on the boat. Many of the traditional state space problems such as Father and Son, Missionaries and Cannibals, and Husband and Wife can be implemented using this class. This class was implemented by Greg.

### 3.2 Board games

The board games are implemented as an ‘abstract base class’. That is, a class which provides the functions common to all board games where a board game can override the default whenever required. For example, the default board piece shape is circular, however this can be modified by a particular board game.

#### 3.2.1 Who wrote it

Stephen thought of the idea of making the boardgame class an abstract base class. However there were three major disadvantages in the original design.

**Non general** as the board description used was such that games such as Wei Qi could not be implemented.

**Inefficient** with both memory and processing. This made searching impractical.

**Not an effective base class** in that many functions which are game independent such as `MouseDown` were not included.

As a result of these problems I rewrote the class using a different design.

#### 3.2.2 Functions provided

The following utility functions are provided to programmers using the boardgame class:

- Handling a mouse click, determining which square it is in.

- A `GetChildren` function (see glossary) which is used in searching and extending the tree.
- Complete `GetState` and `PutState` functions (see glossary) which are used by searching and the undo tree.
- Draws the board
- Draws simple circles for pieces.
- Displays the winner of the game.

### 3.2.3 Restrictions

The boardgame class is built around the assumption that a move can be made by a single click in a rectangular cell. For many games that is not the case and so the class would be inappropriate. The sliding boardgame class in section 3.3 is an alternative class where moves are defined by mouse drags on a piece. Board games with non rectangular cells are not currently supported although this would be a trivial extension.

Games which have moves not fitting these definitions, or which require both clicking and dragging in different situations, cannot easily be implemented using these classes. For example, a player of chess can select any piece to promote a pawn to. Therefore an implementation of chess using Game Solver would not be able to include this rule because it would require the boardgame class to input information which is not a state transition.

### 3.2.4 Go Moku

This game is intended to provide a simple example of how to program a game.

The class stores the number of stones in a row needed to win in the external game file. As a result Tic Tac Toe which Stephen originally implemented separately can now be defined as a Go Moku game.

```

For each direction do
    count := 0
    while the next piece in this direction is our colour then
        advance
        count++
    endwhile
    if count ≥ the number of pieces to win
        winner := our colour
    end if
endfor

```

Figure 2: Go Moku ‘Get Winner’ algorithm

### 3.2.5 Othello

Othello was also originally implemented by Stephen. I rewrote it to enhance efficiency by removing the memory allocation which was making searching impossible as well as removing the dependence on  $8 * 8$ . The disadvantage of dynamically allocating memory is that this operation is very slow. The new algorithm is described in figure 3. It has the advantage that it is board size independent and that the pieces to be flipped do not have to be stored and so avoiding dynamic memory allocation.

```
For each direction do
    if the first piece in this direction is the opposition's then
        while the next piece in this direction is the opposition's, continue
            advance
        endwhile
        if the current piece is our colour then
            go to the start point
            while the current piece is the opposition's colour do
                flip the piece
                advance
            endwhile
        endif
    endif
endfor
```

Figure 3: Othello piece flipping algorithm

The algorithm now checks if there is a continuous line of pieces of the opposition's colour followed by a friendly piece. If so, it flips the opposition pieces in that direction.

### 3.2.6 Wei Qi

This is a simplified version of Wei Qi or Go. The capture algorithm is a standard graphical filling algorithm such as that used in painting programs. The filling algorithm was taken from [FvDFH90].

Two simplification to the rules of Wei Qi were made to simplify the implementation,

- The game does not check if a state has already been visited although this could be implemented using the extent tree.
- The scoring algorithm does not remove dead groups. This is very difficult to implement in a computer [GW77].

The evaluation function for Wei Qi is also very simple and so this implementation could not be expected to ever win a game. This function is complicated by Go's branching factor (see glossary) which restricts the search depth to two.

### **3.3 Board games with sliding pieces**

Many popular board games involve moving pieces around a board (eg Chess, Lido) which is not well supported by the boardgame abstract base class.

For this reason I wrote a slidingboardgame abstract base class which provides the same functions as the boardgame class<sup>3.2</sup>. It is not totally general, Shogi for example could not easily be implemented using the class as it requires pieces to be placed back onto the board.

#### **3.3.1 Nuftafle**

The game Nuftafle was implemented to test the sliding board game class. It was also used to test the searching function and for this reason includes a relatively advanced state evaluation function.

The piece taking algorithm is the same as Othello's except that pieces are not taken on diagonals and only a single piece may be taken in any direction.

## 4 The undo tree

The undo tree or ‘extent tree’ provides a graphical representation of the explored state space where each node it stores the state of the game so that edges represent transitions between states. Using this representation, the user can click on a node in the tree and have the game restored to that point.

The benefit of the tree is that users can explore a sequence of moves, return and explore a different sequence without loosing the first sequence of moves. Additionally, the extent tree is used for the output of the other state exploration functions such as the computer’s hint and the `ExpandTree` function.

### 4.1 Implementation

Internally the tree is stored using a binary tree with a left son, right sibling approach. This approach was suggested by Dr Wyvill and taken from Knuth [Knu73]. Using this approach the third child of a node is represented as `RSib(Rsib(LSon(Node)))`.

#### 4.1.1 Graphical display

The algorithm for displaying the tree was originally copied directly from an algorithm published by Walker [Wal90]. Unfortunately I made a bug in implementing this algorithm which was not found. As a result Dr Wyvill suggested a simpler algorithm, outlined in figure 7. This produces pictures such as that in figure 4 which demonstrates that the algorithm is still good at positioning trees. The primary disadvantage of the algorithm is the ‘left to right gluing problem’ originally described by Tilford [RT81], figure 5 highlights this problem. This figure shows how the rightmost subtree is being pushed too far right because the center subtree is expanding.

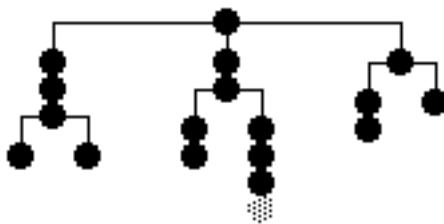


Figure 4: Example from Dr Wyvill’s algorithm



Figure 5: Example of the left to right gluing problem

```

foundNode := find newNode in the tree
if foundNode then
    the current node := foundNode
else
    if the current node has any children then
        let current child be the leftmost child of the current node
        while current child has a right sibling
            current child := the right sibling of current child
        endwhile
        the parent of newNode is the current node
        the right sibling of current child is newNode
    else
        the left child of the current node is newNode
    endif
    the parent of newNode is the current node
    the current node := newNode
endif

```

Figure 6: Tree insertion algorithm

## 4.2 State space expansion

This function finds the set of leaf nodes in the tree and then inserts all possible states which can be reached in one move from these nodes. This function can be used to examine how quickly the state space expands, its ‘branching factor’, as well as such questions as ‘how large is the state space of *game*?’ Finally it is useful for testing a game’s `GetChildren` function (see glossary) which is required for searching.

## 4.3 Choice of a tree over a rooted graph

Using a graph to represent the explored state space instead of a tree was not considered when the tree was first implemented. The advantage of using a tree is that it is much easier to understand what the graphical tree means than the graphical representation of a complex graph. The disadvantage is the graph provides information such as the number of different paths to a given state.

As project does not depend on a tree being used, the tree could be reimplemented as a graph by rewriting the template class.

```
if the currentNode is a leaf node then
    the width of the current node is nodeWidth
else
    for each child do
        the width of the current node += the width of the current child
        if the current child is a leaf node then
            the width of the current node += the sibling separation
        else
            the width of the current node += the subtree separation
        end if
    endfor
    the width of the current node -= the sibling separation
endif
```

Figure 7: Tree drawing algorithm

## 5 Searching

The searching algorithm was designed to find the optimal move in the game so that the current state is closer to some ‘goal’ state. This is the definition of ‘searching’ in artificial intelligence.

### 5.1 Evaluation function

Each game must supply an evaluation function which provides an estimation of how ‘good’ a given state looks. For example, in Chess if the opponents Queen has been captured, then the state will almost certainly have a high evaluation.

The benefit of using an evaluation function over searching until the end of the game is found, is that the size of the state space in almost all games is too large to be searched. In effect, the evaluation function provides an estimation of the chance of reaching a win from the current state.

A hypothetical evaluation function for Othello is given in figure 8 this demonstrates how the expected future (in terms of the corner ownership) generates the evaluation function.

```
if the number of moves to go  $\leq$  16
    value := the number of our pieces minus the number of their pies
else
    ourValidMoveCount := 0
    for each square on the board do
        if we can move on this square then
            ourValidMoveCount++
        endif
        if they can move on this square then
            ourValidMoveCount--
        endif
    endif
    bonus := 0
    for each corner square
        if we own it then
            bonus += 5
        if we probably own it then
            bonus += 5
        endif
        if they own it then
            bonus -= 5
        endif
        if we probably own it then
            bonus -= 5
        endif
    endif
    endfor
    value := ourValidMoveCount + bonus
endif
return value
```

Figure 8: Sample Othello evaluation algorithm

## 5.2 Using a minimax searching algorithm

The algorithm used is a minimax search with alpha beta tree pruning. Minimax is the process where one player is trying to maximise their evaluation function and the other player is trying to minimise the opponents evaluation function. The main benefit of using a minimax approach is that it means a two player game only needs one evaluation function. Unfortunately minimax cannot be extended to games with more than two players. For this reason all sample games only support one or two players. An example minimax search for Tic Tac Toe is given in figure 9. This figure shows how the highest value that the first player can obtain has been propagated up the tree.

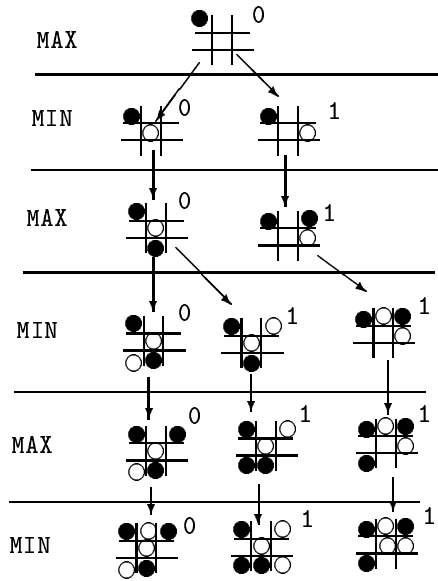


Figure 9: Sample Minimax search

Including probabilistic based pruning has also been considered but is not currently implemented. There are two main benefits of using such a pruning algorithm.

- The branching factor of games is not known to the searching function, however as the size of the state space grows exponentially with the branching factor and so games with a different branching factor cannot be searched to the same depth. This means that a game with a large number of bad moves such as Go Moku cannot be played effectively.

If probabilistic based pruning was incorporated then after states most likely to be the best are found, the algorithm can disregard all other children.

- In two player games, having a totally deterministic computer player leads to a very boring game and using probabilistic based pruning would result in the occasionally computer playing differently.

However, the only paper available that discussed this [Dic84] did not discuss implementation details and I decided it was too fragile to implement without such a paper. The reason it is fragile is that it attempts to cut branches not likely to have the solution based only on a preliminary evaluation of that branch, so a branch could be cut before it is known whether or not it contains the solution, resulting in a bad move being made.

### 5.2.1 Alpha beta pruning

One of the major disadvantages of simple minimax searching is that it ignores the reality that players will make the optimal move. This results in it needlessly expanding a number of states. The  $\alpha\beta$  *pruning* algorithm greatly decreases the number of states that are considered without changing the final result [Nil71].

Basically there is no point considering less optimal moves than the best move we have found so far. For example in the game of Chess, if one player can win in one move then there is no point in wasting time by considering other moves they may make.

Luger, Stubblefield [SL93] and Nilsson [Nil71] provide more detailed descriptions of how the alpha beta pruning algorithm works.

## 5.3 Metagame

The application METAGAME which is mentioned in section 2 strongly discourages this strategy: If it was possible to avoid an evaluation function then that would greatly enhance the highly general nature of the application, one application which does this is METAGAME [Pel92] which uses a learning algorithm for determining how to play each game.

Current approaches advocate the use of a *brute-force* search method (e.g., *minimax* with  $\alpha\beta$  pruning and singular extensions), combined with extremely fast routines for updating board positions. Although this technique has proven effective on several games, it presupposes that the researcher has a good *evaluation function*, which requires specific knowledge of the game.

METAGAME itself uses a learning algorithm to develop an evaluation function rather than obtaining one from the game definition.

## 5.4 Implementation

The alpha beta search procedure is implemented using a simple recursive tree traversal. Figure 10 contains the pseudocode for this algorithm.

The most difficult problem that had to be solved was finding the reachability set of a given state. The `GetChildren` function performs this task and is the most difficult function which every game new must implement

```
if the current state is a leaf state then
    return the current state
else
    if the evaluation function of the current state  $\geq$  alpha then
        return nothing
    end if
    if the evaluation function of the current state  $\leq$  beta then
        return nothing
    end if
    ...
end if
```

Figure 10:  $\alpha \beta$  searching algorithm

## 6 Conclusion

Game Solver is a large and powerful application and although it is called Game Solver it should be remembered that it does not just solve traditional games but a wide variety of state space problems, such as mathematical integration and network flow problems. It has the ‘look and feel’ of a complete Macintosh application, and supports several features not available in other applications. In particular, only one other application is known that supports branch based undo, and few applications support multiple games.

The application is stable and provides an extensive resource of code to assist a programmer in writing a new game. While some minor improvements could be made, if major changes were envisaged it would be better to redesign the project rather than attempt to modify existing code. This is because the Game Solver class structure is already compromised (see section 2.4.1), and further modifications would be likely to cause a structural break down.

The suggested minor changes are as follows:

- An improved user interface, for example, using colour.
- Implementing some more samples, particularly non game based samples such as the traveling salesperson problem.
- Designing some workaround of the C++ class structure so as to support high level events (see glossary).
- Improving the dialogs and other user interface elements.
- Improving the internal class structure to that which is actually used.
- Moving the game definitions from external files to STR resources.
- Working around the memory allocation bug in CodeWarrior so that the hint can be restored.

### 6.1 The extent tree

The extent tree has generally been a success. It is relatively intuitive to users and is an excellent exploration tool.

There are two major deficiencies in the tree. In some cases the current drawing algorithm produces poor results. This could be fixed by reimplementing Walker’s algorithm. The second deficiency is that it internally stores the entire state instead of a definition of the ‘edge’ that moves between the states. This means that the tree can use a very large amount of memory. The complete tree for a game of Wei Qi, for example, could exceed half a megabyte of RAM. It would be difficult to fix this problem without greatly slowing down searching as the evaluation function used in searching can only be used on complete states.

Overall the undo tree is more advanced than undo in other applications where few support more than single level undo.

## 6.2 Searching

Searching was only integrated with the application very near the completion of the project and as a result it is somewhat under developed.

Due to the simplicity of writing a 'game', it can be used for solving many simple problems. However in the conventional games, determining the children of a given state is too slow for the algorithm to be of much use in its current state.

Before any of the following improvements could be considered, a more efficient method of determining children states would have to be developed.

- Rather than expanding each subtree to a fixed depth, expanding them to a variable depth based on time would allow much more sophisticated moves to be made by the computer.
- While the alpha beta procedure is a significant advance over the basic minimax procedure, it is still highly vulnerable to games with many children (such as Wei Qi). Using a probability based edge cutting algorithm would help considerably. Such an algorithm would effectively ignore more than the best few children unless they have very similar evaluations. This would also remove the current determinism in the algorithm which makes the program less useful for game playing.
- The searching algorithm has a high memory requirement and this could be a significant problem. Again the complication comes from games with a large branching factor (see glossary).
- Much of the quality normally associated with the alpha beta procedure comes from having a good evaluation function. A large amount of effort has been spent taking work off the game programmer and requiring a good evaluation function is a step in the wrong direction. It may be possible that a different technique such as neural networks would be more effective

In summary, the group has produced a useful application and learned a lot at the same time. The most significant learning point is that planning does make large projects easier. Overall however the project has been a success and the application is excellent at its purpose of being a state space problem solver.

## A Glossary of terms

### A.1 Branching Factor

In every game there are different moves available at each state. The branching factor of a game is the number of these moves. In games with a very low branching factors searching is often simple, while games with a high branching factor tend to have to rely more on a good evaluation function.

### A.2 GetChildren

This function returns the list of states which can be reached in one move from a given state, otherwise known as the reachability set of a state.

### A.3 PutState

This function is used for setting the internal state of a game. This is required after several operations such as loading a saved game, or clicking on a node in the extent tree.

### A.4 GetState

This function is the opposite of PutState, it returns the current state of the game for saving to an external file, or storing in the extent tree.

### A.5 High Level Events

The Macintosh defines some special events, called “High Level Events” to represent such user operations as dropping a document onto the application or choosing print from the finder. Game Solver does not currently support these.

## References

- [BLSS96] D Burrell, C Lakeland, G Scott, and S Stuart. *Game Solver user manual*. University of Otago, Computer Science Department, University of Otago, Dunedin, 1996.
- [CDPSV92] T Christaller, F De Primolo, U Schnept, and A Voss. *The AI workbench babylon*. Academic Press, 24/28 Oval Road, London NW1, 1992.
- [Dic84] D Dickinson. *An expert system using fuzzy set representation for rules and values to make management decisions in a business game*. PhD thesis, University of Michigan, Ann Arbor, Michigan: University microfilms international, 1984. MFc 5T/D j08640527i.
- [FT96] W Fond and W Topp. *Data Structures with C++*. Simon and Schuster, 1996.
- [FvDFH90] J Foley, A van Dam, S Feiner, and J Hughes. *Computer Graphics, Principles and Practise*. The system programming series. Addison-Wesley, 2<sup>nd</sup> edition, 1990.

- [GW77] Y Gateley and L Williams. *GO Manual*. Raman Sciences Corporation, Colorado springs, 1977.
- [Knu73] D Knuth. *The Art of Computer Programming, Fundamental Algorithms*, volume 1. Addison-Wesley, 2<sup>nd</sup> edition, 1973.
- [NE69] A Newell and W Ernst. *GPS: a case study in generality and problem solving*. Academic Press, New York, 1969.
- [Nil71] N Nilsson. *Problem solving methods in Artificial Intelligence*. Computer Science. McGraw Hill, 1971.
- [Pel92] B Pell. Metagame: A new challenge for games and learning. *Artificial Intelligence 3 – The third computer olympiad*, page 15, 1992. Journal is actually a conference.
- [Ras94] E Rasmusen. *Games and Information*. Blackwell Publishers, 238 Main St, Cambridge, Massachusetts, 2<sup>nd</sup> edition, 1994.
- [RH91] C Rose and B Hacker. *Inside Macintosh*, volume 1-6. Apple Computer, 1 Infinite Loop Road, Capachino, USA, 1985-1991.
- [Row88] N Rowe. *Artificial Intelligence Through Prolog*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1988.
- [RT81] E Reingold and J Tilford. Tidier drawing of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.
- [SL93] W Stubblefield and G Luger. *Artificial Intelligence, structures and strategies for complex problem solving*. The Benjamin/Cummings Publishing Company, 390 Bridge Parkway, Redwood City, California 94065, 2<sup>nd</sup> edition, 1993.
- [Str91] B Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2<sup>nd</sup> edition, 1991.
- [Wal90] J Q Walker. A node-positioning algorithm for general trees. *Software Practice and Experience*, 20(7):685–705, 1990.