Diss. ETH No. 10927

Harnessing Computational Resources for Efficient Exhaustive Search

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZÜRICH

for the degree of
Doctor of Technical Sciences

presented by

RALPH UDO GASSER
Dipl. Informatik Ing. ETH

born October 3, 1966
citizen of Diepoldsau (SG)

Accepted on the recommendation of

Prof. Dr. J. Nievergelt, examiner
Prof. Dr. J. Schaeffer, co-examiner
Prof. Dr. J. Waldvogel, co-examiner

1995

to
Mom and Dad

Acknowledgements

Many people contributed their time to improving this thesis in various ways. Either providing valuable comments regarding content, style and presentation or implementing examples and porting code. Most important was the help people gave me with their presence and support when times were rough and I needed someone to talk to.

Special thanks go to *Jürg Nievergelt* for making this research possible. His interest in game-playing sparked my own, and his expertise often prevented me from heading off in the wrong direction. I am especially grateful for the time he spent teaching me better ways of presenting my ideas.

Although he may not be aware of it, *Jonathan Schaeffer* was a source of inspiration. By showing interest in my research and encouraging me to "get it written", he provided a much-needed push. He is also a competent judge of the technical aspects of this thesis, and let me in on some of his own ideas for the 15-Puzzle in addition to verifying results.

Jörg Waldvogel also deserves special mention for agreeing to be my co-examiner on such short notice. I particularly appreciate our interesting discussions and his introducing me to a related problem.

Many thanks go to the students who assisted me in various phases of this project: *Gerhard Balz*, *Christian Kunz*, *Thomas Lincke* and *Matthias Müller*. Their contributions made this thesis more well-rounded.

Special thanks are also due to the people in our group, who for years put up with my irksome ways:

- *Adrian Brüngger*, who vainly tried to awaken my cultural interests, and all the educational discussions this led to.
- *Michele De Lorenzi*, who let me misuse his social connections in order to gain access to powerful computers.
- *Ambros Marzetta*, unsurpassed for answering obscure technical and theoretical questions.
- *Fabian Mäser*, who allowed himself to be convinced that database verification was worthy of his attention.
- *Martin Müller*, for our many talks on games and game-playing, and for letting me beat him at Go.
- *Christoph Wirth*, without his constant ranting and raving, life at the office would have been considerably less fun.

Finally, I wish to thank *Monika* for periodically reminding me that there are things in life unrelated to this thesis…

Table Of Contents

Abstract

Searching is a central paradigm of computer science. In the past, search problems were often solved heuristically because of limited computational resources. Now that faster machines with larger memories are available, it is increasingly becoming possible to solve these problems optimally using exhaustive search.

In this thesis, we examine how forward and backward search techniques can be combined to efficient exhaustive search algorithms. To this end, we developed the SearchBench, a tool for exhaustive search. It allows the programmer to concentrate on the problem-specific aspects of her task, by pre-implementing the essential problem-independent algorithms. In addition to being suited for rapid prototyping, the SearchBench offers a simple interface, making it a useful teaching tool.

Two examples demonstrate the strength of our combined approach. Nine Men's Morris was proven a draw, making it the first non-trivial game to be solved with exhaustive search. Other games solved to date rely on knowledge-based methods to reduce the search space. For the 15-Puzzle, our approach reduces the size of the search tree needed to optimally solve a position. Progress was also made in finding the hardest 15-Puzzle position. We found positions requiring 80 moves to solve optimally and proved that any position can be solved in at most 87 moves.

Kurzfassung

Suchen ist ein zentrales Paradigma der Informatik. Bisher wurden Suchprobleme oft heuristisch gelöst, da nur beschränkte Rechenressourcen vorhanden waren. Durch die vermehrte Verfügbarkeit von schnelleren Prozessoren mit grossem Speicher, wird es zusehends möglich, diese Probleme optimal zu lösen.

Diese Dissertation untersucht Methoden um Vorwärts- und Rückwärtssuche effizient zu kombinieren. Zu diesem Zweck haben wir die SearchBench entwickelt. Dieses Werkzeug erlaubt es dem Programmierer, sich auf die problemspezifischen Aspekte seiner Arbeit zu konzentrieren. Dies wird erreicht, indem die problemunabhängigen Algorithmen bereits von der SearchBench zur Verfügung gestellt werden. Neben ihrer Eignung, schnell Prototypen zu erstellen, ist die SearchBench auch im Unterricht nützlich, da sie auf einer einfachen Schnittstelle basiert.

Zwei Anwendungen zeigen die Stärke von kombinierten Suchverfahren. Einerseits wurde bewiesen, dass das Mühlespiel unentschieden endet. Andere Spiele die bisher gelöst wurden, beruhen auf wissensbasierten Methoden, die den Suchraum einschränken. Andererseits hat unser Verfahren für das 15-Puzzle die Grösse der benötigten Suchbäume wesentlich reduziert. Ebenso haben wir Fortschritte bezüglich des Auffindens der schwierigsten Stellung gemacht. Insbesondere fanden wir Stellungen, die 80 Züge benötigen um optimal gelöst zu werden, und wir haben bewiesen, dass jede Stellung in höchstens 87 Zügen gelöst werden kann.

1
Introduction

1.1 Exhaustive Search

In spite of continuing hardware advances, one increasingly hears of computations which take months or even years to complete. When confronted with such long run-times, efficiency and optimization become critical. Traditional code-tuning techniques are not enough. Rather we must focus our attention on algorithmic improvements and parallelism. These have the potential to reduce run-times by polynomial or even exponential factors (versus constant for code-tuning).

Many large problems, for instance integer factorization, traveling salesperson or molecular modeling, can be formulated as search problems. Therefore many researchers focus on problem-independent search algorithms. Naturally, not all search problems yield to the same algorithms. We distinguish the following search characteristics:

- *Search completeness:* Is an approximate solution satisfactory or must it be provably optimal? Heuristics are often used to restrict the search to parts of the state space, thereby sacrificing accuracy for speed. If a solution must be proven optimal, exhaustive search is used.
- *Search direction:* Forward search starts from states with unknown value and tries to find their values. In contrast, Backward search starts from states whose value is known and propagates this information to unknown states.

In this thesis we will study exhaustive search problems. While the software tool we developed focuses on backward search, the main thrust of this thesis is to show that the combination of forward and backward search leads to more efficient algorithms than using either alone. Bi-directional search [Pohl 71] is an example of such an approach. The search starts at the known states and proceeds backwards, but also starts from the states whose value is needed, proceeding forward. The idea is for the two searches to meet in the middle. Bi-directional search trades memory usage for run-time, an idea found throughout computer science. Examining the best way of utilizing memory has recently become of renewed importance, because progress in hardware design has resulted not only in increased processor speed, but also makes substantially larger memories feasible.

Finding the combination of algorithms best suited to the problem at hand not only depends on the available memory, but on other criteria as well, for instance the number of states for which values must be computed.

If the optimal value for only a single state is needed, it is sensible to use as much pruning as possible to rapidly compute this value. Typical examples for such algorithms are alpha-beta and branch-and-bound. Improvements to these algorithms are mainly

achieved by finding ways to increase the amount of pruning, in other words finding the state's value while computing as few other state values as possible. The success of such improvements can be seen in the progress made from the first minimax algorithm to alpha-beta [Knuth 75] or proof-number search [Allis 94a].

At the other extreme, it may be necessary to find the optimal value for all states. Because of interdependencies between states, this can often be performed more efficiently than by applying a single-state search to each state. The large storage requirements involved in storing the values for all states is probably one reason this type of search has developed more slowly. An example of such an algorithm is retrograde analysis, which was pioneered by Ströhlein [Ströhlein 70] to compute the values of chess endgame positions. A further example for all-state search are sieves, for instance Eratosthenes' prime sieve, which are used primarily in number theory.

1.2 SearchBench: A Tool for Exhaustive Backward Search

A number of tools have been designed with the aim of providing a framework for general search problems. Most of these implement forward search. For example, the Smart Game Board [Kierulf 90] supports two-player games of complete information, and provides alpha-beta search. Another example is Metagame [Pell 93], a program which, when given the rules of a game, analyses them and attempts to find heuristics enabling it to play well.

To the best of our knowledge, no such tool exists for backward search. There have been various programs that solve dynamic programming problems [Smith 91]. However, dynamic programming requires the search graph to be acyclic, a simplification that is not generally possible. There have also been problem-dependent programs, based on retrograde analysis, designed to compute chess [Thompson 86] or checkers [Lake 94] endgame databases.

In contrast, SearchBench is designed to provide a problem-independent framework. This entails a trade-off between the generality of problems that can be solved and the amount of built-in support that is provided. The following key criteria were identified:

- *Rapid Prototyping:* A useful tool allows the programmer to focus on the problem itself and ignore file I/O, algorithm implementation and other side issues. Because the tool must implement these in a problem-independent fashion, an increase in run-time usually results. However, if this overhead is kept minimal, it may be unnecessary to rewrite the prototype, because more time is needed to do so than is potentially gained by the speedup.
- *Simple Interface:* The requirements for the tool interface are twofold. For teaching purposes it should be kept as simple as possible. At the same time, efficient problem implementation must be possible.
- *Generality and Portability:* We are not aware of any other general tool for backward search. Therefore the requirements of such a tool must first be determined. The SearchBench should then show the viability and usefulness of such a general approach.

The SearchBench kernel provides functions common to all problems. While it is difficult to foresee the requirements of future problems, the following are essential:

- *Computation:* Algorithmic and file support for database computation is provided. In particular the user does not need to know whether a database fits in primary memory or not.
- *Verification and Correction:* Various errors can occur during a search. These include software, hardware and handling errors. To eliminate these, the verification and computation algorithms share as little code as possible. If errors are found they must be corrected, preferably without redoing the entire calculation.
- *Compression:* Problem-independent compression is provided by an implementation of the Lempel-Ziv-Welch (LZW) algorithm. Problem-dependent compression is achieved with move ordering heuristics, i.e. instead of storing a state's value directly, we store the index of the best move in the heuristically ordered move list. A good heuristic reduces the entropy of the database, allowing greater LZW compression.
- *Interaction:* A problem display window allows database positions and their values to be shown. Support for user input, optimal play sequences and statistic functions is provided.

1.3 Games and Puzzles

Games and Puzzles have long been embraced by the mathematics [Berlekamp 82] and artificial intelligence (AI) communities as ideal testing environments. This is because their rules are simple to describe and improvements are easy to quantify. Especially chess was long regarded as the drosophilia of artificial intelligence. At present it is recognized that brute-force search suffices to play this game well, so some AI researchers now prefer using Go for their experiments. Research in this area has allowed programs with impressive playing abilities to develop. Some games have been solved, for instance Qubic [Patashnik 80], Connect-4 [Allen 89] and Go-Moku [Allis 94b], while the strength of current chess [Hsu 90] and checkers [Schaeffer 92] programs is comparable with top-level human play.

Search algorithms have also been applied to puzzles, which can be considered to be one-player games. For the Rubik's cube there is an algorithm by Morwen Thistlethwaite which provably solves any position in at most 52 moves, while a recently introduced algorithm seemingly solves any position in at most 21 moves [Kociemba 92].

This thesis examines two large search problems, Nine Men's Morris and the 15-Puzzle. The game of *Nine Men's Morris* was proven a draw. This is the first non-trivial game solved using a combination of alpha-beta and retrograde analysis. The other solved games mentioned above, all used knowledge-based methods to reduce the size of the search tree. This approach was not possible with Nine Men's Morris.

For the *15-Puzzle* smaller search trees are now needed for the optimal solution of a given position. Some progress was also made in finding the most difficult 15-Puzzle position, in particular we discovered positions requiring 80 moves to solve, previously the best known was 78. We also proved that no position requires more than 87 moves to solve.

1.4 State of the Art

In this section, we compare our contributions for the SearchBench, Nine Men's Morris and the 15-Puzzle with related efforts.

As described in section 1.2, the SearchBench provides a more general framework for backward exhaustive search than any other package we are aware of. It is interesting to compare the size of the problems solved with *SearchBench* to other problem-specific solutions:

- *Chess:* Ken Thompson computed many four and five piece chess databases, which have been released on two compact disks [Thompson 91], [Thompson 92]. Lewis Stiller used a Connection Machine CM-2 to compute six-piece chess databases, each of which consists of up to 6'185'385'360 positions [Stiller 91]. For a summary of results see [Herik 86].
- *Checkers:* An 8x8 checkers project is headed by Jonathan Schaeffer [Schaeffer 92]. His group has computed all 7-piece positions and are currently working on 8 and 9 piece positions. Their databases contain about $1.5 \cdot 10^{11}$ positions. A workstation cluster (25-90 machines) and a BBN TC2000 allow an average of 425 million positions to be computed per day [Lake 94].
- *Awari:* Victor Allis first computed Awari endgame databases for use in his playing program Lithidion [Allis 90]. We have extended this work and now have all positions with 22 or fewer stones ($5.5 \cdot 10^8$) at our disposal.

This summary shows that the size of present-day computations is on the order of 10^{10} positions. This is the exact size of the Nine Men's Morris databases. Thus even when taking our more general software package and slower machines (Macintosh) into account, our results are quite comparable to similar efforts.

Nine Men's Morris has not received much attention from the computer science community. Only a few computer programs have been developed [Ramseier 80], [Brod 89], [Weilenmann 91], and their playing strength is fairly weak. This allowed our early heuristic program to shine [Levy 91]. Except for an independent effort by Ingo Althöfer and Torsten Sillke [Althöfer 89], who computed the 3-3 database, no researchers have exhaustively analyzed any Nine Men's Morris subspace.

Research on the *15-Puzzle* can be classified into three areas:

- *Theory:* Examining the structure of the state space [Wilson 74] shows that it decomposes into two independent subspaces. Finding optimal solution is NP-complete [Ratner 90], while non-optimal strategies for solving the puzzle are polynomial [Sucrow 92].
- *Search algorithms:* Much work in this area has been done by Korf, who applied the IDA* [Korf 85] and RBFS [Korf 93] algorithms to this problem.
- *Lower bounds:* Instead of the standard Manhattan distance bound more informed bounds have been developed. For instance linear-conflict [Hansson 92] or fringe and corner databases [Culberson 94].

Our work is restricted to finding better lower bounds. It is similar to [Culberson 94], however we managed to further reduce the search tree size while using only 15% as much memory.

1.5 Thesis Structure

Chapter 2 introduces the general search model that provides the basis throughout this thesis. We examine the conditions that must hold for the solvability of a given problem

(fixpoint existence) and also examine methods of computing the fixpoint. We give evidence that finding a fixpoint with purely forward or backward search is less efficient than using a combined approach.

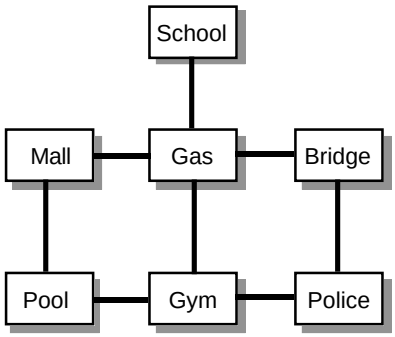
Chapters 3 and 4 describe the SearchBench, a software tool supporting exhaustive backward search. Chapter 3 examines the architecture and interface provided by this tool. Chapter 4 takes a closer look at the basic algorithms, in particular how the search model is efficiently implemented in the SearchBench. We also show how run-times can be estimated and compare different methods of parallelizing the algorithms.

Chapter 5 explains how the game of Nine Men's Morris was solved. This solution is the most computation intensive contribution of this thesis, involving various machines over a time period of approximately three years.

Chapter 6 shows another example for the applicability of our main idea (combining forward and backward search), yielding an improved algorithm for the optimal solution of the 15-Puzzle. Additionally we describe how this idea can be extended to the 24-Puzzle.

Chapter 7 concludes this thesis with a summary and a look at directions for future research.

2
Search Problems



Small town model

As described in the introduction, a variety of algorithms for solving search problems exist. Examples are the alpha-beta, branch-and-bound, retrograde analysis, dynamic programming and sieve algorithms. This chapter defines a common search model that lies at the heart of all these algorithms. We discuss the conditions that must hold, so that consistent values for all states (fixpoint) can be found. This leads to the comparison of two algorithms for computing such fixpoints. We then demonstrate that the combination of these two algorithms leads to more efficient algorithm than using either alone.

We illustrate the theory in this chapter with an example of a small town consisting of various buildings and facilities. Because this old-fashioned town has not yet seen a need for one-way streets, it can be modeled by an undirected graph, where the vertices are the buildings and the edges are the connecting roads. We assume that all roads are one kilometer in length. The search problem is to find the distance of any town location to the pool, the focus of summer activity.

2.1 Search Model

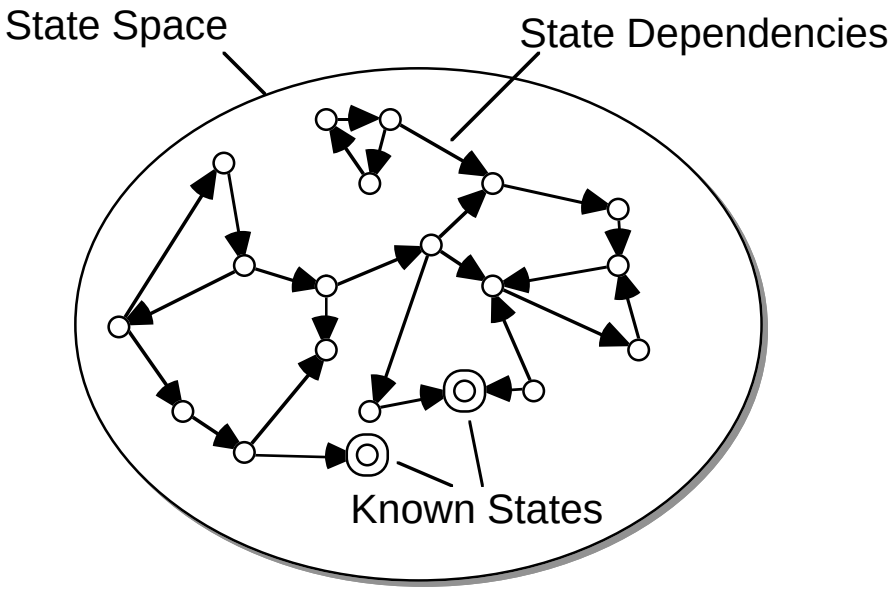
Problem solving can be regarded as finding an object or problem state with given characteristics. For this purpose, the artificial intelligence literature [Pearl 84] introduces three components:

- *Database or State Space:* capable of representing all states.
- *Operators or Production Rules:* enabling the entire state space to be traversed.
- *Control Strategy:* operator scheduling designed to quickly find the desired state.

In this thesis, the goal of searching is not viewed as finding a specific state, but rather as finding the *values* of states. This is a more powerful paradigm, because the AI model can be simulated by letting the values be pointers to the goal state. In this context we do not need a variety of production rules, but rather a definition of which set of states the value of a given state depends on and a function for combining the values of the dependent states into a value for the given state. The operators and production rules can therefore be replaced by:

- *State Dependencies and Value Gather Function (vgf).*

Both these models make no distinction between heuristic and exhaustive search. In general, heuristic search can be regarded as an exhaustive search that terminates at non-leaf nodes or that prunes certain dependencies. The first case can be modeled by defining additional known states. The second by using a modified dependency function.



Search Model

State Space

The state space *S* is the finite collection of all states in the search problem, where a state represents a unique problem configuration. The choice and construction of the state space is an important design criteria. In our small town example, we want to know the distances from any given vertex to the pool. Therefore, we introduce a state representing the problem “distance school-pool”, another for “distance gym-pool”, etc. Often the construction of the state space is non-trivial. For instance, a naive representation of many games includes numerous symmetric positions, while an improved state space only includes one representative from each set of symmetric positions.

State Dependencies

The state dependency function and its inverse are defined as:

$$dep(s) \stackrel{Def}{=} \{ s_1, s_2, \ldots, s_n \} \qquad \text{where } s, \forall_i s_i \in S$$

$$dep^{-1}(s) \stackrel{Def}{=} \{ s_1, s_2, \ldots, s_m \} \qquad \text{where } s, \ \forall_i s_i \in S, \ \forall_i \ s \in dep(s_i)$$

Because the number of states is finite, *dep(s)* and *dep^{-1}(s)* are always computable, however they may not be equally complex. For instance, factorization is more dif∗cult than multiplication. We also refer to the states generated by *dep(s)* as successors and the states generated by *dep^{-1}(s)* as predecessors. The state space and state dependencies together de∗ne the directed *State Space Graph*.

State Values

All states have a value taken from a ∗nite value range *V*. The goal of the search problem is to compute the values of a given subset of all states. The size of this subset may vary from only one (single-state search) to all states (all-state search).

$$val(s) \stackrel{Def}{=} v \qquad \text{where } s \in S,\ v \in V$$

At the start of the search, the *val* function is initialized. The known states are set to their prede∗ned values and the remaining states are set to a default value. During the search *val* is updated using the value gather function. The value gather function (vgf) is a total function which computes a given state s new value based upon the dependent states and the old value function. It returns a new updated value function.

$$vgf\ (s,\ val_{old}) \stackrel{Def}{=} val_{new}$$

In our example, the initial value function returns zero for the single known state ∗distance pool-poolτ, all other states have their value set to unknown. *Vgf* sets a state s value to be one kilometer greater than the minimum value of any dependent state.

$$val_{init}(s) = \begin{cases} 0 & \text{if } s = pool \\ unknown & \text{otherwise} \end{cases}$$

$$vgf(s,\ val)(x) = \begin{cases} 0 & \text{if } val(x) = 0 \\ \min\limits_{s_{dep} \in dep(s)} (val(s_{dep}) + 1) & \text{if } s = x \\ val(x) & \text{otherwise} \end{cases}$$

So for instance vgf(gym, val_{init}) returns the following val function:

$$val(s) = \begin{cases} 0 & \text{if } s = pool \\ 1 & \text{if } s = gym \\ unknown & \text{otherwise} \end{cases}$$

The value gather function assumes that it can access any dependent state or value it needs. In practice such an assumption cannot always be ef∗ciently implemented. In addition it is often possible for the value gather function to determine a state s value without considering all dependents. Therefore it is useful to have a partial value gather function, i.e. one that does not require all dependent values at once but one at a time. The partial vgf shown below uses one state (s_{dep}) and its value to update the value for state *s.*

$$vgf_{part}(s,\ s_{dep},\ val_{old}) \stackrel{Def}{=} val_{new} \qquad \text{where } s_{dep} \in dep(s)$$

Any value gather function can be rewritten in this fashion, but usually the value range *V* must be extended. Consider a value gather function which computes a state s value to be the average of all dependent values. Val(s) in this case is simply a number. However, in order to turn this vgf into a partial vgf, val(s) must contain the values of *all* dependent

states. Only storing the average is insuf∗cient, because if the value of a dependent state changes, its previous value is needed in order to update the average accordingly.

In general the val function of a partial vgf will need to store the values of all dependent states as well as its own. However, many partial vgf s require val to store little or no additional information. Our town example is such a case. Its vgf_{part} leaves the value range unchanged and can be written as:

$$vgf_{part}(s,\ s_{dep},\ val)(x) = \begin{cases} \min(val(s),\ val(s_{dep}) + 1) & \text{if } s = x \\ val(x) & \text{otherwise} \end{cases}$$

2.2 Fixpoint Existence

Before attempting to solve a given search problem, it is prudent to assure that it is solvable. In other words, we must verify that the given graph topology, initial value function and value gather function can theoretically lead to state values that remain unchanged, regardless which state the vgf is applied to. For this purpose, we de∗ne a function:

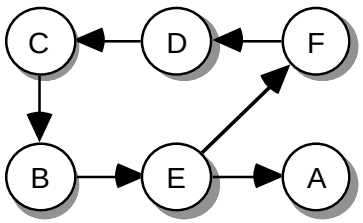
$$f(val) = \forall\ s \in S :\ val :=\ vgf(s, val) \qquad \{\text{parallel application}\}$$

In the simple case where the state space graph is acyclic, a ∗xpoint of *f* exists for any initial value and value gather function. Any topological sorting of the states leads to a sequence in which the vgf can be applied once to each state, ensuring a ∗xpoint.

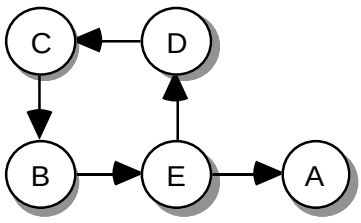
In general the state space graph will contain cycles. In this case the existence of a ∗xpoint depends on the interplay between the state space graph topology, the initial value function and the value gather function. Consider the following vgf and graphs:

$$val_{init}(s)\ = \begin{cases} 0 & \text{if } s = A \\ \text{unknown} & \text{otherwise} \end{cases}$$

$$vgf(s,\ val)(x) = \begin{cases} (\max\limits_{s_{dep} \in dep(s)} (val(s_{dep}) + 1)) \bmod 5 & \text{if } s = x \\ val(x) & \text{otherwise} \end{cases}$$

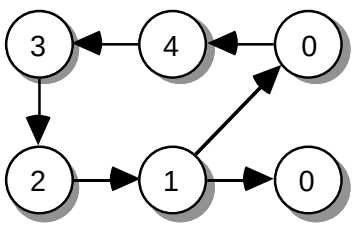


Graph 1



Graph 2

The ∗xpoint of the ∗rst graph is shown on the right. For the second graph none exists. Further analysis reveals that a ∗xpoint only exists if all cycle lengths are multiples of ∗ve.



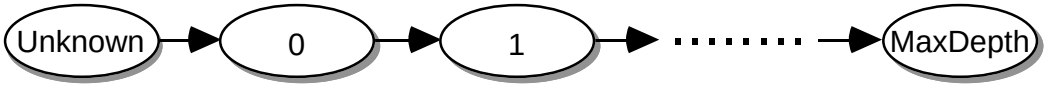
Fixpoint of Graph 1

This example shows that determining whether a ∗xpoint exists can be dif∗cult in general. In this thesis, we restrict our attention to a subset of all value gather

functions that can provably be shown to have a ∗xpoint on any graph topology.

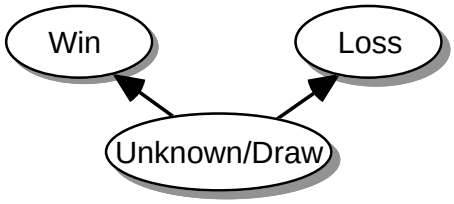
Speci∗cally, we de∗ne a partial order with a single global minimum on the value range *V*. A ∗xpoint exists if we can show that applying vgf(s, val), will not decrease val(s) in regard to the partial order. Because the value range is ∗nite, a time must come when no more values can be increased. Then a ∗xpoint has been found.

In the small town example, we postulate the partial order shown below. The state values are ∗rst set to the global minimum (unknown), except for the known state ∗distance pool-poolτ which is set to zero. Any application of the vgf can now only increase a state s value or leave it unchanged. What effect will an increase have on the other state values? The only directly affected states are the predecessors. From their viewpoint, a dependent state s value has increased. This will either leave their own value unchanged (if the changed dependent state did not have the minimum value) or increase their state value. It remains to be shown that all state values remain in a ∗nite range. This is the case since no value can be greater than the number of states Therefore this vgf will have a ∗xpoint on any graph.



Partial order of values

The value gather function for two-player games also plays an important role in this thesis. Here the state value should reτect the outcome of the game, i.e. does the player to move win, lose or draw. We postulate the partial value order shown on the right. Since the values of all states that are not known should be set to the minimum value, the following initial val function results:



Partial order of game values

$$\mathrm{val}_{\mathrm{init}}(s) = \begin{cases} \text{win} & \text{if s recognizable as won} \\ \text{loss} & \text{if s recognizable as lost} \\ \text{unknown} & \text{otherwise} \end{cases}$$

Clearly, a state s value is a win if there is a dependent state where the opponent will lose. And a state is a loss only if all dependent states are wins for the opponent. This leads to the following value gather function:

$$\mathrm{vgf}(s,\ \mathrm{val})(x) = \begin{cases} \mathrm{val}(x) & \text{if } s \neq x \\ \text{win} & \text{if } \underset{q \in \mathrm{dep}(s)}{\exists} \mathrm{val}(q) = \text{loss} \\ \text{loss} & \text{if } \underset{q \in \mathrm{dep}(s)}{\forall} \mathrm{val}(q) = \text{win} \\ \text{unknown} & \text{otherwise} \end{cases}$$

To assure that a ∗xpoint exists, we must show that the two legal value transitions (Unknown→Win and Unknown→Loss) can only cause further legal transitions in the state space graph. The following table shows the possible transitions a value may make and the effect this has on its predecessors states.

State Transition	Predecessor State Value	Predecessor Transition	State Transition	Predecessor State Value	Predecessor Transition
U→L	U	U→W	U→W	U	U→U, U→L
U→L	L	never occurs	U→W	L	never occurs
U→L	W	no change	U→W	W	no change

Possible value transitions

If any state changes its value from unknown to a loss, the predecessor state values are won. Before the value could either have been won or unknown, neither case violates the non-decreasing condition. If a state changes from unknown to win, the predecessor values either remain unknown or becomes a loss. Again these transitions are allowed according to the partial order of game values. The table also contains two cases that cannot occur. If a state value is a loss, this implies that all successors are won for the opponent, therefore no successor can make a transition Unknown→Win or Unknown→Loss. Since any legal state transition only induces further legal transitions, a ∗xpoint exists on any state space graph.

2.3 Computation Strategies

Our search model solves a problem by repeatedly applying the value gather function to all states. A computation strategy dictates in which order the vgf is applied to the states. Naturally, the object of a computation strategy is to ∗nd a ∗xpoint with as few vgf applications as possible. The following simple computation strategy gives an upper bound for the number of vgf applications needed to ∗nd a ∗xpoint. The idea is to cycle through all states, applying the vgf until no more changes to the val function occur.

```
val := val_init;
s := *rst;
*nal := s;
REPEAT
      val_new := vgf(s, val);
      IF val_new  val THEN
         *nal := s;
         val := val_new;
      END;
      s := nextstate(s);
UNTIL s = *nal;
```

If a ∗xpoint has not yet been found, at least one state must change its value when the vgf is applied to all states. Since each state can change its value at most |value|-1 times, no more than |states|·(|value|-1) changes can occur. Therefore, at most $O(|state|^2 \cdot |value|)$ vgf applications are needed to ∗nd a ∗xpoint.

However, at least for acyclic graphs, it would be ludicrous to determine the ∗xpoint in this fashion. For these cases, it is suf∗cient to apply the vgf to each state at most once, if done in topologically sorted sequence. The following two fundamental computation strategies solve this and other cases more ef∗ciently.

Forward Search

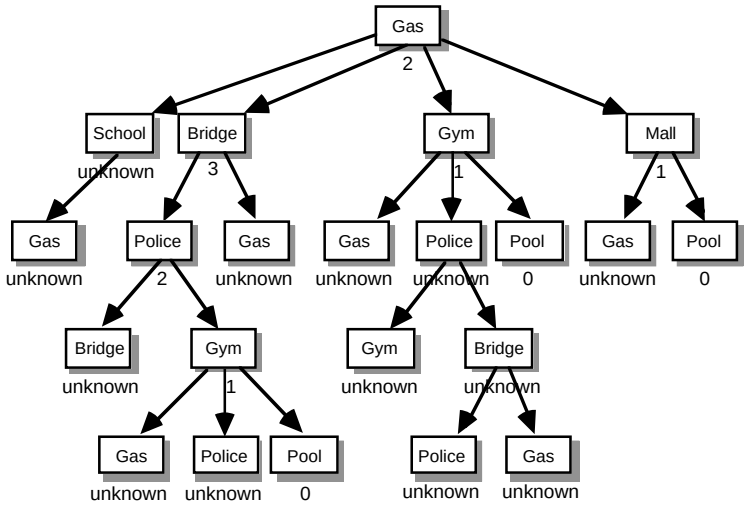
The forward or request-driven computation strategy starts with a state whose value is unknown. It then attempts to ∗nd this state s value by ∗nding the values of its dependent

states. This leads to a recursion, ending at states whose value is known. In pseudo-code, the algorithm can be written as follows:

```
FUNCTION Expand(state): ValT;
BEGIN
      IF Known(state) THEN
         RETURN val_init(state)
      ELSIF NOT visited(state) THEN
         visited(state) := TRUE;
         FOR ALL s_dep ∈ dep(state) DO
            val(s_dep) := Expand(s_dep)
         END;
         visited(state) := FALSE;
         RETURN vgf(state, val);
      ELSE
         RETURN unknown;
      END;
END;

FOR ALL states DO
      visited(s) := FALSE;
END;
val(*rst) := Expand(*rst);
```

A nice feature of this algorithm is that it uses little memory. Only a stack for recursive calls is required. In the worst case, this stack may hold all states in the state space, but typically it will be much less. Usually when *Expand* is called, not all states are evaluated, because only a subset of the states inτuence a state s value. This is advantageous when the value of only a single state is needed. If all state values are required, this procedure must be called for each state. The *dep* and *vgf* functions will then be called exactly once for each unknown state. The number of vgf applications for single-state search can often be further reduced by pruning some dependencies. For example, the alpha-beta algorithm does not examine other moves as soon as a winning move has been found.

Applying this algorithm to our town example in order to ∗nd the ∗distance Gas-Poolτ, results in the following search tree:



Forward search tree

In this example the correct value for ∗distance Gas-Poolτ state is obtained. A disadvantage of forward search can be seen in this example, namely some states are expanded more than once (∗distance Police-Poolτ).

Another disadvantage is that this simple search strategy can only guarantee a ∗xpoint for acyclic graphs, because any dependencies leading to cycles are pruned. It is sometimes possible to transform a problem on a cyclic graph to an analog problem on a acyclic graph. This is often done in order to apply dynamic programming techniques. If cyclic dependencies are unavoidable, a different method for computing a ∗xpoint is needed. One possibility is to store temporary values for the states. Instead of completely pruning a dependency, the temporary value for that state is used and the search reiterates until the state values remain stable. In the worst case, forward search must then store all state values, so maybe it would be better to use backward search instead.

Backward Search

The forward search strategy explained above has a number of disadvantages, for instance some states are expanded multiple times. Another problem is the time spent searching for states whose value can be computed. This can be eliminated by the observation that the only states whose values can change are predecessors of states whose value has just changed. In particular, the only states the vgf should initially be applied to are the predecessors of known states.

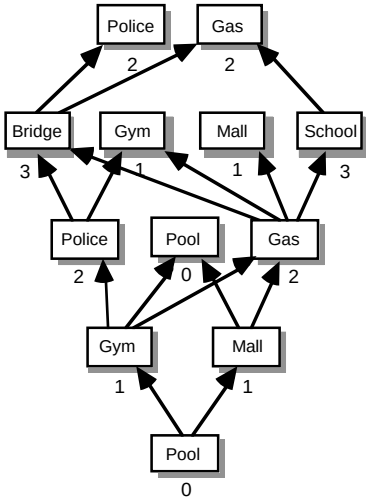
This idea forms the basis of backward or data-driven search. Instead of starting from the unknown nodes and trying to prove values for them, the search starts with the known nodes and tries to prove values for their predecessors. Any predecessor whose value is changed in this fashion may then propagate its value on towards its predecessors, etc. The following algorithm uses a priority queue to hold at most one instance of each state that must be expanded. The priority of the state expansion might be dictated by the partial value order, as this can lower the number of times the vgf function is applied.

```
val := val_init
FOR ALL states DO
      IF Known(state) THEN
         InsertQueue(state)
      END;
END;
WHILE NOT EmptyQueue DO
      GetQueue(state)
      FOR ALL s_pred ∈ dep^-1(state) DO
         val_new := vgf_part(s_pred, state, val);
         IF val  val_new  THEN
            InsertQueue(s_pred)
            val := val_new
         END;
      END;
END;
```

One advantage of this backward approach is the low search overhead. In contrast to forward search, this algorithm needs no stack to keep track of the sequence in which states were expanded, because they will not be contracted again later on.

A disadvantage of backward search is that there is no analogy to the search restrictions possible using forward search. While it would be possible not to place all predecessors in the queue, we have no way of knowing which predecessors lead to the desired state. The required queue size also imposes limitations on the size of the problem.

Applying this algorithm to our town example computes the optimal values for all states from the bottom-up as follows:

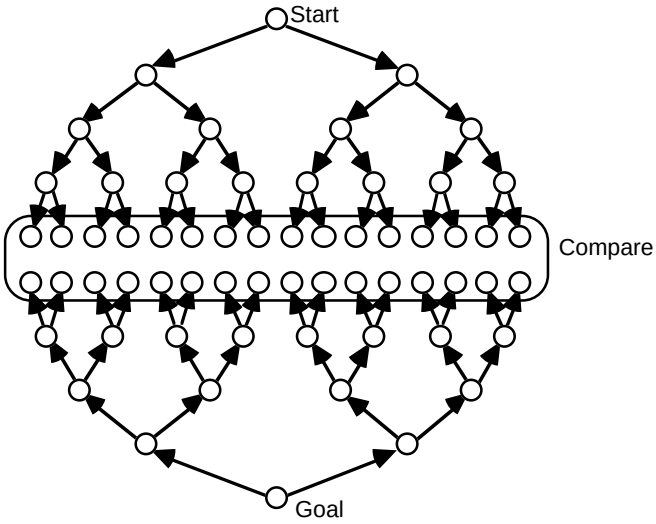


Backward search tree

2.4 Combining Backward and Forward Search

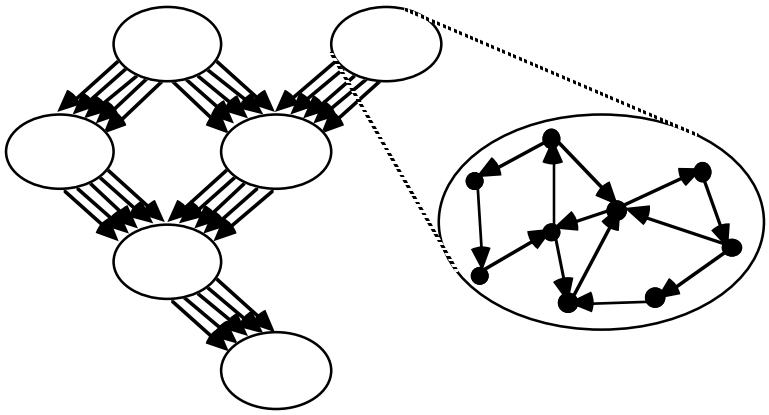
As shown in the preceding sections, forward search uses little memory and backward search is computationally ef∗cient. A plausible thought is to combine these advantages to new computation strategies. This can be done in numerous ways.

Bi-directional search [Pohl 71] is one common method. In order to ∗nd a path from start to goal in a tree with depth *d* and constant branching factor *b*, $O(b^d)$ states must be considered. By simultaneously searching backward from the goal and letting the searches meet halfway, only $O(b^{d/2})$ states are examined. This reduction in tree size is attained by increased memory usage, since all midway states must be stored for comparison purposes. Bi-directional search is ef∗cient when the total number of states is much larger than the number of states at depth *d/2*, otherwise backward search is preferable.



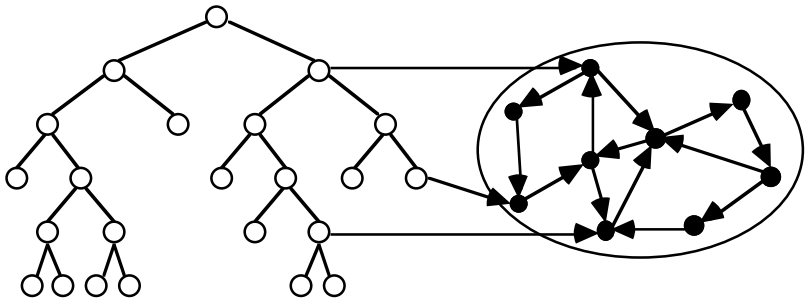
Bi-directional search

A different combination was used for solving Nine Men s Morris. In general the state space graph can be viewed as an acyclic graph of cyclic graphs. The cyclic graphs (compare blow-up in diagram) consist of states that are interdependent, i.e. any two states are dependent on each other via transitive links. Determining the precise structure of the state space graph so that it conforms to this de∗nition may be dif∗cult. If this can be accomplished, a reasonable strategy is to use forward search for the acyclic parts and backward search for the cyclic parts.



Acyclic graph with cyclic subgraphs

A further combination is the branch-and-bound algorithm we use to solve Sliding Tile Puzzles. Branching is forward search and for lower bounding, we simplify the problem by relaxing constraints. The relaxed problem are optimally solved using backward search.



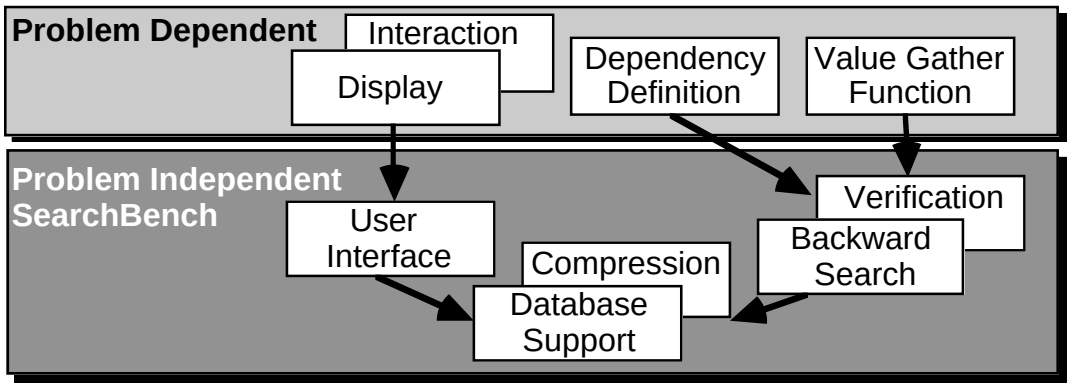
Bounding with relaxed problem

3
SearchBench: Architecture and Interface

In this chapter we introduce the SearchBench. The ∗rst section gives a project overview, describing the main design goals. For the casual user, a typical interactive session is shown. For the programmer, we develop and integrate a simple example into the SearchBench. Finally some standard modules that are useful in conjunction with the SearchBench are discussed.

3.1 Project Overview

The last chapter de∗ned a common model for exhaustive search problems. The advantage of this model is that it hides many implementation speci∗c details, for instance ∗le I/O, value gather application sequence or the data structures used to store states and their values. This greatly simpli∗es the understanding of the basic ideas and algorithms. One of the main design goals for the SearchBench was to provide a tool for exhaustive search that hides as many of these implementation details as possible. This leaves the user, and especially the programmer, free to concentrate on the problem at hand.



SearchBench Architecture

As shown in the diagram, a project can be divided into problem-dependent and independent aspects. There are three problem-dependent aspects:

- Display
- State space graph
- Value gather function

These must be implemented by the programmer and plugged into the SearchBench, which provides the algorithms and data structures. The SearchBench was designed with the following key goals in mind:

- *Rapid Prototyping:* A project can divided into two phases, development and application. In the development phase, code is written and debugged. The SearchBench is useful in this phase because it provides the problem-independent algorithms in tested and debugged condition. This dramatically reduces the coding that remains to be done. After the development phase, the code is put to practical use. In this phase one may argue that it is inef∗cient to use the SearchBench for the actual computation, because of the overhead this entails. However, depending on the size of the problem, it may be more ef∗cient to use the SearchBench, because no CPU cycles are lost during the re-implementation.
- *Simple Interface:* An important application domain for backward search is solving games and puzzles. This makes the SearchBench predestined to be used as a teaching tool, because games are highly motivating to students and the problem domain is usually small and uncomplicated. Its use as a teaching tool encourages a simple interface, so that students are not scared off after their ∗rst session. Therefore, the SearchBench offers an environment with only two windows, a text and a problem display. Because students also write their own applications, the programming interface, i.e. the procedures that must be implemented, must also be kept to a minimum.
- *Generality and Portability:* In spite of the simple programming interface, the problems that can be solved should not be restricted to the aforementioned games and puzzles. The implementations in [Müller 93] demonstrate that other problems can be solved as well, although not all exhaustive search problems are equally well suited for the SearchBench. The SearchBench was originally implemented on the Apple Macintosh in Pascal. It has since been ported to C with an even simpler command line interface [Wirth 94] and should therefore run on most machines.

3.2 User Interface

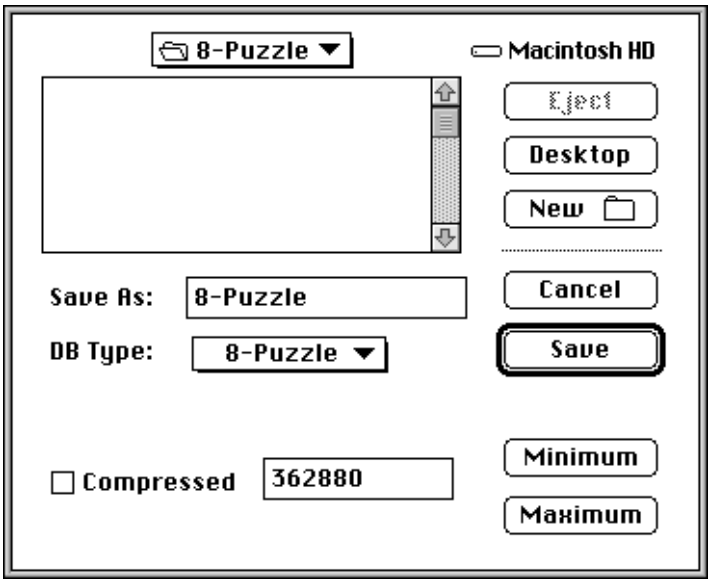
In this section we describe an interactive session on the SearchBench. We ∗rst compute and verify the 8-Puzzle database, then examine the database statistics and most dif∗cult position.

Database Initialization

To compute a database, it must ∗rst be initialized. The initialization dialog box asks for three inputs:

- *Database Name:* The ∗le name the database is to be stored under.
- *Database Type:* Typically a search problem can be split into numerous, acyclic dependent subproblems, each of which is modeled by its own database. This makes it necessary to specify the subproblem being initialized.
- *Desired Compression:* To save disk space, it is possible to compress the database ∗le. Lempel-Ziv-Welch compression [Welch 84] is implemented in the SearchBench. This algorithm splits a ∗le into multiple segments and then independently compresses these segments. The size of the segments can be speci∗ed in the dialog

box. Larger segments tend to yield higher compression ratios, while smaller segments result in faster access. The segment length may be varied in the range given by the Minimum and Maximum buttons.



Initialization dialog

Database Computation and Veri∗cation

A database may depend on other databases, so to start the computation the user ∗rst selects all support databases and ∗nally the initialized database to be computed. If support databases are omitted, none of their values will be propagated.

During any processing phase, the SearchBench shows a progress display. It summarizes the elapsed run-time and number of positions calculated so far. It also estimates the remaining run-time. This estimate is based on an extrapolation of the number of states computed so far. This is not always precise, because there is no way of knowing the total number of states that will be expanded and ∗le I/O tends to bias the estimate especially at the beginning and end of the computation.

Progress display

To verify the completed database, it and the support databases must again be selected. The veri∗cation and calculation use different algorithms. Veri∗cation checks that the value of a state is consistent with the values of its successor states. This does not necessarily mean they are correct, because the initialization may be wrong. If inconsistencies are found, they are output to the Log window.

Database Compression

Two compression methods are available to the user. Any database can be compressed or expanded with a Lempel-Ziv-Welch (LZW) algorithm. Most SearchBench features work with either LZW or standard ∗les. However, some algorithms, for instance retrograde analysis, do not accept LZW main ∗les for ef∗ciency reasons.

A second form of compression is offered by the ∗Move Heuristicτ menu option. It is based on the observation that optimal moves can often be accurately predicted. Instead of storing the state values, the database stores the index of the ∗rst optimal move found in a heuristically ordered move list. If the heuristic order is good, the database will contain many runs where the optimal move was correctly predicted. This in turn makes Lempel-Ziv-Welch more effective. Even if no heuristic is known, substantial savings occur if the value range is larger than the move range, for instance 33 different values occur in the 8-Puzzle, but there are only four different moves.

Database Interaction

SearchBench provides a number of ways to extract information from the ∗nal database. The simplest is the statistics output. SearchBench already implements a simple form of statistics, namely counting the number of occurrences of each value. If this is executed for the 8-Puzzle database, the output shown on the right will be written to the Log window. This shows that there are two hardest states, requiring 31 moves to solve. For some problems additional statistics may be useful, so SearchBench also provides a means of installing user-de∗ned statistic routines (see section 3.3).

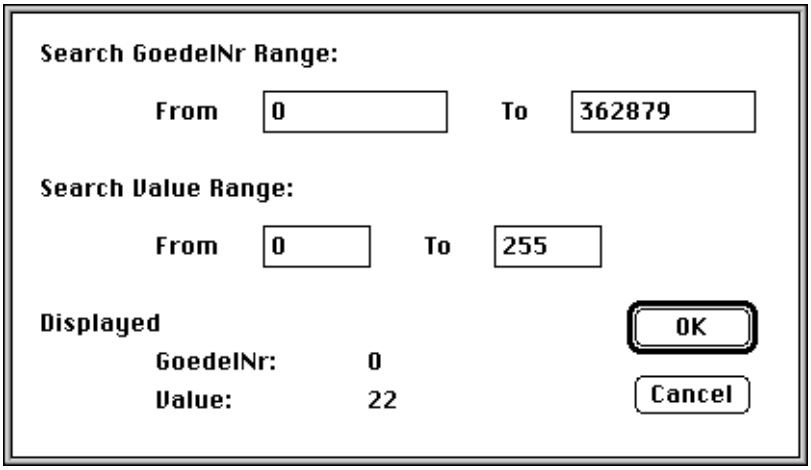
By choosing ∗Select DBτ in the menu bar and selecting our database ∗le, SearchBench opens the problem window for this database. Selecting ∗Findτ then displays a dialog box, where a range of states and state values to search for can be selected. To ∗nd the hardest 8-Puzzle state, we specify a search in the entire state range for a state value 31. SearchBench then ∗nds and displays such a state in the problem window. To ∗nd the second, simply choose ∗Find Nextτ. To see an optimal sequence, we repeatedly select ∗Optimal Stepτ. The user can also enter a move by clicking on the tile to be moved, dragging it and releasing the mouse button.

Log

Statistics for 8-Puzzle

Value	Amount
0	1
1	2
2	4
3	8
4	16
5	20
6	39
7	62
8	116
9	152
10	286
11	396
12	748
13	1024
14	1893
15	2512
16	4485
17	5638
18	9529
19	10878
20	16993
21	17110
22	23952
23	20224
24	24047
25	15578
26	14560
27	6274
28	3910
29	760
30	221
31	2
255	181440
Total:	362880

Default statistics

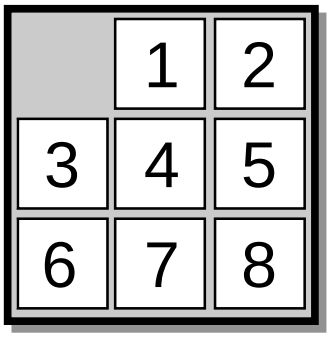


Find dialog

3.3 Programmer Interface

This section describes the programmer s interface to the SearchBench. The procedures are grouped by function. In order to illustrate the use of the procedures, we develop sample code for the 8-Puzzle, a smaller version of the classic Sam Loyd puzzle.

The 8-Puzzle is played on a 3x3 board, in which eight tiles are placed. A move consists of sliding one of the eight tiles into the remaining space. The object of the puzzle is to reach the goal state shown on the right.



Goal state

Problem De∗nition Routines

These routines de∗ne the state space and state dependencies.

```
TYPE
      DBT = INTEGER;
      GoedelNrT = LONGINT;
      PosT =      RECORD
                     db: DBT;
                     goedelNr: GoedelNrT;
                  END;
      PosArrT =   RECORD
                     nrElements: INTEGER;
                     data: ARRAY [0..MaxFanIO] OF PosT;
                  END;
      ValT = ARRAY [0..MaxValueLength] OF BYTE;

FUNCTION DBValid(db:DBNrT; VAR name:String; VAR needsDef:BOOLEAN): BOOLEAN;
FUNCTION UseDefFile(db:DBNrT; ∗leData:StringPtr): BOOLEAN;
PROCEDURE DBInfo(db:DBNrT; VAR nrStates:LONGINT; VAR valueSize:INTEGER);

PROCEDURE InitialValue(pos:PosT; VAR length:LONGINT; VAR value:ValT);

PROCEDURE BackupPos(pos:PosT; wantedDB:DBNrT; VAR results:PosArrT);
PROCEDURE ForwardPos(pos:PosT; VAR results:PosArrT);
```

The state space of a problem can be split into numerous databases, which must be numbered contiguously beginning with zero. *DBValid* returns true for each valid database

number and suggests a ∗lename. Our sample project only has one database, so the database number must be zero.

```
FUNCTION DBValid(db:DBNrT; VAR name:String; VAR needsDef:BOOLEAN): BOOLEAN;
BEGIN
      name := ∗Loyd Puzzleτ;
      needsDef := FALSE;
      DBValid := (db = 0);
END;
```

In some cases a database may require additional data, for example we may later wish to write more general sliding tile routines, that take the size of the board as an input. While it would be possible to de∗ne a unique database number for each possible board size, a more elegant approach is to read in a parameter ∗le de∗ning the length and breadth. This is possible with the *UseDefFile* function. If true is returned in the *needsDef* parameter, *UseDefFile* will be called with a pointer to the parameter ∗lename.

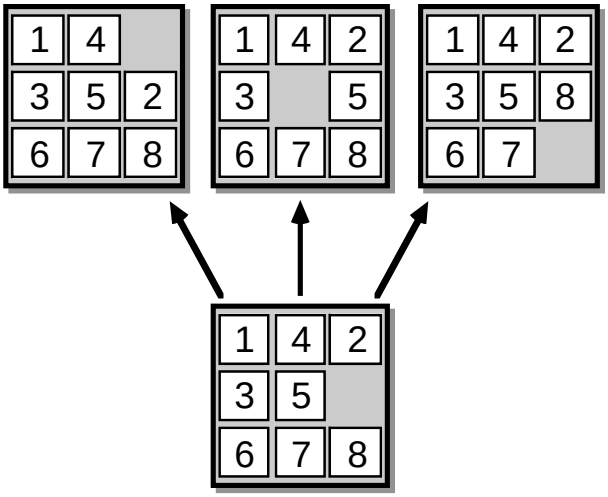
DBInfo returns additional information about the database, namely the number of states in the database and the number of byte needed to represent the state values. Clearly, there are *9!* different ways of placing eight tiles on the board, so this is *nrStates*. If no state is further than 255 moves from the goal state, one byte is suf∗cient to store this information.

```
CONST
      StateSpaceSize = 362880; {=9!}

PROCEDURE DBInfo(db:DBNrT; VAR nrStates:LONGINT; VAR valueSize:INTEGER);
BEGIN
      nrStates := StateSpaceSize;
      valueSize := 1;
END;
```

InitialValue de∗nes an initial value for all states. In the case of the 8-Puzzle, the state value reτects the number of moves from the goal state. Therefore, *value* is zero for all goal states. The maximal one byte value of 255 is assigned to all other states, signifying that the number of moves needed to reach them is unknown. As in this example, the initial database often contains many states with identical values. Therefore it is inef∗cient to call *InitialValue* for each state. In these cases, the initialization can be accelerated by using the *length* parameter to indicate the number of sequential positions with the same value.

```
CONST
      Goal = 46233; {Goedelnr of the goal state}

PROCEDURE InitialValue(pos:PosT; VAR length:LONGINT; VAR value:ValT);
BEGIN
      IF pos.db = 0 THEN
         IF pos.goedelNr < Goal THEN
            length := Goal-pos.goedelNr;
            value[0] := 255;
         ELSIF pos.goedelNr = Goal THEN
            length := 1;
            value[0] := 0;
         ELSE
            length := StateSpaceSize-pos.goedelNr;
            value[0] := 255;
         END;
      END;
END;
```

ForwardPos and *BackupPos* de∗ne the dependencies between states in the state space. In general the state dependencies can be described with a directed graph. *BackupPos* then generates all states that are dependent on a given state (predecessors) and *ForwardPos* generates all states a given state is dependent on (successors). The diagram shows how *BackupPos* is applied to a state, generating three predecessors.



A state and its three predecessors

More speci∗cally, *BackupPos* is passed a state and *wantedDB* parameter and returns all predecessor states in the *wantedDB* database. For the 8-Puzzle, all predecessor states lie in database number zero, so predecessors are only generated if *wantedDB* is zero.

```
CONST
      BoardT = ARRAY [0..8] OF INTEGER;
      DirT = (north, east, south, west);

PROCEDURE BackupPos(pos:PosT; wantedDB:DBNrT; VAR results:PosArrT);
VAR
      board: BoardT;
      empty: INTEGER;
      dir: DirT;
BEGIN
      IF wantedDB = 0 THEN
         NumToBoard(pos, board);
         empty := 0;
         WHILE board[empty]  0 DO
            empty := empty+1;
         END;
         results.nrElements := 0;
         FOR dir := north TO west DO
            IF Adj[empty, dir]  border THEN
               board[empty] := board[Adj[empty, dir]];
               board[Adj[empty, dir]] := 0;
               BoardToNum(board, results.data[results.nrElements]);
               results.nrElements := results.nrElements+1;
               board[Adj[empty, dir]] := board[empty];
               board[empty] := 0;
            END;
         END;
      END;
END;
```

BackupPos uses the procedures *NumToBoard* and *BoardToNum* to translate between the internal representation of the board, an array with nine elements, and the external representation for the SearchBench, a number. A more detailed description of these procedures is given in section 3.4.

Because the 8-Puzzle is described by an undirected graph, i.e. a move can be undone with a second move, *ForwardPos* and *BackupPos* generate the identical set of states. Note that *ForwardPos* must return all successors, whereas *BackupPos* only returns predecessors in the *wantedDB* database. Another difference is the importance of state ordering in *ForwardPos*. While this is irrelevant for *BackupPos*, state ordering in

ForwardPos is used for database compression in the ∗Move Heuristicτ menu option. The ∗betterτ states should be in the lower *results* indices. In our example no special move ordering heuristic is implemented.

```
PROCEDURE ForwardPos(pos:PosT; VAR results:PosArrT);
BEGIN
      BackupPos(pos, 0, results);
END;
```

Value Propagation Routines

These routines handle the information propagation in the state space. They are independent of state dependencies and are solely concerned with how the value of a given state is computed from the values of its dependent states.

```
TYPE
      ValArrT = ARRAY [0..MaxFanIO] OF ValT;

VAR
      changed: BOOLEAN;
      passID: INTEGER;

PROCEDURE StartPass(mainDB:DBT; saveData:String);
PROCEDURE InterruptPass(VAR saveData:String);

FUNCTION Backupable(db:DBT; VAR value:ValT): BOOLEAN;
FUNCTION AnotherPass(db:DBT; VAR doFinalConvert:BOOLEAN): BOOLEAN;

FUNCTION InfoProp(succ, pos:PosT; succVal:ValT; VAR value:ValT): BOOLEAN;
PROCEDURE InfoGather(pos:PosT; VAR value:ValT; VAR successors:PosArrT;
                     VAR succVals:ValArrT; VAR best:INTEGER);

FUNCTION FinalConvert(VAR value:ValT): BOOLEAN;
```

StartPass is called at the beginning of the database calculation. The *mainDB* parameter contains the database number of the database to be computed. The *saveData* string is used to restart interrupted calculations. Normally it will contain the empty string. For the 8-Puzzle, *StartPass* initializes two variables. *PassId* indicates which state values can be propagated and *changed* indicates whether any state values must be propagated in the next pass. For computation intensive databases it is useful to be able to interrupt the calculation and continue later. When a calculation is restarted, *StartPass* extracts the *changed* and *passID* information from the *saveData* string.

```
PROCEDURE StartPass(mainDB:DBT; saveData:String);
VAR
      i: INTEGER;
BEGIN
      changed := (Pos('TRUE', saveData) > 0);
      passID := 0;
      FOR i := 1 TO Length(saveData) DO      {String to number conversion}
         IF (saveData[i] >= '0') AND (saveData[i] <= '9') THEN
            passID := 10*passID+ord(saveData[i])-ord('0');
         END;
      END;
END;
```

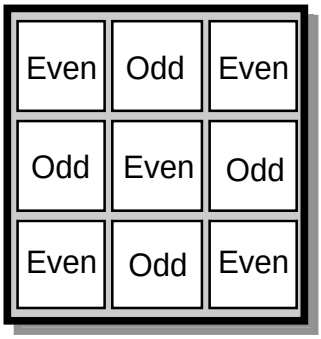
To support interruption of database calculations, *InterruptPass* must also be implemented. This procedure returns a string containing the private data needed to restart the computation where it was left off. In the case of the 8-Puzzle the private data we need to save is *passID* and *changed.*

```
PROCEDURE InterruptPass(VAR saveData:String);
BEGIN
      IF changed THEN
         saveData := concat(*AnotherPass: TRUE  PassID: *, StringOf(passID:1));
      ELSE
         saveData := concat(*AnotherPass: FALSE  PassID: *, StringOf(passID:1));
      END;
END;
```

The *Backupable* procedure determines which state values can be propagated. This is any state whose value is equal to *passID*. A simple implementation for *Backupable* can be written as follows:

```
FUNCTION Backupable(db:DBT; VAR value:ValT): BOOLEAN;
BEGIN
      Backupable := (value[0] = passID);
END;
```

However, the 8-Puzzle has the interesting property, that the parity of the solution length can be determined by the position of the space. The space can only return to the same position in an even number of moves. Similarly, it can only move to an adjacent position in an odd number of moves. Since the space must ultimately move to the upper left corner, the diagram at right shows the parity of the possible solution lengths.



Parity of solution length depending on the space position

Because of this parity argument, any solution sequence for a state can never be improved by only one move. Therefore any state which has a value equal to *passID* or *passID+1* can be propagated, leading to an improved *Backupable* procedure.

```
FUNCTION Backupable(db:DBT; VAR value:ValT): BOOLEAN;
BEGIN
      Backupable := (value[0] = passID) OR (value[0] = passID+1);
END;
```

After completing a pass, i.e. propagating all states with value *passID* or *passID+1*, the *AnotherPass* procedure is called to determine whether the calculation is *nished. The *changed* parameter, which is set in *InfoProp*, determines whether an additional pass is needed. Since two values are propagated per pass, *passID* is incremented by two at the end of a pass. When *AnotherPass* returns false, the *doFinalConvert* parameter indicates whether the *FinalConvert* procedure must be called. *FinalConvert* can be used to delete unneeded information from the state values at the end of the computation. It is unused for the 8-Puzzle, but important for two-player value propagation.

```
FUNCTION AnotherPass(db:DBT; VAR doFinalConvert:BOOLEAN): BOOLEAN;
BEGIN
      passID := passID+2;
      doFinalConvert := FALSE;
      AnotherPass := changed;
      changed := FALSE;
END;
```

The two value propagation routines, *InfoGather* and *InfoProp*, implement the value gather function (vgf) and partial value gather function (vgf_{part}).

Given a state, its value, all dependent states and their values, *InfoGather* computes the new state value and best state transition, i.e. which successor state represents the best move. *InfoGather* is used in conjunction with *ForwardPos. ForwardPos* determines the states dependent on a given state and then *InfoGather* determines the state s value from the dependent state values.

Since the 8-Puzzle values represent the number of moves needed to reach the goal state, the best move is to choose the successor with the lowest value. The state value is then one move more than the best successor. Exceptions to this rule are when all successors values are unknown (255) in which case the state value is also unknown, or when the state is the goal state in which case the value is zero.

```
PROCEDURE InfoGather(pos:PosT; VAR value:ValT; VAR successors:PosArrT;
                     VAR succVals:ValArrT; VAR best:INTEGER);
VAR
      i: INTEGER;
BEGIN
      best := -1;
      IF value[0] <> 0 THEN
         value[0] := 255;
      END;
      FOR i := 0 TO successors.nrElements-1 DO
         IF succVals[i, 0]+1 < value[0] THEN
            best := i;
            value[0] := succVals[i, 0]+1;
         END;
      END;
END;
```

Given the value of a successor state, *InfoProp* incorporates this information into the state value and returns true if the state value has changed. The *changed* variable is updated to reτect if further passes will be needed, i.e. if the new state value cannot be propagated immediately, it must be propagated in a later pass.

```
FUNCTION InfoProp(succ, pos:PosT; succVal:ValT; VAR value:ValT): BOOLEAN;
BEGIN
      InfoProp := FALSE;
      IF value[0] > succVal[0]+1 THEN
         value[0] := succVal[0]+1;
         changed := changed OR (Value[0] > passID+1);
         InfoProp := TRUE;
      END;
END;
```

Display Routines

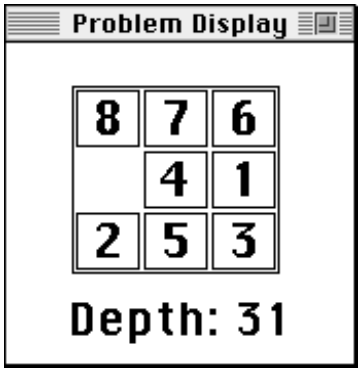
These routines are responsible for board display and move tracking.

```
VAR
      position: PosT;
      value: ValT;             {READ ONLY}
      dBType: DataT;           {READ ONLY}
      displayarea: Rect;       {READ ONLY}

PROCEDURE DrawState(redraw, log:BOOLEAN);
PROCEDURE Mouse(what:MouseT; downClick, lastMouse, thisMouse:Point);

FUNCTION OpenDialog(VAR interact:DialogPtr): BOOLEAN;
PROCEDURE HandleEvent(theItem:integer; VAR theEvent:EventRecord);
PROCEDURE CloseDialog(VAR interact:DialogPtr);

FUNCTION StartSelection(VAR start, length:LONGINT): BOOLEAN;
FUNCTION Select(goedelNr, length:longint; aValue:ValT): LONGINT;
```

The display routines use global variables to pass information between the SearchBench and the problem-dependent code. Either one of these may set the *position* variable to indicate which position should be displayed in the problem window. The other three variables may only be set by the SearchBench. *DBType* indicates whether *position* was found in a database and, if so, whether the database contains the real value of the position or the best move number. The *value* variable contains the value of *position* if it was found. *Displayarea* holds the size of the current display window.

DrawState is responsible for displaying a graphical state view. For this purpose the procedure is passed a *redraw* and *log* parameter. *Redraw* indicates whether the entire display must be redrawn or if it is only necessary to draw any changes that occurred since the last display update. If *log* is set, a textual state view should also be written to the Log window. The sample procedure below does not use *redraw*. This makes for a simple, albeit jittery, display.

```
PROCEDURE DrawState(redraw, log:BOOLEAN);
VAR
      i: INTEGER;
      board: BoardT;
      r: Rect;
BEGIN
      NumToBoard(position, board);
      IF log THEN
         WriteToLog(board);
      END;
      EraseRect(displayarea);
      DrawFrame(displayarea);
      FOR i := 0 TO 8 DO
         GetTileRect(displayarea, i, r);
         DrawBox(r, board[i]);
      END;
      DrawValue(displayarea, value[0]);
END;
```



The effect of *Drawstate*

To interact with the state display, the *Mouse* procedure must be implemented. *Mouse* is called whenever a mouse event (down, drag or up) occurs. The procedure provides the necessary visual feedback and updates the *position* variable.

```
PROCEDURE Mouse(what:MouseT; downClick, lastMouse, thisMouse:Point);
VAR
      board: boardT;
      tilePos, newTilePos: integer;
      r: Rect;
BEGIN
      NumToBoard(position, board);
      tilePos := GetTile(displayarea, downClick);
      IF (tilePos >= 0) AND (board[tilePos]  0) AND (what  down) THEN
         GetRectPos(displayarea, lastMouse, downClick, r);
         DrawBox(r, board[tilePos]);
         IF what = drag THEN
            GetRectPos(displayarea, thisMouse, downClick, r);
            DrawBox(r, board[tilePos]);
         ELSE { what = up }
            newTilePos := GetTile(displayarea, thisMouse);
            IF (newTilePos >= 0) AND (board[newTilePos] = 0) THEN
               board[newTilePos] := board[tilePos];
               board[tilePos] := 0;
               BoardToNum(board, position);
            END;
            DrawState(displayarea, FALSE, FALSE);
         END;
      END;
END;
```

In the default problem display, these two procedures operate on a simple window with no buttons or other items. If the user desires a more fancy display, she can de∗ne her own dialog window using *OpenDialog* and *CloseDialog*. When these routines are called, the user application passes SearchBench a *DialogPtr* to the user s window. This window can then include radio buttons, check boxes and all other indispensable tools. If dialog items are included in the window, *HandleEvent* will be called each time such an item is activated.

In addition to changing the display interactively with the mouse or dialog items, it is possible to search for states with speci∗c characteristics and have them displayed. This is done with the ∗Findτ menu item. In its default implementation the user can select a range of states and a range of values to be searched for. When a matching state is found in the database, it is displayed. The user may wish to de∗ne more application dependent search mechanisms. For instance in chess, we may wish to search for states where a king is outside the square of his opponent s passed pawn. In this case the *StartSelection* and *Select* procedures must be overridden. In *StartSelection* the user may display her own input dialog and must return a range of states. SearchBench then passes all values of the states in this range to *Select*. Any non-negative number returned by *Select* is taken to be the selected state and displayed.

Miscellaneous Routines

User-de∗ned Menu

In addition to the prede∗ned menu items, the user can de∗ne her own problem-speci∗c menu item for added functionality or test purposes.

```
PROCEDURE ProblemMenu(VAR title, list:Str255);
PROCEDURE TestMenuCall(item:INTEGER);
```

ProblemMenu returns the *title* and *list* of items the user wants installed. The format is identical to the one used by the Macintosh operating system. Whenever an item is selected, *TestMenuCall* will be called with the index of the selected menu item.

Output and User alert

SearchBench also provides prede∗ned procedures for writing to the Log window and displaying Alert dialogs. Writing to the Log window is required in *DrawState*, when the *log* parameter is set, or for statistics and user-de∗ned menu calls. An Alert should be generated whenever errors occur in the problem-dependent code, for instance while parsing a ∗le in *UseDefFile*.

```
PROCEDURE LogWrite(aString:String);
PROCEDURE LogWriteTime;
PROCEDURE LogWriteLn;

FUNCTION ShowAlert(module, error:String): Button;
```

Statistics

After completing the database calculation, it is interesting to examine the value distribution. This can be done with the ∗Statisticτ menu item. In its default implementation, this command outputs the value distribution of the ∗rst value byte. If more detailed

statistics are necessary, it is possible to override the standard statistics with the following three routines:

```
FUNCTION StartStat(db:DBT; dBType:DataT): BOOLEAN;
PROCEDURE StatData(goedelNr:GoedelNrT; value:ValT; length:LONGINT);
PROCEDURE EndStat;
```

SearchBench calls *StartStat* with the number and type of the database the statistics are to be performed for. *StartStat* can then prompt the user for the problem-speci∗c type of statistics she wants generated. If *StartStat* returns true, *StatData* is repeatedly called with database values. After the entire ∗le has been scanned, *EndStat* is called to output the results to the Log window.

3.4 Indexing Functions

Many games, puzzles and other problems internally represent their states with an array. In the case of the 8-Puzzle the array contains the position of the tiles on the 3x3 board. SearchBench requires the states to be represented by numbers. It uses these numbers as an index of where the state s value should be stored in the ∗le. Therefore all problem-dependent code will need procedures for converting between the two representations. In our implementation of the 8-Puzzle we used the *BoardToNum* and *NumToBoard* procedures to do these conversions. Since all SearchBench applications require similar routines, it seems reasonable to offer general purpose support code. The ∗ConversionSupportτ module offers the following procedures:

```
PROCEDURE Insert(amount, what:INTEGER; VAR index:GoedelNrT; arr:ArrPtr;
                 VAR len:INTEGER);
PROCEDURE Extract(amount, what:INTEGER; VAR index:GoedelNrT; arr:ArrPtr;
                  VAR len:INTEGER);
```

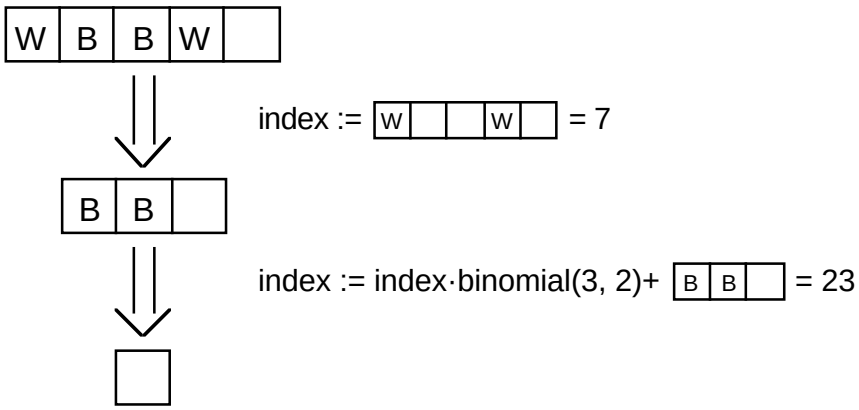
The *amount* and *what* parameters de∗ne the number and type of elements to be inserted or extracted from the array. *Index* is the converted number, i.e. SearchBench representation. The internal array is addressed via a pointer to the array and the array length. For the 8-Puzzle, the conversion routines can then be written as follows:

```
PROCEDURE BoardToNum(board:BoardT; VAR pos:PosT);
VAR
      i, length: INTEGER;
BEGIN
      pos.db := 0;
      pos.goedelNr := 0;
      length := 9;
      FOR i := 1 TO 8 DO
         Extract(1, i, pos.goedelNr, board, length);
      END;
END;

PROCEDURE NumToBoard(pos:PosT; VAR board:BoardT);
VAR
      i, length: INTEGER;
BEGIN
      IF pos.db = 0 THEN
         board[0] := 0;
         length := 1;
         FOR i := 8 TO 1 BY -1 DO
            Insert(1, i, pos.goedelNr, board, length);
         END;
      END;
END;
```

It remains to be explained how *Insert* and *Extract* function. The basic problem can be reduced to converting an array of *n-k* zero s and *k* one s into a number and vice versa. Clearly there are *binomial(n, k)* different ways of placing the one s. We must ∗nd a function that maps a given con∗guration to a number in this range. If the ∗rst position in the array is a zero, then *binomial(n-1,k)* ways of placing the one s remain. A possible enumeration is to map con∗gurations where the ∗rst position is zero to the ∗rst *binomial(n-1,k)* numbers and the con∗gurations where the ∗rst position is one to the last *binomial(n-1,k-1)* numbers. This leads to the following code:

```
index := 0;
FOR pos := 1 TO n DO
      IF arr[pos] = what THEN
         index := index+Binomial[n-pos, k];
         k := k-1
      END;
END;
```

In general, there are different types of pieces in the array, so this procedure must be applied repeatedly. The diagram below shows how an array containing two white and two black pieces is converted to a number. When a piece is extracted, the array is reduced in size. Also note that the index is multiplied by the size of the range before adding the next number.



Converting an array to an index

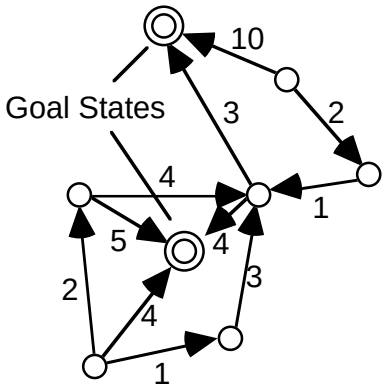
3.5 Value Gather Functions

Because many different problems often use the same value gather functions (vgf), the one-player and two-player vgf are included with the SearchBench. Both can be written as partial vgf s with the use of almost no additional memory. For a look at a greater variety of value gather functions, including some that cannot be ef∗ciently rewritten, see [Müller 93].

One-Player Value Gather Function

This propagation rule is used for problems where the shortest distance between two states is needed. Such a value gather function has already been described and implemented in section 3.3 for the 8-Puzzle. In general, the problem is posed on a directed graph, where each edge has a given cost or length greater zero. A subset of the

vertices are de∗ned as goal states and assigned cost zero. The question is to ∗nd the minimal cost from any state to the nearest goal state.



Directed graph with edge costs

The cost of reaching a goal state via a speci∗c neighbor can be recursively de∗ned as the cost of traversing the edge to that neighbor plus the cost of getting to a goal state from there. Since the cheapest neighbor is chosen, the cost of the state is the minimum cost over all possible neighbors. The range of the val function is [unknown, 0, 1, , MaxCost], where MaxCost is the value of the state with maximum cost. The vgf can thus be written as follows:

$$\mathrm{val}_{\mathrm{init}}(s) = \begin{cases} 0 & \text{if s is a goal state} \\ \mathrm{MaxCost} & \text{otherwise} \end{cases}$$

$$\mathrm{vgf}(s,\ \mathrm{val})(x) = \begin{cases} 0 & \text{if } \mathrm{val}(x) = 0 \\ \min\limits_{s_{dep} \in \mathrm{dep}(s)} (\mathrm{val}(s_{dep}) + \mathrm{cost}(s, s_{dep})) & \text{if } s = x \\ \mathrm{val}(x) & \text{otherwise} \end{cases}$$

This can be rewritten as a partial vgf without changing the value range or $\mathrm{val}_{\mathrm{init}}$. The idea is to change the value of *s* only if the path via the neighbor s_{dep} leads to a lower cost than any other path seen thus far.

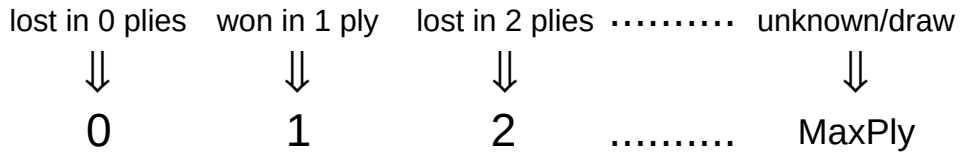
$$\mathrm{vgf}_{\mathrm{part}}(s,\ s_{\mathrm{dep}},\ \mathrm{val})(x) = \begin{cases} \min(\mathrm{val}(s),\ \mathrm{val}(s_{dep}) + \mathrm{cost}(s, s_{dep})) & \text{if } s = x \\ \mathrm{val}(x) & \text{otherwise} \end{cases}$$

Two-Player Value Gather Function

Chapter 2 describes the two-player value propagation if only win, loss and draw values are needed. In general it is also of interest to know the number of moves needed to win or lose. In this case, the simple three-valued range (loss, draw, win) must be extended. The draw value remains unchanged but the wins and losses have depth information added.

Without loss of generality we can assume that the last player to move wins or draws. If the last player to move would lose then there is no point in her making the move, so the previous player makes the last move and wins. This has the consequence that the

value of a position for the player to move is either won in an odd number of plies or lost in an even number. Therefore the value range can be mapped to the positive integers as follows:



Mapping the value range to the positive integers

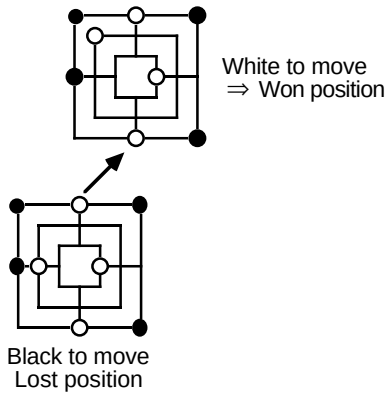
The val function is initialized to contain zero for all immediately lost positions and MaxPly for all others. The unknown and drawn states are mapped to the same value because if the unknown states are not proven a win or loss during the computation, then by default they are drawn.

$$\mathrm{val}_{\mathrm{init}}(s) = \begin{cases} 0 & \text{if s is lost} \\ \mathrm{MaxPly} & \text{otherwise} \end{cases}$$

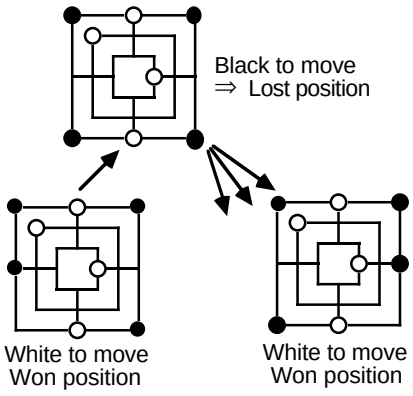
The value gather function attempts to maximize a state s value. If the state is won, the vgf tries to minimize the number of plies needed, whereas if the state is lost, the vgf prefers to maximize the number of plies. The vgf changes the value of state *s* as follows:

$$\mathrm{vgf}(s,\ \mathrm{val})(s) = \begin{cases} 1 + \min\limits_{q \in \mathrm{dep}(s) \wedge (\mathrm{val}(q) = \mathrm{loss})} \mathrm{val}(q) & \text{if } \exists\limits_{q \in \mathrm{dep}(s)} \mathrm{val}(q) = \mathrm{loss} \\ 1 + \max\limits_{q \in \mathrm{dep}(s)} \mathrm{val}(q) & \text{if } \forall\limits_{q \in \mathrm{dep}(s)} \mathrm{val}(q) = \mathrm{win} \\ \mathrm{MaxPly} & \text{otherwise} \end{cases}$$

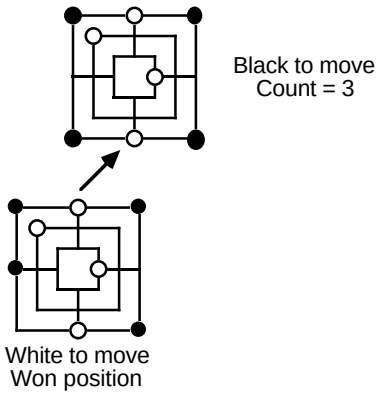
The two-player $\mathrm{vgf}_{\mathrm{part}}$ function is more complex than its one-player counterpart. If a state is lost in *n* plies, then all predecessors must be won in at most *n+1* plies. If a state is won in *m* plies, all predecessors are potential losses (*>m* plies) for the opponent. They are only true losses if all their successors are also won for the player. The drawn states will not influence the value of predecessor states, i.e. the predecessors will never be shown to be won or lost because of a drawn successor. Therefore these states are not propagated, leading to substantial savings.



Loss backup



Win backup

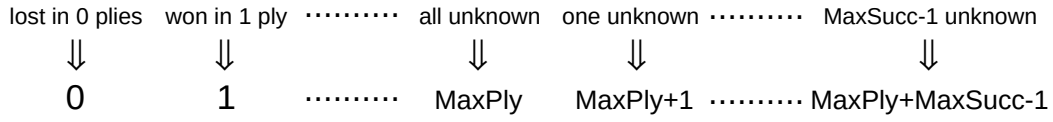


Using the count

The previous diagram suggest checking all successors when propagating a win. This would be inef∗cient. Assuming the predecessor will eventually be proven a loss, all successors will send their value to this state. Each time a value is received, the state would then check the values of all successor to see if it is now a proven loss. This can be more easily accomplished by the use of an additional count ∗eld (diagram at right). The ∗rst time a potential loss is encountered, its count ∗eld is initialized to the number of successor states that have not yet sent their value. Then each time a won value is backed into this state, the count is decremented by one. If the count reaches zero, all successors are wins, and the state is a loss. This assumes that each state does not propagate its value more than once to all predecessors, otherwise a predecessor s count might be decremented twice by the same state.

This explains how vgf_{part} determines the wins, losses and draws. It does not describe how the depth information is updated. If a lost position with depth *d* is propagated, then its predecessors are certainly wins. In addition they can be set to be wins in depth *d+1*, unless they already have a shorter win, in which case the distance information is left unchanged. Won states are more dif∗cult to propagate. If a state s count ∗eld goes to zero, the state is lost and the depth should be as high as possible in order to prolong the ordeal. Therefore, we must store the maximum depth seen thus far along with the count.

We can avoid storing this maximum depth by placing greater restrictions on the state expansion sequence. If we ∗rst propagate the losses in zero, then the wins in one, losses in two, etc. then we can guarantee that the successor state that decrements a state s count to zero is also one with the maximum depth. Then no intermediate maximum must be stored, rather we use the last successor s depth plus one. Thus the value range must only be extended to include the count ∗eld. Since the count is only needed if a state s value is still a potential draw, it is suf∗cient to add MaxSucc-1 values to indicate that one, two up to MaxSucc-1 successors have not yet been seen.



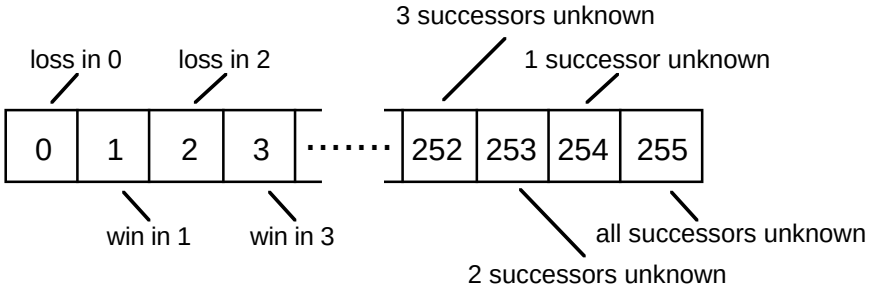
Mapping game values to the positive integers

The initialization remains unchanged and vgf_{part} changes the value of state *s* as follows:

$$\text{vgf}_{\text{part}}(s,\ s_{\text{dep}},\ \text{val})(s) = \begin{cases} \min(\text{val}(s),\ \text{val}(s_{\text{dep}}) + 1) & \text{if } \text{val}(s_{\text{dep}}) = \text{loss} \\ \text{MaxPly} + \text{succ}(s) - 1 & \text{if } (\text{val}(s) = \text{MaxPly}) \wedge (\text{succ}(s) > 1) \\ \text{val}(s) - 1 & \text{if } \text{val}(s) > \text{MaxPly} + 1 \\ \text{val}(s_{\text{dep}}) + 1 & \text{otherwise} \end{cases}$$

In the implementation supplied with the SearchBench, the extended value range is coded into one byte (diagram below). During the computation, wins and losses with ever greater depths are found, while simultaneously the count ∗eld extends down from the

higher values. This may lead to collisions if the number of successors a state has is too large. If the danger of a collision is great, it may be preferable to code the value range into two byte.



Mapping game values onto byte range

4
SearchBench: Algorithms and Implementation

SearchBench was developed under THINK Pascal for the Macintosh. It consists of 18 problem-independent modules that can be grouped as follows:

User Interface	Algorithms	Kernel
4505	1125	3017

SearchBench: lines of code

The size and complexity of the problem-dependent code depends on the complexity of the search problem and user interface. The user interface has an especially large impact on the code size. For instance, retrograde analysis for chess and Nine Men s Morris is similarly complex, however the chess user interface is much more polished. The following table shows the size of some of the examples implemented on the SearchBench

8-Puzzle	Nine Men s Morris	Awari [1]	Chess [2]
698	3611	1189	7644

Applications: lines of code

4.1 Retrograde Analysis

Retrograde analysis is the central algorithm implemented by the SearchBench. Much care was taken to assure that it is ef∗cient and versatile. In this section the basic algorithm and ef∗ciency considerations that went into its implementation are discussed.

As described in chapter 2, retrograde analysis is a simple algorithm that can be clearly separated into problem-dependent and independent parts. In the ∗rst algorithm we consider, the problem-dependent part must supply the following procedures:

- *InitialValue:* returns the initialization value (val_{init}) of a state.
- *BackupPos:* generates all predecessors (dep^{-1}).
- *InfoProp:* updates a state s value with the dependent state information (vgf_{part}).
- *Priority:* returns a expansion priority dependent on the state and its value.

[1] Implemented by T. Lincke / 22 stone database computed.
[2] Implemented by C. Wirth / All four and some five piece databases computed.

The following algorithm combines these four procedures with a priority queue to determine an expansion sequence that leads to a ∗xpoint.

```
FOR ALL states DO
      val(s) := InitialValue(s);
      InsertPQ(s, Priority(s, val(s));
END;

WHILE Not_EmptyPQ DO
      s := GetFirstPQ;
      pred := BackupPos(s);
      FOR ALL p ∈ pred DO
         ExtractPQ(p);
         val(p) := InfoProp(p, s, val(p), val(s));
         InsertPQ(p, Priority(p, val(p));
      END;
END;
```

This pseudo-code has some drawbacks. The algorithm stores a priority and value for each state in the queue. Because the problems solved with SearchBench generally have a large state space, a real machine may not have enough memory to hold all this information. It would be preferable only to store the state values. In addition, because the priority queue must be able to extract either the state with highest priority or any given state, an ef∗cient implementation uses additional memory and complicated data structures.

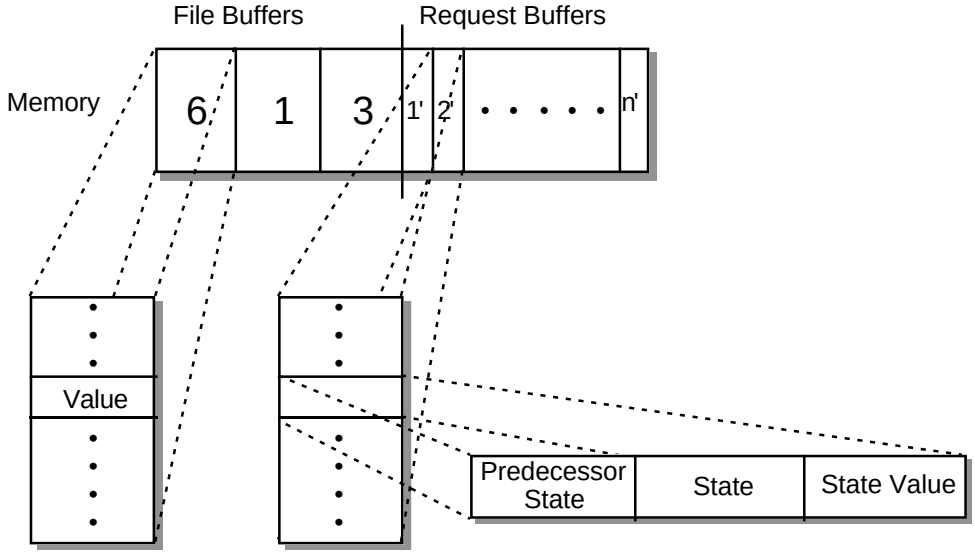
This leads to a revised algorithm that only stores the value of each state. Instead of a priority queue that holds the next state to expand, the entire ∗le is scanned for states that can be expanded. Instead of *Priority* we use *Backupable* to determine if a state can be expanded on the current scan. The *pass* variable counts the number of scans. The following algorithm terminates as soon as a scan ∗nds no more states to expand:

```
PROCEDURE Check(state, limit)
BEGIN
      if Backupable(state, val(state), pass) THEN
         done := FALSE;
         pred := BackupPos(state);
         FOR ALL p ∈ pred DO
            old := val(p);
            val(p) := InfoProp(p, state, val(p), val(state));
            IF (p ≤ limit) AND (old  val(p)) THEN
               Check(p, limit);
            END;
         END;
      END;
END;

FOR ALL states DO
      val(s) := InitialValue(s);
END;
pass := 0;
REPEAT
      done := TRUE;
      FOR ALL states DO
         Check (s, s)
      END;
      pass := pass+1;
UNTIL done;
```

This is essentially the algorithm that is implemented in the SearchBench. An important detail remains hidden in this pseudo-code, namely the fact that each *val* call typically incurs a ∗le access. If this is done inef∗ciently it will dominate the entire computation. The ∗le accesses can be reduced by keeping parts of the ∗le in a memory buffer. Also,

there are usually a number of different expansion sequences that lead to a ∗xpoint, so we attempt to choose one that minimizes the number of ∗le accesses.

The following diagram shows how the SearchBench divides main memory into two partitions (their relative size can be set by the user). The ∗rst is reserved for ∗le buffers and the second for request buffers. The ∗le buffers contain various blocks from the ∗le. This reduces disk I/O, especially during the scan phase, because one ∗le access will load in many needed values. When a state is expanded by *BackupPos*, *InfoProp* needs the value of that state and its predecessors. The state value is in memory because it was just used for the *Backupable* call, but the predecessor values may not be. Instead of loading their value immediately, a request consisting of the state, the predecessor state and its value is generated. The request is stored in the request buffer corresponding to the ∗le buffer the predecessor value is in. As soon as a request buffer is full, it becomes worthwhile to load the corresponding ∗le buffer into memory, because many values in that ∗le block will be used. *InfoProp* is thus delayed until the values are available.



Memory structure for retrograde analysis

This approach also supports the use of compressed data. Since entire blocks are read into the ∗le buffers, they can be decompressed when read into memory and compressed when written to disk. The success of this approach is documented by the fact that the run-times for all applications were dominated by computation, not disk I/O.

4.2 Run-Time Prediction

Since the run-time for interesting retrograde analysis problems is typically quite long, good run-time estimates are important. They facilitate the decision of whether to run a program on a given machine, port the code to a different machine or delay the project until faster hardware is available.

The run-time of retrograde analysis is composed of two parts:

- *Calculation time:* The time needed for all problem-dependent computation, especially dependency generation and value propagation. It is independent of memory and hard-disk size.

- *SearchBench overhead:* Consists of disk I/O and internal overhead needed to synchronize the computation.

Since the SearchBench is quite effective at minimizing disk I/O, the dominant component for most problems is the calculation time. As a ∗rst approximation the run-time can therefore be estimated by the time needed to generate all the predecessors plus the time needed for them to propagate their value:

$$t_{calculation} = states \cdot (t_{backup} + f \cdot t_{prop})$$

states:	number of states to be expanded.
t_{backup}:	average predecessor generation time.
t_{prop}:	average value propagation time.
f:	average number of predecessors.

The last three parameters can be accurately estimated by averaging a few random samples. The number of states that must be expanded will sometimes also be simple to estimate, for instance in the 8-Puzzle we know that exactly half the states are reachable [Wilson 74]. In general it can be dif∗cult to estimate this exactly. However, it is often possible to use an upper bound, for instance in two-player games, no state must be expanded more than once.

SearchBench overhead consists of the time needed to simulate the priority queue by scanning across the entire database, the time needed to store data in the request buffers and the time needed to periodically τush the request buffers. In the estimate shown below, these components are weighted with a constant factor.

$$t_{overhead} = k_1 \cdot size \cdot passes + k_2 \cdot states \cdot f$$

size:	number of states in the state space.
passes:	number of scans needed.
k_1, k_2:	machine dependent factors.

K_1 and k_2 are problem-independent factors resulting from the SearchBench implementation. They will vary depending on the size and access times of the hard-disk and main memory.

In the following example we will estimate the problem-dependent parameters for the 8-Puzzle. The *size* of the database is *9!* states, but because exactly half the states are reachable from the goal state, *states* will be *9!/2*. The fan-out *f* can also easily be determined. There are three different types of tile positions on the board: corners, edges, and central positions. The fan-out of each of these is two, three and four respectively. The space is in the corner for *4/9* of the states and on the edge for *4/9*, thus the average fan-out is:

$$f = \frac{4}{9} \cdot 2 + \frac{4}{9} \cdot 3 + \frac{1}{9} \cdot 4 = \frac{8}{3}$$

Using the implementation of the 8-Puzzle described in chapter 3, we determine t_{backup} and t_{prop} by averaging the results of 100 random calls. This gives the following result:

$$t_{backup} = 0.343 \text{ ms} \qquad t_{prop} = 0.0043 \text{ ms}$$

From this we ∗nd the calculation time to be:

$$t_{calculation} = 181440 \cdot (0.343ms + 8/3 \cdot 0.38ms) = 64\ s$$

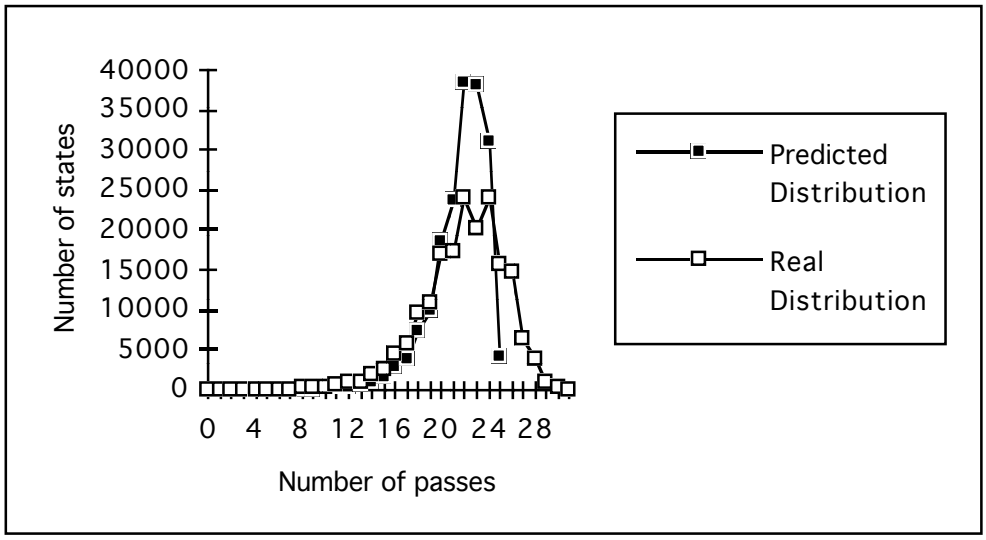
Since the measured run-time for the 8-Puzzle is 106 seconds, the calculation time alone is already a good approximation.

In general we will also want to compute $t_{overhead}$. The only remaining game-dependent parameter is *passes*. To determine the number of passes, we can attempt to model the 8-Puzzle. By incorporating some of its special characteristics, we may be able to predict its value distribution. Some of these characteristics are:

- undirected graph.
- neighbors of states with value *n*, are either value *n-1* or *n+1*.
- fan-out is dependent on the parity of the number of passes.

This model is implemented in the following Maple program. The value distribution it predicts is compared to the real distribution in the diagram below.

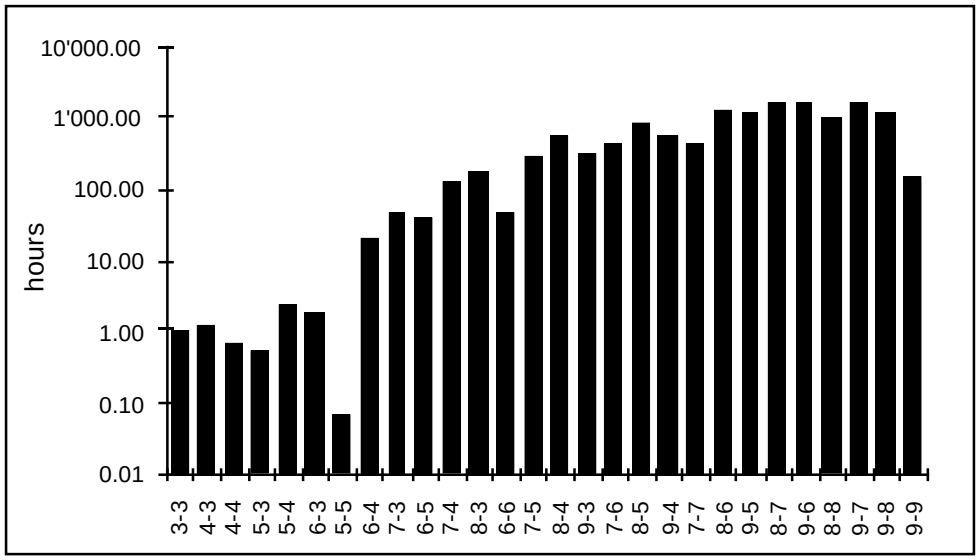
```
8Puzzle := proc()
      leftA := 80640;
      leftB := 80640;
      leftC := 20160;
      rAB[0] := 0;  rBA[0] := 0;      rBC[0] := 0;  rCB[0] := 0;
      A[0] := 1;    B[0] := 0;        C[0] := 0;
      for i from 0 while 0 < A[i]+B[i]+C[i] do
         leftA := leftA-A[i];  leftB := leftB-B[i];  leftC := leftC-C[i];
         temp := 2*A[i]-rBA[i];
         rBA[i+1] := 2*B[i]-rAB[i];
         rAB[i+1] := temp;
         temp := 4*C[i]-rBC[i];
         rBC[i+1] := B[i]-rCB[i];
         rCB[i+1] := temp;
         A[i+1] := evalf(leftA*(1-evalf(1-1/leftA)^rBA[i+1]));
         C[i+1] := evalf(leftC*(1-evalf(1-1/leftC)^rBC[i+1]));
         B[i+1] := evalf(leftB*(2-(1-1/leftB)^rAB[i+1]-(1-1/leftB)^rCB[i+1]));
         print(i,round(A[i]+B[i]+C[i]))
      od
end
```



Comparison of 8-Puzzle distribution

While the model predicts 25 passes instead of the correct 31, it is a surprisingly good ∗t. It will not always be possible to ∗nd a simple model that approximates the real distribution this closely. However, because the SearchBench does a good job reducing ∗le I/O, the calculation time, which is easier to estimate, will usually dominate.

In 1991, when we had only computed up to the 9-3 Nine Men s Morris database, we attempted to predict the run-time for the remaining databases [Gasser 91b]. The following diagram shows the results. The run-time for the larger databases was estimated to be between 1 000 and 1 500 hours on a Macintosh IIfx. Our effective run-time was between 500 and 700 hours on a two to three times faster machine.



Nine Men s Morris run-time estimate

4.3 Veri∗cation

Solving problems with retrograde analysis leads to complex software packages and long run-times. This makes the computation error prone. The possible errors can be divided into three categories:

- *Software errors:* Retrograde analysis relies on many different procedures to compute the state values. These cannot all be exhaustively tested, so there is always the possibility that some bugs remain. Not even the system software and compilers are exempt.
- *Hardware errors:* Data is stored on a hard-disk both during and after computation. Since both the retrograde analysis and compression algorithms generate much ∗le I/O, it is not uncommon for a bit to be τipped, especially if the data is unused for longer periods. Even cosmic radiation may cause a glitch in the CPU or memory.
- *Handling errors:* Long computations are seldom run without interruptions. During this time, the user will move data to different hard-disks, compress ∗les or resume computation on a different platform. All these actions are error prone, for instance continuing with an outdated database version, omitting support ∗les, etc.

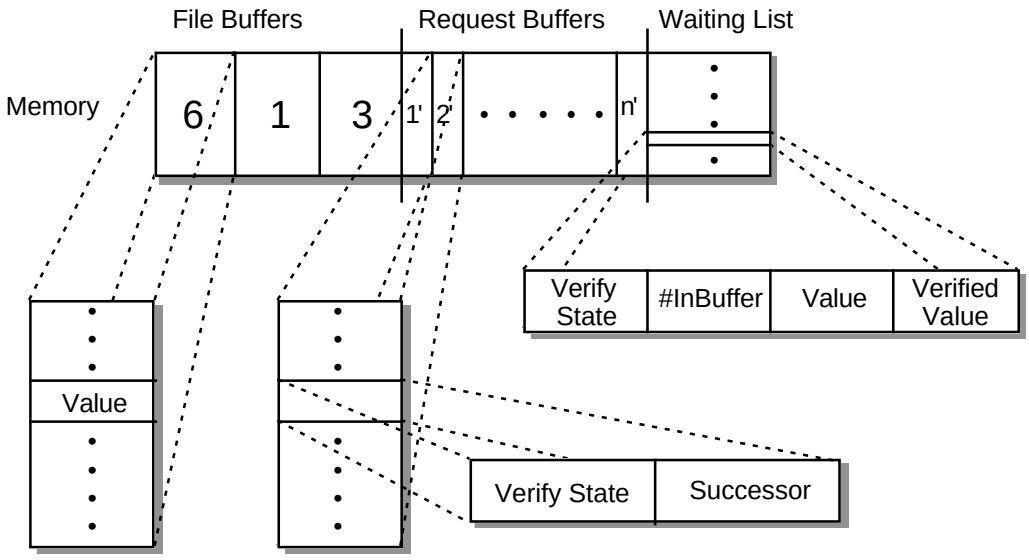
There are a number of methods for reducing the frequency and effect of these errors. Certain consistency checks can be done at run-time and will uncover some errors.

Redoing the entire computation on a different machine should ∗nd most if not all handling and hardware errors. However software errors can be more insidious. The best way of discovering these is to make sure that no code is shared between the veri∗cation and retrograde analysis implementation. This is dif∗cult, because some central routines, for instance the translation between internal and external state representation, must be used for any algorithm. Ultimately the best check is for a third party to verify the results.

SearchBench provides a veri∗cation algorithm that is based on forward search. Each state is considered individually to determine if its value is consistent with the values of its successors. Every single state must be checked, because a state might have an inconsistent value that does not effect the value of its predecessors. The veri∗cation algorithm may thus be slower than the original computation, since retrograde analysis does not always expand all states.

Since the veri∗cation is also a large computation, it must be ef∗ciently implemented. The algorithm as implemented in the SearchBench divides main memory into three parts:

- *File buffers:* store the most recently used blocks of the database.
- *Request buffers:* contains states whose values are not in the File buffer, but are needed to verify a state in the waiting list.
- *Waiting list:* contains states whose veri∗cation is in progress.



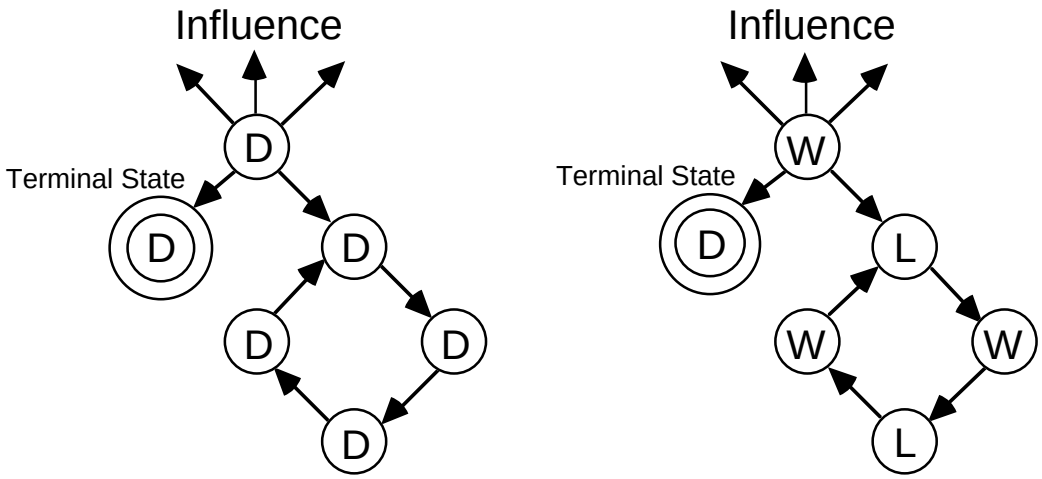
Data Structure for veri∗cation algorithm

As in the retrograde analysis algorithm, the ∗le buffers contain a number of state space blocks taken directly from the database ∗le. If a value of a state to be veri∗ed is not yet in the ∗le buffer, the block it is in is loaded. Then all successors are generated. The values of these successor states are then needed. It would create too much disk I/O if all these values were immediately loaded. For this reason, the loading is delayed until many other values from the same ∗le block are needed. All required successor state values are inserted in the request buffer along with the state that needs their value. If any of a state s successor values are delayed, the state is inserted in the waiting list along with its real value, the value that has been proven with the successors seen thus far, and the

number of successors that are delayed in the request buffer. The following pseudo-code explains this more succinctly:

```
PROCEDURE InsertRequest(thisSucc);
BEGIN
      buffer := WhichBuffer(thisSucc);
      Add(buffer, thisSucc);
      IF Full(buffer) THEN
         LoadFileBuffer(buffer);
         FOR ALL elements in buffer DO
            thisElemValue := GetValue(thisElement);
            UpdateVeri*edValue(listindex, thisElemValue);
         END;
         ClearRequestBuffer(buffer);
      END;
END;

FOR ALL states to verify DO
      succ := ForwardPos(thisState);
      InsertList(thisState, NumSuccs, thisValue, unknown)
      FOR ALL Succ DO
         IF InFileBuffer(thisSucc) THEN
            thisSuccVal := GetValue(thisSucc);
            UpdateVeri*edValue(thisState, thisSucc, thisSuccVal);
         ELSE
            InsertRequest(thisSucc);
         END;
      END;
END;
```

This algorithm will ∗nd all errors in the database if the dependencies form an acyclic graphs. However, if cycles are present it is possible that values will be consistent but incorrect. The following diagram shows such an example for a two-player game where only win, loss and draw occur. There is an in∗nite cycle in this state space, where neither player can escape. In most games, the positions in this cycle would be de∗ned as draws. However, alternately classifying these positions as wins and losses is also consistent. This cycle can then cause an arbitrarily large part of the state space to be incorrectly veri∗ed. In the case of two-player games this problem can be eliminated by extending the value range to include the depth of the win and loss. This extended range will then no longer show this problem, but in general, cycles remain problematic.



Correct Values Incorrect but consistent

4.4 Data Compression

Four types of redundancy can be observed in most data:

- *Character distribution:* some characters are used more often than others.
- *Character repetition:* repeating single character can be more compactly encoded.
- *High-usage patterns:* some character sequences appear more often than others.
- *Positional redundancy:* the appearance of certain characters may be predictable.

Various algorithms have been proposed for dealing with some or all these redundancies. For instance Huffman coding combats the character distribution redundancy, or run-length coding deals with character repetitions.

The initial version of the SearchBench used run-length compression. However the observed compression factors were extremely dependent on the input. Initialized database ∗les showed excellent compression, because most states had the same (unknown) value. However the ∗nal databases could often not be compressed. This led to the idea of writing special compression routines for database ∗les, to exploit all known redundancies. However, determining the types of redundancy that will occur is not an easy task. It is much simpler to use an adaptive approach. For this reason (and supporting experiments) Lempel-Ziv-Welch was included in the SearchBench. A detailed description of the algorithm can be found in [Welch 84], so only a short summary is given here.

The algorithm is organized around a string table that maps strings of characters into 12-bit codes. The table has the pre∗x property that if string *sK* (string *s*, character *K*) is in the table, string *s* is also included. Because the string table only contains strings that have been previously encountered in the input data, it reτects the data statistics.

The string table is initialized with all single character strings. The input data is parsed by taking off the longest recognized string that is in the table. Then the table is extended by the string consisting of the parsed string plus its next character. The complete compression algorithm can be written as follows:

```
Initialize Table;             {all single character strings included}
s := Empty;
WHILE NOT eof DO
      read(K);
      IF InTable(sK) THEN
         s := sK;
      ELSE
         Output code(s);      {12-bit code}
         AddTable(sK);
         s := K;
      END;
END;
Output code(s);
```

The decompression algorithm similarly builds the string table during data decompression. When the code has been decoded, its ∗rst character is used together with the last code to extend the string table. When decompressing, it is possible that the code that is read is not yet in the table. For instance, this can be the case when decompressing the string *KsKsK*. After compressing the string *Ks*, the compression algorithm adds the string *KsK* to its string table and outputs it as the next code. In the same situation the decompression algorithm reads *Ks*, but cannot add the string *KsK* to its table, because the next character is unknown. However this only happens when the last code is extended,

so the ∗rst character of the next code must be the ∗rst of the last code. This leads to the following algorithm:

```
Initialize Table;             {all single character strings included}
last := Empty;
WHILE NOT eof DO
      read(code);
      IF InTable(code) THEN
         OutPut(Decode(code));
         F := First(code);     {∗rst character of code}
      ELSE
         Output(Decode(last));
         Output(F);
      END;
      IF last  Empty THEN
         AddTable(lastF);
      END;
      last := code;
      END;
END;
```

For use with the SearchBench it is important that rapid random access to the ∗le is possible. Since database ∗les can be hundreds of MByte large, the ∗le cannot be compressed in one piece, because then each access would entail decompressing the entire ∗le. Therefore the ∗le is split into a number of pieces (up to 4095 in our implementation) and each piece is compressed separately. As individual pieces become smaller, the compression ratio typically declines, because there is not enough data for the algorithm to adapt to. Thus there is a trade-off between high compression ratios and fast random access.

Experience with this algorithm shows that for database ∗les a more uniform compression ratio is achieved than with run-length compression. The typical compression ratio seems to be about a factor ∗ve for this type of data.

Initially all SearchBench algorithms supported both compressed or non-compressed ∗les in a fashion that was totally transparent to the user. However this had adverse effects on the run-time. In particular, for database computation, where the main database must be opened with write access, this became unacceptable. SearchBench still allows support database ∗les to be compressed, but the main database must be expanded. This also allowed the affected algorithms to be simpli∗ed.

4.5 Parallelization

SearchBench does not yet support parallel computing. However the retrograde analysis and veri∗cation algorithms have been ported to parallel machines with varied success. There are essentially two different ways this can be done:

- *Master/Slave:* This scenario assumes that there is one master processor in charge of all state values. This processor parcels out predecessor and successor generation to the slave processors. The slaves send their results back to the master without communicating among themselves.
- *All equal:* Here each processor is responsible for a subset of the state values. All predecessor and successor generation for these states is done by the processor itself. If these computations produce or require the values of foreign states, a message is sent to the processor in charge of those values.

A master/slave retrograde analysis algorithm for Nine Men s Morris was implemented on Transputers [Kunz 92] and the MUSIC system (developed at ETH). The memory per processor on both these systems was limited (1-2 MByte). The number of processors (8-30) was also fairly small. Because neither of these systems allow the processors to directly access a disk, not all states could be distributed among these processors. The master/slave algorithm that was implemented used a Macintosh as the master. The Macintosh scanned the ∗le for states to be propagated and periodically sent any idle slave some of these states. The slave generated all predecessors and sent these back to the Macintosh. The Macintosh then propagated the information between the states and their successors.

Since retrograde analysis spends most of its time generating predecessors, the computation load could be evenly distributed with this algorithm. However, the master became an I/O bottleneck. This might not be critical if the Macintosh were able to access all state values rapidly, but since they were stored on disk it was dif∗cult to keep all slaves fully utilized.

After ∗nishing the Nine Men s Morris databases, more powerful parallel machines became available, just in time for the database veri∗cation. The parallel veri∗cation algorithm was ∗rst implemented on a cluster of SUN workstations [Balz 94]. It was later ported to the Intel Paragon XP/S5+ [Mäser 94]. These systems offer enough storage that the state space can be distributed. While the workstations must store some of their states on disk, the Paragon can load all state values into main memory. The basic algorithm for these machines is the same as described in 4.3. However when a request buffer is full, instead of loading the state values from the disk into memory, a request is sent to the processor responsible for the states. The experimental results show a nearly linear speed-up for the Nine Men s Morris databases.

We have just recently implemented a parallel all equal retrograde analysis algorithm on the Cenju-3 [Wirth 94]. This machine has 128 processors with 64 MByte per processor. Thus far the algorithm has only been tested on small databases, but initial results are promising. Other researchers have also implemented such algorithms [Lake 94], [Stiller 91], but they used different two-player value propagation functions. Our function may be superior, because it evaluates fewer states, thereby reducing the computational complexity.

Both retrograde analysis and veri∗cation seem easier to parallelize ef∗ciently than alpha-beta or branch-and-bound algorithms. One reason may be that alpha-beta and branch-and-bound do not always have enough work to keep all processors busy and are forced to expand states that a sequential algorithm might prune. Better parallelization is often attained with better heuristics for selecting states to expand [Hsu 90]. The veri∗cation algorithm in contrast must always check each and every state, no pruning is possible. Similarly with retrograde analysis, there are usually enough states that must be expanded to keep all processors busy.

5
Nine Men s Morris

The game of Nine Men s Morris, also known as Merrills or Mill, is one of the oldest games still played today [Bell 69]. Boards have been found on many historic buildings throughout the world. The oldest (about 1400 BC) was found carved into a roo∗ng slate on a temple in Egypt. Others have been found as widely strewn as Ceylon, Troy and Ireland. In 1880 a board was found on a Viking ship buried 870 AD in Norway.

Because Nine Men s Morris is often wrongly thought of as a child s game, it has fallen somewhat out of fashion and is now seldom played. This may explain why the literature that goes beyond the rule explanation stage is fraught with errors. And although a world championship is held yearly in northern England, competitors consist almost exclusively of locals, so it is dif∗cult to judge whether their play is the best humans are capable of.

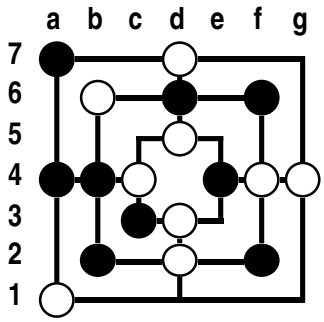
In this chapter we describe how a combination of retrograde analysis and alpha-beta search was used to solve the game. Nine Men s Morris is a draw. Other games that have been recently solved, for instance Qubic [Patashnik 80], Connect-4 [Allen 89], [Allis 89] and Go-Moku [Allis 94b] pro∗t from knowledge-based methods because they all have a low ∗decision complexityτ, i.e. the right move is often obvious. Nine Men s Morris is the ∗rst non-trivial game to be solved that does not seem to bene∗t from such methods.

5.1 Rules

Nine Men s Morris is played on a board with 24 points where stones may be placed. Initially the board is empty and each of the two players hold nine stones. The player with the white stones starts. During the *opening*, players alternately place their stones on any vacant point.

After all stones have been placed, play proceeds to the *midgame*. Here a player may slide one of her stones to an adjacent vacant point. Any time a player succeeds in arranging three of her stones in a row, also called ∗closing a millτ, she may remove an opponent s stone which is not part of a mill. If White closes a mill in the opening by playing to b6 (diagram on next page), she can now remove Black s stone on a1, but not the one on d2.

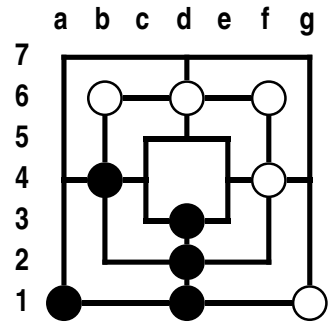
As soon as a player has only three stones left, the *endgame* commences. When it is her turn, the player with three stones may jump one of her stones to any vacant point on the board.



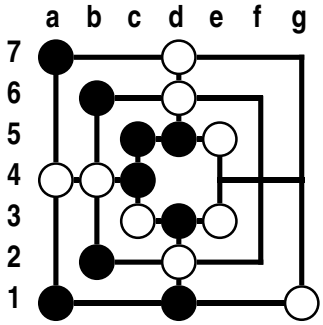
After the opening

The game ends in the following ways:

- A player who has less than three stones loses.
- A player who cannot make a legal move loses.
- If a position is repeated, the game is a draw.



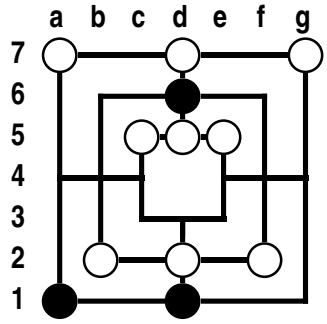
After b6, White removes a1 or b4.



Black has no legal moves.

Two points are subject to debate among Nine Men s Morris enthusiasts. The ∗rst hinges on the observation that in the opening it is possible to close two mills at once. Should the player then be allowed to remove one or two opponent s stones? In our implementation, only one stone may be removed. This is not really an issue, since our observations show that such play never occurs in real games.

The second point concerns positions where the player to move has just closed a mill, but all the opponent s stones are also in mills. May she then remove a stone or not? In our implementation she may remove any stone. Although this makes the rules more complex, most games seem to be played by this rule. This rule can make a difference in some positions. For instance, the diagram on the right shows a position where White can win by our rules, but only draws if stones in mills may not be removed. However, it seems unlikely that these differences affect the value of the game.



White to move

5.2 Bushy

The Nine Men s Morris program Bushy, was originally implemented as part of my diploma thesis [Gasser 90a]. It uses a standard alpha-beta search algorithm, combined with heuristics. It is implemented in Modula-2 under the Smart Game Board [Kierulf 90]. Some heuristics that work well are:

- Do not close more than one mill in the opening.
- Do not reduce opponent to three stones.
- Occupy central points.

Initial playing strength measurements were performed by pitting the program against itself or other programs. The ∗rst real test came in the summer of 1990 at the Second Computer Olympiad in London. There Bushy played against Mike Sunley, the reigning World Champion. Of the six games played, Bushy won four and drew two [Levy 91].

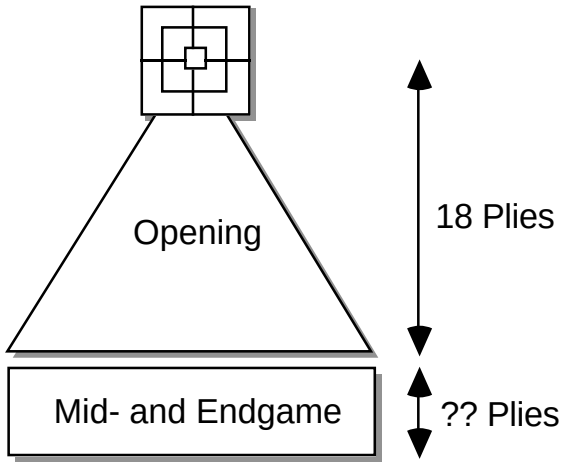
Bushy has also been incorporated into a robot designed in conjunction with the Institute for Robotics at ETH. This robot is on display at the Technorama museum in Winterthur. It has played hundreds of games against visitors and seldom loses, often winning in spite of the little time available for searching. This version of Bushy only uses the smaller databases described in [Gasser 90b].

Unlike chess, Go or other well-studied games, there is little literature on Nine Men s Morris [Müller 87], [Schürmann 80]. Nor is there an established rating system or well-publicized tournaments. This makes it dif∗cult to assess the strength of a computer program. The original reason we decided to compute endgame databases was not with the intention of solving the game, but rather to have an objective measure for judging the program s strength.

After solving Nine Men s Morris, we incorporated the results into a perfect-playing program [Lincke 94]. This program repeatedly demonstrated its ability to attain a draw against a group of strong players from Bern. Although the humans never won, they tended to dominate because the program randomly played any non-losing move. This effect can be avoided by using a combined approach, where the perfect program determines the optimal moves and the heuristic program selects the move that is most dif∗cult to respond to.

5.3 Solving Nine Men s Morris

Nine Men s Morris can be divided into two distinct phases. The opening, where stones are placed on the board and the midgame and endgame, where stones are moved. These two phases lead to a state space graph with speci∗c characteristics. Most importantly, the opening phase induces an acyclic graph, whereas in the midgame and endgame phase move cycles can occur. Another difference is the search depth. The opening is clearly de∗ned to be 18 plies deep for every path, however depending on the chosen path, the number of moves spent in the midgame and endgame will vary.



Two distinct phases

These different properties suggest the use of different search methods. We decided to use retrograde analysis to construct databases containing all midgame and endgame positions. This seems advantageous because retrograde analysis handles cycles more ef∗ciently than forward search. In addition the opening search will require the values of many positions. Because of the interdependencies, this probably means computing the value for all or nearly all midgame and endgame positions, an ideal task for retrograde analysis.

Computing further databases for the opening is unreasonable, because their size would be even larger than for the midgame and endgame. In addition, because the value of only a single opening position (the empty board) is of interest for solving the game, no intermediate position values must be stored. Alpha-beta search [Knuth 75] is ideal for this type of problem.

5.4 Midgame and Endgame State Space

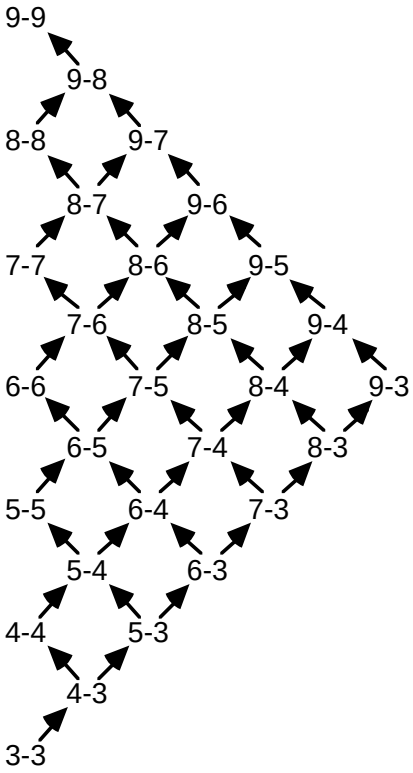
To determine if applying retrograde analysis is feasible, we must construct an appropriate state space. Each of the 24 board points can either be unoccupied or occupied by a black or white stone, so an upper bound for the state space size is 3^{24} ($2.8 \cdot 10^{11}$) states. This rather loose bound can be improved by taking the following game constraints into account:

- Players have 3 to 9 stones on the board.
- There are unreachable positions.
- The board is symmetrical.

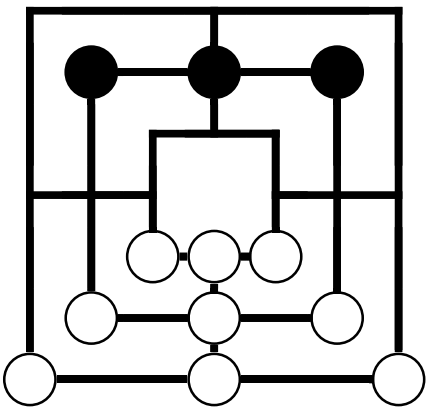
The ∗rst observation allows us to split the state space into 28 subspaces. The diagram shows these subspaces and their dependencies, i.e. the 7-5 positions (7 stones against 5 stones) can only be computed after the values for all 7-4 and 6-5 positions have been determined.

Not all states in these subspaces can be reached in the course of a legal game. For example, if one player has a closed mill, then her opponent cannot have all nine stones on the board (diagram). Similar considerations can be made for two or three closed mills. The 9-9, 9-8, 9-7, 9-6, 9-5, 9-4, 9-3, 8-8, 8-7 and 8-6 subspaces contain such unreachable states.

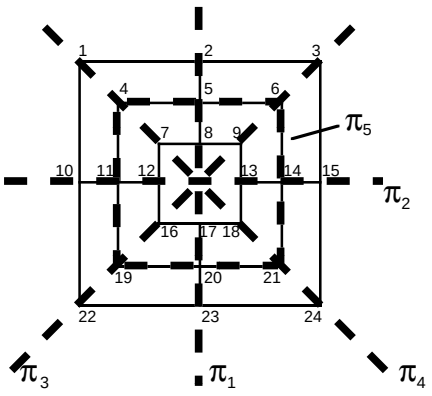
The subspaces also contain symmetrical positions, which can be eliminated using Polya s theorem. To apply this method, we must ∗rst determine the symmetry axes of the game. In general there are ∗ve symmetrical axes (diagram). In the special case of the 3-3 subspace, there are additional symmetries, because all rings are interchangeable. This special case will be ignored in the course of the following explanations.



Database dependencies



Unreachable state



Symmetry axes

The symmetry axes generate the group consisting of the following 16 permutations:

Permutation	Cycle Index	Permutation	Cycle Index
identity	x_1^{24}	π_5	$x_1^8 x_2^8$
π_1	$x_1^6 x_2^9$	$\pi_5\pi_1$	$x_1^2 x_2^{11}$
π_2	$x_1^6 x_2^9$	$\pi_5\pi_2$	$x_1^2 x_2^{11}$
π_3	$x_1^6 x_2^9$	$\pi_5\pi_3$	$x_1^2 x_2^{11}$
$\pi_1\pi_2\pi_3$	$x_1^6 x_2^9$	$\pi_5\pi_1\pi_2\pi_3$	$x_1^2 x_2^{11}$
$\pi_1\pi_2$	x_2^{12}	$\pi_5\pi_1\pi_2$	x_2^{12}
$\pi_1\pi_3$	x_4^6	$\pi_5\pi_1\pi_3$	x_4^6
$\pi_2\pi_3$	x_4^6	$\pi_5\pi_2\pi_3$	x_4^6

Symmetry permutations

The Cycle Index describes the cycle structure of the permutation. The terms involve indeterminates of the form x_k, where the exponents of x_k are obtained by counting the number of cycles of length k the permutation consists of. By summing up all the cycle indices, the cycle index of this group is obtained:

$$P_G(x_1,x_2,x_4) = 1/16\cdot(x_1^{24}+4x_1^6x_2^9+4x_1^2x_2^{11}+2x_2^{12}+x_1^8x_2^8+4x_4^6)$$

The cycle index is used to compute the exact number of positions in any subspace (still includes unreachable positions). The following example for the 4-4 subspace shows how this can be done. For more details see [Reingold 77].

$$\frac{1}{16}\cdot\left(\begin{array}{l}\binom{24}{4}\binom{20}{4}+4\cdot\left(\binom{11}{2}\left(\binom{9}{2}+\binom{9}{1}\binom{2}{2}\right)+\binom{11}{1}\binom{10}{2}\right)+2\cdot\binom{12}{2}\binom{10}{2}+4\cdot\binom{6}{1}\binom{5}{1}\\ +4\cdot\left(\binom{9}{2}\left(\binom{7}{2}+\binom{7}{1}\binom{6}{2}+\binom{6}{4}\right)+\binom{9}{1}\binom{6}{2}\left(\binom{8}{2}+\binom{8}{1}\binom{4}{2}+\binom{4}{4}\right)+\binom{6}{4}\left(\binom{9}{2}+\binom{9}{1}\binom{2}{2}\right)\right)\\ +\left(\binom{8}{2}\left(\binom{6}{2}+\binom{6}{1}\binom{8}{2}+\binom{8}{4}\right)+\binom{8}{1}\binom{8}{2}\left(\binom{7}{2}+\binom{7}{1}\binom{6}{2}+\binom{6}{4}\right)+\binom{8}{4}\left(\binom{8}{2}+\binom{8}{1}\binom{4}{2}+\binom{4}{4}\right)\right)\end{array}\right) = 3'\,225'\,597$$

The following table shows the number of states that remain when taking all three considerations into account. This is the exact number of legal states, since no positions are counted twice and all are reachable.

Database	Legal States	Database	Legal States	Database	Legal States
3-3	56 922	7-4	103 033 056	8-6	735 897 724
4-3	1 520 796	8-3	51 527 672	9-5	452 411 838
4-4	3 225 597	6-6	156 229 360	8-7	1 046 720 480
5-3	5 160 780	7-5	267 821 792	9-6	692 194 494
5-4	20 620 992	8-4	167 411 596	8-8	568 867 453
6-3	13 750 640	9-3	73 827 570	9-7	778 293 448
5-5	30 914 424	7-6	535 586 576	9-8	608 860 826
6-4	51 531 584	8-5	401 705 976	9-9	95 978 501
7-3	29 451 376	9-4	216 132 556		
6-5	144 232 144	7-7	420 793 096	Total	7 673 759 269

Number of legal states

Indexing function

For databases of this size, a memory ef∗cient storage method is needed. The standard method used in retrograde analysis is to construct a perfect hash function, which, given a state, returns a unique index. This makes it unnecessary to store the state description along with the state value, because the state description is encoded in the index. Ideally the perfect hash function has a value range *0 |states|-1* and can be computed rapidly. Constructing such a hash function can be dif∗cult. Because rapid computation is critical, it seems reasonable to loosen the requirements and allow hash functions with a slightly larger range. Thus some unused entries are included in the ∗le, but this will not impede the computation as long as the ∗le size remains suf∗ciently small.

For Nine Men s Morris, we chose a simple index function of the form:

$$\text{Index} = f(W) \cdot \binom{24-|W|}{|B|} + g(B, W) + h(\text{player}, W, B)$$

W:	Set of white stones identi∗ed by their index on the board.
B:	Set of black stones identi∗ed by their index on the board.
f(S):	A unique integer for the stone set S.
g(S, T):	A unique integer for the stone set S, assuming the stone set T is already on the board.
h(P, S, T):	An offset depending on whose turn it is to play.

There are *binomial(24, |W|)* ways of placing *|W|* stones on a board with 24 points. Therefore a simple function *f* could use this encoding. However, this does not take board symmetries into account. The table below shows that the number of placements with symmetries taken into account is much smaller:

Stones	1	2	3	4	5	6	7	8	9
Con∗gurations	4	30	158	757	2 830	8 774	22 188	46 879	82 880

Number of non-symmetrical placements

In our encoding the function *f* maps the white stones to this reduced range. Of the remaining *24-|W|* points on the board, *|B|* are black. We use a simple function *g* to map the black stone set to the *binomial(24-|W|, |B|)* possibilities. Most symmetrical positions are thus eliminated.

Lastly, the function *h* adds an offset depending on whose turn it is to play. If both players have the same number of stones on the board, the stone colors can be τipped so that White is to move and the offset is zero. Otherwise the offset is either zero (White to move) or the number of different boards (Black to move).

The following table compares the range of two indexing functions with the number of legal positions computed earlier using Polya s theorem. The ∗rst uses the function *f* described above. This has the disadvantage that many unreachable states are included. The second indexing function eliminates most of these. For instance in the 9-9 database, the improved function *f* only encodes the stone sets where no mills are closed (38 040 instead of 82 880 different sets). The improved indexing function still includes some symmetrical and unreachable positions. However, the number of excess states is reasonably small, so further improvements were deemed unnecessary.

Original range	Excess	Improved range	Excess
9 074 932 579	18.26%	8 102 965 583	5.59%

Range of two indexing functions

5.5 Database Computation

Initialization

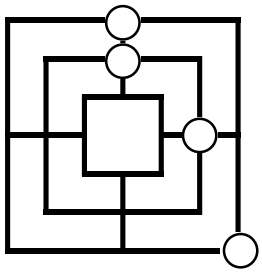
The ∗rst step in retrograde analysis is to initialize all easily recognizable won and lost positions. In Nine Men s Morris, there are two types of terminal positions, both losses for the player to move. The ∗rst type are positions where the player cannot move, the second are positions where the player has less than three stones. Thus if the player to move is blocked, we initialize that state as a loss. We would also like to mark positions where the player to move has less than three stones as losses, but we eliminated these positions from the state space. Therefore positions where the player to move can close a mill and reduce the opponent to two stones are marked as wins instead. We also mark positions where the player with three stones loses in two plies. Such positions are easy to recognize and belong to one of the following categories:

- the opponent has two open mills with no stones in common.
- the opponent has two open mills and the player to move cannot close a mill.
- the player to move must remove a stone blocking an opponent s mill.

The value of all other positions is set to unknown.

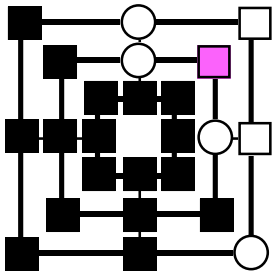
Only databases where at least one player has exactly three stones will contain won or lost positions due to the second type of loss. What is more interesting, is that these databases contain no blocked positions. A player with three stones cannot be blocked because she is allowed to jump. And the opponent cannot be blocked because three stones cannot block more than three stones.

A simple initialization looks at every position and determines if it is won, lost or unknown. This is adequate when at least one player has exactly three stones, because many positions can be initialized as won or lost. However, in the other databases, most states are initialized to unknown. This can lead to very inef∗cient run-times, for instance in the 4-4 subgame, only six of the 3 225 597 positions are blocked. Preferably, the initialization time should be proportional to the number of initialized states.



Four stone con∗guration

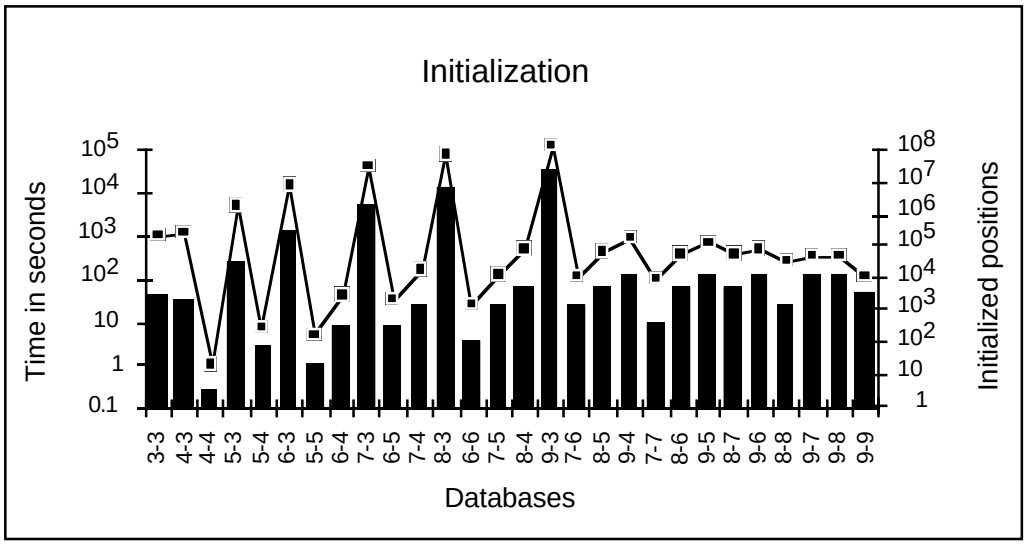
The diagram on the right shows a possible con∗guration of four white stones. If it is White s turn to move, there are 4 845 possible ways of placing four black stones. In the simple 4-4 initialization, all these positions would be checked to see if White is blocked. Taking a closer look at the white stone con∗guration shows that at least nine black stones are needed to render White immobile. Therefore, White to move will not be blocked in any of the 4 845 positions.



White stones partition the board

A similar idea can be used when it is Black s turn to move. The diagram on the left shows how the four white stones partition the board into three independent areas. The areas contain 1, 2 and 17 empty points respectively. Therefore Black can be blocked if any sum of these numbers (i.e. 1, 2, 3, 17, 18, 19 or 20) equal the number of Black stones. In our case, this proves that none of the 4 845 positions with Black to move are blocked.

The diagram below shows that with this improved method, the initialization times are indeed proportional to the number of initialized won or lost states.



Initialization time(bar) and number of states(line)

Calculation

The ∗rst databases were computed in 1989 as part of my diploma thesis on a Macintosh IIx, the last on a Macintosh Quadra 800 in 1993. In between, the algorithm was improved and ported to various machines. These include a Cray X-MP/28, a DEC VAX 9000-420, an IBM RS/6000, a 16-Transputer (T805) system, a 30-Processor MUSIC system (Motorola DSP 96000) and a DEC 3000 (Alpha). Only the DEC 3000 showed better performance than the Apple Macintosh. There are various reason for this surprising result. Many machines executed our code at low priority because of their multi-user environments. Also, many of the more powerful machines are optimized to deal with τoating-point and vector operations, whereas their integer performance is not as highly tuned. The parallel machines additionally suffered from insuf∗cient main memory and no direct disk access. This made the Macintosh front-end an I/O bottleneck.

All the calculations were done with the standard retrograde analysis algorithm (chapter 4) and the two-player value propagation function (chapter 2). One unfortunate incident occurred in the 8-7 database. Using one byte for the value range proved to be too small, consequently the count ∗eld collided with the won and lost values. Positions with high count ∗elds had to be manually scanned and their count ∗eld decreased. This allowed the computation to be successfully concluded.

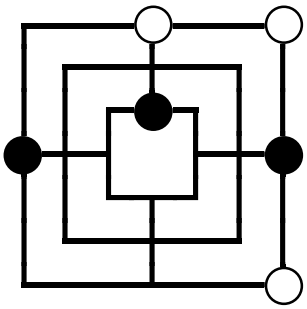
Veri∗cation

Computations of this size almost inevitably contain some errors. The most common cause are software bugs, either in the retrograde analysis code, the system software or the compiler. Hardware problems occur as well, for instance disk errors or memory glitches. For these reasons, the databases were veri∗ed on a cluster of 30 SUN SPARC workstations [Balz 94]. This took approximately three months. The veri∗cation found half a dozen hardware errors. Additionally, an error was found in the Nine Men s Morris code, which allowed some positions to be classi∗ed as draws instead of losses. These errors induced further inconsistencies so that thousands of positions had to be corrected.

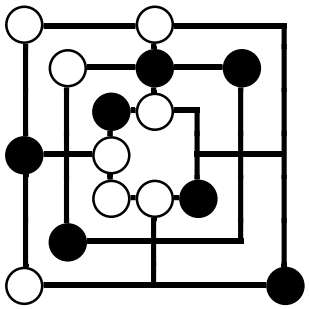
The veri∗cation only checked that the wins, losses and draws were consistent. The depth information was not checked because the additional memory needed to store the databases increases the disk I/O and slows the veri∗cation dramatically. Presently we are running a depth veri∗cation algorithm on a new parallel machine which may enable the complete veri∗cation. Even if depth veri∗cation ∗nds no additional errors, how can we be certain that none exist? A basic idea behind the veri∗cation is to use a different algorithm (forward search) and independent code to verify the data (see 4.3). However there are still some procedures common to both algorithms. For instance the ∗le indexing function or the initial state values. Ideally, our results should be independently veri∗ed.

Results

Sifting through the ∗nal databases in search of ∗interestingτ positions is a daunting task. Because we have no real intuition what makes a position interesting, we decided to gather results in a more statistical fashion. The ∗rst two diagrams show examples of positions with long winning sequences. The 3-3 position has the longest sequence in that database. The 8-7 position has the longest conversion distance among all databases.



Black to move
Loss in 26 plies

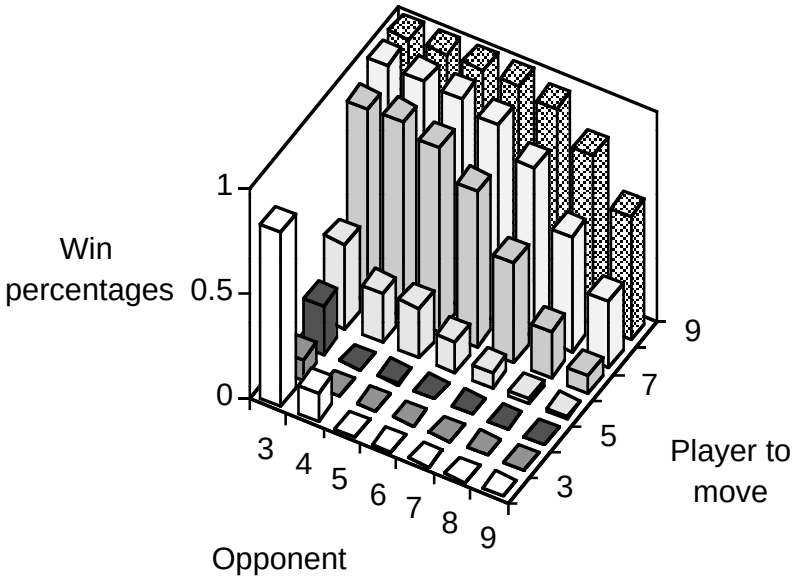


White to move and win
Conversion in 187 plies

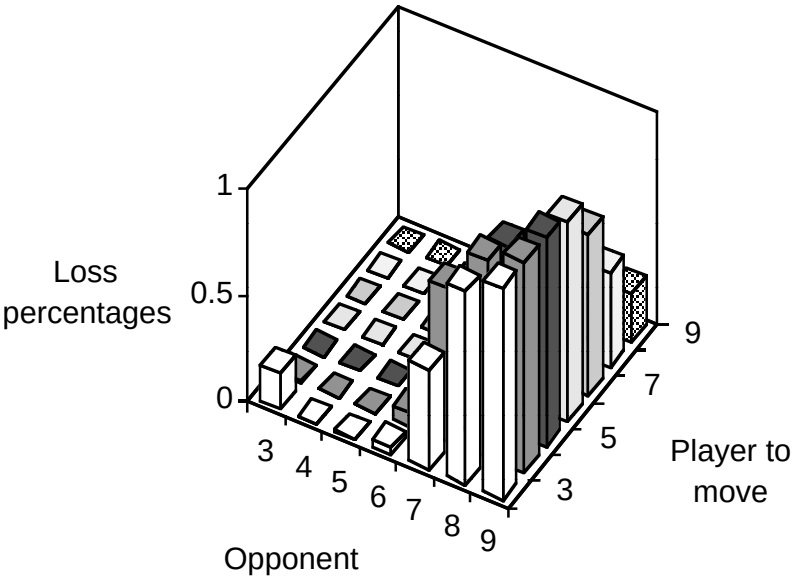
The following diagrams show the percentage of positions that are won (respectively lost) for the player to move. They are grouped according to the number of stones on the board and normalized accordingly.

They show that the win probability strongly correlates with the stone difference, i.e. more stones are better. Comparing the win chances of the player to move with the win chances of her opponent (loss for player to move) shows that having the right to move is advantageous. Also there seems to be a cut-off in both diagrams at about 7 stones. Having less than this number makes winning dif∗cult. The statistics also show that the

3-3 database is special. This is probably because both players are allowed to jump, essentially making this a different game.



Win percentages for the player to move



Loss percentages for the player to move

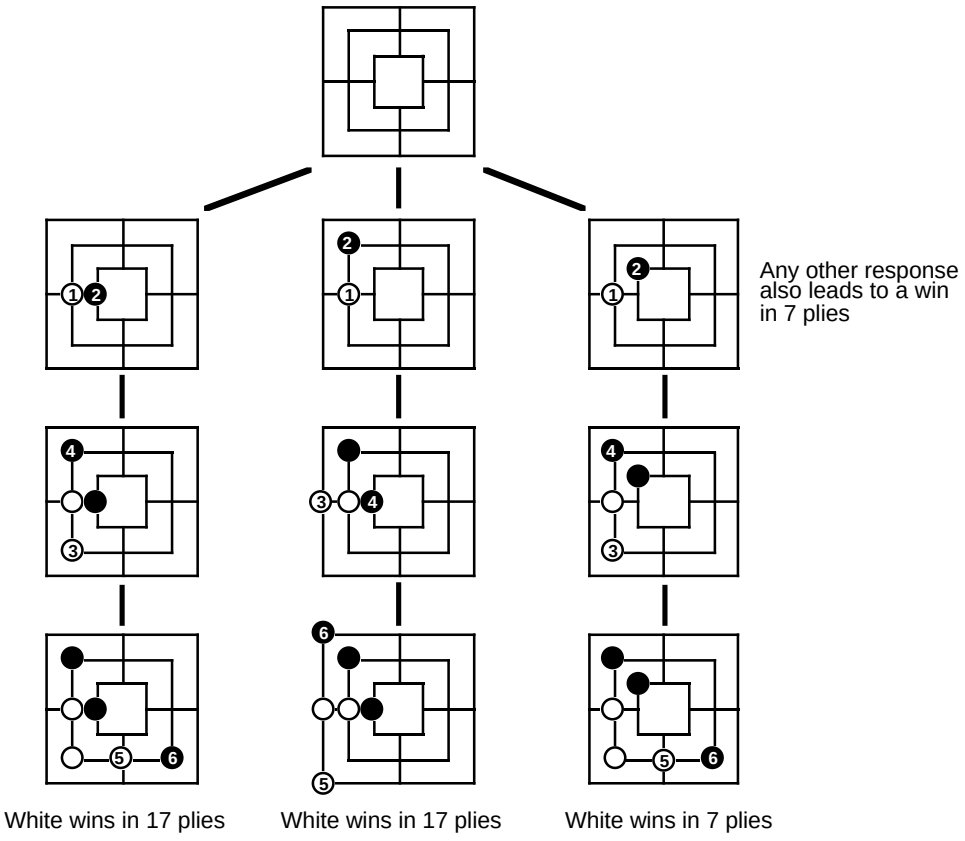
One must keep in mind that these statistics can be misleading. For instance, in the 4-3 database if the player with four stones is to move, it seems she has a small chance of winning. A closer examination of the database shows that all wins are trivial positions where the player can close a mill immediately. These positions will not occur in a real game, so actually the player with three stones has an advantage. Similar reservations apply to all these databases. It is unclear how the realistic positions can be ∗ltered from the rest. The win, loss and draw distribution for each database is listed in appendix A.

5.6 Opening Search

Nine Men s Morris, as the name implies, is usually played with nine black and nine white stones. However, it is also possible to play the game on the same board, but with a different number of stones. In this chapter we ∗rst examine the games where players start with three, four or ∗ve stones. This is interesting because the smaller games can be solved with simpler, more elegant techniques than what is needed to solve the nine stone variant.

Three Stones

This game is a win for whoever goes ∗rst. To prove this, it is suf∗cient to show that White (the ∗rst player) wins if she ∗rst moves to a central point (diagram). Whatever Black replies, White s strategy for the rest of the opening is to make mill threats, forcing Black to block them. After the opening, a look-up in the 3-3 database shows that all positions are won for White.



Opening solution tree

Four Stones

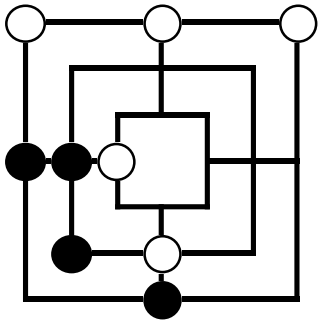
Since there are few non-drawn positions in the 4-4 database, it is easy to prove the four stone game a draw. A simple idea is to look at stone placement. It turns out, that if a player has at least one stone on a central point in a 4-4 position, the opponent cannot win. Thus, if neither player closes a mill, a simple strategy is to place at least one stone

on a central point and leave it there during the entire game. If a player decides to close a mill, she cannot win either. This is because all 4-3 positions, where the player with four stones has a closed mill, are drawn. So if the opponent at any time closes a mill, just stop her from closing another. This is easy as there can be at most one threat at a time.

Five Stones

The same idea used for four stones can be applied here as well. The game is also a draw. We assume that the player to draw will not close any mills during the opening. The opponent may then close zero, one or two mills. We must show that the player will not lose in any of the 5-5, 5-4 or 5-3 positions that can result after the opening.

The 5-5 database con∗rms that any player who occupies at least two central points cannot lose. Since there are four central points, each player easily gets two by placing her ∗rst two stones on these points. In addition, all 5-3 positions, where the opponent has two closed mills, are drawn. This leaves the 5-4 positions. If the opponent closes exactly one mill in the opening, the player will still occupy at least one central point in the resulting 5-4 position. Of all these positions, only one is lost for the player (shown at right). But in this particular position, the player (Black) had the last move, so she would have been able to avoid this position and get a draw by placing her last stone on a different point.



White to move and win

Therefore, we have shown how the player achieves at least a draw. Since either player can follow this strategy, the game is a draw. Note that this strategy only covers the opening, and makes no statements about optimal play for the midgame and endgame.

Nine Stones

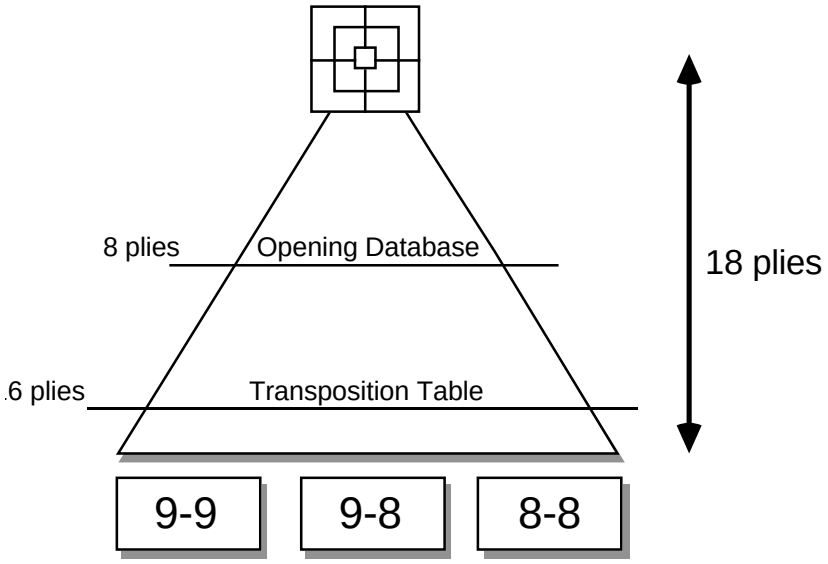
The proof that nine stones is a draw is similar to the ∗ve stone proof in the sense that we again show that either player can attain at least a draw. However, instead of simple pattern matching techniques, we use search to prove this.

In principle, the 18-ply opening can lead to any of the databases except 8-3, 9-3 and 9-4. This comprises approximately nine GByte of data. Since our machine only had 72 MByte, accessing database values becomes an I/O bottleneck. The following three methods were applied to alleviate this problem:

- reduce the size of the databases.
- reduce the number of used databases.
- reduce the number of disk accesses.

Because no cycles occur in the opening, it is unnecessary to compute the depth of the win or loss. Therefore, a ∗rst idea is to reduce the size of the databases by packing ∗ve positions into one byte (3^5=243). However, to further minimize storage requirements, we decided to pack eight states into a byte. Although we then have only one bit per state, this is suf∗cient to compute the game-theoretic value if we do *two* alpha-beta searches. In the ∗rst search, we determine if the initial position is won or not. If it is not won, we do a second search (with different databases) to determine if the initial position is lost or not.

While this reduces the size of the databases by a factor of eight, it still leaves ∗les totaling about 1 GByte. To Nine Men s Morris players, it is clear that most reasonable games will result in at most one or two mills being closed during the opening. For this reason we decided to try to prove the draw using only the 9-9, 9-8 and 8-8 databases, a total of 115 MByte. It is then no longer possible to compute the correct value for every position. But in a position with White to move, we can compute an upper bound on the correct value by assuming all positions not in the databases are won for white and a lower bound if we assume all such positions are won for black. To prove that the game-theoretic value is a draw, we must show that White has both an upper and lower bound of a draw.



Opening Search

We inserted a transposition table at the 16-ply level to further reduce the number of database accesses. Normally, a transposition table includes positions from any ply, possibly even giving priority to those higher up in the tree. We chose to include only the 16-ply positions, because it is far more important to reduce disk I/O than the total number of nodes in the tree.

Finally, all positions that were visited by the alpha-beta searches at the 8-ply level were stored in an intermediate database. This allows games to be played in real-time, because for the ∗rst eight plies only a search to the intermediate database must be performed. The table shows the number of 8-ply positions alpha-beta visited in each of the two searches. We differentiate between positions that could be proven to be at least (most) a draw and those that could not. This does not mean the positions are lost (won), only that using three databases was insuf∗cient to show otherwise.

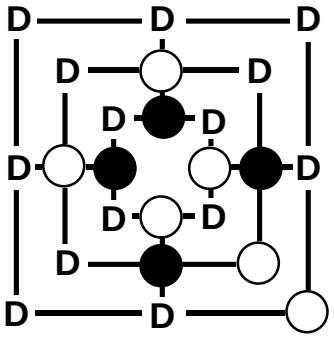
	proven	not proven
White at least draws	15 501	12
White at most draws	4 364	29

Number of searched 8-ply positions

Of the approximately 3.5 million different positions at the 8-ply level, the table shows that only 19 906 were evaluated. The time for a node evaluation ranged from 2 seconds to 15 minutes. The total run-time was about three weeks on a Macintosh Quadra 800. Examining the table, one might be inclined to assume that most positions are drawn. This must not necessarily be the case, perhaps the move ordering heuristic was just successful in selecting drawish moves.

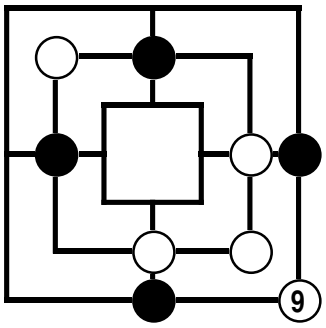
Results

It is now possible to examine some common opening positions. Between equally matched players, games tend to be very drawish during the opening, i.e. most, if not all moves result in a draw (right). However, occasional blunders do occur. The diagram on the lower left shows a position that occurred in a match our Nine Men s Morris program played against Mike Sunley (British Champion) at the Second Computer Olympiad in London, 1990. The program, which was then still based on heuristics, won four games and drew two.

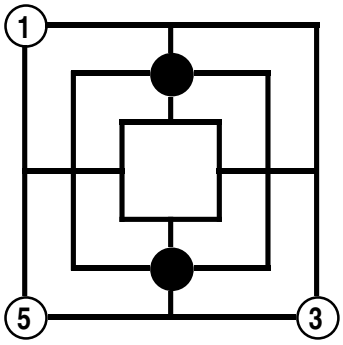


Black to move

Since the initial position is a draw, a natural question to ask is how soon a mistake can be made. The position on the lower right shows one of the earliest losing moves we have found. This positions refutes a common misconception, namely that closing mills is a desirable opening strategy. White follows this strategy with her ∗rst two moves, while Black ignores the potential mill threats. Subsequently, White s third move loses even though she will now easily be able to close two mills during the opening.



White 9 loses



White 5 loses

5.7 Database Compression

During the database computation one byte of information per state is used. Afterwards we can make do with less, for instance for the opening search it is suf∗cient to know whether a position is won or not. The depth information and the loss/draw differentiation can be discarded, allowing eight states to ∗t in a byte.

In this chapter we examine other methods of database compression. These can be divided into game-independent and game-dependent methods. Game-independent methods work reasonably well without knowledge of the underlying data, while game-

dependent methods make use of some Nine Men s Morris knowledge. For a more thorough discussion see [Gasser 91a].

Game-Independent compression

We have experimented with various forms of data compression for Nine Men s Morris. A ∗rst attempt stemmed from the observation that many of the smaller databases consisted largely of draws. This suggested that some form of run-length encoding would be useful. Initial experiments were encouraging, but for the larger databases there are fewer draws and deeper wins and losses. The compression factor (original size divided by compressed size) went down accordingly.

For this reason, we now store the databases using Lempel-Ziv-Welch compression [Welch 84]. This leads to the results shown in the table below:

Database	Compression Factor	Database	Compression Factor	Database	Compression Factor
3-3	2.85	7-4	4.82	8-6	5.92
4-3	12.00	8-3	5.45	9-5	12.05
4-4	100.39	6-6	7.33	8-7	3.68
5-3	26.05	7-5	5.26	9-6	7.63
5-4	32.22	8-4	12.89	8-8	3.60
6-3	26.13	9-3	7.79	9-7	4.93
5-5	62.04	7-6	3.97	9-8	4.44
6-4	5.35	8-5	10.28	9-9	10.20
7-3	5.10	9-4	16.29		
6-5	7.86	7-7	3.32	Total	5.27

LZW compression ratios

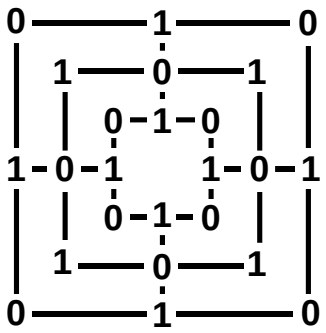
Further reductions can be attained by only storing the values of a few, carefully selected states. The values for the other states can then be computed from the stored states using a shallow search. Which states can be deleted while guaranteeing that a search to a predetermined depth will ∗nd the correct value? If searches up to a depth *d* are allowed, only every *d th* value needs to be stored. As an example, consider the 6-4 database. In addition to draws, it contains values that range from depth 0 (blocked position) to 157 (win in 157 plies). If a 4-ply search is allowed, only every fourth value needs to be stored in order to ∗nd the optimal value and move. For instance, we could only store positions with values [0, 4, 8, , 156]. Three other sets are possible as well, namely [1, 5, 9, , 157], [2, 6, 10, ,154] or [3, 7, 11, ,155]. The set which leads to the best compression can then be chosen. The positions whose values are not needed can be deleted. However, the positions cannot be simply removed from the database, because the ∗le indexing function would be affected. Instead, we set all the ∗deletedτ values to some pre-determined value. The standard data compression techniques will then be more effective.

Note that the above method is not completely game-independent because we use the semantics of the values to provably eliminate certain *values*. Another drawback is that a database look-up must be done for all encountered positions, as we do not a-priori know if the database will contain a deleted value or not.

This second drawback can be avoided by using a similar approach to eliminate certain *positions*. For instance, it is suf∗cient to either store all Black to move or White to move positions. This combined with a two-ply search will ∗nd the best move for any

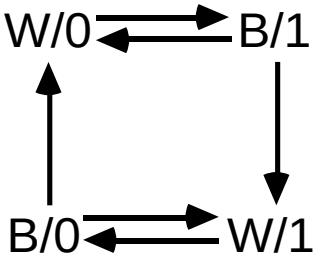
state. And deleted positions can be recognized without accessing the database. In Nine Men s Morris, this method can be further improved because the board graph is bipartite.

We de∗ne the values of the 24 points as shown on the right. The parity of a position is then de∗ned as the sum of the values of all points occupied by White. Any move by White results in the parity going from an odd to an even number or vice-versa. A move by Black does not change the parity unless Black closes a mill.



Position parity

The second diagram shows the parity cycle a game will go through. The letter denotes the player to move (White or Black). The number gives the parity (mod 2) for the white stones. Note that with Black to move, the counterclockwise arrow can only be followed if Black closes a mill. It is suf∗cient to store any one of these four partitions. Whatever partition a position is in, it will return to the same one in an even number of moves. Because Black can close up to six mills before the game is over, a search of up to 16 plies may be needed. But in typical quiescent positions, the search will only be four plies.



Parity cycle

In general, the unused values can be deleted in the same fashion described above. Because of the indexing function chosen for Nine Men s Morris, in this special case it is possible to change the function to eliminate these positions altogether. The table of symmetrical white stone positions is simply reduced to contain only those with the desired parity and only the Black to move or White to move half of the database is used.

Game-Dependent compression

So far, we always saved the value of a position in the database. This makes it easy to determine who has won or lost, while requiring a 1-ply search to ∗nd an optimal move. Alternatively we can store an optimal move. This makes ∗nding an optimal move easy, but to determine if a position is won or lost we must follow an optimal sequence until a terminal position is reached.

In some cases this method may offer further storage savings. For instance if the number of different moves is smaller than the number of database value, a saving occurs immediately. But even if this is not the case, we may be able to improve the compression by using a move generation heuristic. This heuristic orders the possible moves. Then instead of storing the move itself in the database, we store the index of that move in the ordered list. If the heuristic sorts many optimal moves to the front, then the database will consist mainly of low numbers. This again allows the standard compression algorithms to be used to further advantage.

These techniques can be pro∗tably applied to Nine Men s Morris. For instance examining the endgame databases (x-3 databases) shows that a large fraction of all wins and losses occur in one or two plies. An optimal move in these positions is easy to ∗nd. For the wins in one, a move that closes a mill is optimal. For the losses in two any move is optimal.

ID3 Rule Extraction

Up to now we discussed various methods for reducing one of the shortcomings of endgame databases, namely their size. Humans however are not satis∗ed with merely a small database. While this enables humans to play optimally, they do so without real understanding. Humans ask *why* they should play a certain move.

To answer this satisfactorily, we must use general concepts humans readily comprehend. For instance, it is better to explain that a position is won ∗ because white has an open millτ than to say ∗ because white occupies positions a7 and a4 and has a move d1-a1τ. The ∗rst statement is more general and better matches the way humans think about positions.

An example how this can be done is shown by chess grand master John Nunn. In a book of over 300 pages, he extracts knowledge for humans from the KRPKR chess database [Nunn 92]. We would like to do something similar for the 3-3 database. While this is small in terms of the number of positions it contains, our experience has shown that it is quite dif∗cult for humans to play correctly.

We ∗rst de∗ne a set of attributes. The ID3 algorithm [Quinlan 83], [Kunz 91] is then used to generate a decision tree. The attributes are re∗ned until all won, lost and drawn positions can be correctly discriminated.

The following attributes can discriminate between wins, losses and draws in the 3-3 game. The information in parentheses indicates the attribute number and whether the attribute is calculated for the (P)layer to move s stones, the (O)pponent s stones or the (E)mpty stones. For example the ∗rst attribute ∗Number of mill threatsτ is calculated for both the player to move s mill threats (Attribute number 2) and the opponent s mill threats (Attribute number 1).

Number of mill threats (O:1/P:2): A player has a mill threat if there is an empty point on the board, which would create a mill if occupied by the player.

Open forks (O:3/P:4): A player has an open fork if there is an empty point on the board, which would create two mill threats if occupied by the player.

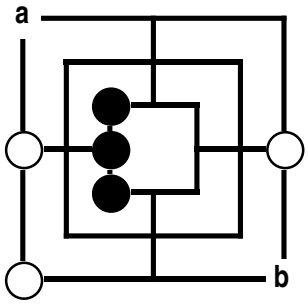
Closed mills (O:5/P:6)

Blocked mill threats (O:7/P:8): A player has a blocked mill threat if her opponent occupies a point on the board, which would be a mill threat if empty.

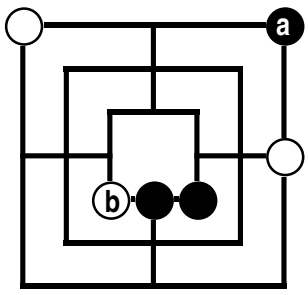
Blocked forks (O:9/P:10): A player has a blocked fork if her opponent occupies a point on the board, which would be an open fork if empty.

Number of empty rings (11): The board consists of three equivalent rings. The number of completely empty rings is counted.

Number of isolated points (O:12/P:13/E:14): A point is isolated, if both lines it lies on are otherwise unoccupied.

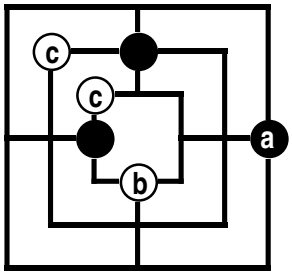


Closed black mill. White mill threat (a) and open fork (b).



Blocked white fork (a). Blocked black mill threat (b). One empty ring.

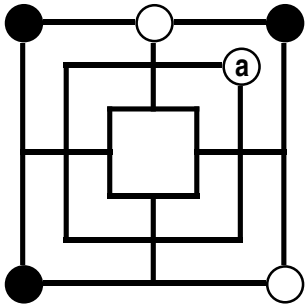
Number of balanced points (O:15/P:16): If an empty point ∗seesτ all the stones of one color and none of the other on the six lines adjacent to it (the two lines this point lies on or the four lines adjacent to these two), then it is a balanced point.



One isolated black (a) and white stone (b). Two white corner points (c).

Number of corner points (O:17/P:18): The points on the board can be classi∗ed according to the number of adjacent points they have. If there are two adjacent points, it is a corner point otherwise a central point.

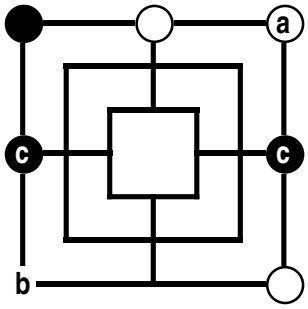
Opponent s mobility (19): If an opponent s stone lies on a line which does not contain any of the player s stones, this line has mobility. If the player moves in such a fashion as to minimize the number of lines with mobility, how many are there?



White to move. The opponent s mobility is zero. White has one (vertical) corner-free line.

Safe threat (20): Can the player to move either close a mill or make a move so that:

- the opponent has no mill threat,
- the player has a mill threat, and
- the opponent cannot create a mill threat by blocking the player s threat or if she can, the player can thereafter block it and recreate one of her own.



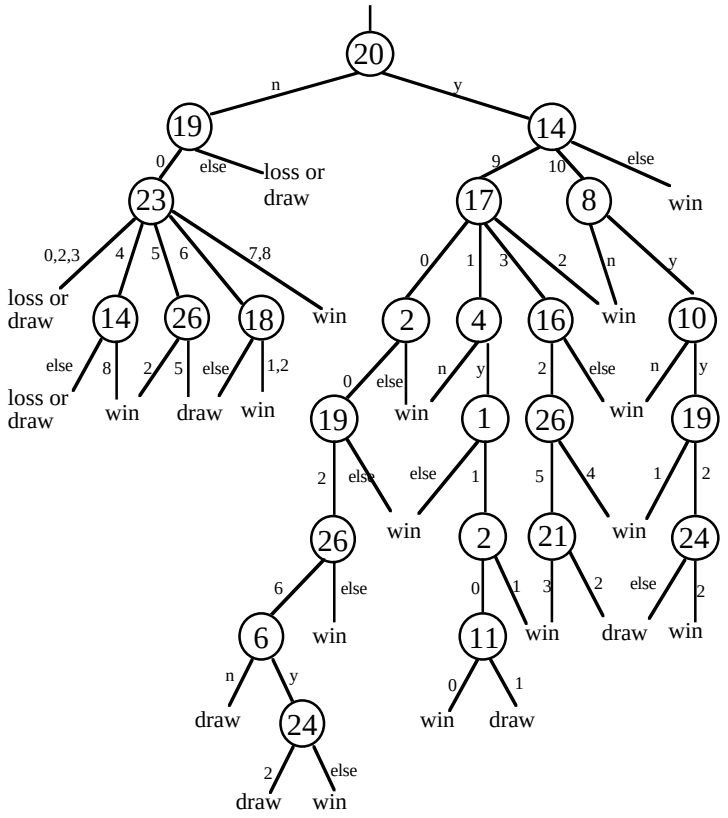
White to move has safe threat (a-b). Black has two central-free lines (c horizontal).

Number of corner-free lines (O:21/P:22/E:23): Consider all the lines that pass through corner points (of the respective color). If the other two points on the line are empty and the two other lines these two points lie on are also empty, it is a corner-free line.

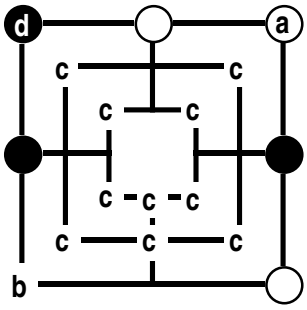
Number of central-free lines (O:24/P:25/E:26): Same as corner-free lines, except lines through central points are considered.

These 26 attributes are suf∗cient to discriminate between wins, losses and draws. But the resulting decision tree is too large for human use. The tree did show that wins and losses are easy to tell apart. The draws cause the most trouble. However for correct play, it is suf∗cient to be able to recognize all won positions. If this can be done, losses and draws can be discriminated by looking at their successors.

Separating the wins from the draws and losses leads to the following tree (Attribute numbers in the circle, values on the edge):



ID3 decision tree



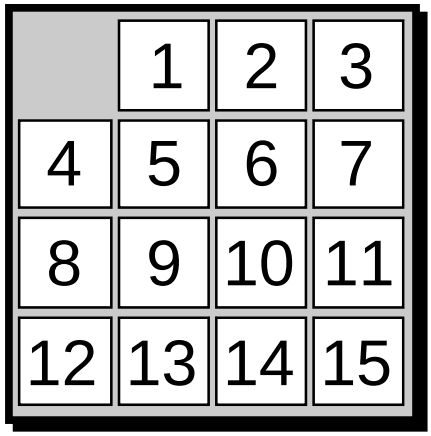
White to move

To better understand this tree, the diagram shows a position, which is won in 25 plies. According to the decision tree, we must check if White has a safe threat. This is the case, for instance the move *a-b* is a safe threat. How many isolated empty points are there? The points marked *c* are isolated and empty. Next attribute: Does White have blocked mill threats? Yes, there is one at *d*. But White has no blocked forks, so this position is correctly classi∗ed a win.

This decision tree is a good example of a compression algorithm. A large effort is involved in ∗nding a good set of attributes. Once found, the attributes and decision tree can be stored using less memory than the original database. But the original objective of explaining the logic behind the optimal moves is not satisfactorily attained. Most positions are easy to classify, but the handful of dif∗cult positions dramatically increase the size of the tree. An alternative for human use may be to allow a few errors in return for a greatly simpli∗ed tree.

6
Sliding Tile Puzzles

Sliding tile puzzles come in various shapes and sizes. They are usually played on rectangular boards with rectangular pieces, but more intricate shapes occur as well. From some random position, the object is to move a tile or set of tiles into pre-de∗ned locations. This is done by repeatedly sliding tiles into vacant spaces. In this chapter we will focus primarily on Sam Loyd s 15-Puzzle. It is played on a 4x4 board with 15 equally sized tiles and one space. The 15-Puzzle goal state is depicted below.



Goal state

Since solving the 15-Puzzle seems easy for humans, it is not surprising that polynomial time algorithms for this and related problems exist [Sucrow 92]. However, solving these problems in the minimal number of moves is NP-hard [Ratner 90].

The 15-Puzzle is often used to test heuristic search methods. It has also been used as a test for exhaustive search techniques by Korf [Korf 85] in his work on the IDA* algorithm or by Hansson, Mayer and Yung who give an improved lower bound measure [Hansson 92]. It seems ideally suited to tests of this nature, because the rules are simple and the dif∗culty of the puzzle can be varied depending on the size of the board.

For exhaustive search, the 15-Puzzle (4x4) seems to generate the right problem complexity. Smaller versions, for example the 8-Puzzle (3x3), are too simple. Since the 8-Puzzle has only 181 440 different reachable positions, the entire space can be searched in a few seconds. The 15-Puzzle has 10^{13} reachable positions, making it a challenging size for exhaustive search. In contrast, heuristic search algorithms are increasingly being tested with the 24-Puzzle [Liempd 93].

In this chapter, we will examine the following two aspects of the 15-Puzzle:

- *Optimal solutions:* The more exactly we can determine a lower bound for a given position, the smaller the search tree that is needed to ∗nd an optimal solution. We show how databases can be used to improve the lower bound.
- *Hardest position:* It is still unknown what the maximum number of moves needed to solve any position is. This problem can be approached from two ends. We search for a position that requires many moves to solve optimally and we develop algorithms that provably solve any position in a certain number of moves.

6.1 Optimally Solving the 15-Puzzle

To ∗nd optimal solutions to the 15-Puzzle, the Depth-∗rst Iterative Deepening algorithm described in [Korf 85] is commonly used. A pseudo-code implementation adapted to the 15-Puzzle is shown below:

```
FUNCTION Solve(state, depth): BOOLEAN;
BEGIN
      IF depth+Estimate(state) <= MaxDepth THEN
         IF GoalState(state) THEN
            PrintSolution;
            RETURN TRUE;
         ELSE
            FOR dir := north TO east DO
               aSucc := Slide(state, dir);
               path[depth] := dir;
               IF (aSucc  none) AND Solve(aSucc, depth+1) THEN
                  RETURN TRUE
               END;
            END;
         END;
      END;
      RETURN FALSE;
END;

MaxDepth := Estimate(InitState);
WHILE NOT Solve(InitState, 0) DO
      MaxDepth := MaxDepth+2;
END;
```

Iterative deepening repeatedly performs a depth-∗rst search. Since *Estimate* returns a lower bound for the number of moves needed to solve the initial position, the ∗rst search depth (*MaxDepth*) is set to that estimate. As long as no solution is found, *MaxDepth* is increased by two and another search is started. *MaxDepth* can be increased by two, because the parity (even or odd number of moves) of the solution length is known from the position of the space.

Solve performs a branch-and-bound (depth-∗rst) search for the goal state. It uses the following functions:

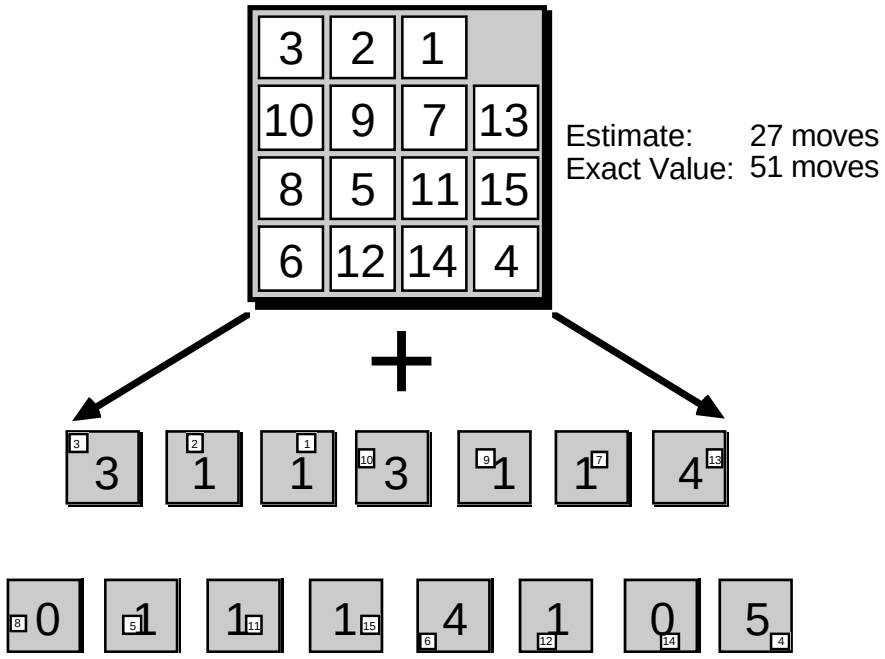
- *Slide(state, dir):* Generates a dependent state by sliding the space in direction of *dir*. If the border is encountered, ≅none is returned. Our implementation ∗rst slides the space to the north, then west, south and east.
- *GoalState(state):* Returns true if *state* is the goal state.
- *PrintSolution:* Prints the solution stored in the *path* variable.
- *Estimate(state):* A lower bound for the cost of moving from *state* to the goal state.

The lower bound returned by *Estimate* is critical to the performance of this algorithm. With an exact lower bound, the search tree degenerates to a linear chain, while a lower bound of zero would lead to a complete tree search, with no pruning whatsoever.

A number of lower bounding mechanisms have been proposed for the 15-Puzzle. One simple, yet effective lower bound is the Manhattan distance [Korf 85]. This heuristic determines the number of moves each tile needs to get to it s goal positions if it were alone on the board. This heuristic seems suf∗cient to solve any 15-Puzzle position, but in some cases rather large search trees are needed. The linear-con τict heuristic [Hansson 92] is a more recently proposed heuristic that reduces the search tree size by approximately a factor ten compared to the Manhattan distance. In the following chapters we show how databases can be used to further improve the lower bound and consequently reduce the search tree size.

6.2 Many-Empty Databases

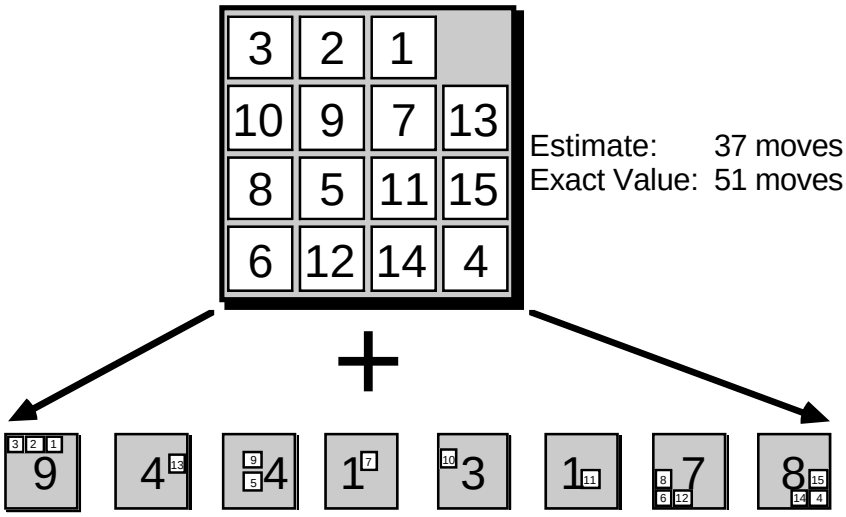
The Manhattan distance lower bound can be regarded as the decomposition of the original problem into ∗fteen 1-Tile patterns. The sum of the number of moves needed to solve all of these 1-Tile patterns is a lower bound to the original problem, because any solution to the original problem also simultaneously solves all 1-Tile patterns.



1-Tile Decomposition

A drawback of the Manhattan distance lower bound is that it neglects all information between tile interaction. For instance, if two tiles are next to each other and must switch places, the Manhattan distance fails to see that it will take at least four moves to do so.

One lower bound that recognizes these and similar situations is linear-con τict [Hansson 92]. Another method is to generalize the decomposition to include 2-Tile and larger patterns. Splitting the position shown above into larger tile patterns improves the lower bound:

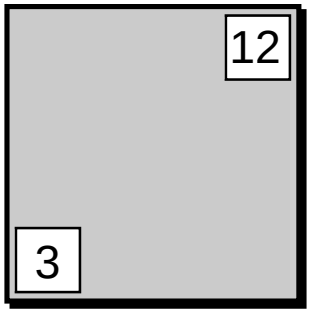


Improved Decomposition

The minimal number of moves needed to solve any n-Tile pattern can be computed using retrograde analysis and stored in databases. The number of different n-Tile patterns are:

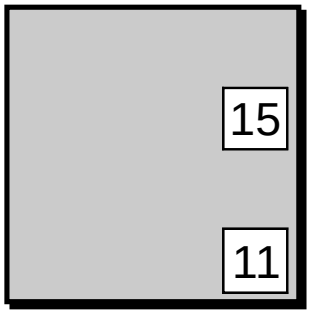
$$\binom{15}{n} \cdot \frac{16!}{(16-n)!}$$

Not all n-Tile patterns are equally useful. For instance, the 2-Tile pattern shown at right requires 12 moves to solve. If we divide this problem into two 1-Tile patterns and apply the Manhattan distance heuristic, we also ∗nd a lower bound of 12 moves. Therefore, when splitting a 15-Puzzle position into n-Tile patterns, there is no point in using this 2-Tile pattern, because the simpler Manhattan distance would have done equally well.

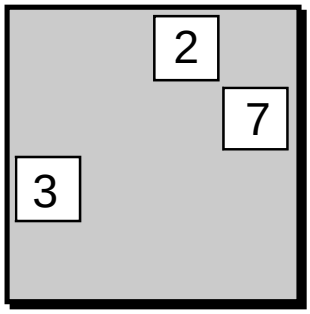


Not a useful 2-Tile pattern

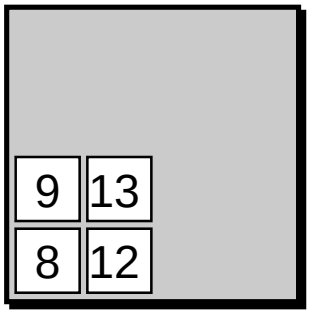
A useful 2-Tile pattern is one which predicts a better lower bound than the Manhattan distance. Analogously, a 3-Tile pattern is only useful if a better lower bound is attained than by splitting it into a 2-Tile and 1-Tile pattern. In general an n-tile pattern is useful, if there is no way of dividing it into a k-Tile and (n-k)-Tile pattern with the same lower bound. The following diagrams show some useful patterns.



Useful 2-Tile pattern requiring ∗ve moves



Useful 3-Tile pattern requiring seven moves



Useful 4-Tile pattern requiring six moves

Since the 1-Tile patterns are equivalent to the Manhattan distance, only the 2-Tile, 3-Tile and 4-Tile databases were computed. The table shows the number of useful n-Tile patterns:

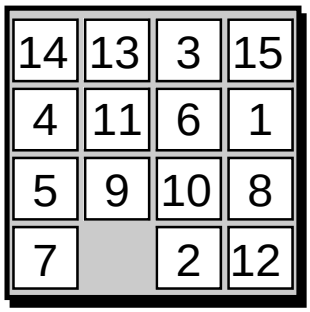
2-Tile Patterns	3-Tile Patterns	4-Tile Patterns
252	5 135	111 930

Number of useful patterns

Computing a lower bound using n-Tile patterns is done in two steps. First all useful patterns into which the position could possibly be split must be found. Then we determine a set of non-overlapping patterns that maximize the lower bound.

Finding the matching patterns for a given position could be implemented by searching through all useful patterns after each move. This is inef∗cient because moving a single tile only affects patterns that require that tile to be at a speci∗c position. Since the speed of the lower bounding mechanism is critical to the run-time of the entire algorithm, an incremental approach is needed, i.e. the list of matching patterns must be updated after each move. This can be done by ∗rst deleting any patterns in the list which depend on the moving tile being at its old position. Then all useful patterns depending on the moving tile being in its new position are added. There are 240 different tile/position combinations (15 different tiles at 16 different positions). For each of these, a trie stores all useful patterns that use this tile/position combination. When a tile is moved only the corresponding trie must be searched for matching useful patterns.

As an example, consider the following 15-Puzzle position and the matching 2-Tile and 3-Tile patterns that were found for it.



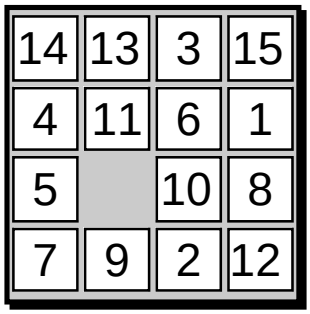
Matching Patterns

```
  9/13     1/ 3/ 6
  2/ 6     1/ 6/15
  2/10     4/ 5/ 9
  8/ 9     6/ 9/11
  8/10     6/10/11
8/9/13     2/ 8/10
```

15-Puzzle position

If tile number nine is moved, ∗ve patterns, namely 9/13, 8/9, 8/9/13, 4/5/9 and 6/9/11 are deleted. Then the trie corresponding to tile number nine at position number thirteen (its new position) is searched for matching patterns. Only one match is found, namely 9/13.

The next diagram shows the position after tile number nine is moved, along with the updated pattern list. The third column lists how many moves these patterns improve the Manhattan distance bound. It is possible for a tile to occur in more than one useful pattern. To obtain a correct lower bound, we choose a set of non-overlapping patterns. We would like to choose a set that maximizes the lower bound. In this example, the best set of non-overlapping patterns improves the Manhattan distance lower bound by eight moves. One example for such a set is 9/13, 1/3/6 and 2/8/10.



Matching Patterns	Improvements
9/13	2
2/ 6	2
2/10	2
8/10	2
1/ 3/ 6	2
1/ 6/15	2
6/10/11	2
2/ 8/10	4

After moving tile nine

Finding this maximum set leads to another, potentially slow, search problem. However, the number of matching useful patterns for a typical position is fairly low, so this search accounts for less than 20% of the time needed to ∗nd the matching patterns.

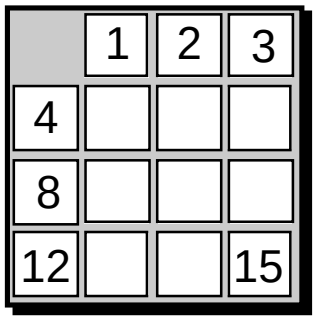
The expected number of matching useful patterns per position can be found by assuming that useful matching patterns occur in a position with the same ratio they do over all patterns.

2-Tile Patterns	3-Tile Patterns	4-Tile Patterns
1.05	1.53	2.56

Expected number of patterns

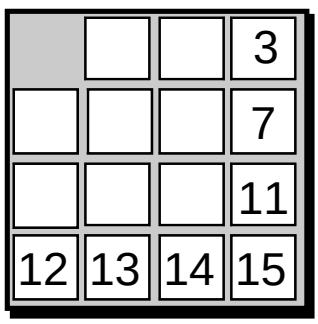
6.3 One-Empty Databases

In the last section we used patterns consisting of a few tiles and many spaces to compute a lower bound. This allowed the lower bound to be computed as the sum of different pattern bounds. An advantage of this method is that all tiles played a part in the lower bound. However, because of the many spaces, the tile mobility is higher than in the real puzzle where only one space can be moved into. This may lead to loose bounds.



Inner database

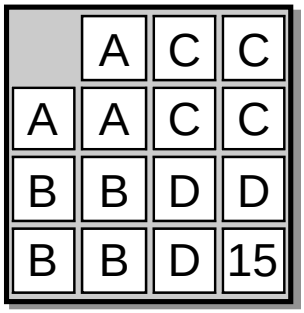
It is also possible to compute databases with only one space. In order that the databases do not grow too large, some tiles are made indistinguishable. The ∗rst diagram shows the Inner database (272 432 160 states). Its goal state only requires the space, the 15-tile and all tiles in the top row and left column to be at their correct positions. The remaining tiles can be at any of the blank positions. The database contains the number of moves needed for any permutation of these tiles (space, blank, 1, 2, 3, 4, 8, 12, 15) to reach this relaxed goal state. The maximum distance any permutation can be from the relaxed goal state is 66 moves and the average is 41.0 moves.



Fringe database

A similar database was ∗rst computed by Culberson and Schaeffer [Culberson 94]. Their so-called Fringe database (272 432 160 states) has a maximum distance from the relaxed goal of 64 moves and an average of 41.5 moves.

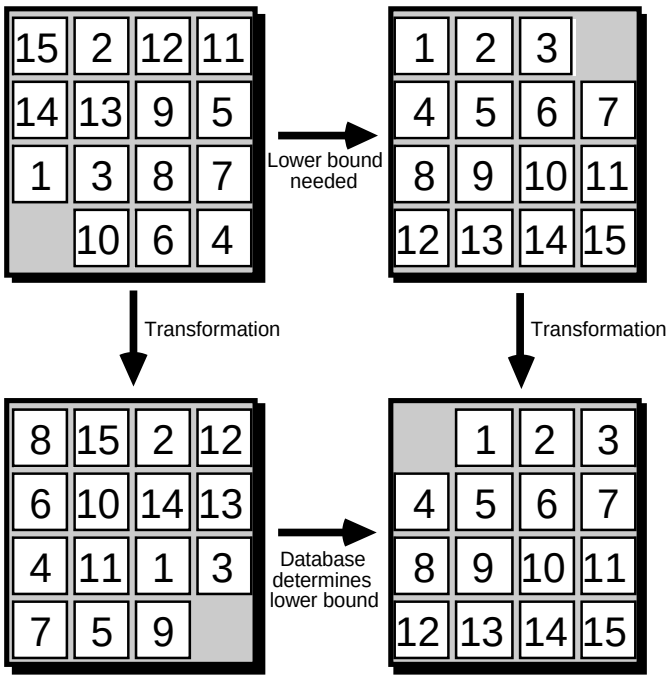
The third example shows the goal state of the Local database (529 729 200 states). Here the space and 15 tile must be in their correct positions, and the remaining tiles must be near their proper positions. For instance, tile number one can be in any position marked *A*, tile six in any position marked *B* etc. In the Local database, the furthest a position can be from the relaxed goal state is 70 moves. The average distance is 42.2 moves.



Local database

A lower bound can be obtained by transforming the position s tiles into the corresponding Inner, Local or Fringe database tiles and looking up the optimal relaxed problem depth in the database. This is a lower bound because any solution to the original problem will also solve the relaxed problem, so the shortest solution to the relaxed problem is a lower bound.

Culberson and Schaeffer [Culberson 94] observed that even more information can be gained from the databases. If the space in the goal position is moved three squares to the right, the upper right position in the following diagram results. If a lower bound *l* for reaching this modi∗ed goal can be found, then a lower bound to the 15-Puzzle goal is *l-3* moves. Such an *l* could be obtained by computing a further database, however Culberson and Schaeffer showed that *l* can be obtained from the original databases.



A position and its goal (top), the analogous position with the 15-Puzzle goal (bottom)

Their observation is that by τipping the modi∗ed goal position across the vertical axis and applying the permutation v to the tile numbers, the original 15-Puzzle goal results.

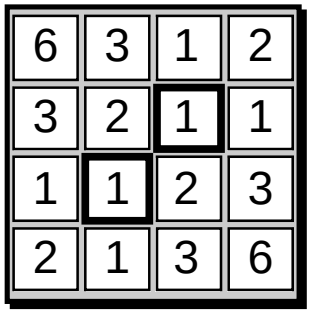
v : (0)(1 3)(2)(4 7)(5 6)(8 11)(9 10)(12 15)(13 14)

This same transformation can be applied to the position, resulting in the modi∗ed position on the lower left. Looking up this modi∗ed position in the database returns a lower

bound for the number of moves needed to reach the 15-Puzzle goal. This bound is equal to *l*, because a path from the modi∗ed position to the goal can be converted (by τipping across vertical axis) to a path that leads from the original position to the modi∗ed goal.

This same idea can be used to create further modi∗ed goal states. Any path from the 15-Puzzle goal state to a state where the space is also in a corner, can be regarded as a modi∗ed goal state. For instance, instead of moving the space to the upper right corner, it can be moved to the lower left corner. Or instead of moving it to the upper right corner in three moves, a path of length ∗ve can be used. In the ∗nal algorithm, all three, ∗ve and six move paths to corners are used. Each of these paths induce a transformation that can be applied to the position, yielding a further lower bound. The maximum of all these bounds is then used.

The Inner, Fringe and Local databases are all fairly large. Unlike Culberson and Schaeffer who use a 1 GByte machine, our machine only has 72 MByte of main memory. Therefore it is not possible to load the entire database into memory, and accessing the ∗le on disk would be too slow. The solution to this problem is only to use a small portion of the entire database. Speci∗cally, for all databases, only the positions where the space is in position number six are loaded. Since our databases are diagonally symmetric, position number nine is also available. To obtain a lower bound if the space is somewhere else, the space is moved to the nearest six or nine square and the lower bound for this position is looked-up. There are often a number of shortest move sequences that place the space in the six or nine position, therefore all the resulting positions are looked-up. A lower bound for the original position is then found by taking the maximum lookup-value and subtracting the distance moved.



Number of database look-ups depending on the space position

6.4 Re∗nements and Results

In two-player games it is often possible for the same position to occur multiple times in the search tree, because players can transpose their moves. In order that the position is not evaluated twice, the result of the ∗rst evaluation is stored in a transposition table for later reuse. Because the 15-Puzzle state space graph is cyclic, it is also possible for the same position to occur numerous times. Therefore, transposition tables can also be applied to the 15-Puzzle [Reinefeld 94]. If the same position is reached twice in the 15-Puzzle search tree, it is possible that the position occurs at different search depths. Clearly, the position with the longer path from the root is of no interest whatsoever. However the transposition table has no control over which position will be found ∗rst in the search. So if the position with the longer path is evaluated ∗rst, it must be reevaluated when the shorter path is found. This leads to the idea of using a ∗nite state automaton to avoid position reoccurrences [Taylor 93]. The automaton is used as a

move generator, and will only allow move sequences that lead to positions which cannot be attained in less moves. If there is more than one way of reaching a position in the same number of moves, exactly one of the sequences is allowed. Since this automaton is ∗nite, it cannot detect transpositions that are deeper than a ∗xed number of moves and thus some positions will still be evaluated more than once.

The number of position evaluations can be further reduced by the observation that there are often forcing moves in the 15-Puzzle. For instance when a move leaves the space in the corner, there is no point in moving back, so there is only one possible continuation. The automaton will also detect additional forcing moves, where only one move will lead to a position that could not have been reached with fewer moves. These forcing moves are especially important in our implementation, because if the space is in the corner we would do up to six database look-ups to ∗nd a lower bound. Because of forcing moves, no lower bound will be computed for this position, rather the forcing move is immediately executed.

To ∗t as many databases as possible into a 72 MByte machine, three database states are stored in two byte. This is possible because all lower bounds in the databases have odd length (because the space is at position six). Thus, there are less than 40 different database values. The memory requirements of these features is as follows:

Many-Empty Databases		One-Empty Databases		Automaton	
2-Tile:	1 512 Byte	Inner:	21 621 600 Byte	12-move:	2 841 168 Byte
3-Tile:	41 080 Byte	Fringe:	21 621 600 Byte		
		Local:	42 042 000 Byte		

Memory requirements

The following table shows the number of tree states expanded for ∗ve lower bounds based on the test suite of 100 problems introduced in [Korf 85]. A more detailed tabulation can be found in appendix B. The ∗rst lower bound uses the 2-tile and 3-tile patterns. Although using 4-Tile patterns reduced the search tree size by another factor two, they were not used because the total run-time was slower. This was due to the large number of useful 4-tile patterns, which made ∗nding all matching patterns more expensive. The next three lower bounds additionally use the automaton and one of the three one-empty databases. The ∗nal algorithm uses all three one-empty databases.

2+3 Tile	Inner	Fringe	Local	Final
743 517 375	70 547 812	45 837 107	72 525 376	13 319 072

Search tree size

Surprisingly, even though the Local database has the highest average lower bound, the search tree is larger than with the Inner or Fringe database. Further experiments are needed to determine whether this is due to the non-randomness of the test suite or an inherent property of the database.

As a comparison, the following table shows the expanded states and memory requirements for the same problems using Manhattan distance [Korf 85], Linear-Conτict [Hansson 92] or Fringe and Corner databases [Culberson 94].

	Manhattan	Linear-Conflict	Fringe+Corner
Expanded states	36 302 808 031	3 759 631 279	19 850 843
Memory requirements	240 Byte	825 Byte	630 MByte

Comparison to other lower bounds

While the number of expanded states using our lower bound is only slightly lower than using the Fringe and Corner lower bound, the memory requirements are about a factor seven lower. Presumably our algorithm is slower because many more database look-ups are performed. However, our algorithm was not optimized for run-time, instead the search tree was minimized. A number of simple improvements would improve the speed considerably, for instance maintaining the modified positions incrementally, or not computing lower bounds for as many modified goal positions. Especially the modified goal states six moves away from the real goal state are more expensive to compute than the resulting gain in tree size.

6.5 Hardest 15-Puzzle Position

Because the smaller sliding tile puzzles have few positions, all positions can be optimally solved. The following table summarizes the maximum distances any position can be from the goal in different sized puzzles:

Board Size	2x2	2x3	2x4	2x5	3x3	3x4
Maximum	6	21	36	55	31	53

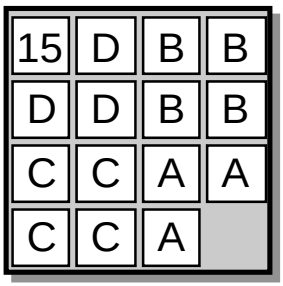
Maximum distance

Unfortunately, the 15-Puzzle has too many positions for all of them to be exhaustively analyzed. Nonetheless we would like to determine a position whose optimal solution requires the most moves.

A method to solve this problem was first proposed by [Schaeffer 94]. First, we attempt to find a 15-Puzzle state with as long an optimal solution as possible. Then, we show how an upper bound for the number of moves needed to solve any 15-Puzzle state can be found. If these two bounds meet, a hardest state has been found.

Lower Bound

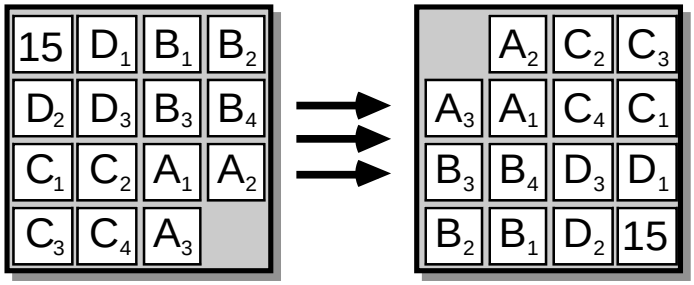
Since we cannot solve all 15-Puzzle states, the databases can be used to find difficult candidates. Of all the databases, the Local database has the state with the highest lower bound, namely 70 moves (diagram). This position can be regarded as a template representing 10 368 (4!·4!·3!·3!/2) actual states. All of these require at least 70 moves to solve, but there will likely be states that require quite a few moves more.



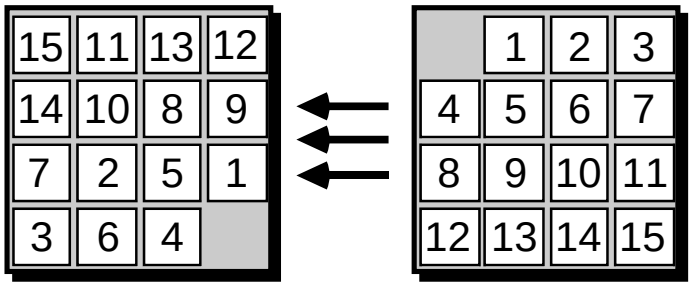
70 moves

What is the most efficient method to compute the true value for all these states? It is not efficient to solve them independently, because experiments showed that many require more than a day to solve. Since solving these positions will cause similar search trees to

be considered, it seems better to use a method that solves all problems simultaneously. We start with the 70 move state and note where all the indistinguishable A s, B s, C s and D s originated from (subscripts in the following diagram). Then we ∗nd all paths to the relaxed goal state that can be reached in a given number of moves using the standard branch-and-bound algorithm. The following diagram shows a possible way of reaching the goal state in 80 moves. The analogous position in the real puzzle is the goal state. By reversing the path used to solve the relaxed problem, a position is obtained that can reach the goal state in at most 80 moves.



80 move solution



Analogous 80 move solution

With this method it is possible to determine the solution length of all 10 368 positions. First all 70 move solutions are generated, there are 188 of these. Then positions that can reach the goal in 72 moves are generated. Included among these 2 126 are also the 70 move positions determined previously. Therefore there are 1 938 positions whose optimal solution requires 72 moves. Running this method for about two months on a Macintosh Quadra 800, we were able to ∗nd 10 349 positions that required 78 moves or less to solve. This left 19 positions that could potentially require more than 78 moves. Each of these positions was solved independently to determine their true depth. Thirteen required 80 moves. These 80 move positions (shown in appendix C) are harder than any known prior to this work. The following table shows the number of positions at each depth (includes symmetrical positions):

Depth	70	72	74	76	78	80
Positions	188	1 938	4 597	3 068	564	13

Number of positions per depth

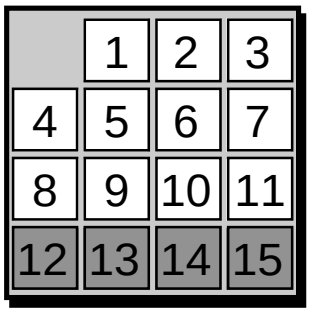
If there are any 81 move positions, there must be a move that leads to an 80 move position. Therefore, we determined the distance of all neighbors of the thirteen 80 move

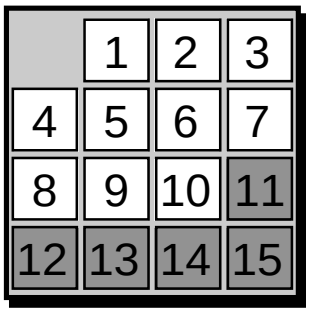
positions. Unfortunately, none of these require 81 moves. However, this does not prove that there are none.

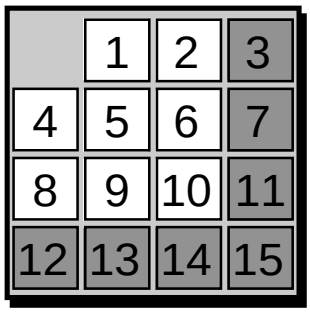
Upper Bound

Humans solve the 15-Puzzle by dividing it into a series of subgoals. For instance they ∗rst ∗x the bottom four tiles, then the next line of four before completing the rest. This method does not lead to optimal solutions, but it can be used to ∗nd an upper bound for the 15-Puzzle [Schaeffer 94]. Speci∗cally, we divide the puzzle into a series of non-conτicting subgoals (i.e. once a subgoal is achieved it will not be destroyed by later moves). An upper bound for the 15-Puzzle is then the sum of the upper bounds for all subgoals.

The diagrams below shows different methods of solving the 15-Puzzle in two steps. The ∗rst subgoal consists of placing all the darkly shaded tiles into place. The second subgoal arranges the remaining tiles. The table lists the number of states in the database and the maximum number of moves needed to solve each subgoal.







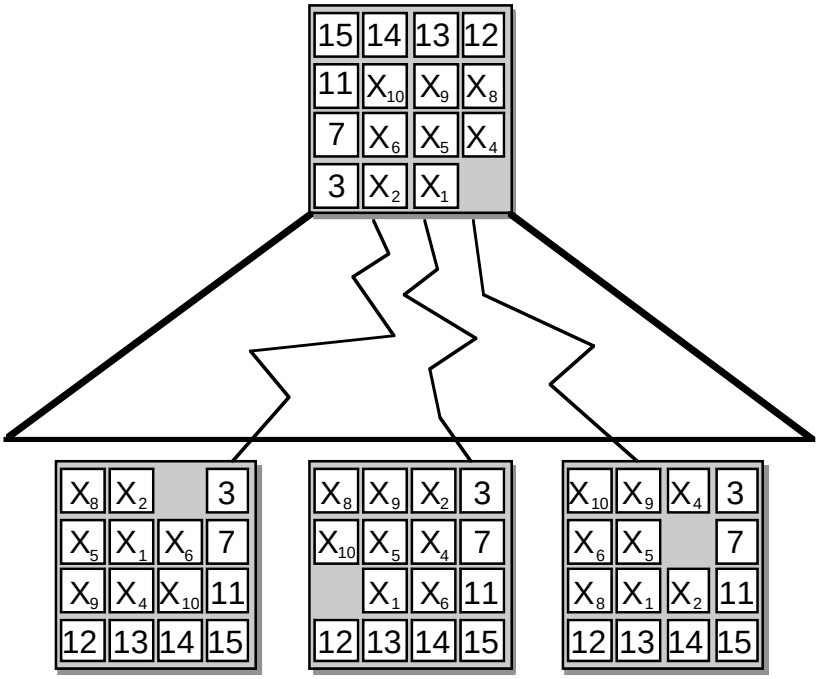
Step 1	States: 524 160	States: 5 765 760	States: 272 432 160
	Longest: 46	Longest: 52	Longest: 61
Step 2	States: 239 500 800	States: 19 958 400	States: 181 440
	Longest: 53	Longest: 49	Longest: 31
Total	States: 240 024 960	States: 25 724 160	States: 272 613 600
	Longest: 99	Longest: 101	Longest: 92

The strictest bound is proved by the third method. From the table we see that it takes at most 61 moves to correctly arrange the bottom row and right column. What remains is the 8-Puzzle, which can be solved in at most 31 moves. This proves that any 15-Puzzle position can be solved in at most 92 moves. This bound can be improved with a simple observation. After the ∗rst step has been completed, the space must be adjacent to one of the shaded tiles. If this were not the case, the shaded tiles could have been arranged correctly with fewer moves. But the 8-Puzzle states that require 31 moves to solve do not have the space on any of these squares. The hardest state with the space on one of these squares requires only 30 moves to solve. This lowers the upper bound to 91 moves [Schaeffer 94].

A method devised by Culberson and Schaeffer for further lowering the upper bound is to generate all possible 91 move positions and compute their correct lower bound. A possible 91 move position can be created by taking a 61-move step 1 con∗guration and placing the remaining tiles in a permutation that leads to a 30-move step 2 con∗guration. Since there are six 61-move positions in step 1 and 221 30-move positions in step 2,

there are at most 1 326 (6·221) positions to be solved. Solving these may increase the lower bound of 80 moves. If no 91-move position exists, all possible 90-move positions must be considered. These can be generated either as a 61-move step 1 and 29-move step 2 position or as a 60-move step 1 and 30-move step 2 position. This continues until the hardest position is found. However, if 80 moves is the hardest position, there will be much computation involved in proving this. In particular, all possible 81-move positions must be solved, and there are billions of these.

A better method is to adapt the algorithm we used to simultaneously solve all 10 368 positions created by the 70-move template. In the present case, we have the 61-move template shown in the following diagram.



61-move solutions

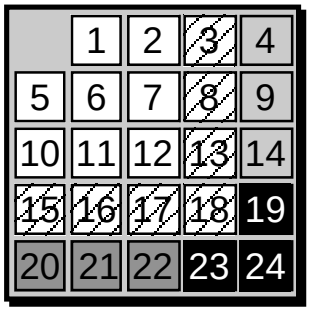
We ∗rst ∗nd all possible 61-move paths that solve this template. Let us assume there are three different paths. If we insert all *8!/2* possible X-tile permutations in the original position, then for each permutation we ∗nd three different positions that can result after 61 moves. Only if all three positions take at least 30 moves to solve, is that permutation a possible 91-move position. If any of the original *8!/2* permutations can not be eliminated, we ∗nd all ways of solving the 61-move template in 63 moves. If any of these 63-move paths lead to positions which can be solved in less than 28 moves, that permutation can also be eliminated. This process continues until all permutations have been shown not to improve the lower bound. If any permutations improve the lower bound, they will not be eliminated with this process. When few permutations remain, it is necessary to solve these individually in order to determine whether they improve the lower bound or not.

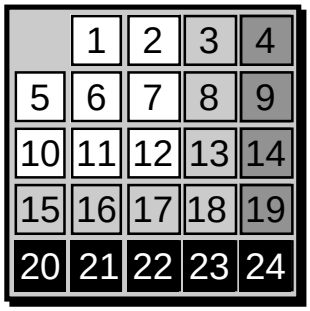
Using this algorithm, we proved that all 15-Puzzle states require less than 88 moves to solve.

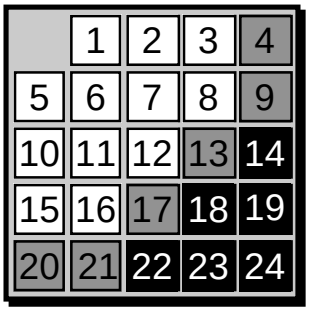
6.6 Solving the 24-Puzzle

At present, the 24-Puzzle cannot be solved optimally in reasonable time, therefore researchers concentrate on finding good heuristic solutions. The best results so far were obtained by [Liempd 93]. He uses the Manhattan distance lower bound and limited-search algorithms to solve randomly generated 24-Puzzle positions. He reports an average solution length of 112 moves with search trees averaging $1.12 \cdot 10^8$ nodes.

His results could be further improved simply by using a more informed heuristic, for instance linear-conflict or many-empty databases. However, he is more interested in methods of constraining the search tree by pruning unlikely solution sequences. A different way of doing this is to divide the final goal into a series of non-conflicting subgoals. Since each of the subgoals are easier to achieve than the final goal, the search trees can be reduced in size. Because of memory limitations, it is not possible to use a 2-step approach as we did for the 15-Puzzle. The following table shows three different multi-step methods. For each step, the longest move sequence and the average number of moves is shown. The tile shading indicates the subgoal in which the tile is placed into position (darker is earlier). The 3-step method uses fairly large databases that have not yet been computed, but would likely lead to further improvements.







Step 1	States: 156 860	States: 127 512 000	States: 1 216 630 800
	Average: 30.5	Average: 43.8	
	Longest: 54	Longest: 74	
Step 2	States: 175 560	States: 1 860 480	States: 147 026 880
	Average: 30.4	Average: 33.3	
	Longest: 53	Longest: 58	
Step 3	States: 93 024	States: 272 432 160	States: 1 616 630 400
	Average: 27.0	Average: 38.5	
	Longest: 47	Longest: 61	
Step 4	States: 272 432 160	States: 181 440	
	Average: 38.5	Average: 22	
	Longest: 61	Longest: 31	
Step 5	States: 181 440		
	Average: 22		
	Longest: 31		
Total	States: 273 039 044	States: 401 986 080	States: 2 980 288 080
	Average: 148.4	Average: 137.6	
	Longest: 246	Longest: 224	

The sum of the longest sequences give an upper bound for the number of moves needed to solve any 24-Puzzle position. The 4-step method thus proves that 224 moves are suf∗cient (actually 219 since the 15-Puzzle can be solved in 87 moves or less). The total average is an estimate for the number of moves needed to solve a random position. Note that no search is needed to attain these averages. A greedy strategy, that simply chooses any move that reduces the distance to the goal of that step suf∗ces.

No experiments have been done, but it is seems likely that the average can be much improved by using search. Instead of using the simple greedy strategy described above, one could grow a narrow search tree containing all optimal moves for the respective subgoal.

7
Conclusion

7.1 Summary and Lessons

This thesis examined how computational resources can be ef∗ciently applied to exhaustive search. Particular emphasis was placed on the skillful combination of forward and backward search, leading to dramatically reduced problem complexity. These aspects were illustrated by the following three contributions:

- *SearchBench:* This teaching and prototyping tool was designed to provide a simple, yet general interface for solving exhaustive search problems. Its success is demonstrated by the numerous students who implemented small examples as part of their course assignments.
- *Nine Men s Morris:* This game was proven a draw using a combination of alpha-beta search and endgame databases. This result was incorporated in a playing program, which never loses.
- *15-Puzzle:* It was demonstrated how endgame databases can be pro∗tably used to signi∗cantly reduce the size of the IDA* search tree. Additionally, the hardest position that is presently known, requiring 80 moves to solve, was found.

Additional, less tangible lessons are provided by this thesis in the form of expertise applicable to similar problems. While these contributions are more dif∗cult to quantify, they can be roughly grouped in the following categories:

- *Ef∗cient algorithms:* In order to develop ef∗cient algorithms for the SearchBench, we were forced to consider methods for reducing disk I/O and similar overhead. This particularly improved our understanding of how main memory should be employed. We also better comprehend the factors the run-time depends upon and methods of ef∗cient value propagation.
- *Managing large computations:* The Nine Men s Morris database computation repeatedly demonstrated the need for a certain amount of redundancy and error correcting capability during the computation. However, because there is a trade-off between an ef∗cient implementation and error correcting capabilities, we tend to shun redundancy during the computation, preferring to verify the computation after it is completed. This tends to be simpler and allows more errors to be found. Because the veri∗cation is performed at the end, it is important to split the state space into small pieces. This reduces the amount of recomputation if an error is found and also the effect of machine crashes.

- *Using computational resources:* Because of their constantly increasing availability, one must consider how memory and processing speed can be ef∗ciently harnessed. Forward search rarely uses large memories to capacity, because transposition tables cannot be ∗lled rapidly enough. Retrograde analysis tends to pro∗t from extended memory and faster processors.

7.2 Future Research

Many subjects broached by this thesis were signi∗cantly advanced. Still, there are further avenues of research to be pursued. For instance, while our Nine Men s Morris program now plays perfectly in the game-theoretic sense, it does not yet choose variants that are especially dif∗cult for humans. This leads to unchallenging matches. The program should be improved by combining the perfect databases with the heuristics developed for Bushy to judge the dif∗culty of positions for humans.

The hardest 15-Puzzle position is now close to being determined. An implementation of our algorithm on a parallel computer, yielding a 1 000-fold speed-up, would probably resolve this question. Signi∗cant reductions in the size of the search tree can also still be attained by using more and larger databases. Another interesting idea is the use of parallel machines to reduce the effective time needed to solve a given problem.

Lastly the facility of parallelizing retrograde analysis merits further study. Initial results attained with an extension to the SearchBench that supports parallel machines are promising. Our experiments suggest that the sequential algorithm can be parallelized with nearly linear speed-up. However, it remains unclear whether this can only be attained using processors with large memory and message passing capabilities, or if a suf∗cient quantity of less powerful processors are also suitable.

Appendix A
Nine Men s Morris Statistics

The following table summarizes the Nine Men s Morris database statistics. Only the win, loss and draw information (for the player to move) is shown, because the depth information has not yet been completely veri∗ed.

Database	Won	Lost	Drawn	Total
3-3	47 167	9 659	96	56 922
4-3	177 678	3 095	1 340 023	1 520 796
4-4	159	29	3 225 409	3 225 597
5-3	586 961	9 677	4 564 142	5 160 780
5-4	9 940	1 518	20 609 534	20 620 992
5-5	28 819	4 289	30 881 316	30 914 424
6-3	2 752 371	192 000	10 806 269	13 750 640
6-4	5 985 315	1 649 976	43 896 293	51 531 584
6-5	17 247 943	4 809 008	122 175 193	144 232 144
6-6	23 250 808	5 273 162	127 705 390	156 229 360
7-3	13 411 581	6 965 784	9 074 011	29 451 376
7-4	47 531 668	37 626 917	17 874 471	103 033 056
7-5	117 750 016	80 868 403	69 203 373	267 821 792
7-6	216 347 851	102 447 808	216 790 917	535 586 576
7-7	197 782 562	63 016 948	159 993 586	420 793 096
8-3	25 697 503	23 397 858	2 432 311	51 527 672
8-4	83 063 764	77 883 075	6 464 757	167 411 596
8-5	197 140 543	175 926 821	28 638 612	401 705 976
8-6	353 632 658	260 086 963	122 178 103	735 897 724
8-7	529 980 084	242 875 900	273 864 496	1 046 720 480
8-8	307 369 402	104 175 471	157 322 580	568 867 453
9-3	36 913 628	36 901 879	12 063	73 827 570
9-4	108 064 788	107 932 986	134 782	216 132 556
9-5	225 971 196	222 904 829	3 535 813	452 411 838
9-6	345 896 267	324 805 074	21 493 153	692 194 494
9-7	402 858 673	300 062 599	75 372 176	778 293 448
9-8	340 418 251	161 643 385	106 799 190	608 860 826
9-9	56 160 179	22 081 037	17 737 285	95 978 501

Appendix B

15-Puzzle Test Suite Results

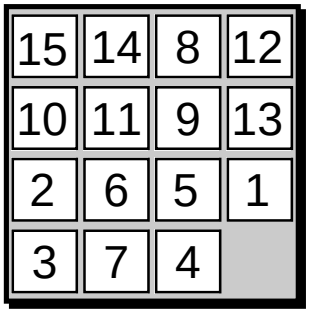
The table shows the results of our two heuristics on the standard test suite of 100 15-Puzzle positions introduced in [Korf 85]. The 2+3 Tile heuristic only uses the 2-Tile and 3-Tile many-empty databases. The ∗nal heuristic also uses a ∗nite state automaton and other re∗nements described in section 6.4.

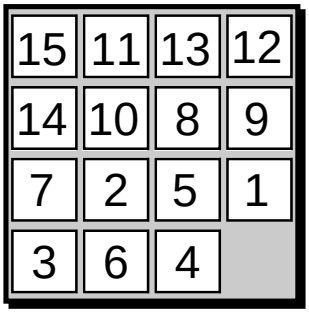
No	Len	2+3 Tile	Final	No	Len	2+3 Tile	Final	No	Len	2+3 Tile	Final
1	57	3 331 490	29 303	35	55	1 916 328	13 448	69	53	1 326 386	22 132
2	55	490 306	89 310	36	52	869 475	21 443	70	52	3 776 883	105 583
3	59	31 373 642	554 615	37	58	5 689 142	140 675	71	44	89 436	2 234
4	56	1 087 529	23 381	38	53	854 390	12 409	72	56	12 539 788	67 538
5	56	690 195	119 762	39	49	514 052	59 351	73	49	294 718	6 168
6	52	1 059 916	15 422	40	54	1 914 985	126 908	74	56	71 677	5 551
7	52	10 012 968	325 237	41	54	2 622 724	44 728	75	48	743 026	115 496
8	50	1 051 828	14 342	42	42	28 715	1 441	76	57	5 544 079	25 780
9	46	19 723	3 543	43	64	2 382 430	98 921	77	54	1 078 034	47 732
10	59	7 360 683	232 171	44	50	1 040 397	36 531	78	53	918 137	56 867
11	57	6 633 964	232 671	45	51	219 057	11 379	79	42	32 293	2 692
12	45	40 044	3 272	46	49	292 504	11 948	80	57	6 896 194	252 201
13	46	235 044	8 035	47	47	20 665	745	81	53	143 332	4 018
14	59	11 953 866	68 992	48	49	59 495	5 917	82	62	52 741 194	412 046
15	62	30 893 444	240 231	49	59	22 645 807	392 788	83	49	578 739	15 600
16	42	658 113	16 614	50	53	2 194 767	54 594	84	55	2 613 407	14 162
17	66	20 859 960	333 053	51	56	896 621	42 850	85	44	222 033	5 683
18	55	1 088 094	7 043	52	56	6 880 594	102 725	86	45	61 955	7 950
19	46	81 868	16 945	53	64	13 378 466	284 818	87	52	2 429 239	16 230
20	52	973 305	44 479	54	56	5 570 113	39 395	88	65	105 319 916	607 548
21	54	6 289 797	241 770	55	41	60 939	2 291	89	54	9 534 684	407 796
22	59	21 103 959	302 725	56	55	26 779 798	294 878	90	50	121 328	9 392
23	49	256 293	13 626	57	50	296 728	8 814	91	57	14 353 477	597 414
24	54	2 810 672	234 798	58	51	500 049	29 758	92	57	7 975 190	247 449
25	52	1 349 067	42 356	59	57	21 977 586	156 547	93	46	50 465	2 938
26	58	6 516 282	204 378	60	66	71 882 129	859 697	94	53	134 627	23 628
27	53	4 536 685	56 774	61	45	148 078	12 721	95	50	225 058	20 114
28	52	157 629	3 197	62	57	2 726 847	14 905	96	49	215 806	29 727
29	54	1 410 590	39 347	63	56	7 434 575	139 676	97	44	34 725	2 413
30	47	52 841	1 753	64	51	6 672 576	192 223	98	54	6 298 396	101 062
31	50	86 925	1 025	65	47	237 845	18 894	99	57	1 411 534	85 499
32	59	39 807 910	1 929 114	66	61	66 379 154	642 855	100	54	2 843 252	299 642
33	60	7 455 524	215 403	67	50	4 629 731	95 076				
34	52	328 277	46 545	68	51	1 123 172	16 206	Total		743 517 375	13 319 072

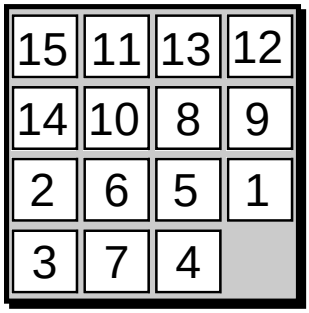
Appendix C

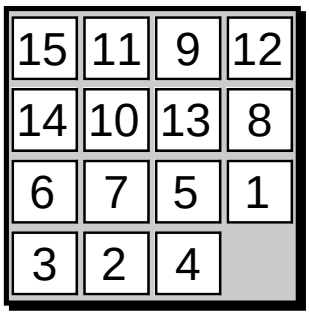
80-Move 15-Puzzle Positions

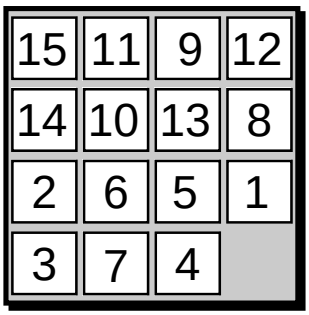
The following positions require 80 moves to solve optimally. Some of the positions can be mirrored along the diagonal, leading to a further position. In total, these nine diagrams thus represent 13 positions.

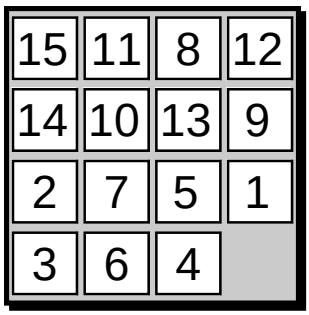


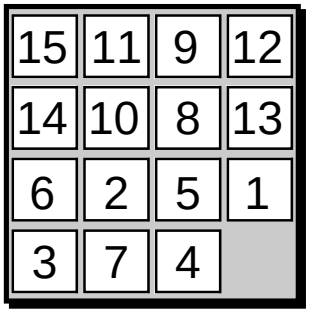


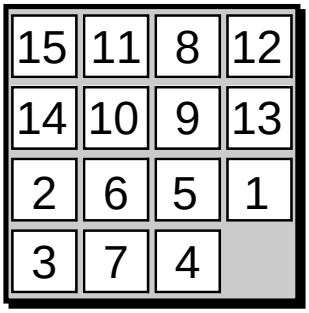


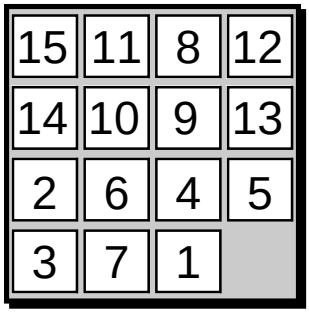












References

[Allen 89] J.D. Allen, A Note on the Computer Solution of Connect-Four, *Heuristic Programming in Arti∗cial Intelligence 1: the ∗rst computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 134-135, Ellis Horwood, Chichester, England, 1989.

[Allis 89] L.V. Allis, M. van der Meulen and H.J. van den Herik, A Knowledge-Based Approach to Connect-Four: The Game is Over: White To Move Wins, *Heuristic Programming in Arti∗cial Intelligence 1: the ∗rst computer olympiad* (eds. D.N.L. Levy and D.F. Beal) pp. 113-133, Ellis Horwood, Chichester, England, 1989.

[Allis 90] L.V. Allis, M. van der Meulen and H.J. van den Herik, Databases in Awari, *Heuristic Programming in Arti∗cial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 73-85, Ellis Horwood, Chichester, England, 1990.

[Allis 94a] L.V. Allis, M. van der Meulen and H.J. van den Herik, Proof-Number Search, *Arti∗cial Intelligence*, Vol. 66, No. 1, pp. 91-124, 1994.

[Allis 94b] L.V. Allis, *Searching for Solutions in Games and Arti∗cial Intelligence*, Doctoral Thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[Althöfer 89] I. Althöfer and T. Sillke, *Eine vollständige Analyse des Mühle-Endspiels*, Universität Bielefeld, 1989.

[Balz 94] G. Balz, *Veri∗cation of State Databases*, Diploma thesis, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1994.

[Bell 69] R.C. Bell, *Board and Table Games from many Civilisations*, Oxford University Press, Oxford, England, 1969.

[Berlekamp 82] E. Berlekamp, J.H. Conway and R.K. Guy, *Winning ways for your mathematical plays*, Academic Press, London, England, 1982.

[Brod 89] C. Brod and M. Ullrich, *Programmierung des Mühlespiels auf verteilter Hardware*, Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, 1989.

[Culberson 94] J. Culberson and J. Schaeffer, *Ef∗ciently Searching the 15-Puzzle*, internal report, University of Alberta, Edmonton, Canada, 1994.

[Gasser 90a] R. Gasser, *Heuristic Search and Retrograde Analysis: their application to Nine Men s Morris*, Diploma thesis, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1990.

[Gasser 90b] R. Gasser, Applying Retrograde Analysis to Nine Men s Morris, *Heuristic Programming in Arti∗cial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 161-173, Ellis Horwood, Chichester, England, 1990.

[Gasser 91a] R. Gasser, Endgame Database Compression for Humans and Machines, *Heuristic Programming in Arti∗cial Intelligence 3: the third computer olympiad* (eds. H.J. van den Herik and L.V. Allis), pp. 180-191, Ellis Horwood, Chichester, England, 1991.

[Gasser 91b] R. Gasser, Run-time Measurements for an Improved Retrograde Analysis Algorithm, *Game Playing System Workshop* (eds. K. Torii et al.), pp. 52-54, Tokyo, Japan, 1991.

[Hansson 92] O. Hansson, A. Mayer and M. Yung, Criticizing Solutions to Relaxed Models Yields Powerful Admissible Heuristics, *Information Sciences* 63, pp. 207-227, 1992.

[Herik 86] H.J. van den Herik and I.S. Herschberg, A data base on data bases, *ICCA Journal* 9(1), p. 29, 1986.

[Hsu 90] F.H. Hsu, *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic Architectural Study with Computer Chess*, Doctoral thesis, Carnegie-Mellon University, Pittsburgh, USA, 1990.

[Kierulf 90] A. Kierulf, K. Chen and J. Nievergelt, Smart Game Board and Go Explorer: A study in software and knowledge engineering, *Communications of the ACM* 33(2), pp. 152-166, 1990.

[Knuth 75] D.E. Knuth and R.W. Moore, An analysis of alpha-beta pruning, *Arti∗cial Intelligence*, Vol. 6, pp. 293-326, 1975.

[Kociemba 92] H. Kociemba, Close to God s Algorithm, *Cubism for Fun* 28, pp. 10-13, 1992.

[Korf 85] R.E. Korf, Depth-∗rst iterative deepening: An optimal admissible tree search, *Arti∗cial Intelligence*, Vol. 27, pp. 97-109, 1985.

[Korf 93] R.E. Korf, Linear-space best-∗rst search, *Arti∗cial Intelligence*, Vol. 62, pp. 41-87, 1993.

[Kunz 91] C. Kunz, *Automatisches Generieren von Entscheidungsbäumen aus einer Datenbank*, Semesterarbeit, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1991.

[Kunz 92] C. Kunz, *Parallel Algorithms for the Exhaustive Analysis of Board Games*, Diploma thesis, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1992.

[Lake 94] R. Lake, J. Schaeffer and P. Lu, *Solving Large Retrograde Analysis Problems Using a Network of Workstations*, internal report, University of Alberta, Edmonton, Canada, 1994.

[Levy 91] D. Levy, Nine Men s Morris, *Heuristic Programming in Arti∗cial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 55-57, Ellis Horwood, Chichester, England, 1991.

[Liempd 93] G. van Liempd, *Limited-Search algorithms*, internal report, PTT Research Tele-Informatica, The Netherlands, 1993.

[Lincke 94] T. Lincke, *Perfect Play using Nine Men s Morris as an example*, Diploma thesis, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1994.

[Mäser 94] F. Mäser, personal communication, 1994.

[Müller 87] R.F. Müller, *Mühle*, ECON Taschenbuch Verlag, Düsseldorf, Germany, 1987.

[Müller 93] M. Müller, *Untersuchung der Rückwärtsanalyse als Lösungsmethode für verschiedene Problemtypen*, Diploma thesis, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1993.

[Nunn 92] J. Nunn, *Secrets of Rook Endings*, Batsford Ltd. UK, 1992.

[Quinlan 83] J.R. Quinlan, Learning Ef∗cient Classi∗cation Procedures and Their Application to Chess End Games, *Machine Learning: An Arti∗cial Intelligence Approach* (eds. R.S. Michalski, J.G. Carbonell and T.M. Mitchell), Tioga, Palo Alto, USA, 1983.

[Patashnik 80] O. Patashnik, Qubic: 4x4x4 Tic-Tac-Toe, *Mathematics Magazine*, Vol. 53 No.4, pp. 202-216, 1980.

[Pearl 84] J. Pearl, *Heuristics*, Addison-Wesley, Reading, USA, 1984.

[Pell 93] B.D. Pell, *Strategy Generation and Evalutation for Meta-Game Playing*, Doctoral thesis, University of Cambridge, Cambridge , England, 1993.

[Pohl 71] I. Pohl, Bi-Directional Search, *Machine Intelligence* Vol. 6 (eds. B. Metzler and D. Michie), pp. 127-140, American Elsevier, New York, USA, 1971.

[Ramseier 80] S. Ramseier, Mühlespiel - eine Computer-Knacknuss?, *Mikro- und Kleincomputer*, Vol. 80 No. 1, 1980.

[Ratner 90] D. Ratner and M. Warmuth, Finding a Shortest Solution for the (NxN)-Extension of the 15-Puzzle is Intractable, *Journal of Symbolic Computation*, 10, pp. 111-137, 1990.

[Reinefeld 94] A. Reinefeld and T.A. Marsland, Enhanced Iterative-Deepening Search, *IEEE Transactions on pattern analysis and machine intel-ligence*, Vol. 16, pp. 701-710, 1994.

[Reingold 77] E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms Theory and Practice*, Prentice Hall, Englewood Cliffs, USA, 1977.

[Schaeffer 92] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu and D. Szafron, A World Championship Caliber Checkers Program, *Arti∗cial Intelligence*, Vol. 53, pp. 273-289, 1977.

[Schaeffer 94] J. Schaeffer, personal communication, 1994.

[Schürmann 80] H. Schürmann and W. Nüscheler, *So gewinnen Sie Mühle*, Otto Maier Verlag, Ravensburg, Germany, 1980.

[Smith 91] D.K. Smith, *Dynamic Programming A Practical Introduction*, Ellis Horwood, Chichester, England, 1991.

[Stiller 91] L. Stiller, Group Graphs and Computational Symmetry on Massively Parallel Architecture, *The Journal of Supercomputing*, 5, pp. 99-117, 1991.

[Ströhlein 70] T. Ströhlein, *Untersuchungen über Kombinatorische Spiele*, Doctoral thesis, Technische Hochschule München, Munich, Germany, 1970.

[Sucrow 92] B. Sucrow, *Algorithmische und kombinatorische Untersuchungen zum Puzzle von Sam Loyd*, Doctoral thesis, University of Essen, Essen, Germany, 1992.

[Taylor 93] L. Taylor and R. Korf, Pruning Duplicate Nodes in Depth-First Search, *AAAI National Conference*, pp. 756-761, 1993.

[Thompson 86] K. Thompson, Retrograde Analysis of certain endgames, *ICCA Journal* 9(3), pp. 131-139, 1986.

[Thompson 91] K. Thompson, Chess Endgames Vol. 1, *ICCA Journal* 14(1), p. 22, 1991.

[Thompson 92] K. Thompson, Chess Endgames Vol. 2, *ICCA Journal* 15(3), p. 149, 1992.

[Weilenmann 91] C. Weilenmann, *Mühle: Beilage zum Programm*, Semesterarbeit, Kantonsschule Winterthur, Switzerland, 1991.

[Welch 84] T.A. Welch, A Technique for High-Performance Data Compression, *Computer*, June 1984, pp. 8-19, 1984.

[Wilson 74] R.M. Wilson, Graph Puzzles, Homotopy, and the Alternating Group, *Journal of Combinatorial Theory* Series B 16, 86-96, 1974.

[Wirth 94] C. Wirth, personal communication, 1994.