Heuristic Search

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90095

December 16, 2008

Contents

Chapter 1

Problems and Problem Spaces

Heuristic Search is a very general problem-solving method in artificial intelligence (AI). The problems we will be concerned with require a sequence of steps to solve, but the correct sequence is not known a priori, and must be determined by a trial-and-error exploration of alternatives.

We will restrict our attention to *systematic search algorithms* in these notes. A systematic search algorithm is one that is guaranteed to find a solution if one exists. If a solution does not exist, a systematic algorithm may run forever, without detecting that there is no solution. Non-systematic algorithms, such as simulated annealing, genetic algorithms, and most local-search algorithms, are often more efficient than systematic algorithms, but cannot guarantee finding a solution, even if one exists.

1.1 Problem Types

The problems that have been addressed by AI search algorithms fall into three general categories: *single-agent path-finding problems*, *two-player games*, and *constraint-satisfaction problems*.

1. gent Path-Finding Problems

On n problems of this type, and the most famous co of our time, is Rubik's Cube (see Figure 1.1), inv rno Rubik of Hungary. The standard version co cube, with different colored stickers on each of the exposed squares of the subcubes, or *cubies*. Any $3 \times 3 \times 1$ plane of the cube can be rotated or twisted 90, 180, or 270 degrees relative to the rest of the cube. In the goal state, all the squares on each side of the cube are the same color. The puzzle is scrambled by making a number of random twists, and the task is to restore the cube to its original goal state.

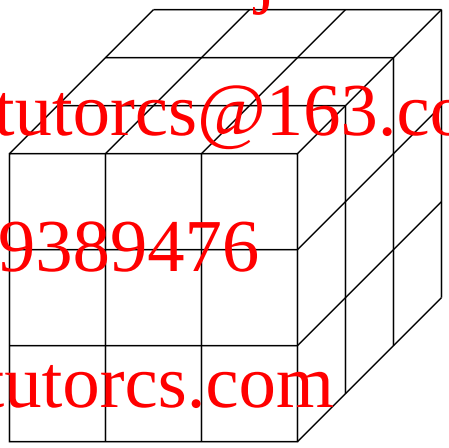


Figure 1.1: Rubik's Cube

The classic example in the AI literature of a single-agent path-finding problem is the sliding-tile puzzles, including the 3×3 Eight Puzzle, and its larger relatives the 4×4 Fifteen Puzzle, and 5×5 Twenty-Four Puzzle (see Figure 1.2). Each consists of a square frame containing a set of numbered square tiles, and an empty position called the blank. The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. The sliding-tile puzzles are common testbeds for research in AI search algorithms because they are very

simple to represe[illegible]vet finding optimal solutions to the $N \times N$ generaliz[illegible]tile puzzles is NP-complete[82].

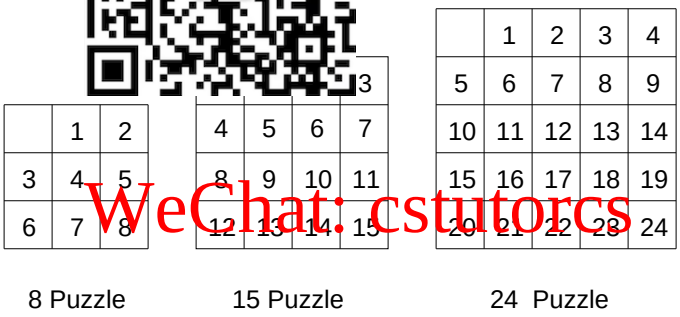


Figure 1.2: Eight, Fifteen, and Twenty-Four Puzzles

The Fifteen Puzzle was invented by Sam Loyd in the 1870s[55], and appeared in the scientific literature in 1879[36]. It became quite a sensation in its day, largely because Loyd offered a $1000 cash prize to transform a particular initial state to a particular goal state. Johnson and Story proved that it wasn't possible, that the entire state space was divided into even and odd permutations, and that there is no way to transform one into the other by legal moves. As a result, however, "The '15' puzzle for the last few weeks has been prominently before the American public, and may safely be said to have engaged the attention of nine out of ten persons of both sexes and of all ages and conditions of the community."[36].

A more practical single-agent path-finding problem is road navigation. Given a starting location and a goal location, the task is to plan a route between the two points, consistent with some objective such as minimizing travel time or distance. Automobiles are now available with navigation systems that solve this problem.

A related and very famous combinatorial problem is the travelling salesman problem (TSP)[54]. Given a set of cities, and the distances between each pair of cities, the task is to plan a route that starts in one city, visits all the cities, and returns to the starting city, while covering a minimum total distance.

These are known as single-agent path-finding problems because in each case we have a single problem solver making the decisions, and

th quence of primitive steps that take the problem sol location or configuration to a goal location or co

1. ger Games

A second class of search problems include two-player perfect-information games, such as chess, checkers, and Othello. Chess was in fact the original AI problem, and the earliest papers on the subject[91] predate the field of AI itself. Chess is considered a quintessential intellectual task, one that aficionados spend a lifetime trying to master, even though it has very simple rules and a clearly defined outcome. As a result, building a computer program to play chess, and ultimately defeat the human world champion, was an obvious grand challenge for AI, one that was achieved in May of 1997 by the defeat of Gary Kasparov by the Deep Blue machine[8]. The best checkers and Othello players in the world are also computer programs[87, 7].

While two-player perfect information games have received the most attention to date, researchers are starting to consider more complex games. Many games involve an element of chance. For example, legal moves in backgammon are determined by the roll of a pair of dice. In addition to a chance element, many games, including most card games like bridge or poker, involve hidden information that is known to some of the players but not to others. These games often involve more than two players as well.

In a two-player game, one must consider the moves of an opponent, and the ultimate goal is a strategy or algorithm that will guarantee a win whenever possible. In practice, the task is often to determine what move to make given a particular state of the game. One wants a program that plays well, usually measured against human performance.

1.1.3 Constraint-Satisfaction Problems

The third category is constraint-satisfaction problems (CSP), of which the Eight-Queens problem is a classic example. The task is to place eight queens on an 8×8 chessboard, such that no two queens are on the same row, column or diagonal. While there exist deterministic

algorithms for so ith an arbitrary number of queens, they are specific roblem. Our interest is in general techniques for so sfaction problems, of which the N-queens problem ent example. There exist general techniques that with over a million queens.

Another example of a CSP is number partitioning. Given a set of numbers, divide them into two mutually exclusive and collectively exhaustive subsets, so that the sum of the numbers in the two subsets are as nearly equal as possible.

Real-world examples of constraint-satisfaction problems are ubiquitous, including planning and scheduling applications. For example, airlines must schedule aircraft, crews, and airport gates to fly a certain set of flights with the fewest planes, crew members, and gates.

In a CSP, there is normally no adversary, but a single agent making all the decisions. What distinguishes these problems from the single-agent path-finding problems is that we are not concerned with the sequence of steps required to reach the solution, but simply the solution itself. For example, in the Eight-Queens problem we are only interested in the final configuration of the queens on the board, and not how it was arrived at. The task is to identify a state of the problem, usually in terms of a set of assignments of values to variables, such that all the constraints of the problem are satisfied. In this case, we want an assignment of each of the eight queens to a particular square of the board, such that no two queens are on the same row, column, or diagonal.

The boundaries of these different problem types are not sharp and well defined. Often a particular problem can be cast as more than one of these different problem types. For example, while we characterized the travelling salesman problem as a path-finding problem, it could also be formulated as a constraint-satisfaction problem, by carefully specifying the constraints that any solution must satisfy, including the optimization criterion. The CSP is probably the most general of these formulations. Most particular problems, however, will fit most naturally into one of these categories.

In the remainder of these notes, we will first address single-agent path-finding problems, then two-player games, and finally constraint-satisfaction problems. Many of the techniques developed for the path-finding problems, such as the brute-force search techniques, will also

be to problems in the latter classes, and will not be re

1. Spaces

A *problem space* is a computer representation of a problem in which a search takes place. The notion of problem solving as search in a problem space is due to Allen Newell and Herbert Simon. An excellent introduction to this view is their Turing Award Lecture[66]. In a later paper[67], Newell articulates directly what he calls the *problem space hypothesis*, which states that, "The fundamental organizational unit of all human goal-oriented symbolic activity is the *problem space*".

A problem space consists of a set of *states* of a problem, and a set of *operators* that change the state. A state is a symbolic structure that represents a single configuration of the problem in sufficient detail to allow problem solving to proceed. For example, in the sliding-tile puzzles, the states are the different possible permutations of the tiles. In chess, a state specifies the locations of the pieces on the board, whose move it currently is, and other details like whether each side has already castled, etc.

For efficiency, states are only represented at a level of detail to allow further problem solving. For example, in Rubik's Cube, one could represent the detailed physical state of the puzzle, including states in the middle of a twist. However, since no further operators can be applied until all the faces are aligned, there is no need to represent these intermediate states. Similarly, in road navigation, one would normally only represent intersections of two or more roads as distinct states.

An operator of a problem is a function that takes a state and maps it to another state. For Rubik's Cube, the operators are the different twists of the faces. For chess, they are the legal moves. Not all operators are applicable to all states. In the sliding-tile puzzles, for example, in order to slide a tile to a particular position, that position must be adjacent to the current position of the tile and empty of any other tiles. The *preconditions* of an operator are those conditions that must be true in order for the operator to be legally applied to a state.

As with state ... of the level of granularity in the representation o... ample, in Rubik's Cube, should a 180 degree twist ... red a single primitive operator, or a sequence of tw... These decisions are usually made to maximize the ... blem solver, as we will see later.

A *problem instance* is a problem space together with an initial state and a set of goal states. In the case of Rubik's Cube, the initial state would be whatever initial scrambled configuration the puzzle starts out in, and the single goal state is the configuration where only a single color shows on each side. For chess, the initial state is the initial board setup, and the goal state is one where the opponent's king is checkmated, which is a state in which the king is unavoidably captured on the next move.

There may be a single goal state, or a set of goal states, any one of which would satisfy the goal criteria. For example, the Four-Queens problem has two different goal states, shown in Figure 1.3. In addition, the goal could be stated explicitly, as in the Eight Puzzle, or implicitly, as in the N-Queens problem, by giving a rule for determining when the goal has been reached. For constraint-satisfaction problems, the goal will always be represented implicitly, since giving an explicit description of the goal is in fact the problem to be solved. Note that whether there is a single or multiple goal states is completely orthogonal to whether the goal is specified implicitly or explicitly, and all four combinations are possible. For example, an implicit goal description may admit one or more goal states, and multiple goal states can be explicitly described simply by listing them all. Furthermore, there may be no solution at all to a problem.

Given a problem space and a problem instance, the problem-solving task is usually to find a sequence of operators that map the initial state to a goal state. In general, more than one operator may be applicable to a given state. For the problems we're interested in, we cannot immediately determine which operator to apply, giving rise to the search behavior that is the subject of these notes.

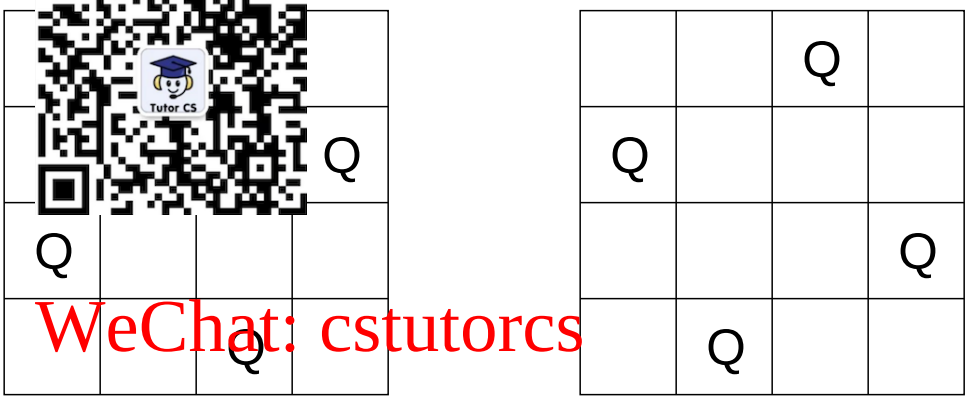


Figure 1.3: Solutions to the Four-Queens Problem

1.2.1 Problem Representation

Note that a problem does not uniquely determine a problem space. For some problems like the sliding-tile puzzles, Rubik's Cube, and chess, there is an obvious problem space. For others, however, like the Eight-Queens problem, the choice of a problem space is not so obvious.

Perhaps the first problem space that comes to mind for the Eight-Queens problem is one that assigns anywhere from zero to eight queens to different squares on the board. Since there are 64 different squares, the number of possibilities with all eight queens on the board is 64 choose 8, which is over four billion possibilities. This doesn't even count the states with fewer than eight queens on the board.

The next thing to notice is that the solution prohibits placing two queens in the same row, so we might as well assign each queen to a separate row initially, so we never violate this constraint. Thus, we only have to choose one of the eight columns for each of the eight queens. This reduces the number of possibilities to 8^8, which is over 16 million possibilities.

Finally, we can observe that no solution can have more than one queen on the same column either, so we shouldn't place queens on columns that are already occupied. This reduces the space to 8!, which is only 40,320 states. Note that this only counts the states with all 8 queens on the board, ignoring those with fewer queens. Thus, by changing the representation of the problem, in this case by incorpo-

rating some of th y into the problem representation, we've reduced th at must be searched from over four billion to about 4 gs of five orders of magnitude. The savings would be more queens. Unfortunately, there is no obvious wa e diagonal constraint directly into the problem representation, or we would be able to directly generate all solutions.

The moral of this story is that the choice of representation for a problem can have an enormous impact on the efficiency of solving the problem[40]. Before embarking on any search, we should try to design the most efficient representation for the problem. While there are no algorithms for problem representation, and precious little theory to guide the process, one general rule is that a smaller representation, in the sense of fewer states, is often better than a larger one.

1.2.2 Problem-Space Graphs

A *problem-space graph* is a mathematical abstraction often used to represent a problem space. The states of the space are represented by *nodes* of the graph, and the operators by *edges* between nodes. Edges may be directed or undirected, depending on whether their corresponding operators can be applied in one direction or both. The task in a single-agent path-finding problem is to find a path in the graph from the initial node to a goal node. Figure 1.4 shows a small part of the Eight Puzzle problem-space graph.

Most problem spaces correspond to graphs with more than one path between a pair of nodes. For example, in Rubik's Cube, making two 90 degree clockwise twists of the same face results in the same state as making two 90 degree counterclockwise twists. In general, detecting when the same state has been regenerated via a different path requires saving all the previously generated states, and comparing newly generated states against the saved states. For simplicity and to save memory, many search algorithms don't detect when a state has previously been generated. This effectively treats the search graph as if it were a tree, and each different path of the tree is explored separately, even if it leads to a previously generated state. The cost of this simplification is that any state that can be reached by two different paths will be rep-

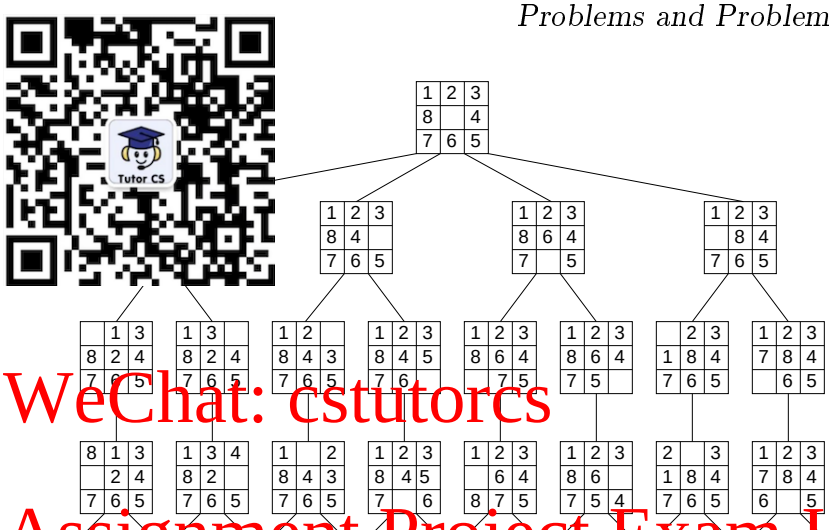


Figure 1.4: Eight Puzzle Search Tree Fragment

resented by two different nodes, increasing the size of the tree relative to the graph. In addition to the memory savings, the benefit of a tree search is that ignoring cycles greatly simplifies many search algorithms.

Branching Factor and Solution Depth

Since most AI problem-space graphs are enormous, and many problem-space trees are infinitely large, our search algorithms can only explore a small part of the graph. The efficiency of most of these algorithms can be characterized in terms of two parameters of the problem-space graph: the branching factor and the solution depth.

The *branching factor* of a node is the number of children it has, not counting its parent if the operators are reversible. The branching factor of a problem space is the average number of children of the nodes in the space. For example, in a binary tree, the branching factor is exactly two.

While the branching factor is generally a function of the problem space, the *solution depth* is a function of the particular problem instance. For a single-agent problem, the solution depth is the length of a shortest path from the initial node to a goal node. For example, if one of the states at the bottom of Figure 1.4 were a goal state, the solution depth would be three moves.

Eliminating D

In many cases, w reduce the size of the search tree by eliminating s te paths. For example, in Figure 1.4, even thoug n be applied in reverse, we don't list the parent o its children. In general, we never apply an operator and its inverse in succession, since no optimal path can contain such a sequence. This immediately reduces the branching factor of the problem by approximately one, a significant improvement.

1.2.3 Types of Problem Spaces

There are several different types of problem spaces, including state spaces and problem-reduction spaces. We consider each in turn.

State Space

In a *state space*, the states represent situations or configurations of the problem to be solved, and the operators represent actions in the problem. For example, Figure 1.4 shows a fragment of a state space for the Eight Puzzle, where the nodes represent permutations of the tiles and the edges represent actual moves in the problem. In a *forward search*, the root of the problem space represents the start state, and the search proceeds forward to a goal state. In a *backward search*, the root of the problem space represents the goal state, and the search proceeds backward to the initial state. In a problem such as Rubik's Cube or the sliding-tile puzzles, the operators are reversible, and there is a symmetry between initial and goal states, suggesting that either a forward or backward search is appropriate.

Problem-Reduction Space

This is best illustrated by the example of the Towers of Hanoi problem. In this problem, there are three pegs, labeled A, B, and C, and a series of discs of different sizes, initially stacked in decreasing order of size on one of the pegs (see Figure 1.5). The goal is to transfer all of the discs from their initial peg, say peg A, to one of the other pegs, say peg C. The move constraints are that only one disc can be moved at a time,

on peg can be moved, and a larger disc can never be smaller disc. In a state-space representation of th he states would represent assignments of discs to pe s would represent legal moves of the discs.

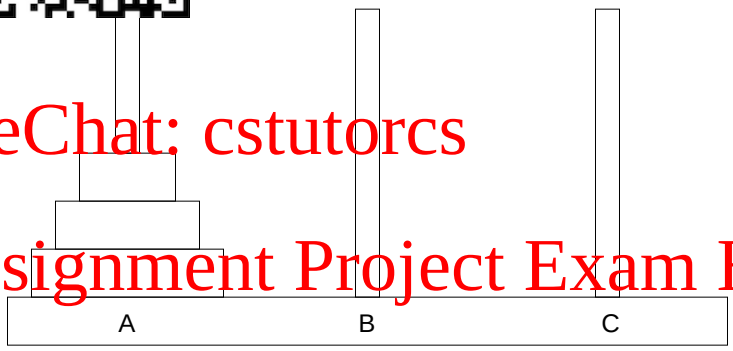


Figure 1.5: 3-Disc Towers of Hanoi Problem

In a problem-reduction space, however, the nodes represent problems to be solved or goals to be achieved, and the edges represent the decomposition of the problem into subproblems. Figure 1.6 shows a complete problem-reduction space for the three-disc Towers of Hanoi problem. The root node, labelled "3AC" represents the original problem of transferring all three discs from peg A to peg C. This goal can be decomposed into three subgoals: move the top two discs from peg A to peg B (2AB), move one disc from peg A to peg C (1AC), and move the remaining two discs from peg B to peg C (2BC). In order to achieve the main goal, all three of these subgoals must be achieved. As a result, this node in the graph is referred to as an AND node, and is indicated by the semi-circular arc connecting the three edges. In addition, in the final solution, the solutions to the subproblems must appear in order from left to right, although the order is not always significant in such graphs.

The goal of moving one disc from A to C (1AC) is a terminal node in this graph, because it can be accomplished by a single legal action in the problem, once its preconditions have been satisfied. However, the goal of moving two discs from peg A to peg B (2AB) is decomposed into three subgoals: move the top disc from peg A to peg C (1AC), move

Figure 1.6: AND Tree for Towers of Hanoi Problem

the next disc from peg A to peg B (1AB), and move the first disc again from peg C to peg B (1CB). The subgoal of moving two discs from peg B to peg C (2BC) is similarly decomposed into three subgoals, each of which can be achieved by an atomic action in the space. Both of these nodes are also AND nodes, since all their subgoals have to be achieved in order to achieve the main goal.

The resulting problem-space graph is referred to as an *AND graph*, since it consists entirely of AND nodes. This is in contrast to graphs such as the one in Figure 1.4, which is referred to as an OR graph, consisting entirely of OR nodes. The difference is that in order to solve a problem represented by an OR node, you only need to solve the problem represented by one of its children, whereas to solve a problem represented by an AND node, you need to solve the problems represented by all of its children.

AND/OR Graphs

A problem-space graph may include both AND nodes and OR nodes. In that case, it is referred to as an *AND/OR graph*. For example, consider the problem of symbolically integrating an expression such as $\int(\sin^2 x + \cos^2 x)dx$. One strategy would be to integrate each term separately, and an alternative would be to make the substitution $\sin^2 x + \cos^2 x = 1$. These two options would give rise to an OR node, since either strategy, if successful, would be sufficient to solve the original problem. If we decided to integrate the two terms separately, then both would have to

be the problem, giving rise to an AND node.

AND/OR graphs is a problem where the effect of an edicted in advance, as in an interaction with the ph ample, in the counterfeit-coin problem, there are tw ar identical, but one of them is slightly heavier or lighter than the others. The only measuring device is a two-platform balance scale that only reports which side is heavier, or that they are equal. The problem is to determine which coin is different, and whether it is heavy or light, in at most three weighings. In this case, the different possible sets of coins that could be placed on the balance platforms would represent OR nodes, since the problem solver can choose its actions. The three possible outcomes of the balance scale would represent AND nodes, since the outcome cannot be predicted in advance, and the problem solver's strategy must work for every possible outcome of each weighing.

Two-Player Game Trees While the above example can be viewed as a game against nature, the most common source of AND/OR graphs is two-player perfect-information games. Consider for example the game of Nim. It is played with two players and a pile of stones. Each player alternately removes one stone or two from the pile, and the player who removes the last stone is the winner of the game. Figure 1.7 shows a state-space tree for the five-stone version of Nim. A state is represented by the number of stones left, and the player to move, which we represent by circles and squares. From the perspective of the square player, square nodes are OR nodes, because the square player can choose its moves. Conversely, circle nodes are AND nodes from square's perspective, since square has no control over circle's moves, and square's strategy must work regardless of what circle does. The AND and OR nodes would be completely reversed from the circle player's perspective.

This tree is also a good example of the difference between a graph and a tree. The two square nodes labelled "2" represent the same state, two stones left and the square player to move. They were each arrived at by a different move sequence. In one case two stones were taken first and then one, and in the other, one stone was taken first and then two.

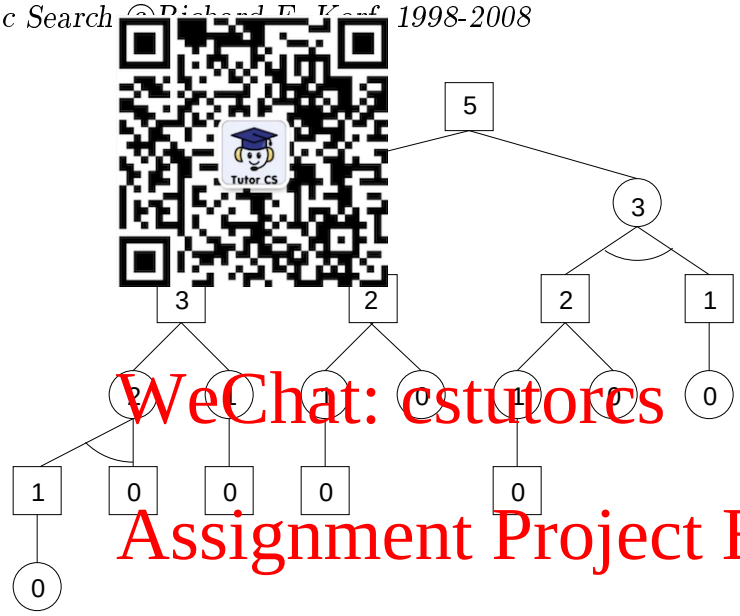


Figure 1.7 Game Tree for 5-Stone Nim

This is referred to as a *transposition* in the two-player game literature. If this were a game graph, they would both be represented by the same node, but since this is a game tree, they are represented by separate nodes, and the subtrees below them are duplicated as well.

Solution Subgraph for AND/OR Trees In a pure OR graph, a solution is simply a path from the root node to a goal node. In a pure AND graph, such as that shown in Figure 1.6, the solution is the entire graph. Both of these are special cases of a solution to an AND/OR graph. For example, a complete solution to a two-player game from the perspective of one of the players is a strategy that will guarantee a win regardless of the moves made by the opponent.

In general, a solution to an AND/OR graph is a subgraph with the following properties. It contains the root node. For every OR node included in the solution subgraph, one child node is included. For every AND node included in the solution subgraph, all the children are included. Finally, every terminal node in the solution subgraph is a solved or winning node. The depth of a solution subgraph is the length of the longest path from the root to a terminal node. The solution depth

of is the minimum depth of all solution subgraphs for nce.

1.3 Problems

As we have seen above, the notion of a solution is different for the different problem types. A solution to a path-finding problem instance is simply a path in the problem space graph from the root node to a goal node. For a constraint-satisfaction problem, a solution is a particular state of the problem that satisfies all of the constraints. For a two-player game, the most general solution is a strategy or solution subgraph of the game tree, in which the states where one player is to move are represented by AND nodes, and states where the opposing player is to move are represented by OR nodes. Alternatively, a solution to a particular game state may simply be a legal move to make in that state.

The quality of a solution often matters in practice. In particular, we will distinguish between an *optimal* solution and a *sub-optimal* solution. For a path-finding problem, an optimal solution is a solution of lowest cost. The cost of a solution may simply be the length of the solution path, if all moves are of equal cost. Alternatively, if different moves have different cost, the cost of a solution is the sum of all the individual move costs on the solution path, and an optimal solution is one of lowest cost. For a CSP, if there is a cost function associated with a state of the problem, an optimal solution would again be one of lowest cost.

For a two-player game, if the solution is simply a move to be made, then an optimal solution would be the best possible move that can be made in a given situation. If the player to move can force a win from the current state, regardless of the opponent's moves, then an optimal move must be to a state that is also a forced win from the perspective of that player. If the best that can be guaranteed is a draw, then an optimal move must be to a state that is still a forced draw. Finally, if the opponent can force a win, then any move for the opposing player may be considered optimal. If the solution is considered a complete strategy subgraph, then an optimal solution might be one that forces a win in the fewest number of moves in the worst case.

1.4 Combinatorial Explosion

A critical property of the problem domains described above is that the number of different states in each problem is large, and grows extremely fast as problem size increases. For example, there are $9! = 362,880$ different possible configurations of the Eight Puzzle, $16! \approx 10^{13}$ states of the Fifteen Puzzle, and $25! \approx 10^{25}$ states of the Twenty-Four Puzzle. Of these, only one half are reachable from any given state. While the $2 \times 2 \times 2$ Rubik's Cube has only 3,265,920 states reachable from a given state, the full $3 \times 3 \times 3$ cube has about 4.3252×10^{19} reachable states. The Saganesque slogan printed on the box, that there are billions of combinations, is a rather considerable understatement. The number of possible tours of an N-city TSP problem is on the order of $N!$. Although the road navigation problem would seem to grow only quadratically with the size of the map, the number of different routes between any two points grows exponentially. The number of different legal checkers positions is estimated at about 10^{20}, while the corresponding number for chess is about 10^{40}. The number of different assignments of N queens to an $N \times N$ chessboard, where no two queens are on the same row or column, is also on the order of $N!$ The number of two-way partitions of a set of N elements is 2^N.

This combinatorial explosion of the number of possible states as a function of problem size is the key characteristic that separates artificial intelligence search algorithms from search algorithms in other areas of computer science. For example, techniques that rely on storing all possibilities in memory, or even generating all possibilities, are completely out of the question except for the smallest of these problems. As a result, the problem-space graphs of AI problems are rarely represented explicitly by listing each state, but rather are implicitly represented by specifying an initial state and a set of operators to generate new states from existing ones.

1.5 Search Algorithms

Most of the rest of these notes will focus on systematic search algorithms that are applicable to these different problem types. For exam-

ple deals with brute-force searches, which are algorit e use of the problem space and a description of th a solution. Chapter 3 considers heuristic search alg dition to the problem space use domain-specific he mprove the efficiency of the search. Chapter 4 is concerned with search algorithms that run in linear rather than exponential space. Chapter 5 is about search algorithms for the case where individual moves of a solution must be executed before a complete optimal solution can be computed. Chapter 6 focuses on methods for deriving the heuristic functions used by the algorithms in Chapters 3, 4, and 5. Chapter 7 deals with two-player perfect information games. Chapter 8 is devoted to the analysis of alpha-beta minimax, the predominant game-tree search algorithm. Chapter 9 deals with games with more than two players. Chapter 10 looks at the decision quality of the minimax algorithm. Chapter 11 is concerned with the automatic learning of heuristic functions for two-player games. Chapter 12 deals with constraint-satisfaction problems. Finally, Chapter 13 is about parallel search algorithms.

Throughout these notes a central concern is with the efficiency of these search algorithms. There are three primary measures of efficiency. One is the quality of the solution returned, and in particular whether it is guaranteed to be optimal or not. The second is the running time of the search algorithm, and the third is the amount of memory required by the algorithm. We will attempt to evaluate each of our algorithms along these three dimensions, either experimentally, analytically, or both.

Chapter 2

Brute-Force Search

Once a problem space is defined for a problem, we can apply a systematic search algorithm to look for a solution. The most general search algorithms are *brute-force* searches, that is, not use any domain-specific knowledge. All that is required for a brute-force search is a state description, a set of legal operators, an initial state, and a description of a goal state. The simplest brute-force techniques are breadth-first, depth-first, depth-first iterative-deepening, and bidirectional search, which we will consider in turn. In this chapter, we will assume that all edges have the same cost, and defer the consideration of algorithms for problems with different edge costs, such as uniform-cost search and depth-first branch-and-bound, to later chapters.

Almost all search algorithms make use of a basic node expansion cycle. In each such cycle, a data structure corresponding to a node of the explicit search tree is selected, and the data structures corresponding to one or more of the children of that node are created. In the descriptions of the algorithms below, to *generate* a node means to create the data structure corresponding to that node, whereas to *expand* a node means to generate all the children of that node.

2.1 Breadth-First Search (BFS)

Breadth-first search (BFS) expands nodes in order of their depth from the root, generating one level of the tree at a time, until a solution

is ot node of the tree is expanded, generating the no hen, each of these are expanded, generating the ch followed by the nodes at depth 3, etc. Figure 2.1 sh tree, where the numbers in the nodes represent th se nodes are generated by BFS.

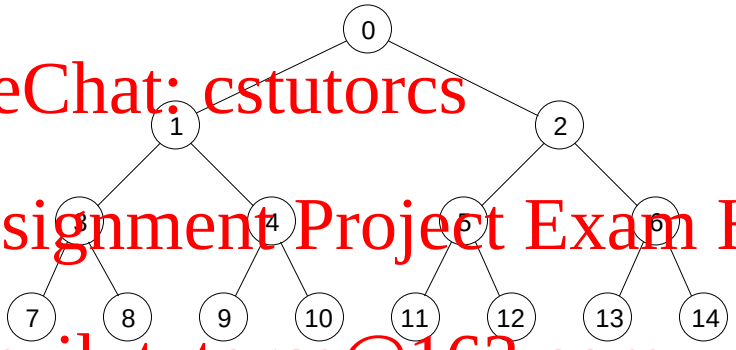


Figure 2.1: Order of Node Generation for Breadth-First Search

The easiest way to implement BFS is with a first-in first-out (FIFO) queue of nodes, initially containing just the root node. At each cycle of the algorithm, the node at the head of the queue is removed and expanded, and its children are placed at the end of the queue. This implementation doesn't require any special processing when completing one level and starting the next. We now consider the performance of BFS in terms of solution quality, time complexity, and space complexity.

2.1.1 Solution Quality

BFS continues until a goal node is generated. There are two ways to record the actual solution path. One is to store with each node the sequence of moves made to reach that node. A more memory-efficient technique is to save all the nodes, and with each node store a pointer back to its parent. Then, when a goal is found, simply trace back through the pointers to recover the actual solution path.

A key property of BFS is that it generates all the nodes at one level of the tree before generating any nodes at a deeper level. As a result, the first time BFS generates a goal node, it has found a shortest path

to a goal, in terms of the number of edges on the path from the root to the goal. Further, since IDA* algorithm is complete, it is guaranteed to find a goal node in a finite tree.

2.1.2 Time

Since the size of a state description remains fixed throughout the search, we assume that each node can be generated in constant time. Thus, the amount of time used by BFS is proportional to the total number of nodes generated, which is a function of the branching factor b and the solution depth d. In addition, the number of nodes generated depends on where the goal node is found within level d. In the *best case*, the first node generated at level d is a goal node. In the *worst case*, the last node at level d is the first goal found. In the *average case*, assuming there is only a single goal node at level d, we would expect to examine half of the nodes at level d before finding a goal. For simplicity, we assume the worst case, generating all the nodes at level d.

Let $N(b,d)$ be the number of nodes generated, in the worst case, in a tree with branching factor b and solution depth d. The number of nodes at level d is b^d. Thus,

$$N(b,d) = 1 + b + b^2 + \cdots + b^{d-1} + b^d$$

Multiplying both sides by b gives

$$b \cdot N(b,d) = b + b^2 + b^3 + \cdots + b^d + b^{d+1}$$

Subtracting the first equation from the second gives

$$b \cdot N(b,d) - N(b,d) = -1 + b^{d+1}$$

$$N(b,d) \cdot (b-1) = b^{d+1} - 1$$

$$N(b,d) = \frac{b^{d+1} - 1}{b-1}$$

Since we are most interested in large search trees, the -1 on top is insignificant, and we can write

$$N(b,d) \approx \frac{b^{d+1}}{b-1} = b^d \left(\frac{b}{b-1}\right)$$

W in this form shows the contribution of the dom-
in mber of nodes at the bottom level of the tree,
an rk due to all the shallower levels. If we hold b
co to grow large, the asymptotic time complexity of
BF

2.1.3 Space Complexity

In order for BFS to generate the nodes at one level of the tree, all the nodes at the previous level must be stored in memory, typically in the FIFO queue. In fact, the most memory-efficient way to report the actual solution is to store all nodes generated, along with a pointer back to their parent node. Since the size of a state description remains constant, we assume that a node can be stored in a constant amount of memory. Thus, the asymptotic space complexity of BFS is the number of nodes generated, which is the same as its asymptotic time complexity, or $O(b^d)$.

As a result, the main drawback of BFS is its memory requirement. To see this, let's consider the implementation of BFS on a modern workstation. Let's assume that our implementation can generate a new state in a hundred instructions. On a two gigahertz machine, this would allows us to generate 20 million nodes per second. Current memory capacities are up to four gigabytes. Let's assume that we can store a node in a word of memory, or four bytes. This gives us a capacity of one billion nodes. With these numbers, we can expect our implementation of BFS to be able to run for only 50 seconds, before exhausting its available memory. If we find a solution before then, fine, but if not, the algorithm will terminate when memory is exhausted.

The numbers above are rough estimates based on current technology. Even if we are off by two orders of magnitude, our BFS algorithm won't be able to run much longer than an hour. If we use disk instead of main memory, this only gives us three orders of magnitude in memory capacity, at a cost of five orders of magnitude in random access speed. As computing technology advances, this problem doesn't go away, since as memories increase in size, processors get faster as well, and our appetite to solve larger problems grows. For example, the Eight Puzzle was the sliding-tile puzzle of choice in the 1960s and 1970s, random

instances of the [illegible] first solved optimally in the 1980s, and random Twe[illegible]ances were solved optimally in the 1990s. The key [illegible]tio of processor speed to memory capacity, a ratio[illegible]ged dramatically over the history of electronic cor[illegible]t, breadth-first search is severely space-bound in practice, and will exhaust the memory available on typical computers in a matter of minutes. This is a problem we will see repeatedly in any algorithm that must store in memory all of the nodes it generates.

2.2 Depth-First Search

The solution to this memory limitation is a different algorithm, called *depth-first search* (DFS). DFS proceeds down the left-most path of the tree until that path terminates, and then backtracks to the last node that has not been completely expanded, and proceeds down from there, backtracking again when that path terminates, continuing until the entire tree has been expanded, or a goal has been found. DFS always generates next a child of the deepest node in the search tree that has been generated, but not completely expanded yet. Figure 2.2 shows the order in which nodes are generated in a depth-three binary tree.

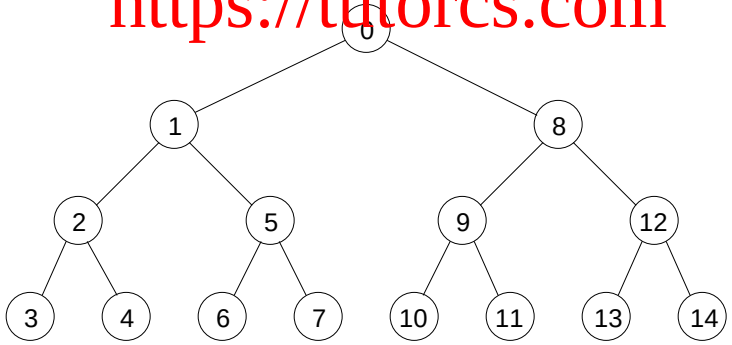


Figure 2.2: Order of Node Generation for Depth-First Generation

There are two ways to implement DFS, which give rise to slightly different node generation orders. One way is to use the same code as for BFS, but change the FIFO queue to a last-in, first-out (LIFO)

sta start with just the root node on the stack, and at ne node at the top of the stack, generate all of its ch e children on top of the stack. This gives rise to th der in Figure 2.3, where the nodes are expanded in d is referred to as depth-first expansion.

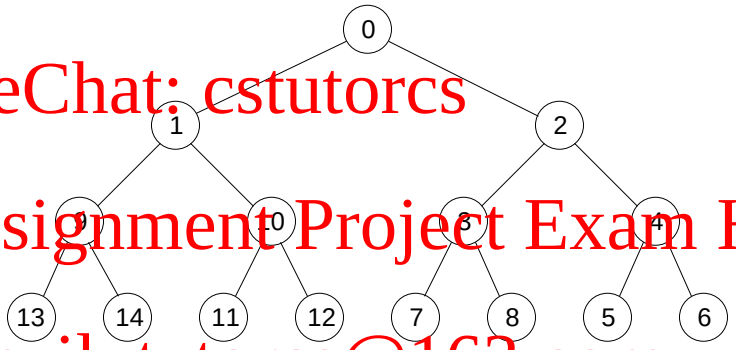


Figure 2.3: Order of Node Generation for Depth-First Expansion

A simpler and more natural way to implement DFS is recursively. Write a function that takes a node as an argument, and performs a DFS below that node. This function will loop through each of the children of its argument node, and make a recursive call to perform a DFS below each of the children in turn. This implementation gives rise to the node generation order in Figure 2.2, which is referred to as depth-first generation. Once a goal state is reached, the solution path is contained in the recursion stack, and can be accessed by returning back up the stack. We will assume this recursive implementation of DFS from here on.

2.2.1 Space Complexity

The primary advantage of DFS is that its space requirement is only linear in the maximum search depth, as opposed to exponential for BFS. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node. For the recursive implementation, this space is occupied by the recursion stack. In particular, if d is the maximum depth of search, and b is the branching factor,

then depth-first search stores $O(d)$ nodes, while depth-first expansion stores $O(d \cdot$ [illegible] stores all the siblings of nodes on the current search p[illegible]ult, memory is not a constraint in practice, and DF[illegible]ng as the computer and the power stay up, which c[illegible]me problems. Thus, as a practical matter, depth-first search is time-limited rather than space-limited.

2.2.2 Time Complexity

The worst-case time complexity of a DFS to depth d is $O(b^d)$, since it generates the same set of nodes as BFS, but simply in a different order, as shown by Figures 2.1, 2.2, and 2.3. This assumes that every branch is terminated at depth d.

On an infinite tree, however, DFS may not terminate, but simply go down the left-most path forever. For example, the state-space graph for the Eight Puzzle contains only 181,440 nodes. In the full state-space tree, however, of which only a tiny fraction is shown in Figure 1.4, every path is infinitely long, eventually generating an infinite number of duplicates of states already generated.

The usual solution to this problem is to impose an artificial cutoff depth on the search. The ideal cutoff is the solution depth d. This value, however, is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, relative to BFS.

2.2.3 Solution Quality

In addition, if the cutoff depth is greater than d, then the first solution found may not be an optimal one. For example, in Figure 2.2, if both node 4 and node 9 are goal states, a left-to-right DFS will return with the depth-3 solution path to node 4, rather than the depth-2 solution path to node 9.

2.3 Depth-first Iterative-Deepening

Th oblem is an algorithm called *Depth-first iterative-de* ich combines the best features of breadth-first an 96, 41]. DFID first performs a depth-first search to depth one, then starts over, executing a complete DFS to depth two, and continues to run a series of depth-first searches to successively greater depths, until a solution is found. Figure 2.4 shows the sequence of node generations, including node regenerations, for a DFID search to depth three.

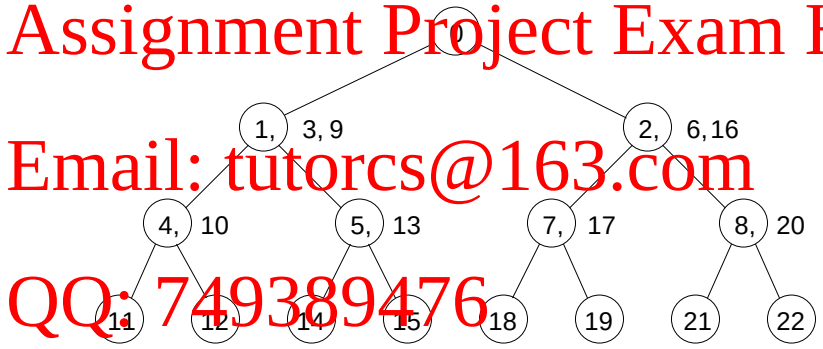


Figure 2.4: Order of Node Generation for DFID Search

2.3.1 Solution Quality

Like BFS, DFID never generates a node until all shallower nodes have already been generated. As a result, the first solution found by DFID is guaranteed to be along a shortest path. Furthermore, DFID is complete and hence guaranteed to find a solution if one exists.

2.3.2 Space Complexity

At any given point, DFID is executing a depth-first search, saving only a stack of nodes. Since the algorithm terminates when it finds a solution at depth d, the space complexity of DFID is only $O(d)$.

2.3.3 Time

Although it appe es a great deal of time in the iterations prior to tl olution, this extra work is usually insignificant. To the number of nodes generated by BFS or DFS to d sponds to the worst-case complexity of the last iteration of DFID, is approximately $b^d \frac{b}{b-1}$. Similarly, the number of nodes generated by DFID in the next-to-last iteration is approximately $b^{d-1}\frac{b}{b-1}$. In general the total number of nodes generated by DFID in a search to depth d is approximately

$$b^d \frac{b}{b-1} + b^{d-1}\frac{b}{b-1} + \cdots + b\frac{b}{b-1} + \frac{b}{b-1}$$

$$= \frac{b}{b-1}(b^d + b^{d-1} + \cdots + b + 1)$$

$$= \frac{b}{b-1}b^d\frac{b}{b-1} = b^d\left(\frac{b}{b-1}\right)^2$$

Assuming that b is fixed and greater than one, the time complexity of DFID as d grows large is asymptotically $O(b^d)$. In other words, most of the work goes into the final iteration, and the cost of the previous iterations is relatively small. In fact, the ratio of the number of nodes generated by DFID to those generated by BFS on a tree is

$$b^d(\frac{b}{b-1})^2 / b^d(\frac{b}{b-1}) = \frac{b}{b-1}$$

This overhead decreases with increasing branching factor. For example, on a binary tree, DFID generates about twice as many nodes as BFS, on a ternary tree only 50% more, and on a quaternary tree only a third more nodes.

2.3.4 Optimality of DFID

In fact, we can prove the following theorem:

Theorem 2.1 *DFID is asymptotically optimal in terms of time and space among all brute-force shortest-path algorithms on a tree with unit edge costs[41].*

lt, we have to verify that DFID is optimal in te ty, time complexity, and space complexity. Since it at a given level before any nodes at the next de olution it finds is arrived at via an optimal path. W plexity next.

Optimality of Time Complexity

To show that DFID is optimal in terms of time complexity, we need to prove that no other brute-force search algorithm that is guaranteed to find shortest-path solutions on a tree with unit edge costs can take asymptotically less time. Assume the contrary, that there exists a brute-force search algorithm A that is guaranteed to find a shortest path to a goal, but whose running time is less than b^d on a tree with branching factor b and solution depth d. Assume that algorithm A works correctly on some problem P with branching factor b and depth d. Since its running time is less than b^d, and there are b^d nodes at depth d, there must be at least one node n at depth d that algorithm A does not generate when solving problem P. Now construct a new problem P' that is identical to problem P except that the goal states of problem P are not goal states in problem P', but node n is a goal in problem P'. Since algorithm A is a brute-force search algorithm, it doesn't use any information to guide it to a goal or to prune any areas of the search tree. Thus, algorithm A will examine the same nodes in problem P' as it examined in problem P. Since algorithm A doesn't examine node n in problem P, then it must not examine node n in problem P'. Thus, algorithm A fails to solve problem P', since node n is the only goal node. This violates our assumption that algorithm A is guaranteed to find a shortest path to a goal if one exists. Thus, no brute-force algorithm that runs in time less than b^d can guarantee finding a shortest solution to a problem with branching factor b and depth d, and hence every such algorithm must take at least b^d time. Since DFID takes $O(b^d)$ time on such a problem, its time complexity is asymptotically optimal.

Optimality of

To show that D complexity is optimal, we rely on a well-known an al result that any algorithm that takes $f(n)$ time g $f(n)$ space. To see this, assume we have an algo some problem, receives all of its input at the beginning of its execution, runs for $f(n)$ time steps, and then terminates. We can model the computer that algorithm A runs on as a finite state machine M. Note that "state" in this context refers to the state of the machine M, including the program status word, all the registers, the entire contents of memory, etc., and not the state of a problem space. Since all the input occurs at the beginning of the run, each machine state is simply a function of the previous machine state. Thus, algorithm A puts machine M through $f(n)$ states before reaching the terminal state. If any two of these $f(n)$ states in the chain were identical, then machine M would be in an infinite loop, violating the assumption that it terminates. Thus all $f(n)$ states must be unique. In order to represent $f(n)$ distinct machine states, $\log f(n)$ bits of memory are required. Thus, any algorithm that takes $f(n)$ time must use at least $\log f(n)$ bits of memory.

Since we showed above that any brute-force shortest-path algorithm must take at least b^d time, any such algorithm must use at least $\log b^d$ space, which is $O(d)$ space. Since DFID uses $O(d)$ space, it is asymptotically optimal in space as well. This completes the proof of our theorem.

2.3.5 Graphs with Cycles

The above theorem relates to a tree-structured problem space with no cycles. On a graph with cycles, however, BFS can be much more efficient than any depth-first search, including DFID. The reason is that a BFS can detect all duplicate nodes whereas a DFS cannot. Thus, the complexity of BFS grows only as the number of nodes at a given depth, while the complexity of DFS depends on the number of paths of a given length. For example, in a square grid, the number of nodes within a radius r of the origin is $O(r^2)$, whereas the number of paths of length r is $O(4^r)$, since there are four neighbors of every node. Thus, in a

gra mber of very short cycles, BFS is preferable to DI ory is available. The problem, of course, is that it

Pr Nodes in Depth-First Search

In some problem spaces, however, we can prune many of the duplicate nodes generated by a depth-first search. Continuing with the grid example, if we eliminate the parent of each node as one of its children, we can reduce the branching factor from four to three. This is easily done with the aid of a *finite-state machine* (FSM) that remembers the last operator applied, and prohibits the inverse of the last operator applied from being applied next. Figure 2.5 shows such a machine, where each state except the start represents the last operator applied, and the transitions represent the legal operators for the search. What is missing from this figure is any edge from an operator to its inverse operator.

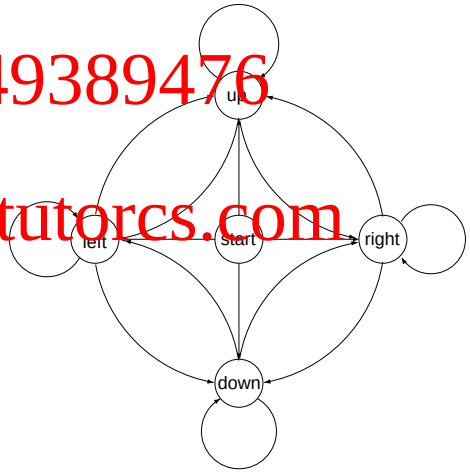


Figure 2.5: FSM for Grid that Rejects Inverse Operators

In fact, we can carry this idea further, as shown by the FSM in Figure 2.6. From the start state, this machine allows an arbitrarily long sequence of moves to the right, or to the left. Following either sequence, a sequence of up moves, or a sequence of down moves is allowed. Thus, the only path allowed by this machine to a grid point (x, y) is all left or

all right moves to[illegible]llowed by all up or all down moves to the y coordina[illegible]y one such path to each grid point, the time comple[illegible]rolled by this finite-state machine is $O(r^2)$, the san[illegible]more general spaces, however, this technique canno[illegible]cate nodes. For more information on this method of duplicate pruning in depth-first search, including how to automatically learn these finite-state machines from a problem description, see [97].

Figure 2.6: More Efficient FSM for Grid Space

2.4 Node Generation Times

We argued above that on a tree, DFS and BFS generate the same number of nodes, and that DFID generates asymptotically the same number of nodes as DFS and BFS. While this is true, the amount of time required to generate each node can make both DFS and DFID more efficient than BFS. BFS must maintain a queue with a copy of each distinct node. Thus, the amount of time to generate a node is proportional to the size of the state representation. In the sliding-tile puzzles, for example, the time to generate a node is linear in the number of tiles, since we must copy the position of each stationary tile,

an ion of the tile that is moved. For our asymptotic an at this time is a constant, but in a puzzle as large as time can be significant.

nted with an explicit stack of nodes, this same ov rred. A simpler implementation of DFS is as a recursive program, however. In that case, the stack of nodes is the recursion stack. In a recursive implementation, we can simply maintain a single copy of the state, changing it as moves are made, and undoing those changes when the search backtracks. For example, in the sliding-tile puzzles, each move requires updating the position of a single tile, and then changing it back to its original value before performing the next move. Thus, a move requires only constant time, independent of the number of tiles in the puzzle. This advantage of all depth-first searches, including DFID, becomes increasingly significant the larger the state description. For example, the state of a chess game consists of an 8×8 array of board squares. In the case of DFID, this advantage often outweighs the additional cost of the non-goal iterations.

2.5 Backward Chaining

So far, we have implicitly assumed that in the problem-space tree the root node represents the initial state, and we search forward until we reach a goal state. Alternatively, the root node could represent the goal state, and we could search backward until we reach the initial state. This strategy is known as *backward chaining* or *backward search*.

Backward chaining is not applicable to all problems. One requirement is that the goal state be represented explicitly. If the goal state is represented implicitly by a set of conditions, it is usually impossible to reason backwards from such a description to the initial state.

It may also seem at first that the operators need to be reversible for backward chaining to be applicable, but this is not the case. All that is necessary is that we be able to reason backwards about the operators. For example, given a road map of all one-way streets, we can plan a route backward from our goal location to our initial location, even though none of the operators can be applied in the reverse direction.

For some problems, such as the sliding-tile puzzles and Rubik's

Cube, the initia[illegible]e symmetric with respect to each other, and there [illegible] to prefer forward search to backward search. On[illegible]olem instance, however, one direction may be mo[illegible] other. For example, if the initial state of the Eig[illegible]olank in the center, and the goal state has the blank in a corner, a backward search would probably be more efficient because the branching factor of the root node in that case would be two instead of four. As a result, a brute-force search tree of a given depth will be about twice as large if the blank starts in the center, rather than in a corner.

For other problems, however, the asymptotic forward and backward branching factors may be quite different. Consider, for example, the problem of theorem proving. A state of the problem would be a set of formulas known to be true, and an operator is a rule of inference that takes one or more formulas and derives a new true formula. The initial state is the set of given formulas of the problem, and the goal state is a set of formulas that includes the theorem to be proven.

A single forward search step is to derive an additional true formula. There are usually a large number of such steps, most of which have no relevance to the theorem to be proven, and thus the forward branching factor is typically large. A backward chaining step would examine the theorem to be proven, and determine what other formulas, if proven, would imply the theorem in a single inference step. There are usually only a small number of such formulas, and hence the backward branching factor in theorem proving is much smaller than the forward branching factor. As a result, backward chaining is a more efficient inference mechanism, and is the method of choice in theorem proving.

2.6 Bidirectional Search

Even if the asymptotic forward and backward branching factors are the same, as in Rubik's Cube or the sliding-tile puzzles, we can still use the idea of backward search to advantage, in an algorithm called *bidirectional search*[77]. The main idea is to simultaneously search forward from the initial state, and backward from the goal state, until the two search frontiers meet at a common state. The solution path from the

ini ncatenated with the inverse of the solution path frc form the complete solution path.

2. Quality

Bidirectional search guarantees finding a shortest path from the initial state to the goal state if one exists. Assume that there is a solution of length d, and that both searches are breadth-first. When the forward search has proceeded to depth k, its frontier will contain all nodes at depth k from the initial state. When the backward search has proceeded to depth $d-k$, its frontier will contain all states at a depth of $d-k$ from the goal state. Consider the state s reached along an optimal solution path at depth k from the initial state. This state is also at depth $d-k$ from the goal state. Thus, the state s is in the frontier of both searches, and the algorithm will find the match and return the optimal solution.

2.6.2 Time Complexity

While the two search frontiers don't need to advance at an equal rate to ensure optimal solutions, the time complexity of bidirectional search is minimized if the two search frontiers meet in the middle. In that case, each search proceeds to depth $d/2$ before they meet. Thus, the total number of nodes generated is $O(2b^{d/2}) = O(b^{d/2})$. This is a very significant savings, amounting to only the square root of the number of nodes generated by a unidirectional search.

This is not necessarily the asymptotic time complexity of bidirectional search however. The problem is that every new node generated has to be compared against the nodes in the opposite search frontier to see if a common node has been found between the two frontiers. If this is done naively, by sequentially comparing each new node against each of the nodes in the opposite search frontier, the time per node generation becomes the size of the opposite search frontier, or $O(b^{d/2})$, and the time complexity of the whole algorithm becomes $O(b^d)$, which is no better than unidirectional search.

This node comparison can be done much more efficiently, however. The best way is to use a function to hash the nodes generated in one direction into a hash table. Then, as each node is generated in the

opposite direction function is applied and compared against the entries. With a good hash function, the time to do the hash comparison will be constant in the average case, although the worst case complexity can be much worse. If we can generate, hash, and check a node in constant average time, the overall asymptotic time complexity of the algorithm becomes $O(b^{d/2})$ again, realizing the savings originally expected.

There are two lessons to learn from this. First, in order to determine the asymptotic time complexity of an algorithm, we have to be sure that we are counting the number of critical operations, and that these operations can be done in constant time. Second, we need to be able to bring the full power of the most efficient algorithms available to bear on any aspect of our search problem.

2.6.3 Space Complexity

The simplest implementation of bidirectional search is to use BFS for both searches. In fact, only one of the searches needs to be a BFS, and the search in the other direction can be a depth-first search such as DFID. In order to detect a match between nodes from the two different searches, however, at least one of the frontiers must be stored in memory, as in BFS. Thus, the space complexity of bidirectional search is dominated by the BFS search and is $O(b^{d/2})$. As a result, bidirectional search is space bound in practice. If there is sufficient memory to store this many nodes, however, bidirectional search is much more time efficient than unidirectional search.

2.6.4 A Finite Graph

The above analysis assumes we are searching a potentially infinite problem-space tree. On a finite graph, however, bidirectional search may be no more efficient than unidirectional search in the worst case. Consider searching a square grid from an initial state in one corner to a goal state in the opposite corner. A unidirectional breadth-first search will generate the entire graph before finding the goal state. A bidirectional breadth-first search, starting from the two corners, will also generate the entire graph before finding a common node.

Chapter 3

Best-First Search

So far, we have assumed that all edges have the same cost, and that an optimal solution is a shortest path from the initial state to a goal state. We now generalize this model to allow individual edges with arbitrary costs associated with them, where the cost of a path is the sum of the edge costs on the path. An optimal solution is a path of lowest cost from the initial state to a goal state. We will use the *length* of a path to denote the number of edges in the path, and the *cost* of a path for the sum of the edge costs on the path.

To handle this case, and others, we introduce *best-first search*, an entire class of search algorithms, each of which employ a *cost function*. A cost function is a function from a node to a number that represents the cost of that node, such as the sum of the edge costs from the root to the node, for example. In these notes, we will assume that a lower-cost node is a better node, but this can easily be transformed to a maximization problem where we have rewards instead of costs. The various best-first search algorithms differ primarily in their cost function.

Best-first search employs two lists of nodes, an *Open* list and a *Closed* list. The Closed list contains those nodes that have already been completely expanded, while the Open list contains those nodes that have been generated but not yet expanded. Initially, just the root node is included on the Open list, and the Closed list is empty. At each cycle of the algorithm, an Open node of lowest cost is expanded, moved to Closed, and its children are inserted into Open. The Open

lis priority queue. The algorithm terminates when a go expansion, or there are no more nodes remaining in

n one example of best-first search, namely breadth-fir st of a node is simply its depth below the root, then best-first search becomes breadth-first search. We do not consider depth-first search to be a best-first search, because in order to run in linear space, DFS doesn't maintain Open or Closed lists.

3.1 Uniform-Cost Search

Let $g(n)$ be the sum of the edge costs from the root to node n. If $g(n)$ is our overall cost function, then best-first search becomes *uniform-cost search*, also known as Dijkstra's single-source shortest-path algorithm[18]. Initially the root node is placed in Open with a cost of zero. At each step, the next node n to be expanded is an Open node whose cost $g(n)$ is lowest among all Open nodes.

Figure 3.1 shows an example tree with different edge costs, represented by the numbers next to the edges. To search this tree with uniform-cost search, we begin with the root node a on the Open list, expand it, generating nodes b and c, and place node a on the Closed list. The cost of node b is 2, and the cost of node c is 1, the costs of their respective edges from the root. Since node c has the lowest cost on Open, we expand it next, generating nodes d and e, and place node c on Closed. The cost of node d is 2, and the cost of node e is 3, the sum of the costs of the edges connecting them to the root node a. At this point, Open contains nodes b, d, and e, and we have a tie between b and d, both with the lowest cost of 2. If we break ties in favor of the shallower node, we will expand node b next, generating nodes f and g, and placing b on Closed. The next node to be expanded will be node d, since its cost of 2 is now the lowest cost of any Open node.

We consider uniform-cost search to be a brute-force search, because it doesn't use a *heuristic function*, which we will introduce in the next section. We cover uniform-cost search in this chapter, rather than the last one, because it is a non-trivial example of best-first search, compared to breadth-first search, for example, which can also be considered

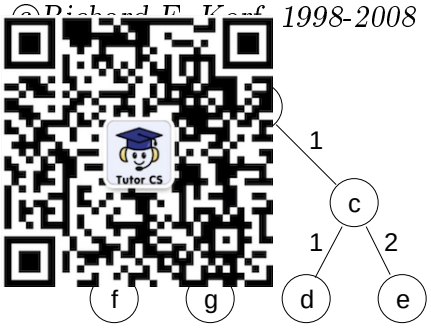


Figure 3.1: Example of Uniform-Cost Search

a best-first search.

The first questions to ask are whether uniform-cost search always terminates, and whether it is guaranteed to find a goal state.

3.1.1 Termination

Since the Open and Closed lists contain all the nodes that have been generated so far, as each new node is generated, it can be compared against the nodes on those lists. If a new state is already on Open or Closed, we don't add another copy to the Open list. Rather, if we have found a lower-cost path to an existing node on Open, the pointer from that node back to its parent is redirected to its parent along the lower-cost path. We will see an example of this below. Under most conditions, detailed below, we cannot find a lower-cost path to a node on Closed. If the new path is of equal or greater cost instead, we simply discard the new node and continue.

Thus, if the problem-space is finite, we will eventually search the entire graph that is connected to the initial state. In some problems, such as Rubik's Cube and the sliding-tile puzzles, there are multiple components of the problem space that are not connected by legal operators. Since the entire connected component of a finite graph will eventually be explored by the algorithm, it will terminate and either return a goal if one exists, or report that there is no goal node in the graph.

We next consider an infinite graph. If there is no goal node, or no finite-length path to a goal node, or no path to a goal of finite cost, best-first search will not reach a goal. A finite-length path may have

inf more operators on the path have infinite cost. Fu orithm will not detect these cases in a infinite gr arching forever, or until space is exhausted.

oal, there must exist a path to a goal with finite ler This is not enough however. In order for the algorithm to eventually reach the goal, there must not be any infinitely long paths of finite cost. For example, there cannot be an infinite chain of nodes with zero-cost edges. A cycle of zero cost is not a problem since new nodes are compared against previously generated nodes on the Open and Closed list. Even requiring all edges to have positive cost is not sufficient. For example, imagine an infinite chain of edges with cost 1, 1/2, 1/4, 1/8, etc. Such a chain can be infinitely long and its total cost will never exceed two. The easiest way to rule out this situation is to assume that all edges have a minimum non-zero edge cost e. In that case, uniform-cost search will eventually reach a goal of finite cost if one exists in the graph.

3.1.2 Solution Quality

The next question is whether uniform-cost search will find an optimal path to a goal state, if one exists.

In order to guarantee optimal solutions, best-first search must terminate only when a goal node is chosen for expansion, and not just when a goal state is generated. Consider the graph in Figure 3.2. Node s is the start state, node g is the goal state, and the numbers next to the edges are the costs of the edges. Uniform-cost search will first expand node s, generating nodes a and b. Since the cost of node a is lower, it will be expanded next, generating the goal state. The cost of the path to node g through node a is 4, which is not optimal. At this point, the Open list contains nodes b and g. Since node b has the lower cost, it is expanded next, generating another path to node g, this one of cost 3. Since we have found a lower-cost path to a node on Open, the parent pointer from node g is redirected to node b instead of node a. At this point, the only node on Open is node g, so we select it for expansion, and since it is a goal node, we terminate and return the path from node s through node b to node g. Note that if we had terminated the algorithm the first time we generated the goal node g, we would

not have found a

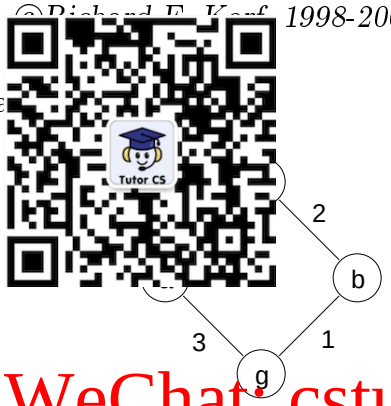


Figure 3.2: Best-first search terminates when a goal is chosen for expansion

In general, uniform-cost search will find a lowest-cost path to a goal node if one exists, as expressed by the following theorem.

Theorem 3.1 *In a graph where all edges have a minimum positive cost, and in which a finite-cost path exists to a goal node, uniform-cost search will return a lowest-cost path to a goal node.*

The first step in the proof of this theorem is to show that, at all times, there exists an Open node which is on an optimal path to a goal. The proof of this fact is by induction on the number of node expansion cycles of the algorithm. The base step is that when the algorithm starts, Open contains just the start node. Clearly the start node is on an optimal path to a goal, if any path to a goal exists.

The induction step is to show that if Open contains a node on an optimal path to a goal before a node expansion, then it must contain one after the node expansion. There are two cases to consider. If the node expanded was not on an optimal path to a goal, then the Open node that was on an optimal path is still in the Open list. Conversely, if the node expanded was on an optimal path, then at least one of its children, which are now in the Open list, is on an optimal path to a goal. Thus, at all times there is an Open node on an optimal path to a goal state, if such a path exists.

The next step is to show that if there is a path to a goal node, the algorithm will eventually find it. The algorithm terminates when it chooses a goal node for expansion. The only ways it can fail to do

th pen nodes, or run forever. Since there is always an ptimal path to a goal, if there is such a path, it ca nodes.

very node in the Open list with finite cost must ev d. The reason for this is that at every node-expansion cycle, a node of lowest cost is expanded, and since there is a minimum positive edge cost, it is replaced by children of greater cost. Thus, the cost of the lowest-cost node in the Open list must increase without bound, eventually exceeding the cost of every node in the Open list with finite cost. Thus, every such node will eventually be expanded, including all the nodes on an optimal solution path, until a goal node is chosen for expansion.

The first time a goal node is chosen for expansion, the algorithm terminates and returns the path to that node as the solution. Let the cost of the path to that goal node be c. Since it was chosen for expansion, all other nodes in the Open list have cost $\geq c$. In particular, any Open nodes on an optimal solution path must have cost greater than or equal to c. Since extending a path can only increase its cost by the minimum edge cost or more, the optimal solution must have cost $\geq c$. Since we have found a path of cost c and the optimal path must have cost $\geq c$, we have found an optimal path to the goal, completing the proof of our theorem. □

3.1.3 Time Complexity

The next question is how long does uniform-cost search take to run? If there is no solution path, and the graph is infinite, it will run forever. If there is a solution, however, it will eventually terminate, returning an optimal solution.

In the worst case, every edge has the minimum edge cost e. To see this, note that an edge with cost greater than e can be replaced by a chain of nodes of lower cost, increasing the number of node expansions required to traverse the chain. If c is the cost of an optimal solution path, once all nodes of cost $\leq c$ have been chosen for expansion, a goal node must be chosen. The maximum length of any path searched up to this point cannot exceed c/e, and hence the worst-case number of such nodes is $b^{c/e}$. Thus, the worst-case asymptotic time complexity

of uniform-cost s ote that the goal may be directly connected to th dge, and hence at depth one, but the cost of that worst-case time complexity. The reason is that th terminate as soon as a goal node is generated, but ode is chosen for expansion.

3.1.4 Space Complexity

The next question to ask is how much memory is required by uniform-cost search. As in all best-first searches, each node that is generated is stored in the Open or Closed lists, and hence the asymptotic space complexity of uniform-cost search is the same as its asymptotic time complexity, $O(b^{c/e})$. As a result, uniform-cost search is memory-limited in practice.

3.1.5 Complexity of Dijkstra's Algorithm

Although Dijkstra's algorithm is the same as uniform-cost search, its time complexity is usually reported as $O(n^2)$, instead of $O(b^{c/e})$. There are several reasons for this discrepancy. The first is that n is the total number of nodes in the graph. Since we assume that the graph may be infinitely large for uniform-cost search, we measure problem size by the branching factor b and solution cost c. Secondly, in Dijkstra's algorithm it is assumed that every node may be connected to every other node, which gives rise to the quadratic complexity. In contrast, for uniform-cost search we assume a constant-bounded branching factor of b.

3.2 Combinatorial Explosion

All the techniques we have considered so far are brute-force methods, in that they rely only on the problem space, the initial state, and a description of the goal state. The problem with brute-force search algorithms is that they are not efficient enough to solve even moderately large problems.

ating the branching factor and solution depth, we he size of a problem simply by the number of sta space. Since a brute-force algorithm uses no ad , it can be expected to examine half the states in on average, to solve a random problem instance.

On current machines, an efficient implementation of depth-first iterative-deepening generate up to about 10^8 nodes per second. At this rate, the Eight Puzzle, with about 10^5 states, could be solved in a millisecond. The Fifteen Puzzle, with a branching factor of 2.13, and an average optimal solution depth of 52.5 moves, would require an average of over 50 years of computation to solve a single random instance. The Twenty-Four Puzzle has a branching factor of 2.3676, and an average optimal solution depth of 101 moves, and would require an average of over 10^{12} years to optimally solve a random instance. Similarly, with 3.67 million states, the $2 \times 2 \times 2$ Rubik's cube would take less than a tenth of a second to solve optimally, but the full $3 \times 3 \times 3$ Cube, with 4.3252×10^{19} states, would require about 57 thousand years to optimally solve a random initial state. Clearly what is needed is a new idea.

3.3 Heuristic Evaluation Functions

The efficiency of a brute-force search can be greatly enhanced by the use of a *heuristic static evaluation function*, or *heuristic function*, often without any sacrifice in solution quality. In the context of a single-agent path-finding problem, a heuristic function takes a state and returns an estimate of the cost of an optimal path from that state to a goal state. There are two ways in which such a function can improve the efficiency of a search algorithm. One is by leading the algorithm toward a goal state, and the other is by pruning off branches that don't lie on any optimal solution paths.

3.3.1 Example Heuristic Functions

For example, given the task of navigating in a network of roads from one location to another, a simple heuristic function is the Euclidean or airline distance between the current location and the goal. This

function estimat e in the road network between the two points, but c onstant time, given the coordinates of the two locati

As another ex euristic function for the sliding-tile puzzles is called . Given a particular current state and goal state, it is computed by taking each tile, counting the number of horizontal and vertical grid units it is displaced from its goal position, and summing these values over all tiles, excluding the "blank". This gives an estimate of the number of moves needed to solve the problem, since each tile must move at least its Manhattan distance to the goal, and each operator moves one tile. This estimate can be computed in time linear in the number of tiles.

Finally, consider the travelling salesman problem (TSP) of finding a shortest tour that covers a set of cities. A good heuristic function for this problem is the cost of a minimum spanning tree (MST) of the cities. A spanning graph is one that connects all the nodes of a graph, and an MST is a spanning graph of minimum total edge cost. A minimum spanning graph will always be a tree, since given any cycle, we can remove any edge, thereby reducing the total cost while still covering all the nodes. Whereas the cost of an optimal TSP tour almost certainly takes exponential time to find, a minimum spanning tree covering N nodes can be computed in $O(N^2)$ time.

3.3.2 Properties of Heuristic Functions

The two most important properties of a heuristic function are that it be a relatively accurate estimator of the cost to reach a goal, and that it be relatively cheap to compute. In practice, the dividing line between cheap and expensive computation is the same as the line between polynomial-time and exponential-time algorithms. As long as a heuristic can be computed in time polynomial in the size of the problem, it will often be useful, at least for large problems.

The other property that all of the above example heuristics have, which will allow us to guarantee optimal solutions, is that they are all lower bounds on actual solution cost. This property is known as *admissibility*. Clearly, the airline distance between two points in a network of roads has to be less than or equal to their distance in the

road sliding-tile puzzles, since each tile must move its e to its goal location, and every operator moves on nhattan distance must be less than or equal to th tion path to the goal. Finally, a solution to the TS graph that includes every node in the problem, in addition to other properties. Since a minimum spanning tree is the lowest-cost subgraph that includes every node, its cost must be a lower bound on the cost of any TSP tour, including the optimal TSP tour. In fact, most naturally occurring heuristics have this property, as we will see in Chapter 6.

3.4 Pure Heuristic Search

Given a heuristic evaluation function, the simplest algorithm that uses it is best-first search where the cost function $f(n)$ is just the heuristic function $h(n)$. We refer to this algorithm as *pure heuristic search*[19] (PHS). In the conventional notation, $f(n)$ refers to whatever cost is associated with a node n in the best-first search, and $h(n)$ is the heuristic estimate of the cost of a path from node n to a goal node, the goal node being left implicit by the notation.

Since best-first search maintains Open and Closed lists of nodes that have been generated, and compares newly-generated nodes against these lists, on a finite graph, PHS will eventually generate the entire graph, finding a goal node if one exists. If the graph is infinite, however, it is easy to construct an infinite path and set of heuristic values that will lead the algorithm away from a goal node forever. Thus, PHS is not guaranteed to terminate on an infinite graph, even if a goal node exists.

Even if PHS does terminate with a solution, the solution is not guaranteed to be an optimal one. For example, consider the graph in Figure 3.3, and assume that the heuristic function is an exact estimator of distance to the goal. After node s is expanded, it leaves nodes a and b on the Open list, with $h(a) = 2$ and $h(b) = 1$. PHS will expand node b next, finding a path of length 4 to the goal node g. Since $h(g) = 0$, the algorithm will then choose the goal for expansion, and terminate, returning a solution of length 4, when one of length 3 exists. The

problem is that the estimated cost $h(n)$ to a goal when choosing a and doesn't consider the cost $g(n)$ from the initial s

Figure 3.3: Pure heuristic search doesn't guarantee an optimal solution

In spite of this, PHS is often an effective algorithm for finding suboptimal solutions to combinatorial problems. For example, on the Fifteen Puzzle, using the Manhattan distance heuristic function, PHS is able to find solutions that are less than three times the length of optimal solutions on average, generating less than seven thousand nodes per problem instance, a small fraction of the hundreds of millions of node generations required to find optimal solutions[42]. On the Twenty-Four Puzzle, PHS also finds solutions that are about three times longer than optimal solutions, generating less than a hundred thousand nodes per problem instance, compared to trillions of nodes for optimal solutions in some cases[48].

3.5 A* Algorithm

In order to find optimal solutions, we have to take into account the cost of reaching each Open node from the initial state, as well as the heuristic estimate of the remaining cost to the goal. The A* algorithm[34] is a best-first search where the cost function $f(n)$ associated with each node n is the sum of $g(n)$, the cost of node n from the initial state, and $h(n)$, the heuristic estimate of the lowest cost from node n to a goal state. For a given $h(n)$, $f(n) = g(n) + h(n)$ is the best estimate of a lowest-cost path from the initial state to a goal state that is constrained to

pa The "A" stands for "algorithm", and the asterisk in y property, discussed below[69].

3. ling Conditions

Like all best-first searches, A* terminates when it chooses a goal node for expansion, or when there are no more Open nodes. In a finite graph, it will eventually explore the entire graph if it doesn't find a goal state. In an infinite graph, if all edge costs are finite and have a minimum positive value, and all heuristic values are finite and non-negative, then the cost of nodes along every path will eventually increase without bound. If in addition there is a finite-cost path to every node in the graph, then every node must eventually be selected for expansion, until a goal node is chosen.

Note that the Closed list is not necessary to keep A* out of infinite loops. The reason is that every trip around a loop will increase the $g(n)$ value, eventually causing A* to leave the loop. The primary purpose of the Closed list in A* is to improve the time efficiency of the algorithm in a graph with cycles, by detecting previously expanded states, and not reexpanding them.

3.5.2 Solution Quality

In general, A* is not guaranteed to return optimal solutions. Consider the graph in Figure 3.4. After the start state s is expanded, it is moved to Closed, and its two children, nodes a and b, are placed on Open. Their costs are computed as follows: $f(a) = g(a) + h(a) = 1 + 1 = 2$, and $f(b) = g(b) + h(b) = 1 + 3 = 4$. Since node a has a lower cost, it is expanded, moved to Closed, and its child node c is placed on Open. Assume that node c is a goal node. Its cost is $f(c) = g(c) + h(c) = 3 + 0 = 3$, since we assume that a goal node has heuristic cost zero, or it wouldn't be recognized as a goal node. The other node on Open, node b, has cost 4, and hence node c is chosen for expansion, terminating the algorithm. The solution returned is the path from the start to node c through node a. This path has a cost of 3, which is suboptimal, since there exists a path to the goal through node b with a cost of only 2. Thus, A* failed to find an optimal solution to this problem instance.

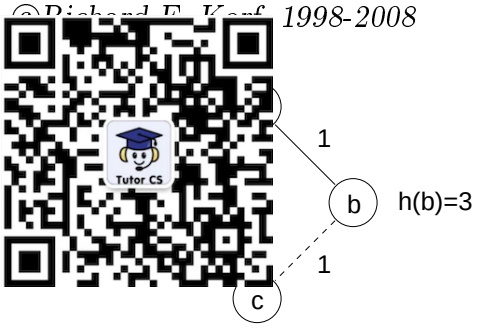


Figure 3.4: A* may not return an optimal solution if the heuristic overestimates

We can blame this failure on the fact that the heuristic value at node b, $h(b) = 3$, overestimates the cost of reaching the goal from node b, which is only 1. Indeed, we can prove that if the heuristic function never overestimates the actual cost of reaching a goal from any node, then A* is guaranteed to return an optimal solution if one exists.

Theorem 3.2 *In a graph where all edges have a minimum positive cost, and non-negative heuristic values that never overestimate actual cost, in which a finite-cost path exists to a goal state, A* will return an optimal path to a goal.*

Much of the proof is similar to the corresponding theorem for uniform-cost search. The first step is to show that there is always an Open node on an optimal solution path. The proof of this is by induction on the number of node expansions of the algorithm. For the base step, when the algorithm starts, Open contains just the start node. Clearly the start node is on an optimal path to the goal, if any path to a goal exists.

The induction step is to show that if the Open list contains a node on an optimal path to a goal before a node expansion, then it must contain one after the node expansion. There are two cases to consider. If the node expanded was not on an optimal path to the goal, then the Open node that was on an optimal path is still on Open. Conversely, if the node expanded was on an optimal path, then at least one of its children, which are now Open, is on an optimal path to the goal. Thus, at all times there is an Open node on an optimal path to a goal state, if such a path exists.

show that if there is a path to a goal node, the alg lly find it. The algorithm terminates when it ch r expansion. The only ways it can fail to do this is nodes, or run forever. Since there is always an Op nal path, if there is a path to a goal, it can't run out of Open nodes.

If it runs forever, every Open node with finite cost must eventually be expanded. The reason for this is that every node expansion expands a node of lowest cost, and since there is a minimum positive edge cost, it is replaced by its children whose g cost is strictly greater than that of their parent by at least the minimum edge cost. While the g value of a child must be strictly greater than that of its parent, its h value could be lower, resulting in an overall f value for the child that is less than that of the parent. This decrease in f values can only be maintained for a limited number of generations, however, since the h values are non-negative, and hence eventually the f values of the descendents must exceed that of their ancestors. Thus, the cost of the lowest-cost node on Open must eventually increase without bound, exceeding the cost of every Open node with finite cost. Thus, every such node will eventually be expanded, including all the nodes on an optimal solution path, until a goal node is chosen for expansion.

The first time a goal node is chosen for expansion, the algorithm terminates and returns the path to that node as the solution. Let the cost of that node be c. Since it was chosen for expansion, all other Open nodes have cost $\geq c$. In particular, any Open node n on an optimal solution path must have cost greater than or equal to c. Since node n is on an optimal path, the path from the initial state to node n must be a lowest-cost path to node n. Furthermore, since the heuristic function $h(n)$ cannot overestimate the cost from node n to a goal state, the total cost of node n, $f(n) = g(n) + h(n)$ must be a lower bound on the optimal solution cost. Since the cost of node n is $\geq c$, and is a lower bound on the optimal solution cost, the optimal solution cost must also be $\geq c$. Since we have found a path to the goal of cost c, it must be an optimal path. This completes the proof of our theorem. □

3.6 Admissible, Consistent, and Monotonic Heuristics

A heuristic function that never overestimates the lowest cost from node n to a goal is known as *admissible*[74]. More formally, if we define $h^*(n)$ as the exact lowest cost from node n to a goal, a heuristic function $h(n)$ is admissible if and only if for all nodes n, $h(n) \leq h^*(n)$.

A related property of heuristic functions is called *consistency*, and is similar to the triangle inequality of all metrics. If $c(n, m)$ is the lowest cost of any path from node n to node m, then a heuristic function $h(x)$ is said to be consistent if for all nodes n and m, $h(n) \leq c(n, m) + h(m)$. To see the relationship to the triangle inequality, consider Figure 3.5. Consistency says that the heuristic estimate of the cost from node n to a goal node g is less than or equal to the sum of the costs from node n to node m, and the heuristic estimate of the cost from node m to a goal.

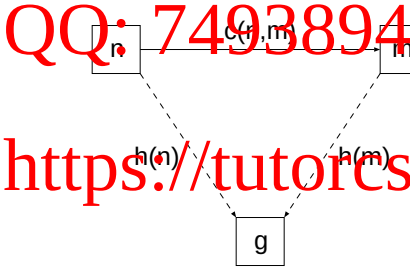


Figure 3.5: The triangle inequality of consistent heuristics

In the global definition of consistency above, nodes n and m can be any nodes in the graph. We can also define a local version of consistency, in which node m is restricted to be an immediate neighbor of node n. Clearly, global consistency implies local consistency. It is also easy to show, by induction on the length of the path from node n to node m, that local consistency implies global consistency. Thus, the two definitions are equivalent, and we will refer to them simply as consistency.

If a heuristic function $h(n)$ is consistent, then the cost function

$f($ *monotonic* nondecreasing, in the sense that the co lways greater than or equal to that of its parent. Fo on $f(n)$ is monotonic if and only if for all children n'). The proof that consistency of $h(n)$ implies m s straightforward, as follows:

$$h(n) \leq c(n, n') + h(n')$$
$$g(n) + h(n) \leq g(n) + c(n, n') + h(n')$$
$$g(n) + h(n) \leq g(n') + h(n')$$
$$f(n) \leq f(n')$$

Applying the same proof in the opposite direction shows that monotonicity of f implies consistency of h. Thus, the two properties are equivalent.

The next thing to notice is that consistency implies admissibility. To see this, we simply replace m with a goal node G, as follows:

$$h(n) \leq c(n, m) + h(m)$$
$$h(n) \leq c(n, G) + h(G)$$
$$h(G) = 0$$
$$h(n) \leq c(n, G)$$
$$h(n) \leq h^*(n)$$

Unfortunately, admissibility does not imply consistency. In that sense, consistency is a stronger property. As we will see in Chapter 6, however, most naturally occurring heuristic functions are consistent. In fact, admissible but inconsistent heuristic functions are rare in practice.

Given an admissible but inconsistent heuristic function h, which gives rise to a nonmonotonic cost function f, we can easily construct a monotonic f function that is still admissible. Whenever the $f(n')$ value of a child node n' is less than the $f(n)$ value of its parent node n, we simply set the $f(n')$ value of the child to the $f(n)$ value of the parent[58]. More generally, we set the f value of a child node to the maximum of the $f(n')$ value of the child node and the $f(n)$ value of the parent. If the heuristic function $h(n)$ is admissible, then this new cost

function $f(n)$ wil as follows. If $h(n)$ is a lower bound on the cost from hen $f(n) = g(n) + h(n)$ is a lower bound on the tot e goal from the initial state via the current path to the total cost of reaching the goal via every child o least as large as the minimum cost through the parent. Thus, setting the f value of the children to be at least as large as the f value of the parent preserves admissibility of the overall cost function. Furthermore, the resulting modified cost function will be more accurate than the original cost function, since both are guaranteed to be lower bounds, but in some cases, the modified cost will be greater than the original cost.

In addition, having a monotonic cost function f simplifies certain details of a best-first search algorithm such as A*. In particular, with a monotonic cost function, the first time a node is expanded, the path to that node is guaranteed to be a lowest-cost path from the root. Thus, whenever a node is moved to the Closed list, we have found an optimal path to it. However, with an inconsistent heuristic function, and hence a non-monotonic overall cost function, the first time a node is expanded it may not have been arrived at via an optimal path from the root. This means that nodes on the Closed list may have non-optimal paths to them, and when the same state is rediscovered via a lower-cost path, the path to the state on the Closed list would have to be updated to reflect the lower-cost path. Due to the rarity of inconsistent heuristics, which we will explain in more detail in Chapter 6, and the above mechanism for transforming an inconsistent but admissible heuristic into a consistent admissible heuristic, we will generally ignore this additional complication in these notes. Note that even with a consistent heuristic, however, the first time a node is generated, it is not necessarily arrived at via an optimal path from the root. Thus, when a state is generated that is already on Open, we need to preserve the lower-cost path to the state.

3.7 Time Complexity of A*

How long does A* take to find an optimal solution? Generally, the running time will be proportional to the number of nodes generated or

ex asymptotically the same. This assumes that the br most a constant, and that the generation and he a node can be done in constant time.

t the Open and Closed lists can be maintained in co e expansion. The Closed list can be organized as a hash table, since we only need to check for the occurrence of a node on Closed. For the Open list, however, we have to be able to insert a node, and retrieve a lowest-cost node, in constant time. For arbitrary values, this would take time that is logarithmic in the size of the Open list. However, in many cases, the heuristic functions and edge costs are integer valued or have a small number of distinct values at any point in time. In such cases, the Open list can be maintained as an array of lists, where each list contains only nodes of the same cost, and there is a separate list for each different cost. This allows constant-time insertion and retrieval from the Open list.

Thus, the question becomes how many nodes A* generates in the process of finding a solution. Clearly, the answer depends on the quality of the heuristic function. The more accurate the heuristic function, the faster A* will run. We begin by considering two simple special cases.

3.7.1 Special Cases

The simplest special case, which is also the overall worst-case assumption for A*, is that the heuristic function returns zero for every node, and hence provides no information to the algorithm, but is still a lower-bound on actual cost. In this case, the cost function $f(n) = g(n)+h(n)$ becomes $f(n) = g(n)$. Best-first search with this cost function is identical to uniform-cost search or Dijkstra's algorithm, which has a worst-case time complexity of $O(b^{c/e})$, where b is the branching factor, c is the cost of an optimal solution, and e is the minimum edge cost.

At the opposite end of the spectrum, the heuristic function could be perfect, and always return the exact optimal cost to a goal state from any given state. This is often written as $h(n) = h^*(n)$, where $h^*(n)$ represents the exact optimal cost to a goal from node n. In this case, $f(n) = g(n) + h(n)$ is the exact cost of reaching a goal from the initial state, while passing through node n. When the root node is expanded, its children are evaluated at their exact costs. Any child on

an optimal path ... lower cost than any children on non-optimal pat... expanded next, and its children will be evaluated ... will continue until a goal node is chosen for expan... node-expansion cycles will be d, the depth of the ... goal. Thus, the asymptotic time complexity of A* with a perfect heuristic function is $O(bd) = O(d)$.

3.7.2 Tie Breaking

There is an important caveat to this result. Consider a problem where every node at depth d is a goal node, and every path to depth d is an optimal solution path. As such a tree is explored, every node will have the same cost. If ties are broken in favor of shallower nodes, or equivalently nodes with lower g costs, the entire tree may be generated before a goal node is chosen for expansion. Thus, the asymptotic time complexity in this case would be $O(b^d)$, in spite of the fact that we have a perfect heuristic function.

A much better tie-breaking rule is to always break ties among nodes with the same $f(n) = g(n) + h(n)$ value in favor of nodes with the smallest $h(n)$ value, or equivalently the largest $g(n)$ value. This ensures that any tie will always be broken in favor of a goal node, which has $h(n) = 0$ by definition. In addition, this tie-breaking rule guarantees that the time complexity of A* with a perfect heuristic function will be $O(d)$. Ties will always be broken in favor of the nodes with the greatest $g(n)$ values, and hence the algorithm will march straight to a goal state, even if all the nodes have the same cost.

This tie-breaking rule makes sense for other reasons as well. To see why, assume that the actual cost to a goal is 10% greater than our estimated heuristic cost, for example. Now consider two nodes on the Open list, nodes a and b. Let $g(a) = 3$, $h(a) = 7$, $g(b) = 7$, and $h(b) = 3$. The total f cost of both nodes is equal ($3 + 7 = 7 + 3 = 10$). However, if we apply our error model, we find that $f^*(a) = g(a) + h^*(a) = 3 + h(a) + .1 \cdot h(a) = 3 + 7 + .7 = 10.7$, whereas $f^*(b) = g(b) + h^*(b) = 7 + h(b) + .1 \cdot h(b) = 7 + 3 + .3 = 10.3$. Thus, we expect the actual cost of reaching the goal through node b to be lower, and we should break ties in favor of the node with the lower h value.

3. ns for Node Expansion by A*

In number of nodes expanded by A*, we begin by ch of such nodes by their costs. To simplify these co that the heuristic function is both admissible and consistent. Given such a heuristic function, A* must expand all nodes whose total cost, $f(n) = g(n) + h(n)$, is less than c, the cost of an optimal solution [74]. Some nodes with the optimal solution cost may be expanded as well, since a goal node will not necessarily be on Open as soon as the lowest-cost node on Open has cost equal to c. The reason for this is that the last few nodes on an optimal path may have $f(n) = c$ until a goal node is chosen for expansion, at which point the algorithm terminates. In other words, $f(n) < c$ is a sufficient condition for A* to expand node n, and $f(n) \leq c$ is a necessary condition. For a worst-case analysis, we adopt the weaker necessary condition.

An easy way to understand the node expansion condition is that any search algorithm that guarantees optimal solutions must continue to expand every possible solution path, until its cost is guaranteed to exceed the cost of an optimal solution, lest it lead to a better solution.

Using these conditions, we will present two different analyses of the time complexity of A*. The first, which has a long historical tradition, is based on an abstract analytical model of the problem space and heuristic. Unfortunately, the assumptions of this model are rarely met in a real problem, and the characterization of the heuristic is very difficult to determine for a real heuristic. We then present a completely different analysis that applies much better to real heuristics for real problems, and can be used to predict with a high degree of accuracy the actual numbers of nodes generated in real problems.

3.7.4 Abstract Analysis

To perform the abstract analysis, we first introduce a model of the problem space, and then consider two different error assumptions on the accuracy of the heuristic, constant absolute error and constant relative error.

Abstract Anal

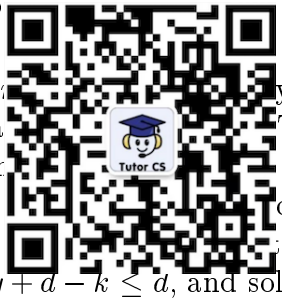
The model of th t is used for this analysis is quite simple, in fact t eal problems (see Figure 3.6). We assume that the tree, with no cycles. There is a uniform branchi g that every node has b children. Every edge or operator costs one unit to apply. There is a single goal node at depth d in the tree, shown as the leftmost leaf node g in the figure. The impact of this assumption is that once we diverge from the optimal path from the root to the goal, the only way to reach the goal is to backtrack until we rejoin the single optimal path.

Figure 3.6: Abstract Model for Analysis of A*

Constant Absolute Error

Our first assumption about the heuristic function is that it has constant absolute error. This means that it never underestimates the optimal cost of reaching a goal by more than a constant. In a worst-case analysis, we assume that it always underestimates by the maximum possible amount. Thus, $h(n) = h^*(n) - k$ for some constant k.

We need to determine how many nodes will be expanded under these assumptions. Consider an arbitrary node n in the tree. Assume without loss of generality that the path from the start node s to node n diverges from the path to the goal at node m. Let d be the distance from s to g, x be the distance from s to m, and y be the distance from m to n. Thus,



$g($ y way to reach the goal from node n is by going ba Thus, $h^*(n) = y+(d-x)$, and $h(n) = y+d-x-k$. Th $+ h(n) = x + y + y + d - x - k = 2y + d - k$.

ode n will be expanded if its total cost is less than or of an optimal solution. Writing the inequality $2y + d - k \leq d$, and solving for y gives us $y \leq k/2$. Thus, nodes off the optimal path will be expanded as long as their depth below the optimal path does not exceed $k/2$.

For any given node m on the optimal path, the number of nodes below it, but off the optimal path, whose depth below m doesn't exceed $k/2$ is $(b-1)b^{k/2-1}$. This is because there are $b-1$ branches immediately below node m that diverge from the optimal path, the remaining child being on the optimal path, and then there are b children below every subsequent node. The number of such nodes m on the optimal path from which we could diverge from it is $d-1$, since the last node on the optimal path is the goal itself. Thus, the total number of nodes whose total cost doesn't exceed d is $(d-1)(b-1)b^{k/2-1}$.

If we add the d nodes on the optimal path, this is exactly the set of nodes that are expanded by A* in the worst case. Since both b and k are constants, as d grows large this function is asymptotically $O(d)$. Thus, the asymptotic time complexity of A* using a heuristic that has constant absolute error is linear in the solution depth. Note that the complexity is actually an exponential function of the error k in the heuristic, and can be very large, but since k is a constant, and the base b is a constant, the entire exponential term is a constant.

Unfortunately, the assumption of constant absolute error, independent of the magnitude of the quantity being estimated, is unrealistic. Consider estimating or measuring distances in the real world. While the distance between atoms in a molecule can be measured with a very small absolute error, the absolute error in measurements of the distance between galaxies in the universe is certainly much greater.

Constant Relative Error

A much more realistic assumption, for measurements or heuristic functions, is constant relative error. This means that the absolute error is a bounded percentage of the quantity being estimated. For example,

if we assume that h never overestimates the actual distance to a goal by no more than α times the actual distance. The worst case model of our heuristic function is $h(n) = \alpha(y + d - x) = .9$ in this case. We now repeat the above analysis for this more general heuristic model, again referring to Figure 3.6.

The depth of node n, $g(n)$, is equal to $x + y$. The exact distance to the goal from node n, $h*(n)$, is equal to $y + (d - x)$. Thus, the heuristic value of node n, $h(n)$, is equal to $\alpha(y + d - x)$. Therefore, the total cost of node n, $f(n)$, is equal to $g(n) + h(n) = x + y + \alpha(y + d - x)$. Nodes will be expanded in the worst case as long as their total cost doesn't exceed the optimal solution cost d. Solving the equation $x + y + \alpha(y + d - x) \leq d$ for y, the depth below the optimal path, gives $y \leq (d - x)(1 - \alpha)/(1 + \alpha)$.

Consider first those nodes that diverge from the optimal path at the start state s. For these nodes, x is zero, and the corresponding depth condition is $y \leq d(1 - \alpha)/(1 + \alpha)$. The total number of such nodes is $(b - 1)b^{d(1-\alpha)/(1+\alpha)}$. For those nodes diverging from the optimal path at depth one ($x = 1$), the depth condition is $y \leq (d - 1)(1 - \alpha)/(1 + \alpha)$, and the total number of such nodes is $(b - 1)b^{(d-1)(1-\alpha)/(1+\alpha)}$. Since this term, and each succeeding term, is asymptotically less than the first term, the total number of nodes is $O((b - 1)b^{d(1-\alpha)/(1+\alpha)})$. This is asymptotically the number of nodes expanded by A* in the worst case, and hence the time complexity of A* is exponential in d. Once again, the complexity is an exponential function of the error in the heuristic, but in this case the error is a linear function of the optimal solution depth, and hence the overall complexity is exponential in the solution depth.

If we rewrite this expression as $O(b^{((1-\alpha)/(1+\alpha))d})$, and notice that $(1 - \alpha)/(1 + \alpha)$ is less than one, we can view the effect of the heuristic as reducing the effective branching factor of the search space from the brute-force branching factor b to the smaller heuristic branching factor $b^{(1-\alpha)/(1+\alpha)}$, and leaving the solution depth unchanged. Reducing the base of an exponential function reduces its asymptotic complexity, while leaving it in the exponential class. Thus, the effect of the heuristic function is not to eliminate the exponential combinatorial explosion, but to muffle it[1].

A[illegible]lt

Th[illegible]e worst-case results. Although we don't include th[illegible]known that under this model, if the error in the he[illegible]$(\phi(d))$, then the average-case time complexity of A*[illegible]ere e is the base of the natural logarithm, and k is a constant[74]. This is consistent with our worst-case results. If ϕ is a constant function, we get complexity that is linear in d, whereas if ϕ is a linear function of d, we get complexity that is exponential in d.

Limitations of this Model

There are several limitations of this model. The first is that the abstract model makes unrealistic assumptions. Most real problem spaces, such as Rubik's Cube or the sliding-tile puzzles, are graphs with cycles as opposed to trees. In such problem spaces, we can reach the goal from any other state without backtracking along the path to the given state.

The second limitation of this model is the characterization of the heuristic function in terms of the accuracy with which it estimates optimal solution costs, averaged over the entire problem space. In order to determine the accuracy of the heuristic on even a single state, we need to determine the optimal solution cost to a goal from that state, which generally requires a great deal of computation. Doing this for a large number of states is impractical for the size of problems we are trying to solve.

While this model gives us some insight into the asymptotic complexity of heuristic search, as a result of these two limitations, it cannot be used to predict the performance of A* on any real problem with a real heuristic. To do that requires a different approach[50].

3.7.5 Heuristic Analysis on Real Problems

Heuristic Distribution

Instead of characterizing a heuristic by its accuracy, for purposes of this analysis, we characterize a heuristic by the distribution of heuristic values over the nodes in the problem space. In other words, we need to know the number of states that have heuristic value 0, how many

states have heur umber with heuristic value 2, etc. Equivalently, we stribution by a set of parameters $D(h)$, each of w of the total states of the problem whose heuristic equal to h. We refer to this set of values as the *ove* he heuristic function. h can range from zero to infinity, but for all values of h greater than or equal to the maximum value of the heuristic, $D(h) = 1$. $D(h)$ can be defined as the probability that a state chosen randomly and uniformly from all states in the problem has heuristic value less than or equal to h.

Overall Distribution Table 3.1 shows the overall heuristic distribution for the Manhattan distance heuristic on the Five Puzzle, the 2×3 version of the sliding-tile puzzle. Manhattan distance is computed by counting the number of cells that each tile is displaced from its goal position, and summing these values for all tiles. The first column of Table 3.1 gives the heuristic value. The second column gives the number of states of the Five Puzzle with each heuristic value. The third column gives the total number of states with a given or smaller heuristic value, which is simply the cumulative sum of the values from the second column. The fourth column gives the overall heuristic distribution $D(h)$. These values are computed by dividing the value in the third column by 360, the total number of states in the problem space. The remaining columns will be explained below.

The overall distribution is easily obtained for almost any real heuristic. As we will see in Chapter 6, for heuristics implemented by table-lookup, or *pattern databases* [13, 45], the distribution can be determined exactly by scanning the table, as shown by the example above. Alternatively, for a heuristic computed by a function, such as Manhattan distance on large sliding-tile puzzles, we can randomly sample the problem space to estimate the overall distribution to any desired degree of accuracy. For heuristics that are the maximum of several different heuristics, we can compute the distribution of the complete heuristic from the distributions of the individual heuristics by assuming that the individual heuristic values are independent of one another. The resulting values will be accurate to the extent that the independence assumption is warranted.

h	N(h)	T(h)	D(h)	Corner	Side	Csum	Ssum	P(h)
0	1	1	.002778	1	0	1	0	.002695
1	2	3	.008333	1	1	2	1	.011028
2	1	4	.016667	1	2	3	3	.016915
3	4	8	.033333	5	1	8	4	.033333
4	30	42	.116667	25	5	33	9	.115424
5	58	100	.277778	38	20	71	29	.276701
6	61	161	.447222	38	23	109	52	.446808
7	58	219	.608333	41	17	150	69	.607340
8	60	279	.775000	44	16	194	85	.775707
9	48	327	.908333	31	17	225	102	.906594
10	24	351	.975000	11	13	236	115	.974503
11	8	359	.997222	4	4	240	119	.997057
12	1	360	1.000000	0	1	240	120	1.000000

Table 3.1: Manhattan Distance Heuristic Distributions for the Five Puzzle

The distribution of a heuristic function is not a measure of the accuracy of the heuristic. In particular, it says little about the correlation of heuristic values with actual costs, and doesn't require the computation of optimal solutions to any problem instances. The only connection between accuracy of a heuristic and its overall distribution is that given two admissible heuristics, the one whose distribution is shifted toward higher values will be more accurate than the one with lower values on average.

Equilibrium Distribution Although the overall distribution is the easiest to understand, the complexity of A* depends on a potentially different distribution. The *equilibrium distribution* $P(h)$ is defined as the probability that a node chosen randomly and uniformly among all nodes at a given depth of the brute-force search tree has heuristic value less than or equal to h, in the limit of large depth.

If all states of the problem occur with equal frequency at large depths in the brute-force search tree, then the equilibrium distribution is the same as the overall distribution. For example, this is the case with the Rubik's Cube search tree. In general, however, the equilibrium

distribution may ll distribution, such as in the Five Puzzle. The ov sumes that all states, and hence all possible blan ally likely. At deep levels in the tree, however, th position in more than 1/3 of the nodes, and in a ess than 2/3 of the nodes. In the limit of large depth, the frequency of side positions is .35321. Similarly, the frequency of corner positions is .64679 [50]. Thus, to compute the equilibrium distribution, we have to take the blank position into account.

The fifth column of Table 3.1, labelled "Corner", gives the number of states with the blank in a corner position (corner states), for each heuristic value. Similarly, the sixth column gives the number of states with the blank in a side position (side states), for each heuristic value. The seventh and eighth columns, labeled "Csum" and "Ssum", give the total numbers of corner and side states with heuristic values less than or equal to each particular heuristic value. The last column gives the equilibrium distribution $P(h)$. The probability $P(h)$ that the heuristic value of a node is less than or equal to h is the probability that it is a corner node, times the probability that its heuristic value is less than or equal to h, given that it is a corner node, plus the probability that it is a side node, times the probability that its heuristic value is less than or equal to h, given that it is a side node. For example, $P(1) = .64679 \cdot (2/240) + .35321 \cdot (1/120) = .011028$. This differs from the overall distribution $D(1) = .008333$.

The equilibrium distribution is not a property of a problem, but of a problem space. For example, including the parent of a node as one of its children affects the equilibrium heuristic distribution in the sliding-tile puzzles, by changing the equilibrium fractions of different types of states. Even when the equilibrium distribution differs from the overall distribution, it can still be computed directly from a pattern database, or by random sampling of the problem space, combined with the equilibrium fractions of different types of states, as illustrated above.

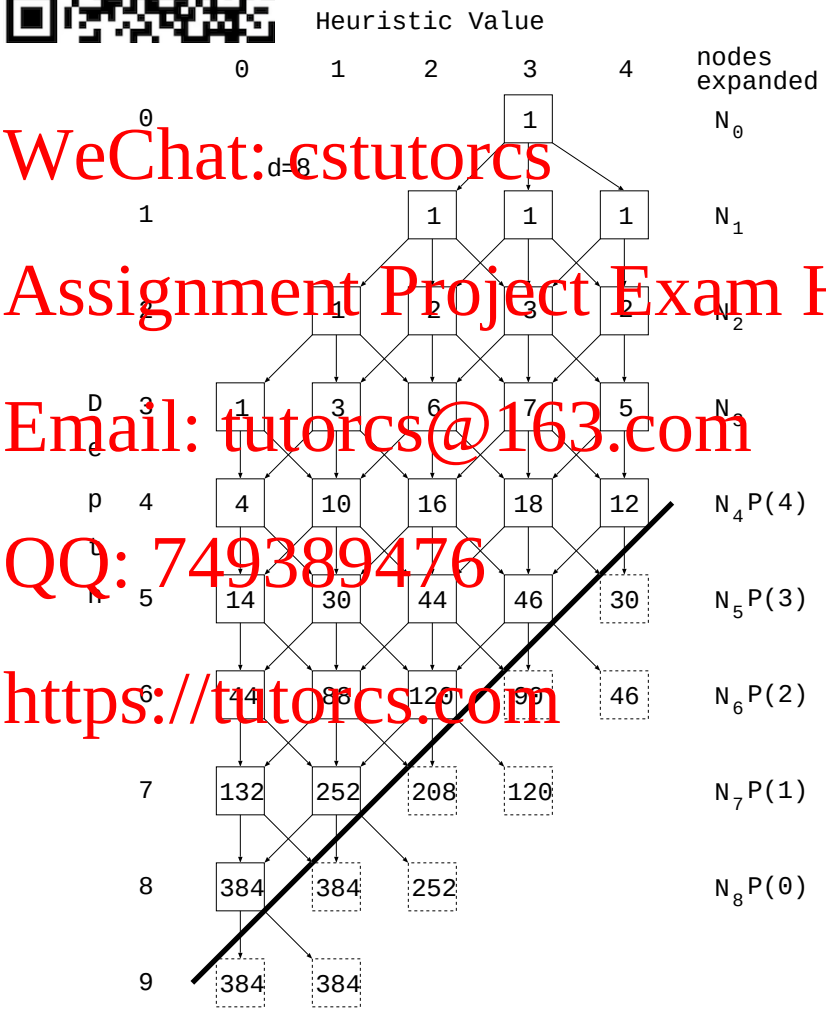


Figure 3.7: Sample Graph for Analysis of A*

3.7.6 An E[illegible] Tree

To provide some[illegible]ur main result, Figure 3.7 shows a schematic rep[illegible]rch tree generated by A* on an abstract problem[illegible] edges have unit cost. The vertical axis represents the depth [illegible] below the initial state, which is also its g value, and the horizontal axis represents the heuristic value of a node. Each box represents a set of nodes at the same depth with the same heuristic value, with the number of such nodes included therein. The arrows represent the relationship between parent and child node sets. These particular numbers were generated by assuming that each node generates one child for each arrow leaving its box. For example, there are 7 nodes at depth 3 with heuristic value 3, 2 whose parents have heuristic value 2, 3 whose parents have heuristic value 3, and 2 whose parents have heuristic value 4. In this example, the maximum value of the heuristic is 4, and the heuristic value of the initial state is 3.

An assumption of our analysis is that the heuristic is consistent. Because of this, and since all edges have unit cost in this example, the heuristic value of a child must be at least the heuristic value of its parent, minus one. We assume a goal state at depth 8 moves for this problem instance. Solid boxes represent sets of "fertile" nodes that will be expanded, while dotted boxes represent sets of "sterile" nodes that will not be expanded, because their total cost, $f(n) = g(n) + h(n)$ exceeds the cutoff threshold of 8. The thick diagonal line separates the fertile node sets from the sterile node sets.

Nodes Expanded as a Function of Depth

The values at the far right of Figure 3.7 show the number of nodes expanded at each depth, which is the number of fertile nodes at that depth. N_i is the number of nodes in the brute-force search tree at depth i, and $P(h)$ is the equilibrium heuristic distribution. The number of nodes generated is simply the branching factor times the number expanded.

Consider the graph from top to bottom. There is a root node at depth 0, which generates N_1 children. These nodes collectively generate

N_2 th 2. Since the goal is at depth 8 moves, in the wc n whose total cost $f(n) = g(n) + h(n) \leq 8$ will be is the maximum heuristic value, all nodes down to l be expanded, as in a brute-force search. Down to nber of nodes expanded at depth d will be N_d, the same as in a brute-force search. Since 4 is the maximum heuristic value, $P(4) = 1$, and hence $N_4 P(4) = N_4$.

The nodes expanded at depth 5 are the fertile nodes, or those for which $f(n) = g(n) + h(n) = 5 + h(n) \leq 8$, or $h(n) \leq 3$. At sufficiently large depths, the distribution of heuristic values converges to the equilibrium distribution. If we assume that the heuristic distribution at depth 5 equals the equilibrium distribution, the fraction of nodes at depth 5 with $h(n) \leq 3$ is $P(3)$. Since all nodes at depth 4 are expanded, the total number of nodes at depth 5 is N_5, and the number of fertile nodes at depth 5 is $N_5 P(3)$.

Although there exist nodes at depth 6 with heuristic values from 0 to 4, their distribution is not equal to the equilibrium distribution. In particular, nodes with heuristic values 3 and 4, shown in dotted boxes, are underrepresented compared to the equilibrium distribution, because such nodes are normally generated by parents with heuristic values from 2 to 4. At depth 5, however, the nodes with heuristic value 4 are sterile, and have no offspring at depth 6, reducing the number of nodes at depth 6 with heuristic values 3 and 4.

The number of nodes at depth 6 with $h(n) \leq 2$ is completely unaffected by this pruning, however, since their parents are nodes at depth 5 with $h(n) \leq 3$, all of which are fertile. In other words, the number of nodes at depth 6 with $h(n) \leq 2$ is exactly the same as in the brute-force search tree, or $N_6 P(2)$. These are the fertile nodes at depth 6.

Due to consistency of the heuristic function, all possible parents of fertile nodes are themselves fertile. Thus, the number of nodes whose heuristic values place them to the left of the diagonal line in Figure 3.7 is exactly the same as it would be in the brute-force search tree. In other words, the pruning of the tree due to the heuristic function has no effect on the fertile nodes, even though the distribution of the sterile nodes is affected. This is the key idea behind our analysis. If the heuristic were inconsistent, then the distribution of fertile nodes would change at every level where pruning occurred, making the analysis much more

complex.

When all edg e number of fertile nodes at depth i is $N_i P(d-i)$, mber of nodes in the brute-force search tree at de ff depth, and P is the equilibrium heuristic distribu ber of nodes expanded by A* in a search to depth d is

$$\sum_{i=0}^{d} N_i P(d-i)$$

.

3.7.7 General Result

Here we make our assumptions explicit, and state and prove our main theoretical result. First, we assume a minimum edge cost, and divide all costs by this value, normalizing it to one, and express all costs as multiples of the minimum edge cost. We allow operators with different costs, and replace the depth of a node by $g(n)$, the sum of the edge costs from the root to the node. Let N_i be the number of nodes n in the brute-force search tree with $g(n) = i$.

Next, we assume the heuristic returns an integer multiple of the minimum edge cost. Given an admissible non-integer valued heuristic, we round up to the next larger integer, preserving admissibility. We also assume that the heuristic is consistent, meaning that for any two nodes n and m, $h(n) \leq c(n,m) + h(m)$, where $c(n,m)$ is the cost of an optimal path from n to m.

Given these assumptions, our task is to determine $E(N, c, P)$, the number of nodes n for which $f(n) = g(n) + h(n) \leq c$, given a problem-space tree with N_i nodes of cost i, with a heuristic characterized by the equilibrium distribution $P(x)$. This is the number of nodes expanded by A* in a search for a solution of cost c, in the worst case.

Theorem 3.3 *In the limit of large c,*

$$E(N, c, P) = \sum_{i=0}^{c} N_i P(c-i)$$

Proof: $E(N, c, P)$ is the number of nodes n for which $f(n) = g(n) + h(n) \leq c$. Consider the nodes n for which $g(n) = i$, which is the set of

no brute-force search tree. There are N_i such nodes. Th that will be expanded by A* searching to cost c ar $n) = g(n) + h(n) = i + h(n) \leq c$, or $h(n) \leq c - i$. By the limit of large i, the number of such nodes *in th tree* is $N_i P(c - i)$. It remains to show that all these nodes in the brute-force search tree are also in the tree generated by A*.

Consider an ancestor node m of such a node n. Since m is an ancestor of n, there is only one path between them in the tree, and $g(n) = i = g(m) + C(m, n)$, where $C(m, n)$ is the cost of the path from node m to node n. Since $f(m) = g(m) + h(m)$, and $g(m) = i - C(m, n)$, $f(m) = i - C(m, n) + h(m)$. Since the heuristic is consistent $h(m) \leq c(m, n) + h(n)$, where $c(m, n)$ is the cost of an optimal path from m to n in the problem graph. Since $C(m, n) \geq c(m, n)$, $h(m) \leq C(m, n) + h(n)$. Thus, $f(m) \leq i - C(m, n) + C(m, n) + h(n)$, or $f(m) \leq i + h(n)$. Since $h(n) \leq c - i$, $f(m) \leq i + c - i$, or $f(m) \leq c$. This implies that node m is fertile and will be expanded during the search. Since all ancestors of node n are fertile and will be expanded, node n must eventually be generated. Therefore, all nodes n in the brute-force search tree for which $f(n) = g(n) + h(n) \leq c$ are also in the tree generated by A*. Since there can't be any nodes in the A* tree that are not in the brute-force search tree, the number of such nodes at level i in the A* tree is $N_i P(c - i)$. Therefore, the total number of nodes expanded by A* in searching to cost c, is

$$E(N, c, P) = \sum_{i=0}^{c} N_i P(c - i) \square$$

.

The Heuristic Branching Factor

The *heuristic branching factor* is the ratio of the number of nodes expanded in a search to cost c, divided by the nodes expanded in a search to cost $c - 1$, or $E(N, c, P)/E(N, c - 1, P)$, where 1 is the normalized minimum edge cost.

Assume that the size of the brute-force search tree grows exponentially with cost, or $N_i = b^i$, where b is the brute-force branching factor.

In that case, the [illegible] factor $E(N,c,P)/E(N,c-1,P)$ is

$$\frac{\sum_{i=0}^{c} b^i P(c-i)}{\sum_{i=0}^{c-1} b^i P(c-1-i)} = \frac{b^0 P(c) + b^1 P(c-1) + b^2 P(c-2) + \cdots + b^c P(0)}{b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0)}$$

The first term of the numerator, $b^0 P(c)$, is less than or equal to one, and can be dropped without significantly affecting the ratio. Factoring b out of the remaining numerator gives

$$\frac{b(b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0))}{b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0)} = b$$

Thus, if the brute-force tree grows exponentially with branching factor b, then the running time of A* searches to successively greater depths also grows by a factor of b. In other words, the heuristic branching factor is the same as the brute-force branching factor.

Previous analyses, based on different assumptions, predicted that the effect of a heuristic function is to reduce search complexity from $O(b^c)$ to $O(a^c)$, where $a < b$, reducing the effective branching factor [74]. Our analysis, however, shows that on an exponential tree, the effect of a heuristic is to reduce search complexity from $O(b^c)$ to $O(b^{c-k})$, for some constant k, reducing the effective depth of search instead. To see this intuitively, consider a problem space where every state has heuristic value k. In that case, a heuristic search to depth c would be the same as a brute-force search to depth $c-k$.

Comparison with Experimental Data

We tested our analysis by predicting the nodes generated by iterative-deepening-A* (IDA*), a linear-space version of A* described in the next chapter, on Rubik's Cube and sliding-tile puzzles, using well-known heuristics. Since all operators have unit cost, the $g(n)$ cost of a node n is its depth d. For N_d, we used the exact numbers of nodes at level d, which were computed in time linear in the depth, by expanding a set of recurrence relations governing the generation of different types of nodes[50].

D	l	Problems	Experimental	Error
	510	1000	1,501	.596%
	169	1000	20,151	.089%
	229	1000	270,396	.433%
	300	100	3,564,495	.815%
14	47,971,732	100	47,916,699	.115%
15	640,349,193	100	642,403,155	.321%
16	8,547,681,506	100	8,599,849,255	.610%
17	114,098,453,567	25	114,773,120,996	.591%

Table 3.2: Nodes Generated by IDA* on Rubik's Cube

Rubik's Cube We first tried to predict results previously obtained on Rubik's Cube[45]. We use the same problem space described in Chapter 1. The median optimal solution depth is 18 moves.

The heuristic we used is the maximum of three different pattern databases[13], described in Chapter 6. It is admissible and consistent, with a maximum value of 11 moves and a mean value of 8.898 moves. The distribution of the individual heuristics was calculated exactly by scanning the databases, then the three heuristics were assumed to be independent to calculate the distribution of the combined heuristic. In this case, the equilibrium distribution is the same as the overall distribution. We ignored goal states, completing the search to various depths.

In Table 3.2, the left-most column shows the cutoff depth, the next column gives the node generations predicted by our theoretical analysis, the next column indicates the number of problem instances run, the next column displays the average number of nodes generated by IDA* for a single iteration to the given depth, and the last column shows the error between the theoretical prediction and the experimental results.

The theory predicts the data to within 1% accuracy in every case. The sources of error include the limited number of problem instances, the independence assumption among the three heuristics, and the difference between the actual heuristic distribution at a given depth and the equilibrium distribution.

Depth	Theore		xperimental	Error
20			393	0.0%
21			657	0.0%
22			1,185	0.0%
23			1,977	0.0%
24	3,561	181,440	3,561	0.0%
25	5,936	181,440	5,936	0.0%
26	10,686	181,440	10,686	0.0%
27	17,815	181,440	17,815	0.0%
28	32,072	181,440	32,072	0.0%
29	53,450	181,440	53,450	0.0%
30	96,207	181,440	96,207	0.0%
31	160,357	181,440	160,357	0.0%

Table 3.3: Nodes Expanded by IDA* on the Eight Puzzle

Eight Puzzle We ran a similar experiment on the Eight Puzzle, using the Manhattan distance heuristic function. Its maximum value is 22 moves, and its mean value is 14 moves. The average optimal solution length is 22 moves, and the maximum is 31 moves, assuming the blank is in a corner in the goal position. Since the Eight Puzzle contains only 181,440 solvable states, the heuristic distributions were computed exactly. Three different distributions were used, depending on whether the blank is in a center, corner, or side position. While we could have used nine distributions, one for each blank position, the four distributions where the blank is in a corner are all the same, as are the four distributions where the blank is in a side position, due to symmetry. The exact number of nodes of each type at each depth of the brute-force tree was computed exactly, based on the initial blank position.

Table 3.3 shows a comparison of the number of node expansions predicted by our theoretical analysis, to the number of nodes expanded by a single iteration of IDA* to various depths, ignoring goal states. Each data point is the average of all 181,440 problem instances. Since the average numbers of node expansions, the size of the brute-force tree, and the heuristic distributions are all exact, the model predicts

D	Problems	Experimental	Error
	100,000	41,973	1.65%
	100,000	91,495	0.66%
	100,000	191,219	1.27%
	100,000	415,490	0.72%
44	100,000	870,440	0.96%
45	100,000	1,886,363	0.75%
46	100,000	3,959,729	0.73%
47	100,000	8,562,824	0.77%
48	100,000	18,003,959	0.55%
49	100,000	38,864,269	0.77%
50	100,000	81,826,008	0.41%

Table 3.4: Nodes Expanded by IDA* on the Fifteen Puzzle

the experimental data exactly, to multiple decimal places, verifying that we have accounted for all the relevant factors.

Fifteen Puzzle We ran a similar experiment on the Fifteen Puzzle, also using Manhattan distance. The mean heuristic value is about 37 moves, and the maximum is 62 moves. The average optimal solution length is 52.6 moves. Since this puzzle contains over 10^{13} solvable states, we used a random sample of ten billion solvable states to approximate the heuristic distributions. Three different distributions were used, one for the blank in a middle, corner, or side position. The number of nodes of each type at each depth was computed exactly, based on the initial position of the blank.

Table 3.4 is similar to Table 3.3. Each line is the average of 100,000 random solvable problem instances. Despite over ten orders of magnitude variation in the nodes expanded in individual problem instances, the average values agree with the theoretical prediction to within 1% in most cases. The ratio between the experimental number of node expansions at the last two depths is 2.105, compared to the theoretical branching factor of 2.130.

3.8 Time Complexity of A*

Both types of a[illegible] that the time complexity of A* depends on the [illegible]stic function. The more accurate the heuristic fun[illegible]ater its values are, assuming they are admissible, the more efficient A* is in finding an optimal solution. An obvious question to ask, however, is for a given heuristic function, does A* make the best use of it? For example, A* is a best-first search, and uses the particular cost function $f(n) = g(n) + h(n)$. Are these the right choices, or in other words, is the time complexity of A* optimal for a given heuristic function?

The answer is yes, under certain conditions. The first is that the heuristic function must be consistent in addition to admissible. The second is that we restrict our attention to algorithms that are guaranteed to find an optimal solution. The way we state this result formally is the following theorem, first proved in [15].

Theorem 3.4 *For a given consistent heuristic function, every admissible algorithm must expand all nodes surely expanded by A*.*

Since the heuristic function is consistent, it must also be admissible. By an admissible algorithm, we mean one that is guaranteed to find an optimal solution, given an admissible heuristic function, if a solution exists. Finally, by the nodes surely expanded by A*, we mean all those node for which $f(n) < c$, where c is the optimal solution cost.

The proof of this theorem is by contradiction. We assume the converse of the theorem and then derive a contradiction. The converse of our theorem is the following statement: there exists an admissible algorithm B, a problem P, a consistent heuristic function h, and a node m, such that node m is not expanded by algorithm B on problem P with heuristic function h, but node m is surely expanded by A*, meaning that $f(m) = g(m) + h(m) < c$.

To derive a contradiction, we construct a new problem P' that is identical to problem P, except for the addition of a single new edge leaving node m, which leads to a new goal node z. Let the cost of the edge from node m to node z be $h(m)$, the heuristic value of node m in problem P, or $c'(m, z) = h(m)$. We use $c'(m, z)$ here to denote the actual cost from m to z in problem P'.

cost of an optimal path from the initial state to th s $g(m) + c'(m, z) = g(m) + h(m) = f(m)$, since ev go through node m. Since $f(m) < c$, the cost of an problem P, the optimal solution to problem P' is art through node m to goal z.

Now consider what happens when we apply algorithm B to problem P'. Since by assumption, algorithm B never expands node m on problem P, it must not expand node m on problem P' either, since they are identical except for the new edge leaving node m, which algorithm B never sees since it doesn't expand node m. Thus, algorithm B must fail to find the optimal solution to problem P'. This violates our assumption that algorithm B is an admissible search algorithm.

The only caveat is that an admissible search algorithm is only required to return an optimal solution if one exists and it is given an admissible heuristic function. If the heuristic function is non-admissible, the algorithm is off the hook. Thus, what is left to show is that the original heuristic function h is admissible in problem P' as well as in problem P.

The only way that the addition of the new edge from node m to the new goal node z could make the original heuristic function non-admissible in problem P' is if the distance to the new goal node z from some arbitrary node n is now less than $h(n)$. Any such path from n to z must pass through node m. Thus, we simply have to verify that in problem P', $h(n) \leq c(n, m) + c'(m, z)$. This is easily done. By definition, $c'(m, z) = h(m)$. Since by assumption h is consistent for problem P, $h(n) \leq c(n, m) + h(m)$, for all nodes n and m. Therefore, in problem P', $h(n) \leq c(n, m) + c'(m, z)$, verifying that h is admissible in problem P' as well, despite the addition of the new edge and goal state. Thus, we have shown that for a given consistent heuristic function, every admissible algorithm must expand all nodes surely expanded by A*. □

3.9 Space Complexity of Best-First Search

The main drawback of best-first search is its space complexity, since it stores all the nodes it generates in either the Open list or the Closed

list. Thus, its asymptotic space complexity is the same as its asymptotic time complexity, assuming that a node can be stored in a constant amount of space. This means that on current computers it will typically exhaust memory in minutes, thus terminating the algorithm. IDA* can easily find optimal solutions to the Eight Puzzle, which has only about 10^5 states, but quickly runs out of memory on many Fifteen Puzzle problem instances. We introduce algorithms that address this limitation in the next two sections and the next chapter.

3.10 Frontier Search

3.10.1 Overview

Best-first search stores both a Closed list of expanded nodes, and an Open list of nodes that have been generated, but not yet expanded. The key idea of *frontier search* is to reduce the memory requirements of best-first search by storing only the Open list and not the Closed list. The term frontier search comes from the fact that the Open list represents the boundary or frontier of the explored region of the problem space. By storing with each Open node which of its neighbors have already been generated, and maintaining a search boundary with no gaps in it, we prevent Closed nodes from being regenerated.

For some applications, such as generating pattern databases, described in Chapter 6, we are only interested in the number of moves needed to reach a goal state. For other problems we need the actual solution path. Since frontier search doesn't store the interior nodes of the search, regenerating the solution path requires additional work. It can be done, for example, by finding the midpoint of a solution path, and then recursively solving the problems of finding a path from the initial state to the midpoint, and from the midpoint to a goal state. This can be done with either a unidirectional or bidirectional search.

At first, we ignore the reconstruction of the solution path, and describe the simplest version of frontier search, which applies to undirected problem-space graphs. Next we consider reconstructing the solution path, first using a bidirectional search and then using a unidirec-

tic n consider several applications of the technique, inc search and A* on the sliding-tile puzzles and the 4-p problem. This work is described more fully in [53

3.10.2 Frontier Search on Undirected Graphs

While frontier search applies to both directed and undirected graphs, we only consider the simpler case here, which is undirected graphs. With each Open node n we associate a state representation, and $g(n)$ and possibly $h(n)$ values. In addition, we store with each node a vector of *used operator* bits, one bit for each legal operator. This bit indicates whether the neighboring state reached via that operator has already been generated or not. For example, for a rectangular-grid problem space, where each node has four neighbors, one in each compass direction, we would store four bits with each node, one for each direction. When we expand a parent node, generating its children, in each child node we mark the operator that would regenerate the parent from that child as used.

For example, consider the graph fragment shown in Figure 3.8, with 5 nodes marked $C(orner)$, $E(ast)$, $S(outh)$, $S(outh)E(ast)$, $F(ar)E(ast)$, and $F(ar)S(outh)$. The algorithm begins with just the start node on the open list, say node C in this case, with no operators marked as used. Expanding node C will generate nodes E and S, and node C will be deleted from memory. The West operator from node E will be marked as used, as will the North operator from node S, since those operators would regenerate node C. Nodes E and S will be evaluated by the cost function, and placed on the open list.

Assume that node E has lower cost and is expanded next. It will generate nodes SE and FE, but not node C, since its West operator is used. Node E is then deleted. The North operator from node SE, and the West operator from node FE will be marked as used. Nodes SE and FE will be evaluated, and placed on the open list. Assume that node S is expanded next, and then deleted. It will generate node FS, with its North operator marked as used. It will also generate another node corresponding to state SE, with its West operator marked as used. It will not generate node C, since its North operator was marked

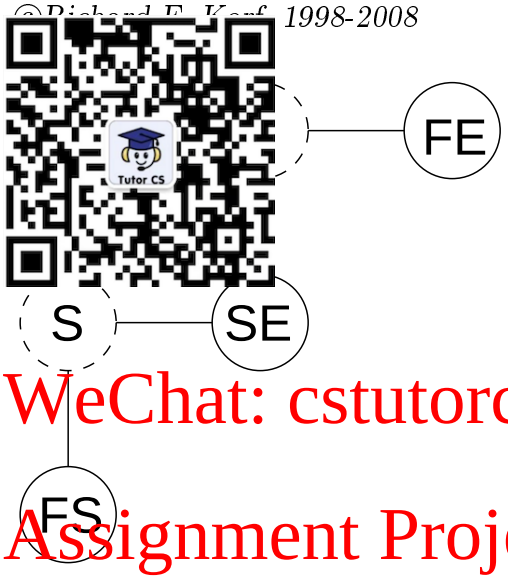


Figure 3.8: Frontier search example on undirected graph

as used.

Once the second node representing state *SE* is generated, the first such node will be found on the open list. Of these two nodes, only the one reached via the lower-cost path will be saved. Both the North and West operators of this node will be marked as used. In general, when duplicate nodes representing the same state are generated, the operators marked as used in the saved copy are the union of the used operators of the individual nodes.

Figure 3.8 represents the state of memory at this point in the search, where the solid nodes are stored, the solid edges represent used operators, and the dotted elements are no longer stored. The algorithm terminates when a goal node is chosen for expansion.

3.10.3 Consistent Heuristic Requirement

Frontier search deletes a node once it is expanded, and never expands a node more than once. Thus, to guarantee the admissibility of frontier-A*, we can only expand a node when we've found a lowest-cost path to it. This requires that any heuristic function be consistent, as described in section 3.6.

3. ing the Solution Path

In tation of best-first search, each node on the Open an a pointer to its parent node. Once the search is co pointers back from the goal state to the initial sta solution path. Since frontier search doesn't save the Closed list, we need an alternative method of recovering the solution path. We describe below two different methods of doing this, one based on bidirectional search and the other based on unidirectional search.

Divide-and-Conquer Bidirectional Frontier Search

If we perform a bidirectional frontier search, starting from both the initial and goal states, when the two search frontiers meet we have a middle node on a solution path from the initial state to the goal state. We can then divide our problem into two recursive subproblems: find a path from the initial state to the middle node, and find a path from the middle node to the goal state. The resulting algorithm is been called *divide-and-conquer bidirectional frontier search.*

If we want optimal solutions, we have to be careful about the terminating condition. For a unidirectional best-first search with a non-overestimating cost function, we can terminate the search whenever a goal node is chosen for expansion. For a bidirectional search, every time we reach a node from both directions we have found a solution path, and we keep track of the lowest-cost solution path found so far. For bidirectional Dijkstra's algorithm, where the cost of a node n is simply $g(n)$ or the cost of the path to node n, we can terminate when the cost of the lowest-cost complete solution found so far is less than or equal to the sum of the costs of the lowest-cost nodes in each direction. The reason is that any additional solutions found must go through an open node in each direction, and will have a cost at least as large as the sum of the minimum g costs in each direction.

For bidirectional A*, where the cost of a node is $f(n) = g(n) + h(n)$, we can terminate the search when the cost of the lowest-cost solution found so far is less than or equal to the minimum f cost of any Open node in either direction. The reason is that with a non-overestimating heuristic function, the lowest f cost in either direction is a lower bound

on any solutions found. Alternatively, we can terminate when the cost of the solution found so far is less than or equal to the sum of the costs in each direction, for the same reason given for the termination condition above. If the heuristic function is relatively accurate, the terminating condition described above is likely to apply first. If the two searches are relatively balanced, and there is relatively little overlap between them, then each of the two searches may do as much work as a unidirectional heuristic search, causing bidirectional search to generate up to twice as many nodes as unidirectional search.

There are also several other drawbacks of bidirectional search. One is that bidirectional search is more complex. In addition to performing forward and backward searches, nodes on both searches must be compared, and the terminating condition is more complex. A consequence is that bidirectional search usually requires more time per node generation. Another factor is that by maintaining two search frontiers, bidirectional frontier search could use up to twice as much memory as unidirectional search. For all these reasons, we'd like a unidirectional version of frontier search.

Divide-and-Conquer Unidirectional Frontier Search

Consider a problem space with a well-defined geometric structure, such as a rectangular grid, and assume we are searching from one corner to the opposite corner. If we draw a midline dividing the rectangle into two equal rectangles, any solution path must cross this midline at some point. Furthermore, the point at which a solution crosses the midline is likely to be near the midpoint of the solution path itself.

Given such a midline, we perform a unidirectional frontier search until we reach the midline. With each node that is beyond the midline, we store the coordinates of its ancestor on the midline. When we merge duplicate nodes that represent the same state, we store the midpoint on a shortest path to the given state. Once the search chooses a goal node for expansion, the midpoint associated with that goal node is approximately half way along the optimal solution path from the initial state to the goal state. The algorithm then recursively computes an optimal path from the initial state to the midpoint, and from the midpoint to

th

etric problems, we may be able to identify the mi . For example, in the Towers of Hanoi problem, th olution path is when the largest disc is moved to th

In a problem space without a clearly defined midline, such as the sliding-tile puzzles, we can use other techniques to determine that a node is approximately half way between the initial and goal states. For example, a node n for which $g(n) = h(n)$ is likely to be approximately half way to the goal. These nodes could then serve as midpoints for the unidirectional search.

Note that there is no need to continue recursive calls all the way to the point where the initial and goal states are identical. Rather, once the resulting problem becomes small enough that we have sufficient memory to store both the open and closed lists, we could call a standard best-first algorithm to solve the remaining subproblem.

In general, the time and space complexity of frontier search is dominated by the top-level call, and the cost of recovering the solution path is not very significant.

3.10.5 Theoretical Properties

Frontier search achieves the goals it was designed for. In particular, it never regenerates a node that has already been expanded, and it faithfully simulates a standard best-first search with a consistent cost function. We focus here on the first pass of the algorithm, from the original initial state to the original goal state, and ignore the recursive calls to reconstruct the solution path. For brevity, we omit the proofs of these theorems.

Theorem 3.5 *In a single pass, frontier search never regenerates a node that has already been expanded.*

Next, we show that even without a closed list, if the heuristic function is consistent, frontier search behaves the same as best-first search.

Theorem 3.6 *With the same consistent heuristic function, or no heuristic function, and the same tie-breaking rule, the first pass of frontier search expands the same nodes in the same order as best-first search.*

3.10.6 Exp[illegible] Frontier-A* on the Fifteen

In addition to b[illegible] frontier search can also be applied to any best-first [illegible]e A* algorithm[34]. We call this combination frontier-A*, and applied it to the Fifteen Puzzle. It is well-known that on this problem A* is limited by the amount of memory available. For example, on a machine with a gigabyte of memory, we can store about 30 million nodes. A* with the Manhattan distance heuristic function is only able to solve 79 of the 100 random problems in [41] before running out of memory. Using frontier-A* with the same amount of memory allows us to solve 94 of the 100 problems. Using disk storage as described below allows us to easily solve all these problems. The fraction of memory saved is the number of nodes expanded, divided by the number of nodes generated, which is about 57% for all 100 problems. In other words, frontier search more than doubles the effective memory on this problem.

While frontier search is a general method for reducing the storage required by best-first searches, often with little or no time penalty, it is still limited by the amount of storage available. In the next section, we consider how to dramatically increase the amount of available storage.

3.11 Storing Nodes on Disk

The limiting resource in any best-first search is memory to store the search nodes. Recently, there have been dramatic increases in the capacity of magnetic disks, and corresponding reductions in the cost per byte. Currently, one terabyte disks are available for about $300. This is about 2.5 orders of magnitude cheaper than semiconductor memory. This suggests the idea of storing search nodes on disk rather than in memory. The main problem, however, is that the latency to access a single byte on magnetic disk can be up to ten milliseconds. Thus, the primary challenge is to redesign our algorithms so that all data accesses are sequential. A complete description of these techniques and experiments can be found in [46].

For simplicity, consider a breadth-first search. The standard imple-

me first search uses a first-in first-out (FIFO) queue of moved from the head of the queue, and appended to ily implemented on disk. The simplest way is to ha a read pointer at the head of the queue, and a wr il. Note that all reads and writes are sequential. If we don't need to keep nodes after they are read, it is more space-efficient to use two files, one for the head, and another for the tail. Nodes are read from the head file, and written to the tail file. Once the head file is exhausted, it is deleted, the tail file becomes the new head file, a new tail file is created, and the search continues.

The remaining difficulty is how to detect duplicate nodes. In memory, in addition to the queue, nodes are normally stored in a hash table. As each node is generated, it it looked up in the hash table, and if it is a duplicate of a node already generated, it is simply deleted. Unfortunately, we can't implement a hash table on disk. The reason is that a hash table is designed to spread its entries randomly for efficiency, but accessing nodes at random on a disk is prohibitively expensive.

3.11.1 Delayed Duplicate Detection

A solution to this problem is that instead of checking each node for a duplicate as soon as it is generated, we delay the detection of duplicate nodes, and then perform this step in large batches using sequential access to the data. This is called *delayed duplicate detection*[51] (DDD). We use the queue data structure on disk as described above, without checking for duplicate nodes, resulting in multiple copies of the same node in the queue. Periodically, such as at the end of each level or *iteration* of the breadth-first search, we merge the duplicate nodes. One way to do this is to sort the nodes in the queue by their state representation, using any of a number of algorithms for sorting disk files. Sorting will bring any duplicate nodes to adjacent positions in the queue. Then, in a single linear scan, we can copy the queue to another file, merging any duplicate nodes together. The drawback of this approach is that it requires logarithmic time to perform the sorting.

A more efficient algorithm uses two orthogonal hashing functions. As child nodes are generated, they are written to different files depending on the value of a first hash function. Thus, any duplicate nodes will

be mapped to th ey will have the same hash value. Then, at the end example, one file at a time is read into a hash tabl second hash function. Duplicate nodes will map n, and can be merged. This eliminates the logari g, and runs in time linear in the number of nodes.

3.11.2 Combining DDD with Frontier Search

Delayed duplicate detection can also be combined with frontier search, further reducing the number of nodes that must be stored on disk. If all cycles in the problem-space graph are even in length, then we only need to store the nodes in the current frontier. For example, in the sliding-tile puzzles, the blank must move an even number of times to return to the same location, and hence all cycles are of even length.

If there are odd-length cycles, such as in the Towers of Hanoi problem for example, then there is an additional complication. In particular, consider two nodes that are at the same depth in a breadth-first search, and are connected by an operator. Assume that each was generated by another operator. In standard breadth-first frontier search, as soon as the first of these nodes is expanded, it will generate the other via the operator between them, and the duplicate node will be dectected immediately. With delayed duplicate detection, however, each node can be expanded and deleted, since they are closed, generating new copies of each other, without ever detecting the duplicates. Effectively, the delayed duplicate detection allows two parts of the frontier to pass through each other, leaking back into the interior. The solution to this problem, for graphs with odd-length cycles, is to keep two levels of the search on disk, and check for duplicates among both levels. Once all duplicates have been merged, the previous level of the search can be deleted.

3.11.3 Interleaving Expansion and Merging

By using disk storage instead of memory, we dramatically increase the size of problems we can search before exhausting the available storage. Storage is still the limiting resource, however. In order to minimize

th of storage needed at any point, we would like to eli des as soon as possible. For example, consider th . Assume that the hash function we use to map no d on the position of the blank. Thus, all the ch file will have the blank in the same position. Furthermore, the parent nodes that generated those children can only have the blank in at most four other positions, and hence must be in at most four other files. Once all those parent files have had their nodes expanded, no more children will be mapped to the given child file, and we can merge the duplicates in that child file, reducing the amount of disk storage required.

3.11.4 Parallel Delayed Duplicate Detection

By using disk storage for nodes, particularly when combined with frontier search, we can implement searches that run for weeks or months at a time without exhausting the available storage. Thus, the running time becomes a prime consideration. Most workstations are available with multiple processors, and most modern processor chips now have multiple cores, which are essentially multiple processors. Thus, parallel processing is readily available and can be used to speed up such long searches. Since delayed duplicate detection is disk I/O intensive, performance with even a single processor will be improved by having multiple threads, so that some threads can still execute when others are blocked waiting for disk I/O.

The natural way to parallelize hash-based delayed duplicate detection is to assign the expansion of the nodes in a given parent file, or the merging of a given child file, to separate processes or threads. This minimizes any potential write conflicts between different threads. Even if two different threads are expanding parent files that generate children in the same child file, the operating system will serialize mulitple writes to the same file, and the order in which they appear in the child file is irrelevant. In order to minimize the amount of disk storage required, we give priority to merging child files, and only expand parent files when no child files are ready to be merged.

3.12 Exp[illegible]ombining Breadth-First Fro[illegible] with DDD

Here we describ[illegible]ts that combine parallel breadth-first frontier sea[illegible]plicate detection on disk, applied to the sliding-tile puzzles and the four-peg Towers of Hanoi problem.

3.12.1 Complete Breadth-First Frontier Search

The simplest case of frontier search is breadth-first frontier search. At first, frontier search would not appear to be very useful for exponential problem spaces. The reason is that in an exponentially growing space with a branching factor greater than two, the frontier of the search space will be larger than the interior, resulting in limited savings in space. When performing a complete breadth-first search of a finite problem space, however, frontier search is indeed useful. The reason is that while the size of the space may grow exponentially initially, eventually the number of nodes at a given depth reaches a maximum value, and then decreases until all nodes have been generated. The maximum number of nodes at any depth in a problem space is called the *width* of the problem space, and is often significantly less than the total number of nodes. The space complexity of a complete breadth-first frontier search is the width of the problem space, rather than the total number of nodes, which is the space complexity of standard breadth-first search.

Sliding-Tile Puzzles

In 1967, Schofield published a paper entitled, "Complete solution of the eight puzzle" [88]. It was based on a complete breadth-first search of the eight puzzle problem space, which contains $9!/2 = 181,440$ states, and gives the number of states at each depth. Using breadth-first frontier search we performed complete breadth-first searches of all sliding-tile puzzles up to the 4×4 and 2×8 Fifteen Puzzles. Table 3.5 shows the results. For the problems larger than the 11 puzzle puzzle, we used delayed duplicate detection on disk.

The first column gives the dimensions of the puzzle, and the second column the number of tiles, which is the product of the dimensions

S			Total States	Max Width	Ratio
2			12	2	6.000
2			360	44	8.182
2			20,160	1,999	10.085
3			181,440	24,047	7.545
2 × 5	9	55	1,814,400	133,107	13.361
2 × 6	11	80	239,500,800	13,002,649	18.419
3 × 4	11	53	239,500,800	21,841,159	10.966
2 × 7	13	108	43,589,145,600	1,862,320,864	23.406
3 × 5	14	84	653,837,184,000	45,136,428,198	14.486
2 × 8	15	140	10,461,394,944,000	367,084,684,402	28.499
4 × 4	15	80	10,461,394,944,000	784,195,801,886	13.340

Table 3.5: Complete Searches of Sliding-Tile Puzzles

minus one. The third column is the maximum depth at which any state appears for the first time, starting from a position with the blank in the corner, except for the 14 puzzle, where the blank started in the center. The fourth column is the total number of states in the complete problem space, which is $(n+1)!/2$, where n is the number of tiles. The fifth column is the width of the problem space, which is the maximum number of states at any given depth. The last column gives the total number of states divided by the width of the space.

While standard breadth-first search stores all the states in the problem space, breadth-first frontier search only stores the number of states in the width of the space. The ratio of these two values gives the memory savings of frontier search. For the 2×8 Fifteen Puzzle, breadth-first frontier search saves a factor of over 28 in memory relative to standard breadth-first search.

4-Peg Towers of Hanoi Problem

The 4-peg Towers of Hanoi problem, also known as Reve's Puzzle, adds an additional peg to the original 3-peg problem. For the 4-peg problem, there exists a strategy that is believed to require a minimum number of moves [26, 95], but this conjecture remains unproven[21], even though the problem has been around for over 100 years [20]. For n discs, the

Discs	Opt. So		Total States	Max Width	Ratio
1			4	3	1.333
2			16	6	2.666
3			64	30	2.133
4			256	72	3.555
5	13	13	1,024	282	3.631
6	17	17	4,096	918	4.462
7	25	25	16,284	2,568	6.341
8	33	33	65,536	9,060	7.234
9	41	41	262,144	31,638	8.286
10	49	49	1,048,576	109,890	9.542
11	65	65	4,194,304	335,292	12.509
12	81	81	16,777,216	1,174,230	14.288
13	97	97	67,108,864	4,145,196	16.190
14	113	113	268,435,456	14,368,482	18.682
15	129	130	1,073,741,824	48,286,104	22.237
16	161	161	4,294,967,296	162,989,898	26.349
17	193	193	17,179,869,184	572,584,122	30.004
18	225	225	68,719,476,736	1,994,549,634	34.454
19	257	257	274,877,906,944	6,948,258,804	39.561
20	289	294	1,099,511,627,776	23,513,260,170	46.761
21	321	341	4,398,046,511,104	77,536,421,184	56.722
22	385	394	17,592,186,044,416	278,269,428,090	63.220

Table 3.6: Complete Searches of 4-Peg Towers of Hanoi Problems

"presumed optimal solution" requires $S(n)$ moves [38], where $S(n)$ is defined recursively as $S(1) = 1$ and $S(n) = \min\{2S(k) + 2^{n-k} | k \in \{1, \ldots, n-1\}\}$. Thus, search algorithms for this problem are still of interest, if only to test the conjecture for larger numbers of discs.

We also implemented a complete breadth-first frontier search on Four-Peg Towers of Hanoi problems with up to 22 discs. The results are shown in Table 3.6. The first column shows the number of discs. The second column gives the optimal solution length, and the third column gives the maximum search depth, which is the maximum distance of any state from the standard initial state with all discs on a single peg. The fourth column gives the total number of states in the space, which is 4^n,

wh of discs. The fifth column shows the maximum wi pace. The last column is the total number of st naximum width, which is the memory savings of br earch over standard breadth-first search. Note th tier search saves a factor of over 63 in memory over standard breadth-first search. The 17 through 22 disc problems required delayed duplicated detection on disk.

In all cases, our experiments confirm the conjectured minimum number of moves. In most cases, the optimal solution length equals the maximum search depth, and prior to this work this was believed to always be the case. For depths 15, 20, 21, and 22, however, the maximum search depth is greater than the optimal solution depth. This means that there are states that are further away from a state with all discs on one pege than the states with all discs on another peg. This is a completely unexpected result.

3.13 Breadth-First Heuristic Search

One of the drawbacks of a best-first search such as A* is the overhead of always expanding next a node of lowest cost. This requires maintaining the nodes in a data structure that makes it easy to find the lowest-cost node, such as a heap. Furthermore, in a parallel implementation, since only one node may be eligible for expansion at any time, this can create a serializing bottleneck. In addition, this makes it difficult to implement an algorithm such as delayed duplicate detection, since we need to expand a large number of nodes before eliminating duplicates.

However, as long as all nodes of cost less than the optimal solution cost are expanded, the order in which they are expanded doesn't matter. An algorithm that takes advantage of this observation is called *breadth-first heuristic search* [104]. Assume that we know the maximum $f(n)$ cost of any node to be expanded. We'll see in the next chapter how to set this value. Breadth-first heuristic search expands nodes breadth-first order, in other words in order of their depth from the root node, but applies a heuristic cost function to every node generated. If the total cost of any node exceeds the maximum cost, that node is discarded. By expanding nodes in breadth-first order, it becomes much

easier to determine when to expand, and in general a large number of nodes can be expanded in parallel. This also facilitates delayed duplicated detection.

3.13.1 Exploring the Four-Peg Towers of Hanoi

In order to verify the presumed optimal solution to the Four-Peg Towers of Hanoi problem with larger numbers of discs, we turned to heuristic search. We used frontier breadth-first heuristic search with pattern database heuristics, which will be described in section 6.2. We also used parallel delayed duplicate detection, storing the nodes on disk. Using these techniques, we were able to verify the presumed optimal solution for up to 31 discs [49].

Chapter 4

Linear-Space Heuristic Searches

The primary limitation of all the best-first search algorithms described in Chapter 3 is that their asymptotic space complexity equals their asymptotic time complexity, which is generally exponential. As a practical matter, if any of these algorithms are run on current machines, they will exhaust the available memory in a matter of minutes.

In this chapter we consider three different classes of heuristic search algorithms that only require space that is linear in the maximum search depth. For these algorithms, memory is never a limitation in practice.

4.1 Iterative-Deepening-A*

The first algorithm we consider is called Iterative-Deepening-A*, or IDA* for short[41]. The best way to think of IDA* is that it applies the iterative deepening idea to simulate the A* algorithm, but using only linear instead of exponential space. In other words, IDA* is to A* as depth-first iterative deepening (DFID) is to breadth-first search.

IDA* proceeds in a series of iterations. Each iteration is a depth-first search, with an associated cost threshold. As each node in the depth-first search is generated, its total cost, $f(n) = g(n) + h(n)$, is computed. If this cost is less than or equal to the cost threshold for the iteration, the depth-first search continues by expanding the node.

If [illegible]s the iteration threshold, however, the branch is pr[illegible] backtracks.

[illegible]ound whose cost does not exceed the threshold fo[illegible]n, the algorithm terminates, returning the path to [illegible] algorithm is depth first, and naturally implemented as a recursive function, when a goal is found, the solution path is contained on the recursion stack. Thus, in a recursive implementation, there is no need to explicitly store previous states or pointers to the parents of a state. Rather, the solution can be output simply by returning back up the call stack.

If an iteration completes its search without returning a solution, another iteration is begun with a greater cost threshold. The cost threshold for each succeeding iteration is set to the minimum cost of all nodes that were generated but not expanded on the previous iteration. The cost threshold for the first iteration is set to the cost of the initial state, which is its heuristic value, and is normally a lower bound on the optimal solution cost. Note that no information is preserved from one iteration to the next, other than the cost thresholds.

Figure 4.1 shows an example search tree generated by the first iteration of IDA* on the Eight Puzzle, using the Manhattan distance heuristic. The root of the tree is the initial state, and the state to its immediate right is the goal state. Since the Manhattan distance between this initial and goal state is 12 moves in this case, the cutoff threshold for the first iteration of IDA* is 12. The equation below each node of the tree gives its cost in the form $g(n) + h(n) = f(n)$. The first iteration performs a depth-first search of the tree, expanding every node with cost less than or equal to 12. During the iteration, IDA* keeps track of the lowest cost among all nodes that were generated but not expanded. In this case, that value is 14.

In the next iteration, the cost threshold will be increased to 14, and IDA* will perform a new depth-first search of the tree, starting from the root, and expanding all nodes whose cost is less than or equal to 14. This will continue until a goal state is reached whose cost is less than or equal to the current threshold.

This pattern of all costs having the same even-odd parity often appears in practice. In the sliding-tile puzzles, for example, every move of a tile increases the $g(n)$ value by one, and either decreases the Man-

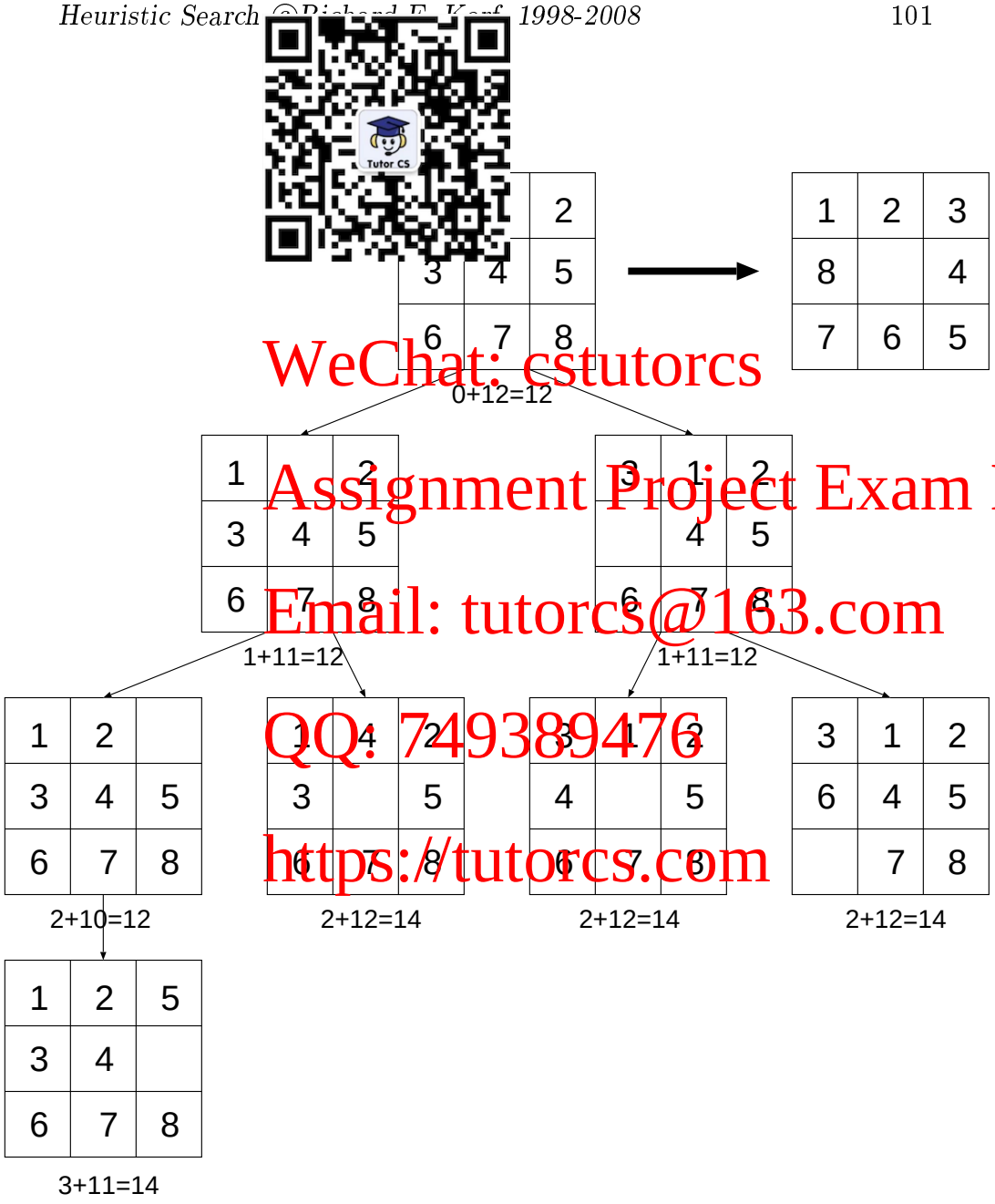


Figure 4.1: Example Search Tree Generated by First Iteration of IDA*

ha by one, or increases it by one. Thus, the $f(n)$ va er the same as that of its parent, or two greater.

4. ion

The first question to ask is whether IDA* is guaranteed to find a solution if one exists. Assume that there exists a solution of finite cost, that the heuristic values along the solution path are all finite, that there is a minimum positive edge cost, and that all heuristic values are nonnegative. A node is pruned when its total cost exceeds the threshold for the current iteration, and the threshold for the next iteration is the minimum cost among all nodes pruned on the previous iteration. Thus the sequence of thresholds is strictly increasing with each iteration, and at least one new node is expanded on each iteration, the one with the minimum cost among those pruned on the previous iteration. Furthermore, since there is a minimum edge cost and all heuristic values are nonnegative, there are no infinite-length paths of finite cost. Thus, unless it terminates with a solution, the costs of the iterations will grow without bound, and every node connected to the root via a path with finite edge and heuristic values will eventually be expanded. Therefore, if there exists a solution of finite cost, it will eventually be found and returned by the algorithm.

4.1.2 Solution Optimality

Next we consider the solution quality returned by IDA*. The result here is the same as for A*, namely that if the heuristic never overestimates actual cost, then IDA* returns an optimal solution if one exists.

Theorem 4.1 *In a graph where all edges have a minimum positive cost, and nodes have nonnegative heuristic values that never overestimate actual cost, in which a finite-cost path exists to a goal state, IDA* will return an optimal path to a goal.*

We already showed above that under these conditions, IDA* will eventually choose a goal node for expansion, and return the path to that goal. It remains to show that when this happens, the solution returned is an optimal one.

Define the *fr*[illegible]y given point in time as the set of nodes that have [illegible]ast once but never expanded. Note that the complet[illegible]tually stored in memory. The first step in our proof [illegible]frontier always contains a node on an optimal solut[illegible]n exists. The proof is by induction on the number of node expansions. Before any expansions, the frontier is just the initial state, and hence this node is on an optimal solution path if a solution exists. For the induction step, assume that at the end of k expansions, there is a node n on the frontier on an optimal solution path. Expansion $k+1$ either expands node n or not. If it doesn't expand node n, then node n is still on the frontier, and is still on an optimal solution path. If it does expand node n, then all of its children are members of the new frontier, and at least one of them must be on an optimal solution path. Thus, there is always a node on the frontier on an optimal solution path.

The next step of the proof is to show that all the cost thresholds are less than or equal to the cost of an optimal solution. Since there is always a node on the frontier on an optimal solution path, this is also the case at the end of an iteration. Call such a node n. Since n is on an optimal solution path, the path from the start to node n must be a shortest such path. Since the heuristic function never overestimates the remaining cost to a goal, the cost of node n, $f(n) = g(n) + h(n)$, must be less than or equal to the optimal solution cost. Since the threshold for the next iteration is the minimum cost of all nodes on the frontier, at the end of the previous iteration, it must be less than or equal to the cost of node n, and hence less than or equal to the optimal solution cost. Thus, every cost threshold must be less than or equal to the optimal solution cost.

The algorithm terminates when it chooses for expansion a goal node m whose cost is less than or equal to the current threshold. Since m is a goal node, its $h(m)$ value must be zero, and its total cost is simply $g(m)$, the cost from the root to node m. Since this cost is less than or equal to the cost threshold, and all cost thresholds are less than or equal to the optimal solution cost, the cost of node m is less than or equal to the optimal solution cost. Since this path is a path from the root to a goal, its cost must equal the optimal solution cost, and hence IDA* returns an optimal solution. □

4.1.3 Space Complexity

We now analyze the space complexity of IDA*. The most natural implementation is as a recursive function. The only memory that is used is the recursion stack. An individual stack frame will occupy a constant amount of memory, and thus the asymptotic space complexity is simply the maximum depth of the stack.

How deep can the stack grow? Assume that the optimal solution cost is c, and that the minimum edge cost is e. No node with cost greater than c will be expanded by IDA*, and the maximum length of any path with total cost less than or equal to c is c/e. Thus, the maximum search depth is $c/e+1$, since nodes of cost c may be expanded. Since e is a constant, the asymptotic space complexity of IDA* is $O(c)$. In practice, this is never a constraint, since the time complexity is typically exponential in the solution cost, as we will see below. Thus, IDA* eliminates the space constraint of A* in practice.

4.1.4 Time Complexity

The next consideration is the time complexity, and particularly the time taken in the iterations of IDA* prior to the last iteration, which is the one that returns the solution.

The Last Iteration

Since the sequence of cost thresholds is strictly increasing, each completed iteration generates more nodes than the previous one. Thus, the last iteration will generate the most nodes, if completed. Since the cost thresholds are always less than or equal to the optimal solution cost c, and the algorithm terminates when it finds a goal node whose cost is less than or equal to the cost threshold, the cost threshold of the last iteration is exactly c. In the worst case, the last node generated in this iteration will be the goal node. Thus, the last iteration of IDA* will expand all nodes whose cost is less than or equal to c.

Strictly speaking, this is only true if the heuristic function is consistent. If the heuristic function is inconsistent, then the total cost function can be nonmonotonic, and there could be a node of cost less

than c that appe[illegible]e as a descendent of a node of cost greater than c. S[illegible]ould not be expanded, the descendent would not [illegible]e heuristic function is consistent, however, the tot[illegible]) = g(n) + h(n)$ will be monotonically nondecreasi[illegible]om the root, and in the worst case the last iteration of IDA* will expand all nodes whose cost is less than or equal to c. This is exactly the set of nodes expanded by A*. Thus, in the worst case, the last iteration of IDA* expands the same set of nodes as A* does.

Time of Previous Iterations

How much time is spent on the iterations of IDA* prior to the final iteration? The answer depends on the heuristic function and the distribution of cost values in the tree. In our time-complexity analysis of A* above, we argued that, for real heuristics on real problems, the heuristic branching factor is equal to the brute-force branching factor. In other words, for large x, the ratio of the number of nodes of cost x, $N(x)$, divided by the number of nodes of cost $x-1$, $N(x-1)$, is the brute-force branching factor b. The number of nodes generated by an iteration of IDA* with cost threshold x is

$$\text{IDA}(x) = \sum_{i=1}^{x} N(x)$$

If $N(x)$ grows exponentially with x, with branching factor b, then $\text{IDA}(x)/\text{IDA}(x-1) = b$. Thus, the number of nodes generated by successive iterations of IDA* also grows exponentially with this same branching factor b. Our analysis of DFID in section 2.3.3 shows that with exponential growth of successive iterations, the overall asymptotic time complexity of DFID is the same as for the last iteration. Similarly, given the assumptions of our analysis, the asymptotic time complexity of IDA* is the same as that of A*.

Thus, IDA* preserves the optimal solutions of A*, with an admissible heuristic, while removing the space constraint, without any sacrifice in asymptotic time complexity. In fact, since it is a depth-first search, the time per node generation is generally less for IDA* than for A*, and in practice it may run significantly faster. An additional feature of

ID[illegible]h simpler to implement than A*, since it is just a [illegible]nd doesn't need to manage any Open or Closed lis[illegible] to its speed as well.

4.[illegible]ns of IDA*

The above analysis assumed a particular model of the heuristic function, and that the goal node was the last node generated in the final iteration. The absolute worst-case for IDA* is much worse however[71]. Consider a tree where every node has a different total cost. For example, Figure 2.1 shows a fragment of such a tree if we interpret the numbers in the nodes as their total cost. In such a tree, there will be a different iteration for every unique node cost, and each iteration will expand exactly one new node that wasn't expanded in the previous iteration. A* will generate $O(b^d)$ nodes in such a tree, if d is the solution depth. IDA* will execute $O(b^d)$ iterations, most of which will contain $O(b^d)$ nodes. Thus, the time complexity of IDA* will be $O(b^{2d})$ on such a tree. In general, if the asymptotic complexity of A* is $O(N)$ on a tree, the asymptotic time complexity of IDA* can be $O(N^2)$ in the worst case. Note that such a scenario may be unrealistic, however, since maintaining unique node costs in an exponentially growing tree requires that the precision of the values continue to increase. In a binary tree, for example, we must add another bit to the representation of the costs with each level of the tree.

Another important caveat to the above results is that the problem space must be a tree. In a graph with cycles, A* will expand all *states* whose cost is less than or equal to c, while IDA* will expand all *nodes* with this same property. The difference is that if the same state can be reached via multiple paths, it will be represented by more than one node in the search tree. By storing the states in memory, A* can detect this duplication and avoid expanding the same state more than once. Since IDA* is a depth-first search, however, it can't detect most duplicates, and expands the graph as a tree, generating a different node for every distinct path to the same state. This can greatly increase the asymptotic time complexity of IDA* compared to A*. Thus, if there are many short cycles in the graph, and sufficient memory is available to run A*, it should be chosen over IDA*. For example, on a dense two-

dimensional roa lots of cycles, but the number of states grows only the size of the map. Furthermore, the map typicall xplicitly in memory anyway. Thus, for a navigation g the best route between a pair of points, A* woul f choice.

4.1.6 Experiments with IDA*

IDA* was first applied to the sliding-tile puzzles. A* with the Manhattan distance heuristic can easily solve the Eight Puzzle, since the average problem instance generates about 1600 nodes. The optimal solution length, averaged over all problem instances, is about 22 moves. On this problem, a good implementation of IDA* runs about three times faster per node generation, than a good implementation of A*, easily making up for the slight increase in node generations. The factor of three speedup will be smaller on problems where node generation and evaluation is more complex, and hence the additional overhead per node of A* will be less significant.

Optimally solving random instances of the Fifteen Puzzle, however, requires generating hundreds of millions of nodes, and thus A* will exhaust the memory on current machines. IDA* with the Manhattan distance heuristic was the first algorithm to optimally solve such problems, generating an average of 400 million nodes per problem[41]. At over seven million nodes per second, these problems take less than a minute to solve on current machines. The average optimal solution length is about 52.5 moves.

IDA* has also been used to solve 50 Twenty-Four Puzzle instances optimally, using much more powerful heuristics that will be described in Chapter 6. These problems generate an average of 360 billion nodes each, and average two days each to solve. The average optimal solution length is about 100 moves [48].

Finally, IDA* has been used to find optimal solutions to the standard $3 \times 3 \times 3$ Rubik's Cube, using a heuristic function that will be described in Chapter 6. These problems generated an average of 350 billion nodes each, and required about a week to solve on average. The median optimal solution length is 18 moves [45].

For all the data mentioned above, the number of nodes generated,

an solve a problem instance, is exponentially distri a large variance among individual problem instances. erage solution time is not indicative of the time to roblem instance, but rather the time to solve a lar ces.

4.2 Iterative Deepening (ID) as an Algorithm Schema

Just as best-first search refers to an entire class of algorithms, differing only in their cost function, iterative deepening (ID) can also be viewed as an algorithm schema, with different instantiations based on the cost function. If cost is depth, we get depth-first iterative-deepening (DFID); if cost is $g(n) + h(n)$ we get IDA*; and if cost is just $g(n)$ we get an iterative-deepening version of uniform-cost search, for example. All these algorithms execute a series of depth-first searches, each with an associated cost threshold. The cost threshold for the first iteration is the cost of the root node, and for each subsequent iteration it is increased to the minimum cost of all nodes generated but not expanded on the previous iteration. Finally, it terminates when a goal node is chosen for expansion. For here on, we will use the term iterative deepening to refer to this class of algorithms.

4.3 Depth-First Branch-and-Bound

IDA* was originally designed for problems, such as the sliding-tile puzzles and Rubik's Cube, where the optimal solution depth is not known in advance, and there are no good upper bounds on solution cost. For other problems, such as the Travelling Salesman Problem for example, the search tree is finite, and the solution depth is known a priori. In the most obvious problem space for the TSP, each node represents a partial tour starting from the initial city, and each operator adds another city to the end of the partial tour. Figure 4.2 shows such a search tree for a four-city TSP problem, where the cities are labeled A, B, C, and D. In a TSP problem with N cities, this search tree is only N

levels deep, and [illegible] depth N. In problems such as this, *depth-first branc*[illegible]B) is often the search technique of choice, particula[illegible]ons are required. DFBnB can also be applied in an[illegible]e is a good upper bound available on the optimal s[illegible]ution cost.

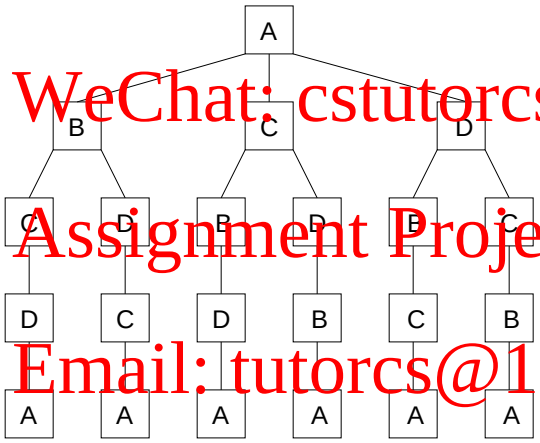


Figure 4.2: Search Tree for TSP on Four Cities

Another reason to prefer DFBnB over IDA* for the TSP is that if the intercity distances are expressed with high precision, there may be almost as many different node costs as there are nodes. In that case, IDA* may only generate a few new nodes on each iteration, approximating its poor performance on a tree with all unique node values.

DFBnB assumes a cost function that can be applied to a partial solution, and is a lower bound on the cost of all completions of that partial solution. For example, in the TSP tree described above, the simplest cost function would be the sum of the costs of the edges in the partial tour so far. Any tours that result from extending the partial tour will have cost at least as large as the cost of the partial tour, assuming nonnegative edge costs.

DFBnB performs a single depth-first search of the problem-space tree, say from left-to-right without loss of generality. As soon as the left-most branch terminates, we have a candidate solution. Let the cost of this solution be α, which is the cost of the best complete solution found so far. Initially, α is set to infinity. As the search continues,

th al node of the tree is compared to α. If the cost of ceeds α, the branch below that node is pruned, be ad to solutions whose cost is equal to or greater th n found so far. If a complete solution is found wh n α, α is updated to this lower value, and the best solution so far is replaced by this new solution. When the search terminates by exhausting the entire tree, the best solution found is returned as the optimal solution.

The pruning efficiency of DFBnB can be greatly improved by several means. The first is called *node ordering*. The idea is to search the children of each parent node in a particular order to improve the overall efficiency of the algorithm. In particular, we would like to find a low-cost solution as quickly as possible, since a lower value of α at a given point during the search results in greater pruning in the rest of the search. For the TSP problem space described above, each node corresponds to a city. Thus, we can order the child nodes of each parent in increasing order of the distance of the child cities from the parent city. This greedy heuristic results in the first tour found, corresponding to the left-most path of the tree, being the *nearest neighbor* tour.

DFBnB can also make use of a heuristic evaluation function. Given a lower-bound heuristic $h(n)$, each node can be evaluated with the A* cost function $f(n) = g(n) + h(n)$. Whenever this cost equals or exceeds the cost of the best complete solution found so far, the search can be pruned below this node. Note that the overall cost function need not be monotonic nondecreasing, but simply not overestimate actual solution costs. Given such a heuristic function, it may be more effective to order the children of a node based on their total cost, rather than simply the edge costs from the parent to the child.

4.3.1 Solution Quality and Complexity

It should be obvious that given an admissible cost function, DFBnB is guaranteed to return an optimal solution. It searches the entire problem space, pruning only those partial solutions whose costs already equal or exceed the cost of the best complete solution already found.

The space complexity of DFBnB is linear in the maximum search depth, since it is a depth-first search. Note that node ordering is per-

formed only on ode once it is expanded, but the entire tree is not branching factor and d the maximum search dep pace complexity is $O(bd)$, since all the children of e rrent path must be generated and stored to order t ly, this is the same as $O(d)$, since we assume that b is a constant.

The time complexity of DFBnB is a more difficult matter. In the best case, the first solution found will be an optimal solution, and thus α will equal c, the cost of an optimal solution, throughout the entire search. The algorithm still must expand every node with cost less than c, and hence its time complexity will be the same as that of A* on a tree, given the same heuristic function. In fact, we showed that A* is optimal in terms of time complexity for any algorithm that guarantees an optimal solution. Note, however, that since DFBnB is a depth-first search, on a graph with cycles it may potentially explore all distinct paths to a given state, resulting in greater asymptotic complexity than A*, which can detect and prune duplicate nodes.

In general, however, the first solution found will not be optimal, and α will exceed the optimal solution cost at least initially. In that case, nodes with cost greater than c will be expanded. Thus, an analysis of the time complexity of DFBnB has to model both the heuristic function and the efficacy of node ordering. The existing analyses on this algorithm are all based on abstract analytic models.

4.3.2 An Analytic Model and Surprising Anomaly

One such model assumes a tree with uniform branching factor and depth, where the edges are assigned costs randomly from some distribution. The total cost of a node is then the sum of the edge costs from the root to the given node. For example, we could assign the edges cost zero or one with probability .5 each. In this case, assuming that the nodes are ordered by their costs in the depth-first search, DFBnB finds the optimal solution at a given depth in a ternary tree faster than in a binary tree! The intuitive reason for this is that a node in a ternary tree is more likely to have a zero-cost child than in a binary tree. As a result, the value of α is likely to be lower at a given point in the search of a ternary tree than in a binary tree. This lower value of α more than

co rger branching factor.

xpected number of zero-cost children of a node is gr nB with node ordering will run in time that is po ch depth[102]. If the expected number of zero-cost children is less than one, then DFBnB runs in exponential time in the search depth. In the exponential case, DFBnB has the same asymptotic time complexity as best-first search, or uniform-cost search in this case. This is encouraging, since we know that best-first search is optimal in time complexity. Thus, we can reduce the space complexity from exponential to linear, without sacrificing optimal solutions or asymptotic time complexity, on a tree with fixed depth.

4.3.3 Truncated Branch and Bound

A very important property of DFBnB is that if there isn't sufficient time to compute an optimal solution, the algorithm can be terminated or truncated at any point in time (truncated branch-and-bound), and return the best solution found so far. This makes DFBnB an *anytime algorithm*[5], in that it computes a solution almost immediately, and the longer it runs, the more it improves on that solution, until eventually finding, and still later verifying, an optimal solution. Truncated branch-and-bound is one of the best known algorithms for solving large instances of the asymmetric Travelling Salesman Problem[103], and the best algorithm for solving large number partitioning problems[44], outperforming incomplete local search techniques. The asymmetric TSP is a version of the TSP where the cost of going from city i to city j is not necessarily the same as the cost of going from city j to city i. The problem of number partitioning is given a set of numbers, divide them all into two nonoverlapping subsets so that the sum of the numbers in each of the subsets are as nearly equal as possible. Both problems are NP-complete.

4.4 Com…erative Deepening and Dept…nch-and-Bound

Iterative deepen… similar but complementary algorithms. Both gu…tions, given a lower-bound heuristic function. Both are depth-first searches, and hence have linear space complexity. Both make use of a global cost bound. In the case of iterative deepening this cost bound is the threshold for a given iteration, and in the case of DFBnB the bound is the cost of the best solution found so far. The fundamental difference between the two algorithms lies in the nature and use of these cost bounds. In iterative-deepening, the cost threshold is always a lower bound on the cost of an optimal solution, and increases in each successive iteration until it reaches the cost of an optimal solution. In DFBnB, however, the cost bound is always an upper bound on the cost of an optimal solution, and decreases over the course of the search until it equals the optimal solution cost.

Both iterative deepening and DFBnB will expand more nodes in general than A*. This is a consequence of their linear space complexity. While iterative deepening never expands any nodes whose cost is greater than c, the cost of an optimal solution, its node generation overhead comes from the fact that every node expanded in each iteration will be reexpanded in each succeeding iteration as well. DFBnB, however, never expands the same node more than once, but since its cost bound in general will not equal c initially, DFBnB will expand some nodes whose cost exceeds c. Note that unlike A*, both iterative deepening and DFBnB are depth-first searches, and will explore multiple paths to the same state in a graph with cycles.

4.5 Nonmonotonic Cost Functions

As with all best-first searches, the main drawback of the weighted heuristic search algorithm is its space complexity, which is the same as its time complexity. Thus, it will exhaust the available memory in a matter of minutes on most computers. The obvious solution at this point is to use iterative deepening (ID), but with the weighted cost function, $f(n) = g(n) + w \cdot h(n)$. The resulting algorithm is identical

to e cost function.

nondecreasing cost function, it is easy to see that no r the first time by iterative deepening in a best-fir words, whenever a node is expanded, all nodes of ady been expanded at least once. Our weighted cost function, $f(n) = g(n) + w \cdot h(n)$ is nonmonotonic with $w > 1$, however, meaning that the cost of a child node can be less than that of its parent. With a nonmonotonic cost function, iterative deepening no longer expands nodes in best-first order.

Consider, for example, the tree fragment in Figure 4.3, where the numbers represent total node costs. A best-first search would expand these nodes in the order 5, 1, 2. With iterative deepening, however, the initial threshold would be the cost of the root node, 5. After generating the left child of the root, node 2, iterative deepening would expand all descendents of node 2 whose costs did not exceed the threshold of 5, in depth-first order, before expanding node 1. Even if all the children of a node were generated at once, and ordered by their cost values, so that node 1 was expanded before node 2, iterative deepening would explore subtrees below nodes 4 and 3 before expanding node 2. The problem is that while searching nodes whose costs are less than the current threshold, iterative deepening ignores the values of those nodes, and proceeds depth-first.

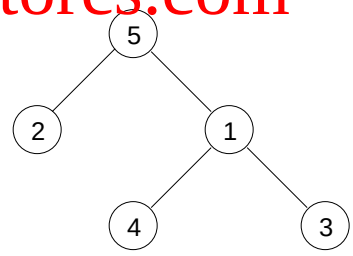


Figure 4.3: Search Tree Fragment with Nonmonotonic Cost Function

This problem is even more serious on a graph with cycles. For example, with a pure heuristic cost function, $f(n) = h(n)$, there may be cycles in the graph in which the maximum node cost is less than the current cost threshold. In that case, iterative deepening will go around such a cycle forever without terminating. Fortunately, since any depth-

first search maintains the search path in memory, this problem could be fixed by comparing each newly generated node with those on the current path, and discarding any node that already appears on the path.

4.6 Recursive Best-First Search

Recursive Best-First Search (RBFS) is a linear-space algorithm that expands nodes in best-first order even with a nonmonotonic cost function, and generates fewer nodes than iterative deepening with a monotonic cost function. For pedagogical reasons, we first present a simple version of the algorithm, and then consider the more efficient full algorithm.

4.6.1 Simple Recursive Best-First Search

While iterative deepening uses a global cost threshold, *Simple Recursive Best-First Search* (SRBFS) uses a local cost threshold for each recursive call. It takes two arguments, a node and an upper bound on cost, and explores the subtree below the node as long as it contains frontier nodes whose costs do not exceed the upper bound. It then returns the minimum cost of the frontier nodes of the explored subtree. Figure 4.4 shows how SRBFS searches the tree in Figure 4.3 in best-first order. The initial call on the root is made with an upper bound of infinity. We expand the root, and compute the costs of the children as shown in Figure 4.4A. Since the right child has the lower cost, we recursively call SRBFS on the right child. The best frontier nodes in the tree will be descendents of the right child as long as their costs do not exceed the value of the left child. Thus, the recursive call on the right child has an upper bound equal to the value of its lowest-cost brother, 2. SRBFS expands the right child, and evaluates the grandchildren, as shown in Figure 4.4B. Since the values of both grandchildren, 4 and 3, exceed the upper bound on their parent, 2, the recursive call terminates. It returns as its result the minimum value of its children, 3. This backed-up value of 3 is stored as the new value of the right child (Figure 4.4C), indicating that the lowest-cost frontier node below this node has a cost of 3. A recursive call is then made on the new best child, the left one,

wi equal to 3, which is the new value of its lowest-co child. In general, the upper bound on a child no inimum of the upper bound on its parent and th s lowest-cost brother.

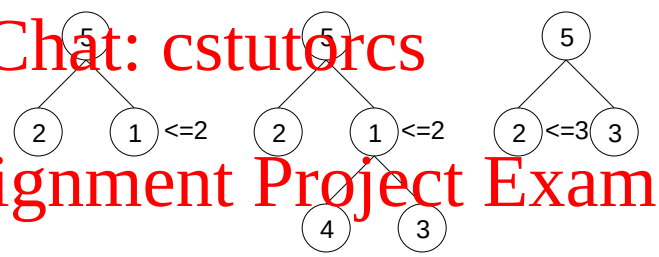


Figure 4.4: SRBFS Example with Nonmonotonic Cost Function

Figure 4.5 shows a more extensive example of SRBFS. In this case, the cost function is simply the depth of the node in the tree, corresponding to breadth-first search. The children of a node are generated from left to right, and ties are broken in favor of the most recently generated node. The reader is encouraged to work through the example in the figure.

Initially, the *stored* value of a node, $F(n)$, equals its *static* value, $f(n)$. After a recursive call on the node returns, its stored value is equal to the minimum value of all frontier nodes in the subtree explored during the last call. SRBFS proceeds down a path until the static values of all children of the last node expanded exceed the stored value of one of the nodes further up the tree. It then returns back up the tree, replacing parent values with the minimum of their children's values and freeing memory, until it reaches the better node, and then proceeds down that path. The algorithm is purely recursive with no side-effects, resulting in very low overhead per node generation. At any point, the recursion stack contains the path to a lowest-cost frontier node, plus the siblings of all nodes on that path. Thus, its space complexity is $O(bd)$, which is linear in the maximum search depth d. In pseudo-code, the algorithm is as follows:

Figure 4.5: SRBFS Example with Cost Equal to Depth

```
SR                                nd: B)
IF                                (N)
IF                                T algorithm
IF                                n, RETURN infinity
FO                                f N, F[i] := f(Ni)
sort N1 and F[1] in increasing order of F[i]
IF only one child, F[2] := infinity
WHILE (F[1] <= B and F[1] < infinity)
   F[1] := SRBFS(N1, MIN(B, F[2]))
   insert N1 and F[1] in sorted order
return F[1]
```


Once a goal node is chosen for expansion, the actual solution path is on the recursion stack, and returning it involves simply recording the moves as the algorithm returns back up the stack. For simplicity, we omit this from the above description.

SRBFS expands nodes in best-first order, even if the cost function is nonmonotonic. Unfortunately, however, SRBFS is inefficient. If we continue the example from Figure 4.5, where cost is equal to depth, eventually we would reach the situation shown in Figure 4.6A, where the left child has been explored to depth 7 and the right child to depth 8. Next, a recursive call will be made on the left child, with an upper bound of 3, the value of its brother. The left child will be expanded, and its two children assigned their static depths of 2, as shown in Figure 4.6B. At this point, a recursive call will be made on the right grandchild with an upper bound of 2, the minimum of its parent's bound of 8, and its brother's value of 2. Thus, the right grandchild will be explored to depth 3, then the left grandchild to depth 4, the right to depth 5, the left to depth 6, and the right to depth 7, before new ground can finally be broken by exploring the left grandchild to depth 8. Most of this work is redundant, since the left child has already been explored to depth 7.

4.6.2 Full Recursive Best-First Search

The way to avoid this inefficiency is for children to inherit their parent's values as their own, if the parent's values are greater than the children's

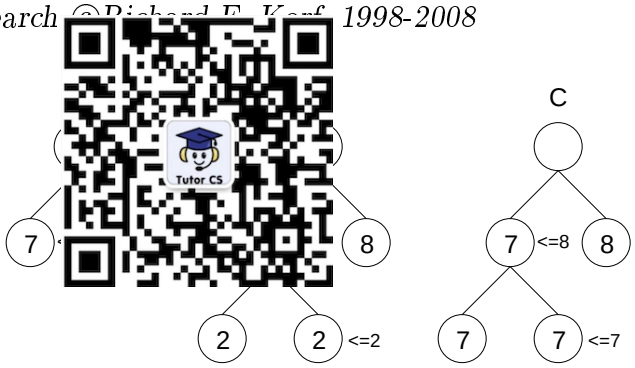


Figure 4.6: Inefficiency of SRBFS and its Solution

values. In the above example, the children of the left child should inherit their parent's value of 7 as their stored value, instead of 2, as shown in Figure 4.6C. Then, the right grandchild would be explored immediately to depth 8, before exploring the left grandchild. However, we must distinguish this case from that in Figure 4.3 where the fact that the child's value is smaller than its parent's value is due to nonmonotonicity in the cost function, rather than previous expansion of the parent node. In that case, the children should not inherit their parent's value, but use their static values instead.

The distinction is made by comparing the stored value of a node, $F(n)$, to its static value, $f(n)$. If a node has never been expanded before, its stored value equals its static value. In order to be expanded, the upper bound on a node must be at least as large as its stored value. The recursive call on the node will not return until the values of all the frontier nodes in the subtree below it exceed its upper bound, and its new stored value will be set to the minimum of these values. Thus, if a node has been previously expanded, its stored value will be greater than its static value.

If the stored value of a node is greater than its static value, its stored value is the minimum of the last stored values of its children. The stored value of such a node is thus a lower bound on the values of its children, and the values of the children should be set to the maximum of their parent's stored value and their own static values. If the stored value of a node is equal to its static value, then the node has never been expanded before, and the values of its children should be set to their static values. In general, a parent's stored value is passed down to

its ...erit the value only if it exceeds both the parent's sta...ild's static value.

...best-first search algorithm (RBFS) takes three ar... its stored value $F(N)$, and an upper bound B. Th...BFS is made on the root node r, with a value equal to the static value of r, $f(r)$, and an upper bound of ∞. In pseudo-code, the algorithm is as follows:


```
RBFS (node: N, value: F(N), bound: B)
IF f(N)>B, return f(N)
IF N is a goal, EXIT algorithm
IF N has no children, RETURN infinity
FOR each child Ni of N,
   IF f(N)<F(N) THEN F[i] := MAX(F(N),f(Ni))
   ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
IF only one child, F[2] := infinity
WHILE (F[1] <= B and F[1] < infinity)
   F[1] := RBFS(N1, F[1], MIN(B, F[2]))
   insert N1 and F[1] in sorted order
return F[1]
```


Figure 4.7 shows RBFS in action, again for the special case where cost is equal to depth. The numbers to the left of each subfigure represent depth in the tree, and hence the static values of the nodes at that horizontal level. Once the search has progressed to the situation in Figure 4.7A, the left child of the root is expanded, with an upper bound equal to its brother's value of 4, and its stored value of 3 is passed down to its children. Since the stored value of the parent, 3, exceeds its static value, 1, and also the children's static values of 2, the children's stored values are set to their parent's stored value of 3, as shown in Figure 4.7B. At this point, the rightmost grandchild is expanded with an upper bound equal to 3, the minimum of its parent's upper bound of 4, and its brother's value of 3, and its stored value of 3 is passed down to its children. Once again, since the stored value of the parent, 3, exceeds the static value of the parent, 2, and is not less than the children's

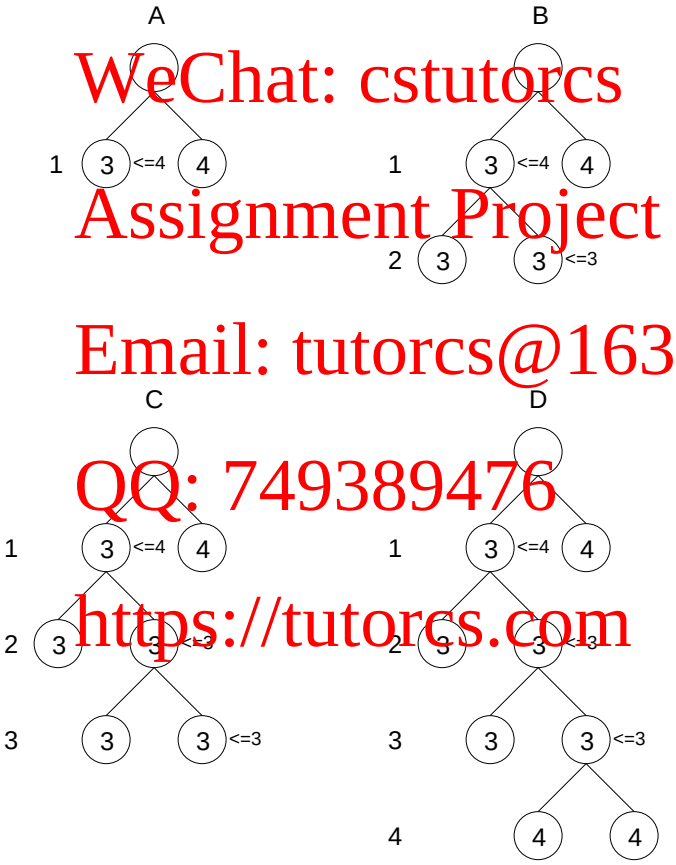


Figure 4.7: Recursive Best-First Search (RBFS) Example

sta ildren inherit their parent's stored value of 3, as sh The rightmost great-grandchild is now expanded wi of 3, which is the minimum of its parent's upper bo other's value of 3, and its stored value of 3 is pa ldren. In this case the parent's stored value of 3 equals its static value of 3, and the children's stored values are set to their static values of 4, as shown in Figure 4.7D. At this point, even if the static values of the children were less than that of their parent, the children would adopt their static values, and the subtrees below them would be explored in best-first order, until the values of all the frontier nodes exceeded the upper bound of 3.

Like SRBFS, RBFS explores new nodes in best-first order, even with a nonmonotonic cost function, and also runs in linear space. Its advantage over SRBFS is that it is more efficient. RBFS behaves differently depending on whether it is expanding new nodes, or previously expanded nodes. In new territory, it proceeds strictly best-first, while on old ground it goes depth-first, until it reaches a lowest-cost node on the old frontier, at which point it reverts back to best-first behavior.

If there are cycles in the graph, and cost does not always increase while traversing a cycle, as with a pure heuristic cost function, $f(n) = h(n)$, in order to avoid infinite loops each new node must be compared against the stack of nodes on the current path, and pruned if it is already on the stack.

4.6.3 Correctness of SRBFS and RBFS

The original design goal of RBFS was to simulate a best-first search, but in linear instead of exponential space. We define a best-first search order as a sequence of node expansions that could be performed by a best-first search. The reason there is more than one best-first search order is that in the case of a tie among nodes of equal cost, best-first search is free to choose among the tied nodes.

For a monotonic cost function, the overall cost of a child node is always greater than or equal to the cost of its parent. In a best-first search order with a monotonic cost function, the sequence of node costs is monotonically nondecreasing. It is easy to show that in this case, iterative deepening simulates a best-first search, in the sense that the

sequence in whi xpanded is a best-first search order. In other wo de is expanded, all nodes of lower cost have already ith a nonmonotonic cost function, however, iterativ er simulates a best-first search, as illustrated above

We can prove that the order in which nodes are first expanded by both SRBFS and RBFS is a best-first search order, even with a nonmonotonic cost function [42]. Ties among nodes of equal cost are broken in favor of nodes that are closest in the tree, in terms of number of edges, to the last node expanded.

4.6.4 Space Complexity of SRBFS and RBFS

The space complexity of SRBFS and RBFS is $O(bd)$, where b is the branching factor and d is the maximum search depth. Each recursive call of either algorithm must store all the children of the argument node, along with their F values. This requires $O(b)$ space. The maximum number of active calls that must be stored on the recursion stack at any point is equal to the maximum search depth d. For the Travelling Salesman Problem, for example, the maximum search depth is typically the number of cities. In the case of an admissible cost function, the maximum search depth is the longest path in which no node costs exceed the minimum solution cost. For example, in a problem with identical edge costs, and an admissible heuristic function, the maximum search depth will not exceed the optimal solution length.

4.6.5 Time Complexity of SRBFS and RBFS

The asymptotic time complexity of SRBFS and RBFS is the number of node generations. The actual number of nodes generated depends on the particular cost function used. We will consider the worst case, the special case where the cost is the depth, and finally compare the time complexity of RBFS and iterative deepening with an arbitrary monotonic cost function.

W[illegible]Complexity

As[illegible]pening, the worst-case time complexity occurs wh[illegible]unique cost values. Furthermore, they must be ar[illegible]ssive nodes in an ordered sequence of cost values ar[illegible]s of the root node. For example, Figure 4.8 shows a worst-case binary tree. In this case, in order to expand each new node at depth d, both SRBFS and RBFS must abandon their current path all the way back to the root. The stored value of one of the remaining children of the root will equal the cost of the next open node, indicating which child it is to be found under. However, there is no information as to where it is to be found under this child. In the worst case, we may have to generate all nodes down to and including depth d under this child, in order to find one of lowest cost. Since they are all descendents of the same child of the root, there are $O(b^{d-1})$ such nodes. Thus, each new node expansion at depth d may require $O(b^{d-1})$ node generations. Since there are b^d nodes at depth d, to completely explore a tree of depth d may require $O(b^{2d-1})$ time. This is only slightly better than the $O(b^{2d})$ worst-case time complexity of iterative deepening[71].

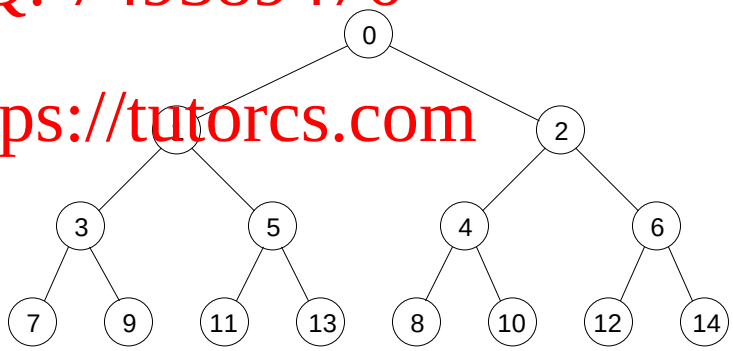


Figure 4.8: Worst-case Binary Tree for SRBFS and RBFS

RBFS vs. ID with Monotonic Cost Functions

How do RBFS and iterative deepening (ID) compare in general? If the cost function is nonmonotonic, the two algorithms are not directly comparable since they explore different parts of the tree and return

different solution... onic cost function, however, they both explore the ... produce lowest-cost solutions, but RBFS generates ... erative deepening on average. The proof of this can ... t we provide an overview below.

In a best-firs... otonic cost function, the sequence of costs of first-time node expansions is monotonically nondecreasing. Each distinct cost value in the tree generates a different iteration of iterative deepening, with a threshold equal to that cost value. We can similarly define an "iteration" of RBFS, corresponding to a particular cost threshold, as that interval during which those nodes being expanded for the first time all have the same cost. The primary difference between the two algorithms is that IDA* starts each new iteration with just the root node, while RBFS starts each new "iteration" with the path to the last node generated on the previous iteration. While this saves a few node expansions between each iteration, its primary advantage occurs when nodes of a given cost are clustered together in the tree.

As an extreme example, consider the depth-first search tree in Figure 2.2, but assume that the numbers in the nodes represent their cost, rather than their order of generation. On such a tree, since every node has a unique cost, each iteration would only expand one new node, and the time complexity of iterative deepening would be $O(b^{2d})$. In such a tree, however, all the descendents of a given node have costs less than that of the brother node with the next higher cost. Thus, RBFS will search this tree in depth-first order, never expanding a node more than once, and its time complexity will be $O(b^d)$.

As another example of the difference between RBFS and iterative deepening with a monotonic cost function, we compared RBFS to IDA* on the Euclidean Travelling Salesman Problem, using the A* evaluation function, $f(n) = g(n) + h(n)$. The heuristic function used was the minimum spanning tree (MST) of the cities not yet visited. Since the MST is a lower bound on the cost of a TSP tour, both IDA* and RBFS find optimal solutions. With 10, 11, and 12 cities, RBFS generated an average of 16%, 16%, and 18% of the nodes generated by IDA*, respectively. The time per node generation was roughly the same for the two algorithms, since the cost of computing the heuristic function dominates the running time. Additional experimental results, on the

asy on random trees, can be found in [102].

ever, both these algorithms generate significantly mo n-first branch-and-bound, with nodes ordered by th heuristic. The reason is that with sixteen bits to nces and for the size of problems that can be practically run, many nodes have unique cost values and there are relatively few ties. Thus, the overhead of node regenerations becomes prohibitive for both RBFS and IDA*, since a new iteration is needed for each distinct cost value. The potential advantage of IDA* and RBFS over depth-first branch-and-bound is that since the former are best-first searches, they never generate any nodes with cost greater than the optimal solution cost. However, depth-first branch-and-bound with node ordering is very effective on the TSP, generating only about 10% more than the minimum number of nodes, i.e. those with cost less than the optimal solution cost, and never reexpanding any nodes.

4.7 Other Limited-Memory Searches

A number of other limited-memory search algorithms, discussed below, have been designed to reduce the node-regeneration overhead of IDA*. The most important difference between these algorithms and RBFS is that none of the other limited-memory algorithms expand nodes in best-first order when the cost function is nonmonotonic. However, many of the techniques in these algorithms can be applied to RBFS to reduce its node regeneration overhead as well. A survey of these techniques can be found in [43].

4.7.1 MREC

The MREC algorithm[90] executes A* until memory is almost full, then performs IDA* below the stored frontier nodes. Duplicate nodes are detected by comparing them against the stored nodes. In Sen and Bagchi's original experiments with MREC on the Fifteen Puzzle[90], they did not check for duplicate nodes, and achieved only a 1% reduction in node generations. By checking for duplicate nodes, however, the reduction in number of node generations more than compensates

for the additional generation time. MREC could be combined with IDA* using a table of previously generated nodes, and checking among newly generated nodes.

4.7.2 MA*, SMA*, and ITS

While MREC statically allocates its memory to the first nodes generated, the MA* algorithm[10] dynamically stores the best nodes generated so far, according to the cost function. It behaves like A* until memory is almost full. Then, in order to free space to continue, it prunes a node of highest cost on the open list, updates the value of its parent to the minimum of its children's values, and places the parent back on the open list. Unfortunately, the constant overhead of this algorithm is prohibitively expensive. Both SMA*[84] and ITS[30] represent attempts to significantly reduce the constant factor overhead of MA*. If it could be made practical, MA* could be combined with RBFS by storing more of the tree than the current path, and checking newly generated nodes for duplicates.

4.7.3 DFS*, IDA_CR, and MIDA*

DFS*[81], IDA_CR[86], and MIDA*[101] are all very similar algorithms, independently developed. All three attempt to reduce the node regeneration overhead of IDA* by setting successive thresholds to values larger than the minimum value that exceeded the previous threshold. This reduces the number of different iterations by combining several together. In order to guarantee optimal solutions, once a goal is found, the algorithms revert to depth-first branch-and-bound, pruning any nodes whose cost equals or exceeds that of the best solution found so far. Setting the best threshold values requires some finesse, since values that are too large will result in excessive node generations on the final iteration.

4.7.4 Iterative Expansion

The algorithm that comes closest to RBFS is Iterative Expansion (IE)[84]. IE is very similar to SRBFS, except that a node always inherits its par-

en t's cost is greater than the child's cost. As a re -first search with a nonmonotonic cost function, bu to RBFS for the special case of a monotonic cost fu

4.7.5 Bratko's Best-First Search

Ivan Bratko[6] anticipated many of the ideas in RBFS, IE, MA*, and MREC, in his alternative formulation of best-first search. In particular, he first introduced the ideas of recursively generating the best frontier node instead of using an open list, using the value of the next best brother as an upper bound, and backing up the minimum values of children to their parents. The main difference between Bratko's algorithm and these others is his use of exponential space. Apparently, most of the inventors of these other algorithms were unaware of Bratko's earlier work.

Chapter 5

Real-Time Heuristic Search

5.1 Introduction

5.1.1 Limitation of Single-Agent Search Algorithms

A serious drawback of most of the algorithm schemas we have considered so far, including best-first search, iterative deepening, and RBFS, is that they typically take exponential time to produce any solution. While DFBnB finds a solution quickly, it also takes exponential time to find an optimal solution. This is an unavoidable cost of obtaining optimal solutions, and restricts the applicability of these algorithms to relatively small problems in practice. For example, brute-force search can solve the Eight Puzzle, IDA* with the Manhattan distance heuristic function can solve the Fifteen Puzzle, IDA* with an improved heuristic, described in the next chapter, can optimally solve the Twenty-Four Puzzle, and also the $3 \times 3 \times 3$ Rubik's Cube. Finding optimal solutions to any larger versions of these problems is intractable on current machines. As observed by Simon[93], however, it is relatively rare that optimal solutions are actually required in practice, but near-optimal or "satisficing" solutions are usually acceptable for most real-world problems.

A related drawback of the above algorithm schemas is that they must search all the way to a complete solution before making a commitment to even the first move in the solution. The reason is that an optimal first move cannot be guaranteed until the entire solution is

found will be at least as good as any other solution. As a result, searches are run to completion in a planning or simulated environment, and only the first move of the resulting solution is executed in the world.

5.1.2 Two-Player Games

In contrast, heuristic search in two-player games, which we will consider in detail in Chapter 7, adopts an entirely different set of assumptions. The canonical example is a program that plays chess. Since the chess graph is so large, about 10^{40} positions, the idea of searching all the way to checkmate is never entertained, except in the endgame. Rather, the problem solver faces a limited search horizon, and makes suboptimal decisions.

A related characteristic is that in a two-player game, actions must be committed before their ultimate consequences are known. For example, in a chess tournament, moves must be made within a certain time limit, and can never be revoked.

Research on single-agent problems has focused on increasing the size of problems for which optimal solutions can be found, and as a result is severely limited in the size of problems that can be solved. Research in two-player games, however, has concentrated on making the best decisions possible given the limited amount of computation available between moves, and has been spectacularly successful. For example, in 1997 a machine called Deep Blue defeated Garry Kasparov, the human world chess champion [8]. The best checkers[87] and othello[7] players in the world are also computers.

5.1.3 Real-Time Single-Agent Search

The goal of this chapter is to apply the assumptions of two-player games, namely limited search horizon and commitment to moves in constant time, to single-agent heuristic searches.

A limited search horizon may be the result of computational or informational limits. For example, even a moderate-size sliding-tile puzzle, such as the 6×6 Thirty-Five Puzzle, may preclude searching all the way from an initial state to a solution, due to the computation

required. In the ...us vehicle navigation without the benefit of compl..., the search horizon is limited by the range of the ...

The commitm...nstant time is a further constraint. In the real world ...orresponds to executing a physical action, such as moving a chess piece. In our experiments, while actions are not physically executed, they are faithfully simulated. For example, when backtracking occurs, both the original moves and the backtracking moves are counted in the solution lengths, much as the odometer of a vehicle records every mile driven, whether ultimately productive or not. Furthermore, the only information available to the algorithm is information that would be available had the simulated moves actually been executed. For example, the information available to the sensors of a vehicle depends on its location.

5.2 Minimin Lookahead Search

The first step is to decide how to make an individual move decision in real time. In two-player games, this is done by a full-width, fixed-depth minimax search. We specialize the minimax algorithm to the case of a single problem-solving agent, and call the resulting algorithm *minimin* search. At first we will assume that all edges have the same cost.

The minimin algorithm searches forward from the current state to a fixed depth determined by the computational or information resources available for a single move, and applies the heuristic evaluation function to the nodes at the search frontier. Whereas in a two-player game we alternately back up the tree the minimum and maximum of these values, to account for alternate moves among the players, in the single-agent setting, the backed-up value of each interior node is the minimum of the values of its children, since the single agent has control over all moves, and we adopt the convention that lower values are better. Once the backed-up values of the children of the current state are determined, a single move is made in the direction of the lowest-cost child, and the process is repeated. The reason for not moving directly to the frontier node with the minimum value is follow a strategy of least commitment, under the assumption that after committing to the first move, addi-

tic m an expanded search frontier may result in a dif second move than was anticipated by the first se

h proceeds in two quite different, but interleaved m ahead search occurs in a planning mode, where the postulated moves are not actually executed, but merely simulated in the machine. After one complete lookahead search, the best move found would actually be executed in the real world by the problem solver. This is followed by another lookahead simulation from the new current state, and another actual move, etc.

In the more general case where the operators have non-uniform cost, we must take into account the cost of the path from the current state to the frontier, in addition to the heuristic estimate of the remaining cost. To do this we adopt the A* cost function of $f(n) = g(n) + h(n)$. The algorithm looks forward a fixed depth, and returns the minimum $f(n)$ value of each frontier node. An alternative scheme to a fixed-depth search would be to search forward to a fixed $g(n)$ cost. We adopt the fixed-depth algorithm under the assumption that in the planning phase, the computational cost is a function of the number of moves, rather than the actual execution costs of the moves. Yet a third alternative would be to search forward to a fixed $f(n)$ cost, and then choose a frontier node of minimum $h(n)$ value. We still prefer the fixed-depth search, in this case because it allows finer-grain control over the amount of computation per move.

If a goal state is encountered before the search horizon, then the path is terminated and a heuristic value of zero is assigned to the goal. Conversely, if a path ends in a non-goal dead-end before the horizon is reached, then a heuristic value of infinity is assigned to the dead-end node, guaranteeing that that path will not be chosen.

5.2.1 Branch-and-Bound Pruning

An obvious question is whether every frontier node must be examined to find one of minimum cost. If we use only frontier node evaluations, then a simple adversary argument establishes that determining the minimum cost frontier node requires examining every one.

However, if we allow heuristic evaluations of interior nodes, then

substantial prun overall cost function is monotonic non-decreasing. ion allows us to apply branch-and-bound, describec gnificantly decrease the number of nodes examined e decisions made. In particular, if any interior nod ost that equals or exceeds the cost of the best frontier node found so far, we can prune the search below the interior node, because none of its descendents can have a lower cost than that of the best frontier node already found.

To find the best child of the root, we could use the best upper bound from the search of previous children in the search of subsequent children. In practice, however we reinitialize the upper bound to infinity at the beginning of the search of each child. While this results in some loss of efficiency, it is required in order to determine an actual value for each child. Otherwise, all the frontier nodes below a particular child may be cut off, leaving it with no backed-up value. Values for each child are required for RTA*, to be described in section 5.3.

5.2.2 Efficiency of Branch-and-Bound

Figure 5.1 shows a comparison of the total number of nodes generated as a function of search depth for several different sliding-tile puzzles, including the Eight, Fifteen, Twenty-four, and 10×10 Ninety-nine puzzle. The straight lines on the left represent brute-force minimax search with no pruning, and represent branching factors of 1.732, 2.130, 2.368, and 2.790 for the Eight, Fifteen, Twenty-four, and Ninety-nine puzzles, respectively. The curved lines on the right represent the number of nodes generated in a minimin search to the given depth with branch-and-bound pruning using the Manhattan distance heuristic function. In each case, the values are the averages of 1000 random solvable initial states.

One remarkable aspect of this data is the effectiveness of branch-and-bound pruning. For example, if we fix the available computation at one million nodes per move, requiring less than a second of CPU time on current machines, then branch-and-bound extends the reachable Eight Puzzle search depth about 40 percent from 25 to 35 moves, more than doubles the Fifteen Puzzle depth from 18 to 40 moves, and triples the Twenty-four Puzzle depth from 15 to 45 moves. Fixing the amount

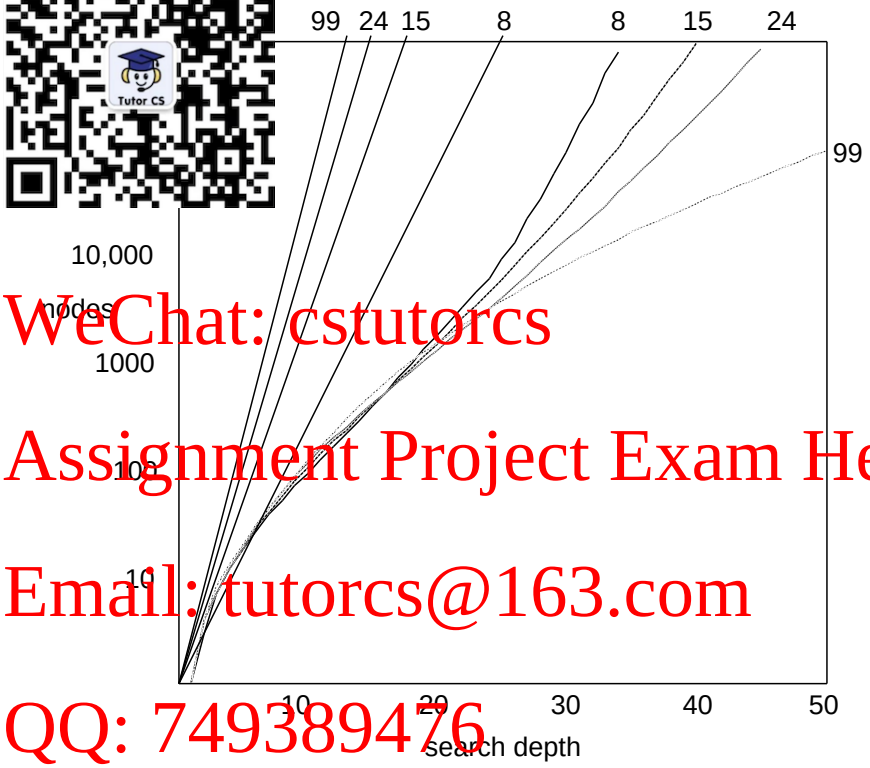


Figure 5.1: Search depth vs. nodes generated for brute-force minimax and depth-first branch-and-bound search.

of computation at 100,000 nodes per move, the reachable Ninety-nine puzzle search depth is multiplied by a factor of five from 10 to 50 moves!

Even more surprising, however, is the fact that given sufficient computation, the search depth achievable with branch-and-bound actually *increases* with increasing branching factor! In other words, we can search significantly deeper in the Fifteen Puzzle than in the Eight Puzzle, and even deeper in the Twenty-four Puzzle, in spite of the fact that the brute-force branching factors are larger. Another way of viewing this is that the effective branching factor is smaller for the larger problems.

5.2.3 An A

Is this paradoxi t of the sliding-tile puzzles, or a more general ph alyze branch-and-bound, we need a model that as ior nodes, and ensures the lower-bound property. Such a model can be found in [37]. It consists of a tree of uniform branching factor and depth, where the edges are assigned a value of zero or one independently with some probability p. The cost of a node in the tree is the sum of the edge costs from the root to that node.

At least at a superficial level, this model characterizes the sliding-tile puzzle. Moving any tile either increases its h value by one, or decreases it by one. Since every move increases the g value by one, the $f = g + h$ value either increases by two or stays the same. If we divide these increments by two, and subtract the h value of the root from every node, we get edge costs of zero or one whose sum represents the f value of a node. Initially at least, the probability of an increment in f is one half.

This model also displays the anomaly mentioned above. For example, if we chose the edge costs to be zero or one with probability .5 each, then branch-and-bound can find a minimum-cost leaf node of a ternary tree of a given depth faster than a binary tree of the same depth!

The complexity of various search algorithms in this model has been analyzed by [102], among others. In particular, it has been shown that if the expected number of zero-cost edges below a node is less than one, then the time complexity of finding a lowest-cost leaf node is exponential in the tree depth, while if the expected number of zero-cost edges is greater than or equal to one, then the problem has only polynomial complexity. The expected number of zero-cost edges below a node is the branching factor times the probability of a zero-cost edge. For example, in a binary tree where edge costs of zero and one occur with probability .5, the expected number of zero-cost edges below a node is $2 \times .5 = 1$, whereas in a ternary tree with the same probabilities, the expected number of zero-cost edges is $3 \times .5 = 1.5$. This suggests that the ternary tree can be searched more efficiently than the binary tree, as is the case.

This also predicts the results for the sliding-tile puzzle, up to a

po verage probability of a zero-cost edge is .5, be-
ca ary tile is equally likely to increase or decrease
th e. The Eight Puzzle branching factor is less than
tw pected number of zero-cost edges less than one.
Th anching factor is slightly greater than two, how-
ever, meaning that the expected number of zero-cost edges is greater than one. This suggests that we can find a lowest-cost frontier node at a given depth in the Fifteen Puzzle faster than in the Eight Puzzle, as our data above shows.

Unfortunately, this analytic model only applies to the sliding-tile puzzles up to a limited depth, since it assumes that the probability of an edge taking on the value of zero or one is the same for every edge in the tree. In practice, however, a string of zero-cost edges, which decrease the Manhattan distance, tends to make a positive-cost edge much more likely. The mismatch between the analytic model and the actual data is supported by the fact that the model predicts that for the Fifteen and larger-size puzzles, the minimum leaf cost should remained bounded with increasing depth, whereas in practice it grows linearly with depth.

In addition to explaining the above anomaly, this analytic model also supports a general analysis of the time complexity of depth-first branch-and-bound[102]. In particular, when the problem complexity is exponential, meaning that the expected number of zero-cost edges is less than one, then DFBnB has the same asymptotic time complexity as best-first search (BFS), which is optimal in time for this problem. This fact, coupled with the linear space complexity of DFBnB compared to the exponential space complexity of BFS, often makes DFBnB the algorithm of choice when applicable.

5.2.4 Minimin Search as a More Accurate Evaluation Function

The effect of minimin search is to compute a backed-up heuristic value for each child of the root node. Thus, this algorithm plus the basic heuristic function, can be encapsulated and viewed as simply a more accurate heuristic evaluation function. This gives rise to an entire fam-

ily of heuristic fu search horizon, that vary in accuracy and comput The greater the search depth, the more accurate th e more expensive it is to compute. For the remaind will adopt this point of view and simply refer to a on, without explicitly mentioning the fact that it may be computed using minimin lookahead search with branch-and-bound.

5.3 Real-Time-A* (RTA*)

So far, we have addressed the issue of how to make individual move decisions, but not how to make a sequence of decisions to arrive at a solution. The obvious approach of simply repeating the minimin algorithm for each decision won't work, since the problem solver may eventually move back to a previously visited state and loop forever. Simply excluding previously visited states won't work either, since all successors of the current state may have already been visited. Furthermore, since decisions are based on limited information, an initially promising direction may appear less favorable after gathering additional information in the process of exploring it, thus motivating a return to a previous choice point. The challenge is to prevent infinite loops while permitting backtracking when it appears favorable.

The basic principle of rationality is quite simple. One should backtrack to a previously visited state when the estimate of solving the problem from that state, plus the cost of returning to that state, is less than the estimated cost of reaching the goal by going forward from the current state. Real-Time-A* (RTA*) is an efficient algorithm for implementing this basic strategy. While the minimin lookahead algorithm is an algorithm for controlling the planning phase of the search, RTA* is an algorithm for controlling the execution phase. As such, it is independent of the planning algorithm chosen.

In RTA*, the merit of a node n is $f(n) = g(n) + h(n)$, as in A*. However, unlike A*, the interpretation of $g(n)$ in RTA* is the actual distance of node n from the current state of the problem solver, rather than from the original initial state. The key difference between RTA* and A* is that in RTA*, the merit of each node is measured relative

to n of the problem solver, and the initial state is irr lates a best-first search using this different cost fu it could be implemented by storing on an Open lis ed states, and every time a move is made, upda f all states on Open to accurately reflect their actual distance from the new current state. Then at each move cycle, the problem solver would select a state with the minimum $g + h$ value, move to it, and again update the g values of all nodes on Open.

The drawbacks of this naive implementation are: 1) the time to make a move is linear in the size of the Open list, since the g value of every node must be updated, 2) it is not clear exactly how to update the g values, and 3) it is not clear how to find the path to the next destination node chosen from Open. All of these problems, however, can be solved in constant time per move, using only local information in the graph.

RTA* maintains a hash table of those nodes that have been visited by a move of the problem solver, together with a stored h value for each of those nodes. At each cycle of the algorithm, the current state n is expanded, generating its neighbors n', and a heuristic function $h(n')$, possibly augmented by lookahead search, is applied to each neighboring state that is not in the hash table. For those states in the table, the stored value of $h(n')$ is used instead. In addition, the cost of the edge to each neighboring state $k(n, n')$ is added to this $h(n')$ value, resulting in an $f(n') = k(n, n') + h(n')$ value for each neighbor of the current state. A neighboring node n' with the minimum $f(n')$ value is chosen for the new current state, and a move to that state is executed. At the same time, the previous state n is stored in the hash table, and associated with it is the second-best $f(n'')$ value. The second-best $f(n'')$ value is the best of the alternatives that were not chosen, and represents the estimated cost $h(n)$ of solving the problem from state n, from the perspective of the new current state n'. If there is a tie among the best values, then the second-best value will equal the best. The algorithm continues until a goal state is reached.

One can construct examples to show that RTA* could backtrack an arbitrary number of times over the same nodes. For example, consider the graph in Figure 5.2, where the initial state is node a, all the edges have unit cost, and the values in each node represent the

original heuristic ost of reaching a goal from those nodes. Since lo s the example more complicated, we will assume is done to compute the h values. Starting at node d are generated and evaluated at $f(b) = g(b) + h($ $) = g(c) + h(c) = 1 + 2 = 3$, and $f(d) = g(d) + h(d) = 1 + 3 = 4$. Therefore, the problem solver moves to node b, and stores with node a the information that $f(c) = h(a) = 3$, the second-best f value. This is its best estimate of the cost of reaching the goal by backtracking through node a. Next, nodes e and i are generated and evaluated at $f(e) = g(e) + h(e) = 1 + 4 = 5$, $f(i) = g(i) + h(i) = 1 + 5 = 6$, and using the stored h value of node a, $f(a) = g(a) + h(a) = 1 + 3 = 4$. Thus, the problem solver moves back to node a, and stores $f(e) = 5 = h(b)$ with node b. At this point, $f(b) = g(b) + h(b) = 1 + 5 = 6$, $f(c) = g(c) + h(c) = 1 + 2 = 3$, and $f(d) = g(d) + h(d) = 1 + 3 = 4$, causing the problem solver to move to node c, storing $f(d) = 4 = h(c)$ with node a. From node c, $f(j) = g(j) + h(j) = 1 + 6 = 7$, $f(k) = g(k) + h(k) = 1 + 7 = 8$, and $f(a) = g(a) + h(a) = 1 + 4 = 5$, causing the problem solver to move back to node a again, and store $f(j) = 7 = h(c)$ with node c. From node a, $f(b) = g(b) + h(b) = 1 + 5 = 6$, $f(c) = g(c) + h(c) = 1 + 7 = 8$, and $f(d) = g(d) + h(d) = 1 + 3 = 4$, causing the problem solver to move to node d and store $f(b) = 6 = h(a)$ with node a. Continuing the example, the problem solver will move from node d back to node a, back to node b, and then to node e.

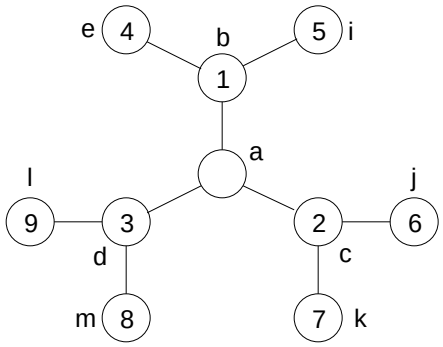


Figure 5.2: Real-Time-A* Example

xample the problem solver repeatedly backtracks ov It is not in an infinite loop, however, since each tir ion, it goes one step further than the previous tir e information about the space. This seemingly irr produced by a rational policy in the presence of a limited search horizon, and a pathological set of heuristic values.

Note that while admissibility of the heuristic function is required for branch-and-bound pruning in minimin lookahead search, RTA* places no such constraint on the values returned by the heuristic function. Of course, the more accurate the heuristic function, the better the performance of RTA* will be, but even with no heuristic information at all, RTA* will eventually find a solution, as the results below will show.

RTA* only needs to store a table of previously visited nodes. The size of this table is linear in the number of moves actually made, since minimin lookahead search saves only the backed-up value of the root node, and not the nodes it generates. Furthermore, the running time is also linear in the number of moves made. The reason for this is that even though any lookahead search requires time that is exponential in the search depth, the search depth is bounded by a constant. Thus, both the time and space complexities of RTA* are linear in the number of moves made. For an actual robot moving in the real world, or a purely computational agent that does a great deal of lookahead for each move, space should not be a significant constraint on the applicability of RTA*.

5.3.1 Completeness of RTA*

Under what conditions is RTA* guaranteed to eventually reach a goal state? One caveat is that the problem space must be finite. In an infinite problem space, unlike A*, RTA* may not find a solution. For example, consider a simple straight-line graph where every node has the same h value, such as zero. Once the algorithm arbitrarily chooses one direction, it will continue in that direction forever, regardless of whether or not the goal lies in that direction. A second caveat is that a goal must be reachable from every state. Otherwise, if there are one-way edges with dead-ends for example, RTA* could end up in a part of

the graph from w goal. Furthermore, there can't be cycles in the gra tive cost, or RTA* could remain in such a cycle fore g its estimate of reaching the goal. Finally, the heu l must be finite, since a barrier of infinite heuristic nd the goal, making it inaccessible to the algorithm. Given these restrictions, RTA* is guaranteed to eventually find a solution if one exists. Note that the only constraint placed on the heuristic evaluation function is that it return finite values. In particular, it need not be admissible.

Theorem 5.1 *In a finite problem space with finite positive edge costs and finite heuristic values, in which a goal state is reachable from every state, RTA* will find a solution.*

Proof: Assume the converse, that there exists a path to a goal state, but that RTA* will never reach it. In order for that to happen in a finite problem space, there must exist a finite cycle that RTA* travels forever, which does not include a goal state. Also, since a goal state is reachable from every state, there must be at least one edge leading away from the cycle. We will show that RTA* must eventually leave any such cycle. At any point in the algorithm, every node in the graph has a value associated with it, either explicitly or implicitly. For the nodes already visited, this will be their stored value in the hash table, and for the unvisited nodes it will be their original heuristic evaluation. Consider an individual move made by RTA*. It reads the value of each of its neighbors, adds the corresponding positive edge costs, stores the second lowest of these values with the current state, and moves to the new state. Thus, the new value stored with the old state must be strictly greater than the value of the state it moves to, since positive edge costs were added to each neighboring value. Now consider a state on the hypothesized infinite cycle with the lowest value. By definition, its value is less than or equal to the value of the state that follows it on the cycle. When the algorithm reaches such a state, it increases its value, since the value written is greater than that of the next state on the cycle that it moves to. Thus, every trip of the algorithm around the cycle increases the lowest value on the cycle. Therefore, the values of all nodes on the cycle increase without bound, since there must be a

mi a finite graph. At some point, since the heuristic va alue of a node on a path that leads away from th than the competing neighbor on the cycle. At th nm will leave the cycle, violating our assumption of us, there can be no infinite loops in a subset of the problem graph. Therefore, in a finite problem space, every node, including a goal node if it exists, must eventually be visited. When the algorithm visits a goal, it will terminate successfully. □

5.3.2 Correctness of RTA*

Since RTA* is making decisions based on limited information, the best we can say about the quality of decisions made by the algorithm is that RTA* makes optimal moves relative to the part of the search space that it has seen so far. This is true in the special case of a tree, but does not extend to graphs with cycles. We begin with some definitions required to precisely state and prove our result. The key difference between these definitions and the standard definitions for A* is that in the case of RTA*, both g and h values are relative to the current state of the problem solver.

As usual, a node is *generated* when the data structure corresponding to that node is created in the machine. A node is *expanded* when all of its children are generated. The search *frontier* is the set of all nodes that have been generated but not expanded since the beginning of the search, including all lookahead phases. This is analogous to the Open list in A*. Let $g_x(n)$ be the cost of the path in the tree from node x to node n. In a tree, there is a unique path between any pair of nodes. Note that $g(n)$ in A* is $g_s(n)$ where s is the initial state. If n is a frontier node, let $h(n)$ be the heuristic static evaluation of node n. If n is an interior node, meaning it has been expanded, and x is the current state of the problem solver, define $h_x(n)$ as the minimum, over all frontier nodes m below node n in the tree rooted at node x, of $g_n(m) + h(m)$. Finally, define $f_x(n)$ as $g_x(n) + h_x(n)$.

Theorem 5.2 *Each move made by RTA* on a tree is along a path whose estimated cost of reaching a goal is a minimum, based on the cumulative search frontier at the time.*

Proof: We v on on the number of moves made by the algorithm urrent state of the problem solver, $h_x(n)$ is the valu ith each previously visited node n in the tree. Con of the algorithm, starting from an initial state s, b stored in the hash table. Minimin search generates a tree rooted at s, terminating in a set of frontier nodes m at the search horizon, possibly including dead-ends and goal states within the horizon. Associated with each frontier node m is a value $f_s(m) = g_s(m) + h(m)$. In the case of a dead-end, this value will be infinity, and in the case of a goal node it will be $g_s(m)$. The backed-up value associated with each child c_i of the start state s is the minimum value of $f_s(m)$ over all frontier nodes m below c_i in the tree rooted at s. Note that branch-and-bound pruning has no effect on the values returned by the lookahead search, but simply computes them more efficiently. The problem solver moves to a child with the minimum value, say c_1 without loss of generality, which is a move along a path whose estimated cost of reaching the goal is a minimum. This move changes the root of the tree from s to c_1, as if the tree was picked up by node c_1. This leaves all parent-child relationships the same, except that c_1 is now the parent of s instead of vice-versa. Therefore, the children of s are now c_i for $i > 1$. At the same time, the second-best $f_s(m)$ value is stored with state s. The second-best value is the minimum value of $f_s(m)$ for all frontier nodes m below c_i for $i > 1$, which is the minimum $f_s(m)$ value of all frontier nodes m below s in the tree rooted at c_1. Thus, the value stored with s is $h_{c_1}(s)$ after the move to c_1.

For the induction step, assume that at a given point in the algorithm, the state of the problem solver is x, and the value stored with each node y in the hash table is $h_x(y)$. For each of the children c_i of x, if c_i is not in the hash table, minimin search will be used to compute $h_x(c_i)$. Otherwise, $h_x(c_i)$ will be read from the table. In either case, $g_x(c_i)$ will be added to $h_x(c_i)$, to produce the correct value of $f_x(c_i)$. Then, by following exactly the same argument as above, replacing s with x, the value stored with x will be $h_{c_i}(x)$ after a move to c_i.

Finally, RTA* always moves from its current state x to the neighbor c_i for which $f_x(c_i)$ is a minimum. This is the same as moving along a path whose estimated cost to the goal is a minimum, given the cumulative search frontier at that point. □

he special case of trees and does not extend to ge cles. Both minimin lookahead search and RTA* mu modified to guarantee locally optimal decisions on not be done in constant time per move[75]. The gu solution (Theorem 1), however, is not specialized to trees, but applies to graphs as well. Thus, these algorithms can be applied to graphs, with the only caveat being that some decisions may not be locally optimal. For example, we have applied the above algorithms to the sliding-tile puzzles, whose problem spaces are graphs with cycles.

5.3.3 Solution Quality vs. Computation

In addition to efficiency and completeness of the algorithm, the quality of solutions generated by RTA* is of concern. The most important factor affecting solution quality is the accuracy of the heuristic function itself. In addition, one would expect that for a given heuristic function, solution quality would increase with increasing search depth. This intuition is supported by experiments on the sliding-tile puzzles using the Manhattan distance evaluation function.

As mentioned above, viewing a heuristic function plus lookahead search as a single, more accurate heuristic function generates a whole family of heuristic functions, one corresponding to each search depth. The members of this family vary in computational complexity and accuracy, with the more expensive functions generally being more accurate. How do we make the best choice from among this family?

Increasing the search horizon increases the computation per move, but decreases the number of moves required to solve the problem. Thus, the choice of how far to look ahead amounts to a tradeoff between the cost of performing the search, and the cost of executing the resulting solution. The optimal search horizon depends on the relative costs of computation and execution, and hence is problem dependent. The fact that computation per move grows exponentially with search depth, but that solution length asymptotically approaches the optimal solution with increasing search horizon, guarantees that the optimal search horizon will remain bounded in any application.

5.4 Lear LRTA*)

We now turn ou oblem of successively solving multiple problem ins problem space with the same set of goals. The key extent performance improvement, or learning, will occur over multiple problem-solving trials. The information saved from one trial to the next is the hash table of heuristic values recorded for previously visited states.

Unfortunately, while RTA* as described above is well suited to single problem-solving trials, it must be slightly modified to accommodate multi-trial learning. The reason is that the algorithm records the second-best estimate with the previous state, which represents an accurate estimate of the previous state, looking back from the perspective of the next state. However, if the best estimate turns out to be correct, then storing the second-best value can result in inflated values for some states. These inflated values will direct the agent in the wrong direction on subsequent problem-solving trials.

This difficulty can be overcome simply by modifying the algorithm to store the best value as the heuristic value of the previous state instead of the second-best value. We call this algorithm Learning-RTA* or LRTA* for short. LRTA* retains the completeness property of RTA* described above (Theorem 5.1), and the same proof is valid for LRTA*. It does not, however, always make locally optimal decisions.

For example, consider the graph fragment shown in Figure 5.3, where the numbers in the nodes represent the initial heuristic values, and all edges cost one unit. Assume that LRTA* starts in state b. From state b, $f(a) = g(a)+h(a) = 1+5 = 6$, and $f(c) = g(c)+h(c) = 1+1 = 2$, causing the algorithm to move to state c, and store $f(c) = 2 = h(b)$ with node b. From state c, $f(b) = g(b) + h(b) = 1 + 2 = 3$, and $f(d) = g(d) + h(d) = 1 + 5 = 6$, causing the algorithm to move back to state b, and store $f(b) = 3 = h(c)$ with node c. At this point, the algorithm has seen both states a and d, and instead of moving back to state b it should have moved to state d, as RTA* would do on this graph. From state b, however, $f(a) = g(a) + h(a) = 1 + 5 = 6$, $f(c) = g(c) + h(c) = 1 + 3 = 4$, and the algorithm moves back to state c again, storing $f(c) = 4 = h(b)$ with state b. From state c, $f(b) = g(b)+h(b) = 1+4 = 5$, $f(d) = g(d)+h(d) = 1+5 = 6$, causing

th back to state b again, and storing $f(b) = 5 = h(c)$ wi his point, with a tie between the heuristic values of LRTA* have a chance of moving on to state a. Nc nces back and forth between states b and c until it or in other words, until their original heuristic values equal or exceed 5, at which point it will escape to the rest of the graph.

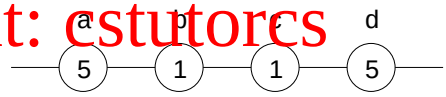


Figure 5.3: Learning Real-Time A* Example

5.4.1 Convergence of LRTA*

An important property that LRTA* does enjoy, however, is that repeated problem-solving trials cause the heuristic values to converge to their exact values. We assume a common set of goal states, and a set of initial states, which are chosen at random. This assures that all possible initial states will actually be visited. We also assume that the initial heuristic values are admissible, or do not overestimate the distance to the nearest goal. Otherwise, a state with an overestimating heuristic value may never be visited and hence remain unchanged. Finally, we assume that ties are broken randomly. Otherwise, once an optimal solution to a goal from some initial state is found, that path may continue to be traversed in subsequent problem-solving trials without discovering additional optimal solution paths. Under these conditions, we can state and prove the following result:

Theorem 5.3 *In a finite problem space with finite positive edge costs, and non-overestimating initial heuristic values, in which a goal state is reachable from every state, over repeated trials of LRTA*, the heuristic values will eventually converge to their exact values along every optimal path.*

Proof: As before, if node n has never been visited, then $h(n)$ is its heuristic static evaluation. Otherwise, $h(n)$ is the heuristic value stored

with node n the ...ed by LRTA*. A node n is *correct* if $h(n) = h^*(n)$, ...ctual cost of an optimal path from node n to a goa... n is visited by LRTA*, its new value $h(n)$ is eq... value of $k(n,n') + h(n')$ over all neighbors n' of n, ...he cost of the edge from n to n'.

The first observation is that when LRTA* visits a node n and updates its heuristic value, it preserves the admissible property of h. In particular, if all the neighboring values of a node are admissible, then its updated value will also be admissible. An optimal path from n must pass through one of its neighbors, say node m. If $h(n)$ is the new value of node n after it is visited by LRTA*, then $h(n)$ by definition is the minimum value of $k(n,m) + h(m)$ over all neighbors m of n, which is less than or equal to $k(n,m) + h(m)$ since m is one of the neighbors of n. This is less than or equal to $k(n,m) + h^*(m)$ since we assume neighboring h values to be admissible, which is equal to $h^*(n)$ since we assume that m is on an optimal path from n. Thus, $h(n) \leq h^*(n)$, and LRTA* preserves admissibility with every move.

Next, consider two adjacent nodes i and j, such that j is the successor of i on a path traversed by LRTA*, and assume that node j is correct. Thus, the new value of node i after updating by LRTA*, $h(i)$, will equal $k(i,j) + h(j)$ since j is the successor of i, which equals $k(i,j) + h^*(j)$ since j is correct. This is less than or equal to the minimum value of $k(i,i') + h(i')$ over all neighbors i' of n_i, since j is the successor of i, which is less than or equal to $k(i,i') + h^*(i')$ since h values are non-overestimating. Therefore, since $h(i)$ is less than or equal to $k(i,i') + h^*(i')$ over all neighbors i' of i, and $h(i)$ is equal to the cost of a path to the goal from node i, namely that through node j, $h(i) = h^*(i)$, and i will be correct after LRTA* updates its value. Thus, if the successor of a given node is correct, the value of the given node will be correct after it is visited by LRTA*.

Since the heuristic value stored with a state by LRTA* is the minimum of the values of its neighbors plus the associated edge costs, the value stored with a state is always strictly greater than the minimum value of its neighbors. Applying the same argument from the proof of Theorem 5.1, this implies that LRTA* will eventually reach a goal state during every problem-solving trial.

h $(n_s, n_2, \ldots, n_g)$ about to be traversed by LRTA* fr n_s to a goal state n_g. Since we assume non-ov ic values, $h(n_g) = 0 = h^*(n_g)$. Working back-wa here will be a non-empty continuous sequence of co incorrect node is reached. Let node j be the last correct node in this continuous sequence, and let node i be the first incorrect node, where j is the successor of i. Once LRTA* reaches node j it will not change its value, or any subsequent value on the path, since they are all correct and their successors are correct. Furthermore, when LRTA* reaches node i, it will correct its value in passing, since its successor, node j, is correct. Thus, every pass of LRTA* over a path to a goal changes another node on the path from incorrect to correct, without disturbing the values of the following nodes. This continues until all nodes along the path are correct.

Assume that there is a node n on an optimal path p from some initial state s to a goal state, and that after an infinite number of trials, its value is not correct. Therefore, node n must only have been visited a finite number of times. Of all nodes on p that have only been visited finitely many times, there must be a node j which is closest to s. It can't be node s, since initial states are chosen randomly and hence every initial state is chosen infinitely often. Therefore, node j must have a predecessor, call it node i, along path p, and by definition, i has been visited an infinite number of times. Thus, at least one of the other neighbors of i, say l, must have been visited an infinite number of times and hence $h(l) = h^*(l)$. Since $h(j)$ does not overestimate, $k(i,j)+h(j) \leq k(i,j)+h^*(j)$, which is less than or equal to $k(i,l)+h(l)$, since j is on an optimal path from i to a goal, which equals $k(i,l)+h^*(l)$, since l is correct. Therefore, j would be chosen as the successor of i at least as often as l since ties are broken randomly, and hence must also have been visited infinitely often. But this violates our assumption that it has only been visited a finite number of times. Thus, there can be no node on an optimal path whose value is not correct after an infinite number of trials. □

5.4.2 Effici*

How long does exact values take? The answer depends on the s em space, the distribution of initial and goal states, and the initial heuristic values. By way of example, we present below the analysis of one special case.

The problem space is a square grid N units on a side, with all edges having unit cost. A single initial state is in one corner and a single goal state in the opposite corner. The initial heuristic values are all zero, to demonstrate that perfect values can be learned with no initial information. When the goal state is reached, the problem solver starts over again at the initial state. Every state in the grid is along some optimal solution path. When learning is complete, the final space contains a regular pattern of diagonal stripes of equal heuristic values.

On the average, each move of each trial increases the value of one state by one unit. This persists even as most of the states reach their correct values, since the algorithm preferentially traverses the incorrect, underestimating paths over the correctly estimating paths. The average final value of a state is N, with the range being between zero and $2N-2$. Since there are N^2 states, the average learning time is $O(N^3)$ moves. This result is borne out by experimental results.

Over the course of the learning time, the average time to reach the goal decreases to the optimal time of $2N-2$. Empirically, solution times very close to this optimal value are actually achieved very early in the learning episode. This result is based on observations of a graphics interface to the system described above. Even though perfect values appear as regular diagonal stripes, long before the learning is complete, an imperfect but very recognizable banding occurs in the remainder of the space. The interpretation of this is that even though only a small percentage of the states have achieved their exact values, the values of a large percentage of the states are consistent with those of their neighbors. This amounts to a relative accuracy that is very effective in driving the problem solver efficiently towards the goal.

5. Conclusions

Existing heuristic search algorithms cannot be used in large applications, due to their computational cost and the fact that they do not commit to an action before its ultimate outcome is known. Minimin lookahead search is an effective algorithm for such problems. Branch-and-bound pruning drastically improves the efficiency of the algorithm without affecting the decisions made. Surprisingly, in some problem domains such as the sliding-tile puzzles, the search depth reachable with this algorithm increases with increasing branching factor. Real-Time-A* (RTA*) efficiently solves the problem of when to abandon the current path in favor of a more promising one, and is guaranteed to eventually find a solution. In addition, RTA* makes locally optimal decisions on a tree. Increasing lookahead search depth increases solution quality, at the cost of more computation per move. Lookahead searches of varying depths can be characterized as generating a family of heuristic functions that vary in accuracy and computational complexity. The optimal level of lookahead depends on the relative costs of simulating vs. executing moves, and hence is a function of the particular application. Finally, Learning-RTA* (LRTA*) is a slight modification of RTA* which preserves its completeness properties while learning exact heuristic values over repeated problem-solving trials.

Chapter 6

Design of Heuristic Functions

Chapters 3, 4, and 5 described algorithms that use heuristic evaluation functions to estimate the cost of reaching a goal from a given state. Here we address the question of how to design such heuristic functions.

6.1 Heuristics from Relaxed Models

The classical answer to this question is that a heuristic function returns the exact cost of reaching a goal in a simplified or relaxed version of the original problem [74]. As we will see below, heuristics derived in this way are both admissible and consistent.

6.1.1 Examples

For example, consider the problem of navigating in a network of roads from an initial location to a goal location. A good heuristic to estimate the cost in the road network between two points is the Euclidean or straight-line distance between the points, which can be computed in constant time. If we remove the constraint that we have to travel along the roads, and allow unrestricted travel between any two points, we get a relaxation of the original problem, which we might call helicopter navigation. The exact cost of an optimal solution to a helicopter navigation problem is the Euclidean distance between the two points. Thus, the exact cost of solving the simplified helicopter navigation problem

is ction for the original road navigation problem.
e, consider the travelling salesman problem (TSP).
W al TSP tour as a graph having three properties:
1) , 2) every node has degree two, an edge entering
th e leaving it, and 3) the graph is connected. In
addition, an optimal TSP tour is a graph of lowest cost that satisfies the above properties. If we delete any of these three constraints, we get a relaxed or simplified problem. In two cases, the optimal solution cost for the simplified problem is a good heuristic function for TSP.

For example, if we remove the constraint that every node has degree two, this leaves the requirement of a connected graph that covers all the nodes. This is called a spanning graph. The optimal solution to this simplified problem is a spanning tree of lowest cost, or a minimum spanning tree (MST). A minimum spanning tree of n nodes can be computed in $O(n^2)$ time, and can be used as a heuristic function for the TSP, for which all known optimal algorithms have exponential complexity.

Alternatively, if we remove the constraint that the graph be connected, we are left with a graph that covers every node, and every node has degree two. Each connected set of nodes in such a graph form a simple loop or cycle, but there can be multiple cycles that aren't connected to each other. Another way to view such a graph is that every city has another city assigned to it, the one that follows it in its cycle. Finding a lowest-cost set of such assignments is known as the assignment problem. Its exact solution can be computed in $O(n^3)$ time for n nodes[70]. The solution to the assignment problem is a heuristic function for TSP. It is particularly effective for the asymmetric TSP (ATSP), in which the cost of the edge from city i to city j is not necessarily the same as the cost of the edge from city j to city i.

If we delete the constraint that the graph has to cover every node, then the optimal solution is the empty graph, which has cost zero and hence is not an effective heuristic function.

Finally, consider a sliding-tile puzzle. One of the constraints on this problem is that a tile can only be slid into the blank position. If we remove this restriction, we allow any tile to be moved to any horizontally or vertically adjacent position. Note that this allows multiple tiles to occupy the same position. The optimal solution cost for an instance

of this relaxed p e Manhattan distance, since every tile must be mo distance to its goal location, and each move can o an adjacent position. Thus, once again we see tha on cost to a simplified version of a problem can be valuation function for the original problem.

6.1.2 The STRIPS Problem Formulation

Can we formalize this idea, and derive such heuristics automatically? To some extent, the answer is yes. Since generating a simplified problem involves manipulating the entire description of a problem, rather than simply individual states, we need a formal problem-description language that is richer than the problem-space graph.

One such language is called the *STRIPS* representation[25]. In this representation, the current state of a problem is described by a set of atomic formulas in predicate calculus. For a sliding-tile puzzle, for example, we can give the location of each tile by a set of predicate formulas of the form $On(x, y)$, meaning that tile x is in location y. In addition, we can specify the position of the blank or empty position by a predicate of the form $Clear(z)$, where z is a particular location. Finally, the structure of the tile locations, and in particular their adjacency, can be specified by a set of predicates of the form $Adj(y, z)$, meaning that location y is horizontally or vertically adjacent to location z. Note that the adjacency predicates Adj are fixed for a given size puzzle, but the On and $Clear$ predicates will differ depending on the current state.

In addition to describing a state of the problem, we also have to describe the legal operators. This is done with a predicate of the form $Move(x, y, z)$, meaning that it is legal to move tile x from location y to location z. In order to define which operators are legal in a given state, and their effect on that state, there are three lists associated with each operator predicate: a *precondition list*, an *add list*, and a *delete list*.

The precondition list tells us what conditions must be true in order for $Move(x, y, z)$ to be a legal move. In particular, tile x must be in location y, $On(x, y)$, location z must be empty, $Clear(z)$, and locations y and z must be adjacent, $Adj(y, z)$. The add list tells us what predicates that were not true before the operator was applied will be true after

it ticular, tile x will be in location z, $On(x, z)$, and lo y, $Clear(y)$. Finally, the delete list tells us what pr rue before the operator is applied will no longer be plied. In particular, tile x will no longer be in lo nd location z will no longer be empty, $Clear(z)$. In general, the delete list will be a subset of the precondition list.

The idea here is that the state is described by a set of predicates that are true at a given point in time. The precondition list is compared to this set to determine what operators can be legally applied, and the application of an operator adds and deletes predicates from the list of true predicates. All predicates that were true before an operator is applied, and that are not on the add or delete lists of the operator, continue to be true.

Once we have formulated our problem in the STRIPS representation, constructing simplified or relaxed problems is easy. We simply remove preconditions on the operators from the precondition list. For example, if we remove the $Clear(z)$ precondition, we can move any tile to any adjacent location, regardless of the position of the blank. This is the relaxed problem described above, for which the exact cost of an optimal solution is the Manhattan distance.

6.1.3 Admissibility and Consistency

The heuristics derived by this method are both admissible and consistent. To see why, consider the effect of removing operator preconditions on the problem-space graph. All the operators that were applicable before the precondition deletion are still applicable. Thus, all previous edges still exist in the new graph. In addition, by deleting some preconditions, we add new operator applications, which is equivalent to adding new edges to the graph. Furthermore, in some cases we add new states as well. For example, by removing the precondition that the destination location of a tile be empty, we effectively add new states where multiple tiles occupy the same location. The key property, however, is that we don't remove any states or edges from the original graph, but simply add new ones.

To see that the resulting heuristic functions are admissible, note that the heuristic is the lowest cost of any path between two states

in the problem-s relaxed problem, while the exact solution cost is y path between the states in the problem-space g problem. Since we only add new states and opera he original graph to the graph of the relaxed prob path in the new graph can only have an equal or lower cost than the lowest-cost path in the original graph. Thus, the heuristic function is admissible, or a lower bound on the optimal solution cost for the original problem.

To see that the heuristic is consistent as well, note that the value of the heuristic is the actual cost of the best solution path in the graph of the relaxed problem. Recall that a heuristic h is consistent if and only if for all nodes n and m, $h(n) \leq c(n,m) + h(m)$, where $c(n,m)$ is the cost of an optimal path from n to m. $h(n)$ is the actual optimal cost of reaching a goal from node n in the graph of the relaxed problem. This cost has to be less than or equal to the cost of going from n to any other node m, plus the actual cost of reaching a goal from node m, with equality for any nodes m on an optimal path to a goal. Thus, any heuristics based on the exact cost of optimal solutions of relaxed problems are guaranteed to be both admissible and consistent.

6.1.4 Automatically Deriving such Heuristics

Given a STRIPS representation of a problem, it is easy to create a description of a relaxed problem, simply by deleting preconditions. While this step is easily automated, automating the entire process of generating heuristic functions is still quite difficult. The hard part is identifying which relaxed problems have the property that their exact solution costs can be efficiently computed, and deriving an efficient algorithm to perform the computation. For example, once we remove the $Clear(z)$ condition from the above description of the sliding-tile puzzles, it is not immediately obvious, at least not to a computer, that the cost of an exact solution to the resulting relaxed problem is easy to compute, any more than it is easy to compute the exact cost of an optimal solution to the original problem. Furthermore, automatically figuring out how to compute what we call Manhattan distance efficiently is also a challenge. The state of the art in this area is represented by [79].

6.2 Database Heuristics

The Manhattan distance heuristic can be efficiently computed because of the fact that the individual tiles do not interact with one another. What makes the sliding-tile puzzle is difficult, and why the Manhattan distance is only a lower bound on actual cost, is that the tiles get in each other's way. By taking into account some of these interactions, we can compute more accurate admissible heuristic functions.

6.2.1 Non-Additive Pattern Databases

Pattern databases[13] are one way to do this, and were originally applied to the Fifteen Puzzle. Figure 6.1 shows the Fifteen Puzzle, with a subset of the tiles labelled, called the *fringe* tiles. For a given state, the minimum number of moves required to get the fringe tiles from their current locations to their goal positions, including any required moves of other tiles, is obviously a lower bound on the number of moves needed to solve the entire puzzle.

Figure 6.1: The Fringe Pattern for the Fifteen Puzzle

This number depends only on the positions of the fringe tiles, and the blank position, but is independent of the positions of the other tiles. While it would be too expensive to calculate the moves needed to solve the fringe tiles for each state in the search, we can precompute all of these values, store them in memory in a pattern database, and

look them up wh during the search. Since there are seven fringe tiles sixteen different locations, the total number of di igurations of these tiles and blank is $16!/(16-8)! =$ ach table entry, we store the number of moves ne nge tiles from their corresponding locations, which takes less than a byte of storage. Thus, we can store the whole table in less than 495 megabytes of memory.

We can compute this entire table with a single breadth-first search backward from the goal state shown in Figure 6.1. In this search, the unlabelled tiles are all considered equivalent, and a state is uniquely determined by the positions of the labelled tiles and the blank. As each configuration of these tiles is encountered for the first time, the number of moves made to reach it is stored in the corresponding entry of the pattern database. The search continues until all entries of the database are filled. Note that this table need only be computed once for a given goal state, and its cost can be amortized over the solution of multiple problem instances with that same goal state.

Once the table is stored, we can use IDA* to search for an optimal solution to a particular problem instance. As each state is generated, the positions of the fringe tiles and the blank are used to compute an index into the pattern database, and the corresponding entry, which is the number of moves needed to solve the fringe tiles, is used as the heuristic value for that state.

Using this pattern database, Culberson and Schaeffer reduced the number of nodes generated to solve the Fifteen Puzzle by a factor of 346, and reduced the running time by a factor of six, compared to Manhattan distance [13]. Combining this with another pattern database, and taking the maximum of the two database values as the overall heuristic value, reduced the nodes generated by about a thousand, and the running time by a factor of twelve.

Rubik's Cube

Non-additive pattern databases were also used to find the first optimal solutions to Rubik's Cube. The standard $3 \times 3 \times 3$ Rubik's Cube, shown in Figure 1.1, contains about 4.3252×10^{19} different reachable states. Of the 27 possible $1 \times 1 \times 1$ cubies, 26 are visible, with one in the

ce[illegible]x cubies in the center of a face rotate, but don't m[illegible]s form a fixed reference framework, disallowing ro[illegible]cube. Of the remaining 20 movable cubies, eight ar[illegible]h three visible faces each, and twelve are on the ed[illegible]e faces each. Corner cubies stay on the corners, and edge cubies stay on the edges. A corner cubie can be oriented in any of three ways, and an edge cubie can be oriented two different ways. Each twist moves four corner cubies and four edge cubies.

The obvious heuristic for Rubik's Cube is a three-dimensional version of Manhattan distance. For each cubie, compute the minimum number of moves required to correctly position and orient it, and sum these values over all cubies. Unfortunately, to be admissible, this value has to be divided by eight, since every twist moves eight cubies. A better heuristic is to take the maximum of the sum of the Manhattan distances of the corner cubies, divided by four, and the Manhattan distances of the edge cubies, divided by four. The expected value of the Manhattan distance of the edge cubies is $22/4 = 5.5$, while the corresponding value for the corner cubies is $12.333/4 \approx 3.08$, partly because there are twelve edge cubies, but only eight corner cubies.

If we consider just the eight corner cubies, they can be in any one of $88,179,840$ different possible permutations and orientations. Using a breadth-first search from the goal state, we can enumerate these states, and record in a table the number of moves required to solve each combination of corner cubies. Since this value ranges from 0 to 11, only four bits are required for each table entry. Thus, this table requires $44,089,920$ bytes of memory, or 42 megabytes, which is easily accommodated on modern workstations. Note that this table only needs to be computed once for each goal state, and its cost can be amortized over the solution of multiple problem instances with the same goal.

The expected value of this heuristic is about 8.764 moves, compared to 5.5 moves for the Manhattan distance of the edge cubies. During an IDA* search, as each state is generated, a unique index into the heuristic table is computed, followed by a reference to the table. The stored value is the number of moves needed to solve just the corner cubies, and thus a lower bound on the number of moves needed to solve the entire puzzle.

We can improve this heuristic by considering the edge cubies as well.

A single pattern database storing the number of moves required to solve all twelve edge cubies takes too much memory, but we can store pattern data for only six of the edge cubies. The number of possible combinations of these twelve edge cubies is 42,577,920. The number of moves required for them ranges from 0 to 10, with an expected value of about 7.668 moves. At four bits per entry, this table requires 21,288,960 bytes, or 20 megabytes. Similarly, we can compute the corresponding pattern database for the remaining six edge cubies.

Unfortunately, the only way to combine these different pattern database heuristics, without overestimating the actual solution cost, is to take the maximum of their individual values. The reason is that every twist of the cube moves four edge cubies and four corner cubies, and moves that contribute to the solution of cubies in one pattern database may also contribute to the solution of the others. The heuristic used for the experiments reported below is the maximum of these three values: all eight corner cubies, and two groups of six edge cubies each. The total amount of memory for all three tables is 82 megabytes. The total time to generate the heuristic tables was about an hour, and this cost is amortized over multiple problem instances with the same goal state. The expected value of the maximum of these three heuristics is about 8.898 moves. Even though this is only a small increase above the 8.764 expected value for just the corner cubies, it results in a significant performance improvement, because the heuristic distribution is more favorable.

IDA*, using the maximum of the three pattern database values described above as the heuristic, was used to find optimal solutions to ten random instances of Rubik's Cube[45]. The median optimal solution length is only 18 moves. One problem instance generated a trillion nodes, and required a couple of weeks to run. With improvements by Michael Reid and Herbert Kociemba, and larger pattern databases, most states can now be solved optimally in a day.

Limitations of Non-Additive Pattern Databases

The main limitation of non-additive pattern databases is that they don't scale up to larger problems. For example, the state space of the Twenty-Four Puzzle is more than 100,000 times larger than that of

Ru he Twenty-Four puzzle contains 25 different po-
sit base covering n tiles and the blank would require
25 s. For example, five tiles plus the blank generate
al sibilities. A pattern database of six tiles and the
bl er 2.3 gigabytes. Furthermore, the values from a database covering only six tiles are smaller than the simple Manhattan distance of all the tiles. Even with multiple databases, the best way to combine them admissibly is to take the maximum of their values, even if the tiles in the different databases don't overlap. The reason is that non-additive pattern database values include all moves needed to solve the pattern tiles, including moves of other tiles.

Instead of taking the *maximum* of different pattern database values, we would like to be able to *sum* their values, to get a more accurate heuristic, without violating admissibility. This is the idea of disjoint pattern databases.

6.2.2 Disjoint Pattern Databases

The easiest way to construct an additive pattern database for the sliding-tile puzzles is to partition the tiles into disjoint groups, such that no tile belongs to more than one group. We then precompute tables of the minimum number of moves of the tiles in each group that are required to solve those tiles. We call the set of such tables, one per group of tiles, a *disjoint additive pattern database*, or a *disjoint database* for short[48]. Then, given a particular state in the search, for each group of tiles, we use the positions of the tiles in the group to compute an index into the corresponding table, retrieve the number of moves required to solve the tiles in that group, and then add together the values for each group to compute an overall heuristic for the given state.

The key difference between these databases and the non-additive databases described in [13], is that their databases include all moves required to solve the pattern tiles, including moves of other tiles not in the pattern set. As a result, given two such databases, even if there is no overlap among their tiles, we can only take the maximum of the two values in an admissible heuristic, because moves counted in one database may be moving tiles in the other database, and hence these

moves would be counted. In a disjoint database, we only count moves of the tiles in the group. While this idea seems remarkably simple, it eluded a number of researchers who worked on this problem [13, 48].

A second difference between these two types of databases is that we don't consider the blank position, decreasing the size of the databases. A disjoint database contains the minimum number of moves over all possible blank positions. While this could also be done in a non-additive database, the position of the blank is less important in a disjoint pattern database.

Manhattan distance is a trivial example of a disjoint database where each group contains only a single tile. While Manhattan distance was initially discovered by hand, it could also be "discovered" automatically by the following process. For each tile in each position, perform a search until it reaches its goal location, in which all other tiles are indistinguishable. A state of this search is uniquely determined by the position of the tile in question and the position of the blank, and only moves of the tile of interest are counted. Since the operators of the sliding-tile puzzle are invertible, we can perform a single search for each tile, starting from its goal position, and record how many moves of the tile are required to move it to every other position. Doing this for all tiles results in a set of tables which give, for each possible position of each tile, its Manhattan distance from its goal position. Since we only counted moves of the tile of interest, and each move only moves a single tile, we can sum the Manhattan distances to get an admissible heuristic.

A non-trivial example of a disjoint database for Fifteen Puzzle is shown in Figure 6.2, where we have divided the tiles into a group of seven and a group of eight. The seven-tile database contains 57,657,600 entries, which range from 0 to 33. The eight-tile database contains 518,918,400 entries, which range from 0 to 38. In neither case is the blank position part of the index to the database. As a general rule, when partitioning the tiles, we want to group together tiles that are near each other in the goal state, since these tiles are most likely to interact with one another.

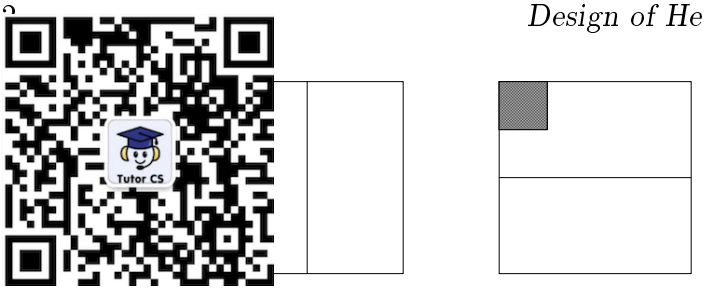


Figure 6.2: Disjoint Database for Fifteen Puzzle and its Reflection

Experiments on Fifteen Puzzle

As a first test, optimal solutions to 1000 random Fifteen Puzzle problem instances were found, using IDA* with a variety of heuristics. The average optimal solution length for these problems is 52.522 moves. Table 6.1 shows the results. The first data column shows the average value of the heuristic function over the 1000 initial states. The second column gives the average number of nodes generated per problem instance. The third column displays the average speed of the algorithm, measured in nodes per second, on a 440 MegaHertz Sun Ultra10 workstation. The fourth column indicates the average running time, in seconds, to optimally solve a problem instance.

Heuristic Function	Value	Nodes	Nodes/sec	Seconds
Manhattan Distance	36.940	401,139,530	7,269,026	55.192
Linear Conflicts	38.788	40,224,625	4,142,193	9.710
Disjoint Database	44.752	136,289	2,174,362	.063
Disjoint+Reflected	45.630	36,710	1,302,233	.028

Table 6.1: Experimental Results on the Fifteen Puzzle

The first row gives results for the Manhattan distance heuristic. The second row is for Manhattan distance enhanced by *linear conflicts*. Historically, the linear-conflict heuristic was the first significant improvement over Manhattan distance[32]. It applies to tiles in their goal row or column, but reversed relative to each other. For example, assume the top row of a given state contains the tiles (2 1) in that order, but in the goal state they appear in the order (1 2). To reverse them, one of the tiles must move out of the top row, to allow the other tile to

pass by, and the e top row. Since these two moves are not counted distance of either tile, two moves can be added to hattan distances of these two tiles without violatin same idea can be applied to tiles in their goal col t, a tile in its goal position may participate in both a row and a column conflict simultaneously. Since the extra moves required to resolve a row conflict are vertical moves, and those required to resolve a column conflict are horizontal moves, both sets of moves can be added to the Manhattan distance, without violating admissibility. The linear-conflict heuristic reduces the number of nodes generated by an order of magnitude, at a cost of almost factor of two in speed, for an overall speedup of over a factor of five compared to Manhattan distance.

The next two rows are for disjoint pattern database heuristics. The third line of the table is for a heuristic which is the sum of the seven and eight-tile database values depicted on the left side of Figure 6.2. We used one byte per entry for efficiency, occupying a total of 550 megabytes, but these databases could be compressed. For example, we could store only the additional moves exceeding the Manhattan distances of the pattern tiles, and separately compute the Manhattan distances during the search. Furthermore, since the parity of the additional moves is the same as that of the Manhattan distance, we could store only 1/2 the number of additional moves.

The heuristic for the fourth line is computed by first computing the heuristic of the third line. We then compute the sum of the seven and eight-tile database values shown on the right side of Figure 6.2. Finally, the overall heuristic is the maximum of these two different sums. Since the two different partitions in Figure 6.2 are reflections of one another, we use the same pair of tables for both databases, and simply reflect the tiles and their positions about the main diagonal to obtain the reflected values.

The disjoint database heuristics provide dramatic reductions both in nodes generated and overall running time. The maximum of the original and reflected database values, shown in the fourth line, reduce the number of node generations by a factor of almost eleven thousand, and the running time by a factor of almost two thousand, compared to Manhattan distance. By contrast, the best non-additive pattern

da d by Culberson and Schaeffer[13], using a similar an enerated almost 10 times more nodes then the dis ran only 12 times faster than simple Manhattan dis

Experiments on Twenty-Four Puzzle

Finding optimal solutions to the Twenty-Four Puzzle is so expensive that it is only practical with the most powerful heuristics. The heuristic used was based on a disjoint pattern database, and its reflection about the main diagonal, shown in Figure 6.3. There is nothing special about this decomposition of the problem, but we chose to group together tiles that are close to each other in the goal state, in the belief that such tiles are more likely to interact with one another. Each group consists of six tiles, requiring 127,512,000 entries each. For a given state, the value from each database is the sum of the number of moves needed to solve each group of tiles from that state. The overall heuristic is the maximum of the values from the original and reflected databases. Since there are only two different-shaped groups, a 2×3 block of tiles, and an irregular block surrounding the blank, we only have to store two different tables, with the remaining values obtained by mapping the tiles and their positions into one of these two tables. The heuristic values for the 2×3 database range from 0 to 35, and for the irregular database they range from 0 to 34. At one byte per entry, the total amount of memory for both tables is 243 megabytes, but these could be compressed as well.

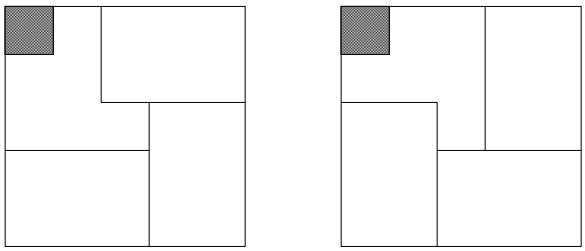


Figure 6.3: Disjoint Databases for TwentyFour Puzzle

Fifty random problem instances were solved optimally with IDA*,

using this heuris[illegible]verage optimal solution length is 100.78 moves. Th[illegible] nodes generated was 360, 892, 479, 671. The program wa[illegible] generates about 2,110,000 nodes per second on a[illegible]n Ultra10 workstation. The running times on in[illegible]anged from 18 seconds to almost 23 days, with an average of two days per problem. The average Manhattan distance for a random sample of 10,000 initial states is 76.078 moves, while for the disjoint database heuristic it is 81.607 moves.

6.2.3 Pairwise and Higher-Order Distances

The main drawback of disjoint databases is that they don't capture interactions between tiles in different groups of the partition. This requires a different approach. Consider a table for a sliding-tile puzzle which contains for each pair of tiles, and each possible pair of positions they could occupy, the number of moves required for both tiles to move them to their goal positions. We call these values the *pairwise distances* of the tiles. For most pairs of tiles in most positions, their pairwise distance will equal the sum of their Manhattan distances. For some tiles is some positions however, such as two tiles in a linear conflict, their pairwise distance will exceed the sum of their Manhattan distances. Given n tiles, there are $O(n^4)$ entries in a complete pairwise distance table. As with all pattern databases, this table is only computed once for a given goal state.

Given a complete pairwise distance table, and a given state of the puzzle, how do we compute the heuristic value for that state? We can't simply sum the table values for each pair of tiles, since each tile participates in many pairs, and the sum of all values will grossly overestimate the optimal solution length. Rather, we partition the n tiles into $n/2$ non-overlapping pairs, and then sum the pairwise distances for the pairs we chose. In order to get the most accurate admissible heuristic, we want to choose a set of non-overlapping pairs for which the sum of their pairwise distances is maximized. For each state of the search, the corresponding set of pairs may be different, requiring this computation to be performed for each heuristic evaluation.

To understand this problem more generally, for a given state, define a graph where each tile is represented by a node, and there is an edge

be nodes, labelled with the pairwise distance of the co that state. The task then is to choose a set of ed so that no two chosen edges are incident to the sa he sum of the weights of the edges is maximized. Th *ximal weighted matching* problem, and can be solved in $O(n^3)$ time[70], where n is the number of nodes, or tiles in this case.

This technique can obviously be extended to triples of tiles, or even higher-order distances. Unfortunately, for triples the corresponding three-dimensional matching problem is NP-Complete [28], as is higher-order matching. This idea is the basis of the heuristic function used in [52], although it was implemented in a domain-specific fashion.

The advantage of this approach is that it produces more accurate heuristic values, and hence fewer node generations, compared to disjoint pattern databases. The reason is that if two tiles are in different groups in a disjoint database, their interactions are not captured by the disjoint database, but they can be captured by a pairwise-distance pattern database. The disadvantage of this approach is that to compute the heuristic value of each state in the search requires solving a matching problem, which is much more expensive than simply adding the values for each group in a disjoint database.

Extensive experiments were performed with pairwise distances and triple distances on both the Fifteen and Twenty-Four puzzles. These heuristics resulted in fewer node generations, but the overhead of computing them resulted in longer running times, compared to the disjoint databases.

6.2.4 Compressed Pattern Databases

The more components of a problem that are taken into consideration, the more accurate the resulting pattern database heuristic is. What limits the number of components that can be included is the memory to store the pattern database. Consider the 4-Peg Towers of Hanoi problem, for example. We can construct a pattern database heuristic by considering the positions of a subset of the discs in the problem. For example, the number of different possible configurations of 15 discs is $4^{15} = 2^{30} \approx 10^9$. The maximum number of moves needed to solve any

state of the 15-c moves, and hence each value can be stored in a b e each disc can be on one of four possible pegs, an y one peg must be in sorted order, we can uniquely of 15 discs by 15 two-bit fields, or a bit string o ore, each such string represents a legal configuration of 15 discs. Therefore, we can use such a bit string as an index into the database, eliminating the need to store the state with each entry. Thus, a 15-disc pattern database will occupy exactly a gigabyte of memory.

Given a problem with more than 15 discs, we can build a simple pattern database using any subset of the 15 discs. However, as we saw in chapter 3, we can actually perform a complete breadth-first search for up to 22 discs of the 4-peg Towers of Hanoi problem, using magnetic disk to temporarily store the nodes during the search. This allows us to construct up to a 22-disc database compressed to the size of a 15-disc pattern database as follows [23]. We initialize a 15-disc database in memory to all empty, except for the entry corresponding to the goal state, which is initialized to zero. Then we perform a complete breadth-first search of the 22-disc problem space. For each state generated, we compute an index based on only 15 of the discs, such as the 15 largest discs for example. If the corresponding entry in the pattern database is empty, we store the current search depth in it, and otherwise leave the value unchanged. When the search completes, each entry in the database will correspond to a particular configuration of the 15 largest discs, and will contain the minimum depth at which this configuration was reached, or equivalently, the minimum distance from the goal of all states of the 22-disc problem with that particular configuration of the 15 largest discs. In general, this value will be larger and more accurate than the value from a simple 15-disc database, because it includes moves of the 7 smallest disks as well. Empirically, such compressed databases are much more effective than simple uncompressed databases of the same size.

6.2.5 Multiple Goal Pattern Databases

For some problems, rather than a single goal state, we may have multiple goal states, and are interested in a shortest path to any goal state.

Fo lance driver may want the shortest route to the clo traditional heuristic functions, to estimate the sh closest goal state would require computing the he rate goal state, and taking the minimum of all th databases provide a much more efficient way to do this [49]. The idea is that instead of starting the breadth-first search to construct the pattern database with just a single goal state in the FIFO queue, we seed the queue initially with all possible goal states, and then simply run a standard breadth-first search, constructing the database as before. Thus, the depth at which we encounter each state for the first time will be the length of a shortest path from that state to any of the goal states. This allows us to estimate the shortest path to any goal state with only a single lookup in the pattern database for each node generated in the search.

Chapter 7

Two-Player Perfect-Information Games

Building a computer program to play a game such as chess was perhaps the first AI challenge, and in fact predates the term artificial intelligence. It is also a classic application of heuristic search algorithms. In May 1997, the chess machine Deep Blue defeated Garry Kasparov, the reigning world champion, in a six-game match[8].

It is easy to see why computer chess was a natural domain for studying artificial intelligence from its earliest days. First, the game is well-structured. There is a small, discrete board, six different types of pieces with well-defined move and capture rules, and a clear goal of capturing the opponent's king. Second, it is a game of perfect information, meaning there is no chance element and all information is available to both players. In spite of this, it is a very difficult to master, and many people devote their entire lives to studying the game. Indeed, chess has always been a prototypical game of the intellect, clearly embodying what we at least used to think intelligence was all about. We now know that other tasks, such as vision, speech, and language production and understanding, are computationally more difficult, but that was not at all obvious in the early days of the field, particularly since small children perform these tasks effortlessly. Finally, it is probably no coincidence that early programmers and AI researchers were often amateur chess players as well, given that computer programming and playing chess are not entirely dissimilar activities. Many of these features are true of

ot

7 story of Computer Chess

We begin with a brief history of computer chess. No such history would be complete without starting with Baron Wolfgang von Kempelen of Austria, in 1769. He built a chess automaton, widely exhibited by Johann Nepomuk Maelzel and known as Maelzel's Chess Machine, which appeared to automatically move the pieces on a board on top of the machine, and played excellent chess[9]. Surprisingly, the essential puzzle of the machine was not solved until 1836, by none other than Edgar Allen Poe, who demonstrated that the moves of the machine were made by a human concealed inside of it[76]. See [17] for a brief description of this saga.

The first serious paper on computer chess was written by Claude Shannon[91], the father of information theory. He laid out the basic theory, described in this chapter, of minimax search with a heuristic static evaluation function, and even anticipated the need for more selective search algorithms. The next most significant event was the invention of alpha-beta pruning, also described below, usually attributed to John McCarthy, around 1956. Surprisingly, although alpha-beta pruning was used in early programs such as Samuel's checkers player[85] and Newell, Shaw, and Simon's chess program[65] in the late 1950's, it wasn't until several years later that anyone fully described the algorithm in a paper[35]. It seems likely that the incredible power of this technique, as we will see below, was not appreciated at the time. The next significant milestone was the development of Belle[12] around 1982 by Condon and Thompson. Ken Thompson was also the co-inventor of C and Unix. Belle was the first machine whose hardware was specifically designed to play chess, in order to achieve greater speed and search depth. Finally, the Deep Blue machine was the first chess machine to defeat the human world champion, Garry Kasparov, in a six-game match in May 1997[8]. This was an achievement that was predicted for about ten years in the future, almost continuously since the late 1950s. It turned out to be harder than most people thought, despite the dramatic increase in the power of computers over that time.

7.2 Othe

Before turning o lgorithms underlying modern two-player game pro rvey the state of the art in other games.

The best checkers player in the world is a program named Chinook[87], developed by Jonathan Schaeffer and his colleagues at the University of Alberta. The last human to seriously challenge the program, Marion Tinsley, was the best player in the world for almost 40 years, losing only 7 games in competition over that time. During a championship match between Tinsley and Chinook in August 1994, Tinsley resigned due to illness, and unfortunately died seven months later. The entire story is well-told in Schaeffer's excellent book[87].

Othello programs have long been recognized as being better than the best humans. One possible reason is that in Othello, a large number of pieces change hands on each move, making it difficult for people to play this game well. The best Othello program in the world today is Logistello[7], written by Michael Buro. The best chess, checkers, and Othello programs use the standard techniques of alpha-beta minimax search described below.

Unlike the above games, backgammon includes a roll of the dice, introducing a random element. The best backgammon program is called TD-gammon[98], written by Gerry Tesauro. It uses a machine-learning technique called temporal-difference learning to learn an evaluation function. It does relatively little search, since the roll of the dice makes the branching factor quite large. TD-gammon is comparable to the best human players today.

In addition to a random element, card games such as bridge and poker, and word games such as Scrabble, introduce hidden information, in the form of cards or tiles that are known to one player and not to others. The best Scrabble program in the world, called Maven, was written by Brian Sheppard[92], and is comparable to the best human players. The best bridge program is called GIB, written by Matt Ginsberg[31]. It is not yet competitive with the best human players. Poker programs are even worse relative to their human counterparts[4]. In addition to the complexities of betting, poker also involves a strong psychological element, at least when played by people.

e the best computer programs don't even play at th ateurs yet. One reason for this is the very large br h is initially 361. As a result, computers can't se must rely on other techniques [63].

le theory to explain the uneven performance of computers against humans across different games is that it depends on the branching factor, at least for the perfect-information games. Since the main technique in the computer arsenal is search, the smaller the branching factor, the deeper computers can search, and the better they play against their human opponents. For example, in games like checkers, with a branching factor of about four, and Othello, with a branching factor less than ten, programs outperform the best humans. In chess, with a branching factor of about 36, the best programs just recently became competitive with the best humans. Finally, in games like Go, with a branching factor in the hundreds, computers play very poorly relative to their human counterparts. Backgammon however, with its large branching factor, is a counterexample to this theory.

7.3 Brute-Force Search

We begin by considering a purely brute-force approach to game playing. Clearly this will only be feasible for small games, but provides a basis for our further discussions.

Consider once again the game of Nim, first described in Chapter 1. It is played with two players and a pile of stones. Each player alternately removes one stone or two from the pile, and the player who removes the last stone wins the game. Figure 7.1 shows an augmented version game tree for the five-stone version of the game, shown in Figure 1.7. If the square player can force a win from a given position, the corresponding node is enclosed in a large square, and similarly for the circle player. The heavy lines show a winning strategy for the first player to move.

This information is computed from the bottom up, as follows. We begin by marking the terminal or leaf nodes at the bottom of the tree with the player who removed the last stone. For example, the deepest node in the tree is marked with a square because the square player took the last stone. Next, we mark those nodes immediately above the

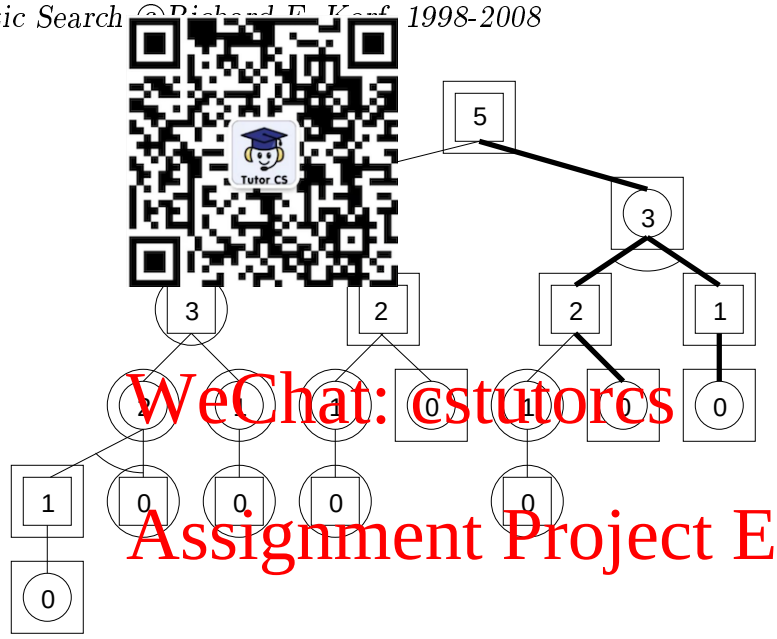


Figure 7.1: Solved Game Tree for Five-Stone Nim

leaf nodes which only have a single move available. For example, at the left-most 1 node in the tree, the game isn't over, but the only legal move is for the square player to take the last stone, and win the game. Thus, from that position, the square player can't help but win.

A more interesting case is the leftmost 2 node in the tree, where it is circle's turn to move. If circle takes 1 stone, leaving 1, then square will win. Alternatively, if circle takes 2 stones, leaving none, then circle wins. We assume that circle will play correctly, and since it is circle's turn to move, we mark this node with a circle, meaning that circle can win the game if it plays correctly.

Finally, consider the topmost 3 node in the tree, the right child of the root, where again it is circle's turn to move. The two children of this node are marked as forced wins for square. Thus, even though it is circle's turn to move, regardless of what circle does, square can force a win, and hence this node is marked as a forced win for square.

The values of all internal nodes of the tree, including the root, are computed in the same way, assuming that both players will play perfectly. The root node is labelled as a forced win for square, and the heavy lines indicate the winning strategy, or the solution subtree of the

Al[illegible] winning strategy is to take two stones initially, an[illegible] what circle does, square can take the remaining sto[illegible] me.

[illegible] described this algorithm as if it were based on a br[illegible] practice the search would be performed depth-first to save memory. This is possible because the value of each internal node can be computed from the values of its children.

In a game such as tic-tac-toe, chess, or checkers, there are also drawn positions. Thus, each node is labelled win, lose, or draw, from the perspective of a particular player. The player to move is assumed to choose a winning move if one is available, and choose a drawing move only if no winning move is available. The root node would also be labelled win, lose, or draw, meaning that either one player can force a win, or either player can force a draw. In fact, we know from the famous *minimax theorem*[100] that every two person zero-sum game is a forced win for one player, or a forced draw for either player, and that in principle these optimal minimax strategies can be computed. For example, performing this algorithm on tic-tac-toe results in the root being labelled a draw, meaning either player can force a draw by playing perfectly.

Since tic-tac-toe has less than $9! = 362,880$ nodes in its game tree, this algorithm can easily be implemented to play this game perfectly. Checkers is estimated to include about 10^{20} positions, however, and chess is believed to include about 10^{40} positions, meaning that brute-force search is out of the question for any non-trivial game.

7.4 Heuristic Evaluation Functions

The solution to this dilemma, as recognized by Shannon[91] in 1950, is to use a heuristic static evaluation function to estimate the merit of a position when the final outcome has not yet been determined. The simplest example of such a function for chess is the relative material on the board. We first count the number of white pieces of each type, and multiply them each by their relative strength. For example, the classic values are that a queen is worth 9 points, a rook is worth 5 points, bishops and knights are worth 3 points, and a pawn is worth

1 point. We do[illegible]on for the black pieces, and then subtract the wei[illegible]e black player from the weighted material of the v[illegible]sulting material evaluation gives a rough estimate c[illegible]th of the position for each player. Large positive v[illegible]ous for white, and negative values of large magnitude are beneficial to black. Henceforth, we will refer to the players as MAX and MIN, since MAX is trying to maximize the evaluation function score, at least as an intermediate goal, and MIN is trying to minimize it.

As in the case of the single-agent heuristic functions we considered earlier, a heuristic static evaluation function for a two-player game is a function from a state to a number. Although we have a precise definition of the meaning of a single-agent heuristic function, namely that it estimates the cost of an optimal path from the given state to a goal node, there is no such generally agreed upon definition of the semantics of a two-player evaluation function. For one thing, the goal of a two-player game is to reach a winning state, but the number of moves required to get there is not important. The best we can do is an informal statement that it estimates the relative merit of the position for the two players.

The material calculation is the most important component of a real heuristic static evaluation function for a game like chess, but it is not the only one. In addition to material, other considerations such as king safety, center control, pawn structure, mobility, etc., all must be taken into account. These other features must be reduced to numbers, and they must be combined with appropriate weights in an overall evaluation function. This is normally done by a human expert in the domain, such as a chess grandmaster, but there also exist automatic techniques for learning such functions, which we will discuss in Chapter 11.

Given a heuristic static evaluation function, it is straightforward to write a program to play a game. From any given position, we simply generate all the legal moves, apply our static evaluator to the position resulting from each move, and then move to the position with the largest or smallest evaluation, depending on whether we are the maximizer or minimizer, respectively.

For example, consider the well-known game of tic-tac-toe. Any

st ld first detect whether the game is over. If X is function should return ∞ if there are three $X's$ in there are three $O's$ in a row. This guarantees th ose a won position if available, and avoid a lost po le. The hard part is how to evaluate all the other positions.

Nilsson[68] suggests the following evaluation function for tic-tac-toe. Count the number of different rows, columns, and diagonals occupied by X, and subtract the number of rows, columns, and diagonals occupied by O. In the tic-tac-toe board, there are three rows, three columns, and two diagonals. If X makes the first move in the center of the board, he occupies 4 lines, while O occupies none yet. Thus the evaluation of that position would be $4 - 0 = 4$, suggesting that it is advantageous for X. Figure 7.2 shows all possible first moves for X, taking into account symmetry, and the corresponding evaluations of the resulting board positions. Since the move to the center has the largest evaluation, X would move there, according to our algorithm.

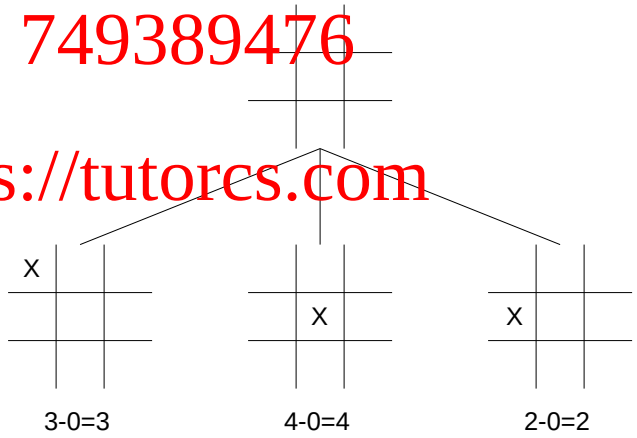


Figure 7.2: First moves of tic-tac-toe

Note that this algorithm is extremely efficient, requiring time that is only linear in the number of legal moves. Its drawback is that it is very short-sighted, since it only considers the immediate consequences of each move. If we used such an algorithm to play chess with a material evaluation function, it could easily be defeated by offering unequal piece

trades, for exam computer a move that captures a pawn, but whicl ure of its queen on the following move. Obviously ead deeper.

7.5 Minimax Search

Figure 7.3 shows a depth-two game tree for tic-tac-toe, where we have evaluated the leaf nodes at the bottom of the tree. For the move to the center, we have shown both possible responses, but for the other two moves we have only included the evaluations of the resulting positions.

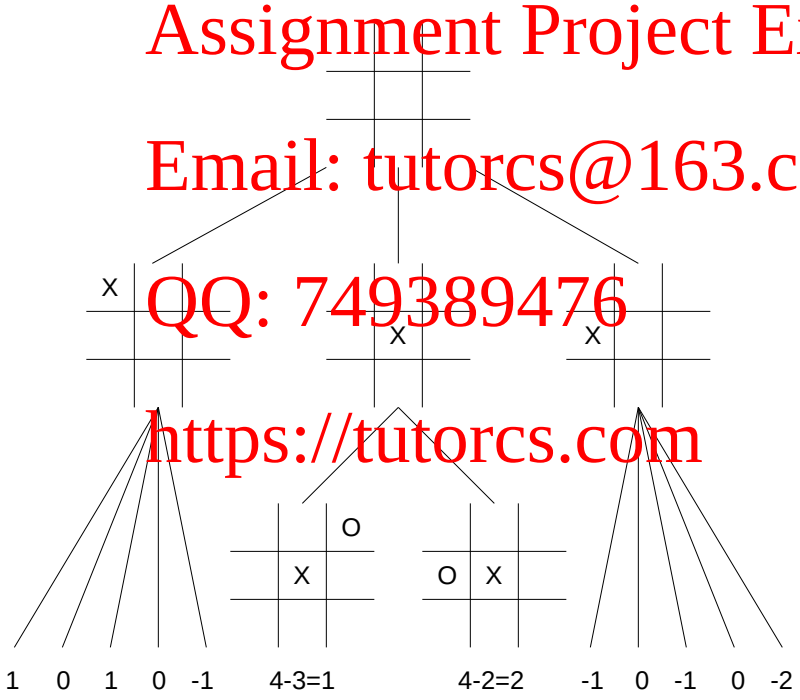


Figure 7.3: First two moves of tic-tac-toe

The question now is what move to make, given this additional information. A naive answer is to move to the center, because this leads toward the largest possible leaf evaluation, a score of two. While this is the correct answer, the reasoning is wrong, because it doesn't take into account the fact that after $X's$ first move, O gets to move. Being

th ume that O will move to the position with the sm e, among its choices. Thus, if X moves to the ce cted to move to the corner, resulting in a score of ve by X. Similarly, if X moves to the corner, O nd by moving toward the minimum score below that node, which is -1. Finally, if X moves to the side, O will probably move to the position with a score of -2. Thus, the *backed-up* score of an initial move to the center, corner, or side, is 1, -1, and -2, respectively, the minimum values below each of those nodes. X should still move to the center, because its backed-up score is 1, the maximum of the three backed-up scores.

In general, given a heuristic static evaluation function, an obvious algorithm is to search as deeply as possible given the computational resources of the machine and the time constraints on the game. Then, the nodes at the search frontier are evaluated by the heuristic function. The values of interior nodes are computed by the minimax rule: at nodes where MIN is to move, the backed-up value is the minimum of its children's values, and at nodes where MAX is to move, the backed-up value is the maximum of the children's values. Finally, a move is made to a child of the root with the largest or smallest backed-up value, depending on whether MAX or MIN is moving, respectively. This algorithm is called *minimax search*.

Note again that while the above description has a breadth-first flavor, the algorithm is actually implemented as a single depth-first search, to save memory. This is possible because there is a limited search depth, and the backed-up value of a node is a function only of its children.

Figure 7.4 shows an example of a depth-four binary minimax tree, where square nodes represent MAX's moves, and circle nodes represent MIN's moves. The numbers in the leaf nodes are the heuristic static evaluations of those nodes, and the numbers in the interior nodes are the backed-up values computed recursively from the leaf values by the minimax rule. This is an abstract game tree, which doesn't represent any particular real game.

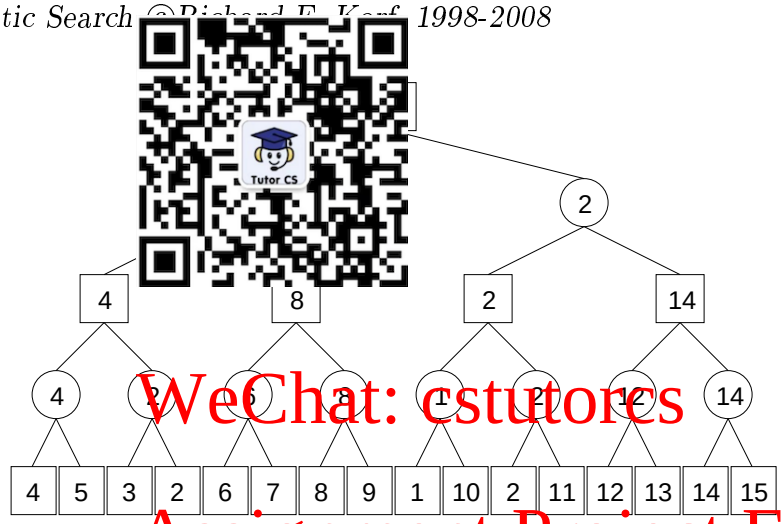


Figure 7.4. Example Minimax Tree

7.6 Alpha-Beta Pruning

One of the most elegant of all AI search algorithms is alpha-beta pruning. It was in use in the late 1950s, but a thorough treatment of the algorithm can be found in [39]. The idea, similar to branch-and-bound, is that the minimax value of the root of a game tree can be determined without examining all the nodes at the search frontier.

Figure 7.5 shows an example of alpha-beta pruning, indicated by the heavy lines, using the same tree as in Figure 7.4. At the square nodes MAX is to move, while at the circular nodes it is MIN's turn. The search proceeds depth-first to minimize the memory required, from left to right, and only evaluates a node when necessary. First, nodes e and f are statically evaluated at 4 and 5, respectively, and their minimum value, 4, is backed up to their parent node d. Node h is then evaluated at 3, and hence the value of its parent node g must be less than or equal to 3, since it is the minimum of 3 and the unknown value of its right child. The value of node c must be 4 then, because it is the maximum of 4 and a value that is less than or equal to 3. Since we have determined the minimax value of node c, we do not need to evaluate or even generate the brother of node h.

Similarly, after statically evaluating nodes k and l at 6 and 7, respectively, the backed up value of their parent node j is 6, the minimum

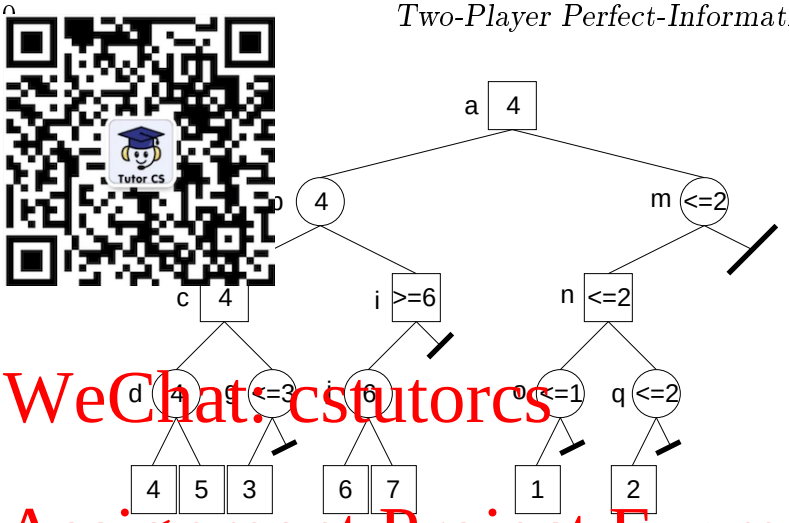


Figure 7.5: Alpha-Beta Pruning Example

of these values. This tells us that the minimax value of node i must be greater than or equal to 6, since it is the maximum of 6 and the unknown value of its right child. Since the value of node b is the minimum of 4 and a value that is greater than or equal to 6, it must be 4, and hence we achieve another cutoff.

The right half of the tree shows an example of *deep pruning*. After evaluating the left half of the tree, we know that the value of the root node a is greater than or equal to 4, the minimax value of node b. Once node p is evaluated at 1, the value of its parent node o must be less than or equal to 1. Since the value of the root is greater than or equal to 4, the value of node o cannot affect the root value, and hence we need not generate the brother of node p. A similar situation exists after the evaluation of node r at 2. At that point, the value of node o is less than or equal to 1, and the value of node q is less than or equal to 2, hence the value of node n, which is the maximum of the values of nodes o and q, must be less than or equal to 2. Furthermore, since the value of node m is the minimum of the value of node n and its brother, and node n has a value less than or equal to 2, the value of node m must also be less than or equal to 2. This causes the brother of node n to be pruned, since the value of the root node a is greater than or equal to 4. Thus, we computed the minimax value of the root of the tree to be

4, by generating n leaf nodes in this case.

Below we give a description of alpha-beta pruning. It is expressed as two mutually recursive and entirely symmetric functions. MAXIMIN assumes that the root node is a maximizing node, and MINIMAX assumes that the root node is a minimizing node. V(N) is the heuristic static evaluation of node N. Alpha is the greatest lower bound on the minimax value of all the ancestors of node N, and beta is the least upper bound on the minimax value of the ancestors of N. Each function returns the backed-up minimax value of node N if it lies within the alpha and beta bounds.


```
MAXIMIN (node: N, lowerbound: alpha, upperbound: beta)
IF N is at the search depth, return V(N)
FOR each child Ni of N
   value := MINIMAX(Ni, alpha, beta)
   IF value > alpha, alpha := value
   IF alpha >= beta return alpha
Return alpha

MINIMAX (node: N, lowerbound: alpha, upperbound: beta)
IF N is at the search depth, return V(N)
FOR each child Ni of N
   value := MAXIMIN(Ni, alpha, beta)
   IF value < beta, beta := value
   IF beta <= alpha, return beta
Return beta
```

7.6.1 Performance of Alpha-Beta

Since alpha-beta performs a minimax search while pruning much of the tree, its effect is to allow a deeper search with the same amount of computation. This raises the question of how much does alpha-beta improve performance? The best way to characterize its efficiency is in terms of its *effective branching factor*, which is the d^{th} root of the number of frontier nodes that must be evaluated in a search to depth d.

alpha-beta pruning depends upon the order in wh untered at the search frontier. For any set of fro here exists an ordering of those values such that alp erform any cutoffs at all. For example, if the tre valuated from right to left, no cutoffs occur. In general, if the children of MAX nodes are generated in increasing order of their backed-up values, and the children of MIN nodes are generated in decreasing order of their backed-up values, then all frontier nodes must be evaluated, and the effective branching factor is b, the brute-force branching factor.

Conversely, if the largest child of a MAX node is generated first, and the smallest child of a MIN node is generated first, then every possible cutoff is realized. In that case, the effective branching factor is reduced from b to $b^{1/2}$, the square root of the brute-force branching factor. For example, since the branching factor of chess is about 36 in the midgame, in the best case alpha-beta reduces the effective branching factor to about 6. Another way of viewing the perfect ordering case is that for the same amount of computation, one can search twice as deep with alpha-beta pruning as without. Since the search tree grows exponentially with depth, doubling the search horizon is a dramatic improvement.

In between worst-possible ordering and perfect ordering is random ordering, which is the average case. Under random ordering of the frontier nodes, alpha-beta pruning reduces the effective branching factor to approximately $b^{3/4}$[72]. This means that one can search 4/3 as deep with alpha-beta, yielding a 33% improvement in search depth.

7.7 Additional Enhancements

While alpha-beta minimax search with heuristic static evaluation is the main technique behind high-performance two-player perfect-information game programs, there are a large number of additional enhancements that have been developed over the years to improve performance with limited computation. We briefly discuss the most important of these below.

7.7.1 Node

There is a large est-case effective branching factor for alpha-beta, l e case of $b^{3/4}$. In actual practice, however, its perf ser to the best case because of *node ordering*. The id is that instead of generating the tree left-to-right, we can reorder the tree based on the static evaluations of the interior nodes. In order to preserve the linear-space complexity of depth-first search, only the immediate children of each node are reordered after the parent node is fully expanded. In particular, the children of MAX nodes are searched in decreasing order of their static values, while the children of MIN nodes are searched in increasing order of their static values. While node ordering with a perfect heuristic would guarantee best-case performance, in practice this technique often generates trees that are only 50% larger than best-case trees[22].

7.7.2 Iterative Deepening

Another important idea is iterative deepening, which was first used in the context of two-player games, to solve the problem of where to set the search horizon[94]. In a tournament game, there is a limited amount of time allowed for moves. Unfortunately, it is very difficult to accurately predict how long it will take to perform an alpha-beta search to a given depth. If we guess a search horizon that is too shallow, we don't get the benefit of a deeper search, even if there is sufficient time. Even worse, if we guess too deep, we have to make a move before the search is completed. Since the search is performed depth-first, this usually means that we have extensive information on the merit of some moves, and almost no information about other moves. The solution to this problem is to perform a series of searches to successively greater depths. When time runs out, the move recommended by the last completed iteration is made.

Note that iterative deepening can be combined with node ordering to improve the pruning efficiency of alpha-beta. In particular, instead of using the static heuristic values of nodes to order them, we can use their backed-up values from the last iteration to order the nodes for the next iteration.

7.[illegible]ce

Th[illegible]ant idea in the development of two-player game pr[illegible]. The notion of quiescence is that the heuristic sta[illegible]d not be applied to positions whose values are unstable, such as those occurring in the middle of a piece trade. In those positions, a small secondary search is conducted until the static evaluation becomes more stable. The nodes at the end of the quiescence search are evaluated, and those values are backed up the tree. In games such as chess or checkers, one simple way to do this is to always explore any capture moves one level deeper.

7.7.4 Transposition Tables

The search graphs of most games contain multiple paths to the same node, often reached by making the same moves in a different order. For example, at depth two in Figure 7.1, there are two nodes representing the state where there are two stones remaining, and the square player is to move. One is reached by first taking two stones and then one, while the other is reached by the same moves made in the opposite order. This situation is referred to as a transposition of the moves.

For efficiency, it is important to detect when a state has already been searched, to avoid researching it. Since alpha-beta minimax is a depth-first search, all such common states cannot be detected without using exponential space. A partial solution is to save a table of some previously generated game states, together with their backed-up minimax values, in what is called a *transposition table*. Whenever a new state is generated, if it's in the transposition table, its stored value is used instead of searching the tree below the node. One strategy for managing the scarce memory resource of the transposition table is to store the shallowest nodes in the tree, on the grounds that detecting duplicate states among them will result in the greatest savings, since the subtrees below such nodes are the largest.

7.7.5 Oper

Most board gam the same initial state. Thus, there is really only one itions near the root occur often in different games. tial moves could be computed by deep searches pe practice, however, most programs use human expertise to develop a table of initial moves known as an *opening book*. This is not unlike the play of human experts, who rely on memorized opening moves acquired from years of study.

7.7.6 Endgame Databases

Near the end of the game, large tables are also useful. For example, the Chinook checkers program owes a great deal of its success to its endgame database[87]. This database stores the exact minimax outcome, win, lose, or draw, of every possible checkers position with eight or fewer pieces on the board. Once the search detects that there are eight or fewer pieces on the board at a given node, it simply looks up the exact game-theoretic value of the position in its database, rather than searching further.

These endgame databases are computed using a technique known as retrograde analysis, originally pioneered by Ken Thompson in chess[99]. The basic algorithm is as follows. First we enumerate all positions of interest, such as those with eight or fewer pieces, and initially label every position as unknown, meaning its exact minimax value is not yet known. Then, we apply the rules of the game to label certain positions as wins, losses, or draws. For example, in checkers, those positions where one side has no pieces on the board, or no legal moves, are labelled losses for that side. Next, we go through each unknown position, and apply all legal moves, generating all its successor positions, and examine their status to see if we can determine the status of the parent position. For example, if the player to move in the parent position has a position labelled win among its children, the parent can be labelled win as well. Alternatively, if all of the child nodes of the player to move are labelled losses for that player, the parent position can be labelled a loss. If there is at least one draw, and all the remaining children are known losses, then the position is a draw for the parent. After applying

th position, some positions will have changed their st o win, loss, or draw, and the algorithm performs an ugh all the remaining unknown positions. The alg when the status of every position is known.

7.7.7 Special Purpose Hardware

Although the basic algorithms are described above, much of the performance advances in computer chess have come from faster hardware. The faster the machine, the deeper it can search in the time available, and the better it plays. Despite the rapidly advancing speed of general-purpose computers, the best machines today are based on special-purpose hardware designed and built only to play chess. The first of these was a machine named Belle, designed by Ken Thompson in 1982[12]. This was followed by Hitech, a machine designed by Hans Berliner and Carl Ebeling in 1987[22]. The latest in this line is Deep Blue, the machine that defeated Garry Kasparov. Deep Blue can evaluate about 200 million chess positions per second[8].

7.7.8 Selective Search

What is most surprising about chess is that some humans, such as Kasparov, are competitive with computers, since a human certainly can't consider more than ten positions in a second, compared to 200 million. The fundamental reason that humans are competitive in these games is that they are very selective in their choice of positions to examine, unlike the programs, which perform full-width fixed-depth searches. The importance of selective search was recognized by Shannon back in 1950. Since then, more than a dozen selective search algorithms have been proposed in the literature.

As an example, we present an algorithm called *best-first minimax* [47], not because it is the best, but because it is the simplest to describe. Given a partially expanded minimax tree, the backed-up minimax value of the root is determined by, and has the same value as, one of the leaf nodes, as is the value of every node on the path from the root to that leaf. That path is known as the principal variation, and the leaf node at the end is known as the principal leaf. For example,

in Figure 7.5, th is the left-most path in the tree. The principal va m's best guess as to the sequence of moves that wi given the partial game tree. The idea of best-first simply to always expand next the principal leaf, w the end of the principal variation. The rationale is that this node has the most influence on the the current move recommendation, and is the unexpanded node most likely to actually be reached. A move can be made when the principal variation reaches a given depth. In general, best-first minimax will generate an unbalanced tree, and make different move decisions than full-width fixed-depth alpha-beta.

Consider the example in Figure 7.6, where again squares represent MAX nodes and circles represent MIN nodes. Figure 7.6A shows the situation after the root has been expanded. The values of the children are their static heuristic values, and the value of the root is 6, the maximum of its children's values. Thus, the right child is the principal leaf, and is expanded next, resulting in the situation in Figure 7.6B. The new frontier nodes are statically evaluated at 5 and 2, and the value of their MIN parent changes to 2, the minimum of its children's values. This changes the value of the root to 4, the maximum of its children's values. Thus, the left child of the root is the new principal leaf, and is expanded next, resulting in the situation in Figure 7.6C. The value of the left child of the root changes to the minimum of its children's values, 1, and the value of the root changes to the maximum of its children's values, 2. At this point, the rightmost path is the new principal variation, and the rightmost leaf is expanded next, as shown in Figure 7.6D.

Unfortunately, very few selective search algorithms are as effective as full-width alpha-beta. The primary reason is that a full-width search is good insurance against missing an important move and making a mistake as a result. Consequently, most game programs that use selective searches use a hybrid algorithm that begins with a full-width search to a nominal depth, and then searches more selectively below that depth.

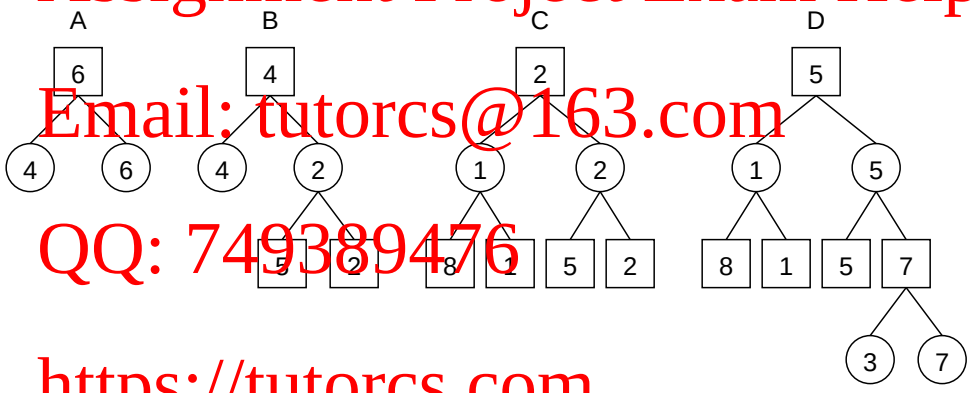


Figure 7.6: Best-first minimax search example

Chapter 8

Performance Analysis of Alpha-Beta Pruning

Since alpha-beta pruning performs a minimax search while pruning much of the tree, its effect is to allow a deeper search with the same amount of computation. This raises the question of how much does alpha-beta improve performance? The best way to characterize its efficiency is in terms of its *asymptotic effective branching factor*, which is the d^{th} root of the number of nodes that must be evaluated in a search to depth d, in the limit of large d. Similarly, it can be defined as the ratio of the number of nodes generated at depth d to the number of nodes generated at depth $d-1$.

As we saw in the previous chapter, the efficiency of alpha-beta pruning depends upon the order in which nodes are encountered at the search frontier. Thus, we consider three different cases: worst case, best case, and average case.

For any set of frontier node values, there exists a worst-case ordering of the values such that alpha-beta will not perform any cutoffs at all. For example, if the tree of Figure 7.4 is evaluated from right to left, no cutoffs occur. In general, if the children of Max nodes are generated in increasing order of their backed-up values, and the children of Min nodes are generated in decreasing order of their backed-up values, then all frontier nodes must be evaluated, and the effective branching factor is b, the brute-force branching factor.

8.[illegible]ound for Minimax Algorithms

Ne[illegible]ower bound on the number of leaf nodes that mu[illegible] any minimax algorithm, including alpha-beta. By [illegible]m, we mean any algorithm that is guaranteed to return the minimax value v of the root node of a game tree. This is the backed-up value of a partially expanded game tree with numeric values assigned to the frontier nodes, assuming that the value of an interior Max node is the maximum of its children's values, and the value of an interior Min node is the minimum of its children's values. Verifying that the minimax value of a partial game tree is equal to v is equivalent to verifying that the value is both $\geq v$ and $\leq v$. To show that the value is $\geq v$, we need to show that no matter what Min does, Max can always achieve a value $\geq v$. This requires demonstrating a strategy for Max, which is a solution subtree from Max' perspective, containing one child of each Max node, and all b children of each Min node. Similarly, to show that the value is $\leq v$, we need to show that no matter what Max does, Min can always achieve a value $\leq v$. This requires demonstrating a strategy for Min, which is a solution subtree from Min's perspective, containing one child of each Min node, and all b children of each Max node. Thus, any correct minimax algorithm must explore a strategy for Max and a strategy for Min.

Figure 8.1 shows a minimal binary minimax tree of depth five. It consists of a strategy for Min, shown in solid lines, and a strategy for Max, shown in dotted lines. Both strategies start at the root node. Since the root is a Max node, both its children are included in the Min strategy, but only the leftmost child is included in the Max strategy in this case. Thus, the right subtree of the root belongs only to the Min strategy, and includes one child of each Min node, and both children of each Max node. Turning now to the left child of the root, since it is a Min node, both children are part of the Max strategy, but only the leftmost child is part of the Min strategy, in this case. Thus, the subtree rooted at the right child of the left child of the root belongs only to the Max strategy, and contains one child of each Max node, and both children of each Min node. Since each possible choice is made in favor of the leftmost node in this case, the leftmost path, shown in both line types, is common to both strategies.

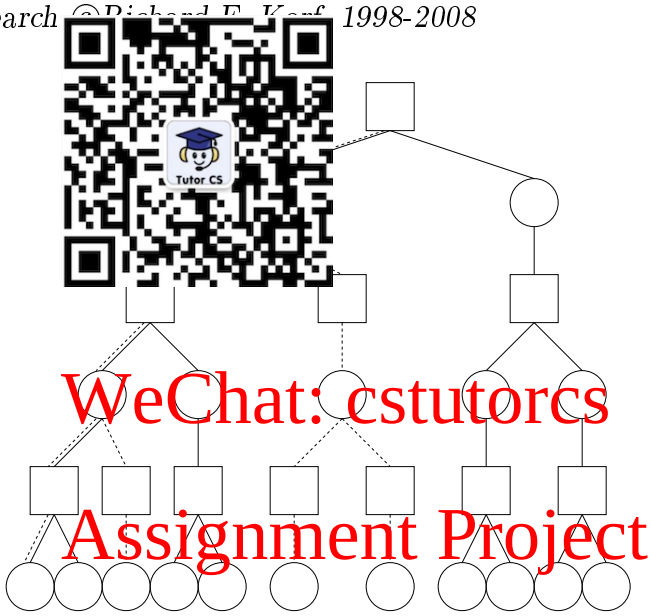


Figure 8.1: Minimal Depth-Five Binary Minimax Tree

Assume that our game tree has a uniform branching factor of b, and a uniform depth of d levels. Since a strategy for Max is a solution subtree containing one child of each Max node, and all b children of each Min node, a Max strategy branches b ways at every other level. Assume, without loss of generality, that Max is to move at the root. If d is even, then there will be $b^{d/2}$ leaf nodes in a strategy for Max, and if d is odd, there will be $b^{\lfloor d/2 \rfloor}$ such leaf nodes.

Similarly, since a strategy for Min is a solution subtree containing one child of each Min node, and all b children of each Max node, a Min strategy also branches b ways at every other level. Again assume that Max is to move at the root. If d is even, then there will be $b^{d/2}$ leaf nodes in a strategy for Min, and if d is odd, there will be $b^{\lceil d/2 \rceil}$ such leaf nodes.

Thus, if d is odd, the sum of the leaf nodes in both strategies will be $b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil}$. If d is even, the corresponding number of leaf nodes will be $b^{d/2} + b^{d/2}$. If d is even, however, $d/2 = \lfloor d/2 \rfloor = \lceil d/2 \rceil$, so we can write the expression as $b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil}$ in either case.

These two strategies, one for Max and one for Min, will have a single path in common, shown as the leftmost path in Figure 8.1. Hence they

wi node in common, the leftmost leaf in the figure, wl whose value progagates up to the minimax value of node was counted twice in the above sum, so the ac t distinct leaf nodes in a strategy for Max and a str $/2\rfloor} + b^{\lceil d/2 \rceil} - 1$. This is the number of leaf nodes that must be examined by any minimax algorithm, and hence a lower bound on the time complexity of any such algorithm. Asymptotically, this is $\mathcal{O}(b^{d/2})$.

8.2 Minimax Value of Game Trees

Since the best and worst cases occur under different orderings of the values of the leaf nodes, the most natural definition for the average case is that the leaf nodes are randomly ordered, with each possible ordering being equally likely. Note that heuristic node ordering would violate this assumption, but it remains the simplest definition of the average case. In general, the average-case performance of an algorithm may not predict its performance in practice. The problem of the average-case complexity of alpha-beta pruning was open for a long time, and not solved until 1982 by Judea Pearl[72]. This entire section is taken directly from Pearl's book [74].

To solve this problem, we first consider what the minimax value of the root of a game tree will be, in the average case of randomly ordered frontier nodes. We then use that result to determine the asymptotic complexity of alpha beta. We begin with the special case of a game tree where the leaf nodes are actual terminal positions, and have the exact values of WIN or LOSS, with no drawn positions. We then consider the more general case of arbitrary leaf-node values.

8.2.1 Win-Loss Trees

Our standard analytic model is a tree with uniform branching factor b, and uniform depth d. We will assume that Max is to move at the root, and that the depth d is even. The terminal nodes will each be labelled WIN with probability P_o, and LOSS with probability $1 - P_o$, independently of each other.

We can cons called board-splitting, that corresponds exactly t l. We'll call our two players Vertical and Horizont sheet of graph paper, $b^{d/2}$ squares on each side, and V with probability P_0, and H with probability $1 - P_0$ each other. When it is Vertical's turn to move, he divides the board vertically into b equal slices, and discards all but one of the slices. Then, Horizontal takes the remaining slice, and divides it horizontally into b equal slices, discarding all but one of them, etc. When there is only one square left, the initial in the square indicates the winner of the game.

For example, if $b = 2$, every move of board splitting splits the board in half, so every node of the game tree has 2 children. The total number of squares in the board is $2^{d/2} \cdot 2^{d/2} = 2^d$, and since each move discards half of the squares, after d total moves, we are left with a single square. Since each of these squares is marked independently as a win for vertical with probability P_0, and a win for horizontal with probability $1 - P_0$, this game corresponds exactly to our analytic model with $b = 2$. The argument for every other branching factor is completely analogous.

As a first step to determining the complexity of evaluating such a win-loss tree, we compute the probability that the backed-up minimax value of a node is a win for one player or the other. Let P_n be the probability that Max can force a win, given that Max is to move and there are $2n$ total moves remaining in the tree. Similarly, let Q_n be the probability that Max can force a win, given that it is Min's turn to move and there are $2n - 1$ total moves left. Thus, Q_n applies to the children of the nodes to which P_n applies. Figure 8.2 shows a tree fragment with the corresponding assignment of probabilities, where square nodes represent moves for Max, and circle nodes represent moves for Min.

Next, we write some equations describing the relationship between these variables, in order to derive a recurrence relation among them. First, in order for a node where Min is to move to be a win for Max, all of its children must be wins for Max. Since the probabilities are all independent, the probability that all b children of a node are wins for Max is the probability that one is a win for Max, raised to the b power. Thus, $Q_n = P_{n-1}^b$. Similarly, in order for a position where Max is to move to be a loss for Max, all of its children must be losses for Max. The probability that a node is a loss for Max is one minus the

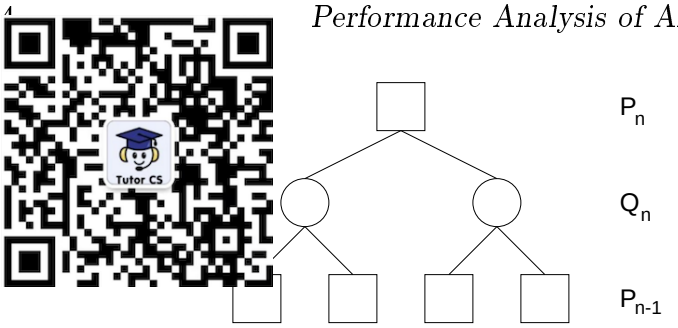


Figure 8.2: Search Tree Fragment Illustrating Definitions of P_n and Q_n

probability that it is a win for Max. Therefore, we can write $1 - P_n = (1 - Q_n)^b$. By substituting the first equation into the second, we get $1 - P_n = (1 - P_{n-1})^b$, or $P_n = 1 - (1 - P_{n-1})^b$.

This expresses P_n as a function of P_{n-1}, and defines a recurrence for P_n. In other words, given P_0, the probability that a leaf node is a win for Max, the above recurrence relation will give us the probability that a Max node at any higher level in the tree will be a win for Max. We are interested in the asymptotic probabilities as the depth of the tree grows large. Thus, we need to understand the behavior of large numbers of iterations of the function $f(x) = 1 - (1 - x^b)^b$.

The s-shaped line in Figure 8.3 shows a graph of this function for the special case of $b = 2$, a binary tree. The straight diagonal line is the function $f(x) = x$. The arrows show what happens as we iterate this function, which is equivalent to propagating the winning probabilities up the tree. For example, if we start with $P_0 = .4$, $P_1 \approx .3$, represented by the vertical arrow from .4 to our function curve. By moving horizontally left from this point to the straight line $f(x) = x$, we reflect the value of $P_1 \approx .3$ onto the x axis. Applying our function to P_1 gives us $P_2 \approx .17$, represented by the vertical line down to our function curve. Repeating this process eventually causes the value of P_n to converge to 0 very quickly. In fact, if we start with P_0 at any point less than the point at which the curve crosses the straight line, P_n eventually converges to zero. Similarly, if we start with P_0 equal to any value greater than the crossover point, such as .8 in the figure, then P_n eventually converges to 1. Furthermore, this convergence is very rapid, as shown by the figure.

What this means is that if the probability of a win for Max at the

Figure 8.3: Graph showing iterations of function $f(x) = 1 - (1 - x^2)^2$.

leaf nodes is chosen below the crossover point, then for a sufficiently deep tree, it is almost certain that the root will be a forced win for Min. For example, if we chose Max to win at the leaves with probability .5, which sounds fair, then a large enough game is almost certainly a win for Min. The reason is that since we assumed a tree of even depth with Max at the root, Min gets to make the last move. Thus, the nodes one level above the terminal nodes are a win for Min if at least one of their children is a win for Min, but a win for Max only if all their children are wins for Max. On the other hand, if the probability of a win for Max at the leaves is greater than the crossover point, then a large enough game tree is almost certainly a forced win for Max.

The point where our function crosses the straight line $f(x) = x$ is the solution to the equation $x = 1-(1-x^b)^b$, and the fixed point of our

ite his value as ξ_b. For $b = 2$, $\xi_2 = (\sqrt{5}-1)/2 \approx .618$. Th wn as the "golden ratio". If we choose $P_0 = \xi_b$, th e tree, the probability that a node is a win for Ma

xpress the probability that the root of a game tree is a forced win for Max, as a function of the probability that a leaf node is a win, in the limit of large depth, by the following equation:

$$\lim_{n\to\infty} P_n(P_0) = \begin{cases} 0 & \text{if } P_0 < \xi_b \\ \xi_b & \text{if } P_0 = \xi_b \\ 1 & \text{if } P_0 > \xi_b \end{cases}$$

This result is rather surprising. It says that even though wins and losses are chosen randomly at the leaf nodes, unless the probability of a win at the leaf is exactly ξ_b, we can predict the win-loss status of the root of a sufficiently deep minimax tree with almost certainty, simply by knowing the probability of a win at the leaves, assuming the leaf values are chosen independently of one another. In some sense, the alternating levels of minimaxing filter out the noise of the leaf nodes, to produce a root whose value can be predicted in advance.

8.2.2 The Minimax Convergence Theorem

We now generalize our result to the case of leaf nodes with arbitrary numerical values. We adopt the same model of a tree with uniform branching factor b, and uniform depth d. The leaves, however, are now assigned random numeric values, independently, but from a common probability distribution function, denoted $Fv_0(v)$. $Fv_0(v)$ is the probability that a particular terminal node value chosen from this distribution has a value less than or equal to v, or $P(v_0 \leq v)$. For example, if the terminal node values were chosen uniformly from the interval zero to one, then $Fv_0(v) = v$.

Given this model, the first thing we want to determine is the probability distribution of the minimax value of the root of a tree, in the limit of large depth, as a function of the probability distribution of the leaf values. We express the distribution of the minimax values at $2n$ levels above the leaves as $Fv_n(v)$, which by definition equals $P(v_n \leq v)$. For-

tunately, we can result we just derived for win-loss trees to this mor

The trick is t lue of v, we can define a leaf node as a win for Max er than v, and a loss for Max if its value is less than , the minimax values propagate up the tree exactly the same as in the win-loss trees. In a win-loss tree, a Max node will be a win for Max if and only if any of its children is a win for Max. Correspondingly, in a general game tree, the minimax value of a Max node will be greater than v if and only if any one of its children has a minimax value greater than v. Similarly, in a win-loss tree, a Min node will be a win for Max if and only if all of its children are wins for Max. Correspondingly, in a general game tree, the minimax value of a Min node will be greater than v if and only if all of its children's minimax values are greater than v.

In a win-loss tree, P_n is the probability that a Max node $2n$ levels above the leaves is a forced win for Max. Correspondingly in our general trees, this is the probability that the minimax value of the root is greater than a particular value v, or $P(v_n > v)$. This is equal to $1 - P(v_n \leq v)$, which is $1 - Fv_n(v)$. If we substitute $1 - Fv_n(v)$ for P_n, and similarly $1 - Fv_0(v)$ for P_0 in the above limit equation for win-loss trees, we can solve for $Fv_n(v)$. The resulting equation gives us the probability distribution of the minimax value of the root of a tree with arbitrary numeric terminal values in the limit of large depth:

$$\lim_{n\to\infty} Fv_n(v) = \begin{cases} 0 & \text{if } Fv_0(v) < 1 - \xi_b \\ 1 - \xi_b & \text{if } Fv_0(v) = 1 - \xi_b \\ 1 & \text{if } Fv_0(v) > 1 - \xi_b \end{cases}$$

This is the famous minimax convergence theorem, proved by Pearl for game trees, but noticed as early as 1956 by Shannon and Moore in a different context we'll describe below[62]. What it says is quite surprising. The probability distribution is zero up to a particular value of v, which we will refer to as v^*. v^* is the value of v for which $Fv_0(v^*) = 1 - \xi_b$. In other words, in a b-ary tree, it is that particular value for which the probability that a leaf node takes on a value less than it is $1 - \xi_b$. Beyond v^*, the probability distribution function of the minimax value of the root is one. The meaning of this is easier to see in the corresponding probability density function, which is the derivative of

th on, and shows the distribution of probability mass ar s of v. The probability density function of the st ion $Fv_n(v)$ is an impulse at v^*. In other words, all ss is concentrated at v^*.

What this means is that if we construct a minimax tree with arbitrary terminal values chosen independently from the same distribution, in the limit of large depth we can predict exactly what the minimax value of the root of the tree will be! For example, in a binary tree where the leaf values are chosen independently from the uniform distribution between zero and one, $v^* = 1 - \xi_2 \approx 1 - .618 = .382$. In other words, in a sufficiently deep such tree, the minimax value of the root will be .382. Note that the particular convergence value is a function of the leaf node distribution, but convergence will occur with any distribution, as long as the same distribution is used for each leaf node, and the values are chosen independently.

The minimax convergence theorem was first noticed by Moore and Shannon in the context of constructing reliable circuits from unreliable components[62]. For example, assume that we want a fuse that will burn out after a specific, predictable interval of time, but we only have fuses that have some broad distribution of burn-out times. If we connect two fuses in series, the time for the complete circuit to burn out will be the minimum of the burn-out times of the individual fuses, since when either fuse burns out, the circuit is broken. Conversely, if we connect two fuses in parallel, the burn-out time of the whole circuit will be the maximum of the burn-out times of the individual fuses, since the circuit will remain closed until both fuses burn out. By constructing a large series-parallel network of fuses, the burn-out time of the entire circuit will correspond to the minimax value of the burn-out times of the individual fuses. For example, Figure 8.4 shows a series-parallel fuse network that is equivalent to a depth four binary tree with a Max node at the root. Because of the minimax converge theorem, in a sufficiently large such network, we can accurately predict the burn-out time of the complete circuit, even though we can't predict when any individual fuse will burn out with any degree of accuracy.

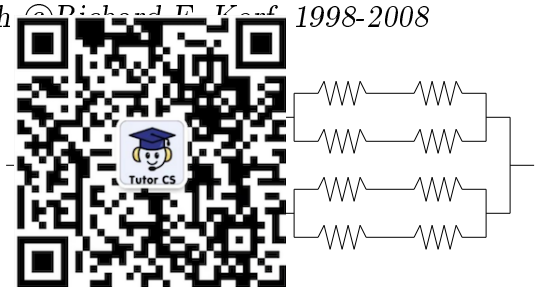


Figure 8.4: Series-Parallel Fuse Network Equivalent to a Depth-Four Binary Minimax Tree with Max at the Root

8.3 Average-Case Time Complexity

We now turn our attention back to the question of the average-case time complexity of alpha-beta pruning. Again we first consider win-loss game trees, and then examine trees with arbitrary numeric terminal values.

8.3.1 Win-Loss Trees

Although alpha-beta pruning was originally described for trees with numeric values, a special case also applies to win-loss trees. In particular, when evaluating the children of a node, as soon as one child is evaluated as a win for the parent, the remaining children are pruned.

Again, we assume our previous model of a tree with uniform branching factor b and uniform depth d, with leaf nodes labelled win or loss independently with probabilities P_0 and $1 - P_0$, respectively. Assume that $P_0 < \xi_b$. In that case, according to the minimax convergence theorem, at sufficiently high levels of the tree, all nodes are losses for Max, and wins for Min. Note that although we know this from analyzing the model, our alpha-beta algorithm doesn't know it, and must search the tree like any other tree. At a Max node, all the children must be evaluated, since each will be a loss for Max. Only one child of each Min node must be examined, however, since the first child will be a win for Min, causing the remaining children to be pruned. Thus, the tree that is actually evaluated contains all children of each Max node, and only one child of each Min node. If we follow any path from the root, it will branch b ways at every other level, the Max nodes, and only one way at the alternating Min levels. Thus, the asymptotic number of leaf

no rge depth will be $O(b^{d/2})$, which is equivalent to an factor of $b^{1/2}$.

that $P_0 > \xi_b$. In that case, at sufficiently high lev st all nodes will be wins for Max, and losses for Mi the game tree means generating only one child of each Max node, and all children of each Min node. As in the above case, this also results in an effective branching factor of $b^{1/2}$.

The only remaining case to consider is what happens when P_0 is exactly equal to ξ_b. Note that this case is likely to be extremely rare, particularly since ξ_b is an irrational number in general. In this case, however, Pearl[74] shows that the effective branching factor is $\xi_b/(1-\xi_b)$. For example, for a binary tree, $\xi_b/(1-\xi_b) \approx 1.618$, compared to $\sqrt{2} \approx 1.414$. For large values of b, $\xi_b/(1-\xi_b) \approx b^{3/4}$.

8.3.2 Trees with Arbitrary Terminal Values

What is the asymptotic effective branching factor of alpha-beta for trees with numeric leaf values chosen randomly and independently from the same distribution? Here we have to distinguish two cases, depending on whether the leaf values are chosen from a continuous distribution, such as a segment of the real number line, or from a discrete distribution, with only a finite number of distinct values.

If the distribution of leaf values is discrete, then the minimax convergence theorem tells us that at sufficiently high levels of the tree, the minimax values of all nodes will be equal. Alpha-beta pruning works by comparing the value of each child of a Min node against a lower alpha bound, using a less-than-or-equal-to test, and each child of a Max node against an upper beta bound, using a greater-than-or-equal-to test. If the test is satisfied, the remaining children are pruned. If all nodes have the same value high in the tree, alpha and beta will have this same value, and every test will be satisfied by equality. Thus, alpha-beta pruning will realize its best-case performance.

On the other hand, if the leaf values are chosen from a continuous distribution, no two leaf nodes will have the same value, since the probability that any node takes on any particular value is zero. These values will be very tightly clustered around v^*, but will remain distinct. In that case, Pearl[74] shows that again the effective branching factor

will be $\xi_b/(1-\xi$

Note that or ter, the heuristic values must be discrete. On the tice we can't search trees that are infinitely deep. T he effective branching factor is $b^{1/2}$ or $b^{3/4}$, or somet pends on the actual search depths and the number of different possible heuristic values.

Remember that all these average-case results are for randomly ordered trees. In practice, however, node ordering is always used to achieve performance close to the best case.

Chapter 9

Multi-Player Games

9.1 Introduction

In the last chapter, we considered algorithms for two-player, perfect-information games. In this chapter, we generalize those algorithms, in particular minimax search and alpha-beta pruning, to the case of non-cooperative perfect-information games with more than two players. In other words, we assume that every player is out for themselves, and that the players don't form coalitions to defeat other players. For example, Chinese Checkers can involve up to six different players moving alternately. As another example, Othello can be extended to an arbitrary number of players by having different colored pieces for each player, and modifying the rules such that whenever a mixed row of opposing pieces is flanked on both sides by two pieces of the same player, then all the pieces are captured by the flanking player.

9.2 Maxn Algorithm

Luckhardt and Irani[56] extended minimax to multi-player games, calling the resulting algorithm max^n. For typographical convenience we refer to it as *maxn*. They assume that the players alternate moves, that each player tries to maximize his or her return, and is indifferent to the returns of the other players. At the frontier nodes, an evaluation function is applied that returns an n-tuple of values, where n is

th with each component corresponding to the estimated position for one of the players. For example, an evaluator for multi-player Othello might return the number of pieces on the board. Then, the value of each interior node for player i is to move is the entire n-tuple of the child for which the i^{th} component is a maximum. Figure 9.1 shows a maxn tree for three players, with the corresponding backed-up maxn values. The numbers in the nodes represent the player to move at that node, and the evaluations for players 1, 2, and 3 appear in that order in the n-tuple. For example, the value of the leftmost interior node of the tree, where player 3 is to move, is the value of its right child, $(1, 7, 2)$, because its third component, 2, is greater than the third component of its left child, 0.

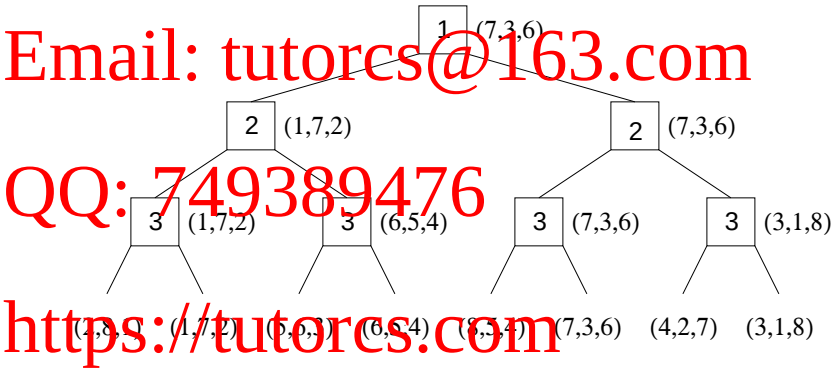


Figure 9.1: Three-Player Maxn Game Tree Fragment

More formally, define $M(x)$ to be the static heuristic value of node x, and $M(x, p)$ to be the backed-up maxn value of node x, given that player p is to move at node x. Each of these values is an n-tuple. $M_i(x, p)$ is the component of $M(x, p)$ that corresponds to the return for player i. We can then define the maxn value of a node recursively as follows:

$$M(x,p) = \begin{cases} M(x) & \text{if } x \text{ is a frontier node} \\ M(x_i, p') & \text{otherwise} \end{cases}$$

where $M(x_i, p') = \max M_p(x_i, p')$ over all children x_i of node x, where p' is the player that follows player p in the move order, and ties are

broken in favor c

Minimax can cial case of maxn for two players, where the evalua s an ordered pair of x and $-x$, and each player max t at its moves.

Luckhardt ar that at nodes where player i is to move, only the i^{th} component of the children need be evaluated. At best, this can produce a constant factor speedup, but it may be no less expensive to compute all components than to compute only one. They correctly concluded that without further assumptions on the values of the components, pruning of entire branches is not possible with more than two players. Thus, they did not explore such pruning any further.

They used the terms "shallow pruning" and "deep pruning" to refer to their techniques of avoiding some partial evaluations. Since these terms had previously been used to describe actual tree pruning in the alpha-beta literature [39], we will use the original meanings of both these terms, sacrificing consistency with Luckhardt and Irani's terminology.

9.3 Alpha-Beta in Multi-Player Games

If there is an upper bound on the sum of all components of a tuple, and there is a lower bound on the values of each component, then tree pruning is possible. The first condition is a weaker form of the standard zero-sum assumption of two-player games. The second is equivalent to assuming a lower bound of zero on each component, since any other lower bound can be shifted to zero by adding or subtracting it from every component. Most practical evaluation functions will satisfy both these conditions, since violating them implies that the value of an individual component can be unbounded in at least one direction. For example, in the evaluation function described above for multi-player Othello, no player can have less than zero pieces on the board, and the total number of pieces on the board is the same for all nodes at the same level in the game tree, since exactly one piece is added at each move.

9.3.1 Immediate Pruning

The only pruning possible under these assumptions occurs when a node has a value of ∞ for one of its components, and the i^{th} component of one of its children equals ∞ based on the sum of all components. In that case, all remaining children can be pruned, since no other child's i^{th} component can exceed the upper bound on the sum. We will refer to this as *immediate pruning*. This is equivalent to situations in the two-player case when a child of a Max node has a value of ∞, or a child of a Min node has a value of $-\infty$, indicating a won position for the corresponding player.

9.3.2 Shallow Pruning

A more complex situation is called *shallow pruning* in the alpha-beta literature. Figure 9.2 shows an example of shallow pruning in a three-player game, where the upper bound on the sum of each component is 9. Note that in this particular example, the sum of each component is exactly 9, but an upper bound is all that is required. Evaluating node b results in a lower bound of 3 on the first component of node a, since player one is to move at node a. This implies an upper bound on each of the remaining components of $9 - 3 = 6$. Evaluating node g produces a lower bound of 7 on the second component of node f, since player two is to move at node f. Similarly, this implies an upper bound on the remaining components of $9 - 7 = 2$. Since the upper bound (2) on the first component of node f is less than or equal to the lower bound on the first component of node a (3), player one won't choose node f, and its remaining children can be pruned. Similarly, evaluating node i causes its remaining brothers to be pruned, in this case because node h can't be better than node b for player one, based on equality of the bounds. This is similar to the pruning in the left subtree of Figure 7.5.

The procedure *Shallow*, shown below, takes a *Node* to be evaluated, the *Player* to move at that node, and an upper *Bound* on the component of the player to move, and returns an n-tuple that is the maxn value of the node. *Sum* is the global upper bound on the sum of all components of an n-tuple, and all components are assumed to be non-negative. Initially, *Shallow* is called with the root of the tree, the player to move

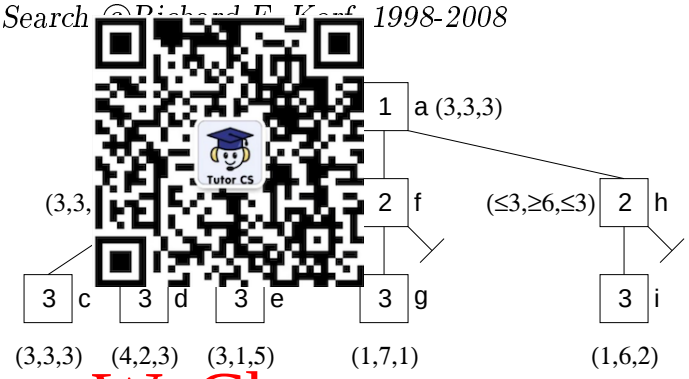


Figure 9.2: Shallow Pruning Example in Three-Player Game Tree

at the root, and *Sum*. Note that the shallow pruning procedure includes immediate pruning as a special case.


```
Shallow(Node, Player, Bound)
IF Node is terminal, RETURN static value
Best = Shallow(first Child, next Player, Sum)
FOR each remaining Child
    IF Best[Player] >= Bound, RETURN Best
    Current = Shallow(Child, next Player, Sum - Best[Player])
    IF Current[Player] > Best[Player], Best=Current
RETURN Best
```


9.3.3 Failure of Deep Pruning

In a two-player game, alpha-beta pruning allows an additional type of pruning known as *deep pruning*. For example, in Figure 7.5, the siblings of nodes p and r are pruned based on bounds inherited from their great-great-grandparent, the root in this case. In general, deep pruning refers to pruning a node based on a bound inherited from its great-grandparent, or any more distant ancestor. In a two-player game tree, this can only occur in trees of height four or greater. Surprisingly, deep pruning does not generalize to more than two players.

Figure 9.3 illustrates the problem. Again, the upper bound on the sum of each component is 9. Evaluating node b produces a lower bound of 5 on the first component of node a, and hence an upper bound of $9 - 5 = 4$ on the remaining components of node a. Evaluating node e

res nd of 5 on the third component of node d, and he l of $9 - 5 = 4$ on the remaining components of no per bound of 4 on the first component of node d i r bound of 5 on the first component of node a, th nnot become the value of node a. Thus, we are tempted to prune node f.

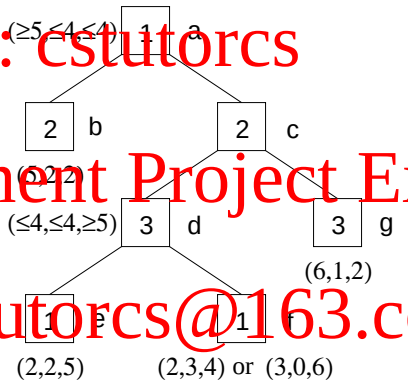


Figure 9.3: Failure of deep pruning for three players

The value of node f could affect the value of the root, however, depending on the value of node g. For example, if the value of node f were (2,3,4), the value of node d would be (2,2,5), the value of node c would be (2,2,5), and the value of node a would be (5,2,2). On the other hand, if the value of node f were (3,0,6), then the value of node d would be (3,0,6), the value of node c would be (6,1,2), and the value of node a would be (6,1,2). Thus, even though the value of node f cannot *be* the maxn value of the root, it can *affect* it. Hence, it cannot be pruned.

9.3.4 Optimality of Shallow Pruning

Given the failure of deep pruning in this example, is there a more restricted form of pruning that is valid, or is shallow pruning the best we can do? The answer is the latter, as expressed by the following theorem:

Theorem 9.1 *I[illegible]orithm that computes the maxn value of a game tree w[illegible]ayers must evaluate every terminal node evaluated b[illegible]nder the same ordering.*

A directional [illegible]e in which the order of node evaluation is independent of the value of the nodes, and once a node is pruned it can never be revisited. For example, a strictly left-to-right algorithm is directional.

Since the complete proof is somewhat tedious and not very revealing, we present here an overview and example of the argument. The main idea is illustrated by the construction in Figure 9.4, which shows a 3-player, 6-level tree. We assume that node n is evaluated by shallow pruning but pruned by another algorithm. We then show that the value of every node above it depends on the value of node n. The letters to the left of the path from the root to node n represent the greatest lower bounds on the components corresponding to the player to move at each node. Since by assumption node n is evaluated by shallow pruning, it must be the case that for any two bounds x and y at adjacent depths, $x + y$ is strictly less than the global upper bound. Since the decision to skip node n is made before any of the nodes to the right of the path are examined, we are free to choose any values for these nodes consistent with the global bounds. For this purpose, $x+$ represents a value greater than x by an arbitrarily small amount. The reader is encouraged to assign each of the two alternative values to node n, and then compute the maxn value of the root, to see that it is different in the two cases. The propagation of values up the tree can be viewed as a "zipper" effect in the sense that the original order of the "teeth" (nodes) at the bottom determines the order of the teeth at the top, even though no individual tooth can move very far. The formal proof is by induction on the height of the tree and generalizes the result to an arbitrary number of players greater than two.

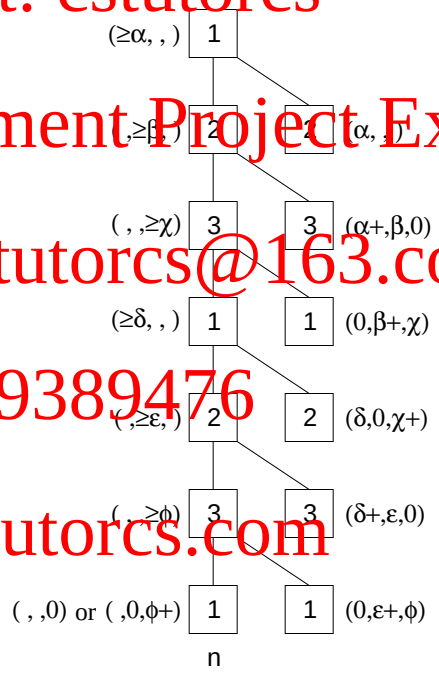


Figure 9.4: Proof sketch of optimality of shallow pruning

Chapter 10

Minimax and Pathology

So far, we have considered the time and space complexities of minimax search. We now turn our attention to the quality of the decisions it makes. Since alpha-beta makes exactly the same decisions as minimax search, the question is the decision quality of minimax.

10.1 Exact Terminal Values

If the leaf nodes of the tree are evaluated exactly, as in the case of win-loss-draw terminal values, then minimax makes optimal moves against an opponent who plays perfectly. In other words, if it can force a win it will, it will lose only if the opponent can force a loss, and will draw in every other situation.

This is not to say, however, that the decision quality is optimal against an imperfect opponent. Consider, for example, a situation where two moves are available, both eventually leading to a forced loss for the player to move. However, one move may lead easily and immediately to a loss, whereas the other may require a long sequence of moves and a great deal of skill on the part of the opponent to force the loss. Against an infallible opponent, it doesn't matter what move is made, and indeed minimax will have no preference for one option over the other. Against an opponent who can make mistakes, however, it is far preferable to choose the move that requires the most skill on the part of the opponent, greatly increasing the chances of an error by the

op win by the player to move.

10 axing of Heuristic Values

W the endgame, the values associated with most nodes in a game tree are heuristic values returned by the static evaluator, and subject to error. Shannon[91] recognized that most positions in a chess tree could not be evaluated exactly, and suggested the idea of a heuristic evaluation function. He then went on to recommend that these values should be minimaxed up the tree as if they were exact values. This isn't necessarily the best thing to do, however.

Consider a Max node with two children, as shown in Figure 10.1. Let x and y be the static heuristic values of the child nodes, and let m be the minimax value of the parent node, or $m = \max(x, y)$. Static evaluations produce an estimate of the value of a node whose actual value is unknown at the time of the estimate. The best way to model the actual value of a node is as a random variable with some probability distribution. For example, assume that the true value of each child node is a random variable uniformly distributed from zero to one, and that the variables are independent of each other. Then, the most natural way to estimate their values would be by their expected values, which would be 1/2. Thus, we would expect the static evaluator to return 1/2 for each child.

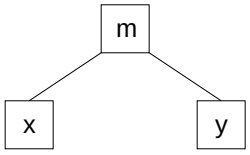


Figure 10.1: Small Game-Tree Fragment

Since the minimax value of the parent node is dependent on the values of the children, m becomes a random variable as well. The minimax algorithm estimates its value as the maximum of the static values of the children, or the maximum of their expected values. The maximum of the expected values of the children is also 1/2 in this example. A better estimate of the value of the parent node, however,

would be its exp expected value of the maximum of x and y. As we e expected value of the maximum of two values is r maximum of their expected values.

The probabil ion of a random variable x, $PDF(x)$, is the probabilit value from the distribution, x_o, is less than or equal to x. Since in this case both x and y are uniformly distributed from zero to one, $PDF(x) = P(x_0 \leq x) = x$, and $PDF(y) = P(y_0 \leq y) = y$. Since m is the maximum of x and y, and x and y are independent, $PDF(m) = P(m_o \leq m) = P(max(x, y) \leq m) = P(x \leq m \text{ and } y \leq m) = P(x \leq m) \cdot P(y \leq m) = m \cdot m = m^2$. The probability density function, $pdf(x)$, of a random variable x is the first derivative of its distribution function. Thus, $pdf(m) = 2m$. The expected value of a continuous random variable is the integral of its value times its pdf over its range of possible values. Thus $E(m) = \int_0^1 m \cdot pdf(m) \cdot dm = \int_0^1 2m^2 dm = \frac{2}{3}m^3]_0^1 = \frac{2}{3}$.

Intuitively, this result is easy to see. If two values chosen from the uniform interval from zero to one were uniformly spaced in the interval, we would expect them to be at 1/3 and 2/3, with 2/3 being the maximum value.

Thus, the expected value of the maximum of two random variables chosen independently from the uniform distribution from zero to one is 2/3, while the maximum of their expected values is only 1/2. The essential error here of minimax is to take the maximum of the expected values, instead of computing the expected value of the maximum. The same error occurs at Min levels of the tree, and these errors propagate with every level. As we search deeper, the minimax values accumulate more and more error.

A natural question is why not compute the exact expected value of the minimax value of the root of a game tree? There are a number of reasons why this is not feasible. The first is that to do this requires the exact distribution function of all the leaf nodes. In a real game, all we have is a numerical heuristic estimate, and very little idea of the actual distribution. Secondly, to do the above calculation, we assumed that the values of the child nodes were independent of one another. This is unlikely to be true in a real game, since nearby nodes in a tree are likely to be correlated, since they represent the values of similar positions. Finally, even if we had the exact distributions and they were

inc butions of the interior nodes become increasingly co d can only be calculated exactly for very small tre nd, minimaxing scalar values is exceedingly simple perhaps most importantly, minimax allows us to us g to almost double the search depth in practice. Any more sophisticated backup rule is not likely to benefit from such pruning, effectively requiring that decisions be made on the basis of much less data than is possible with alpha-beta minimax.

10.3 Game-Tree Pathology

In fact, the above error in minimaxing gives rise to a surprising effect, known as game-tree pathology[64]. Recall the game of board splitting, presented in Chapter 8 as a concrete example of our analytic model of win-loss game trees. We begin with a blank two-dimensional board of $b^{d/2}$ squares on a side. Each individual square is marked as a win for one player or the other, independently with some probability P. The two players, called vertical and horizontal, alternately split the remaining board into b equal slices, and save one slice, discarding the rest. The game is over when there is only a single square left, with the winner determined by the marking in that square. An obvious heuristic evaluation function for this game is to count the total number of winning squares in each remaining board.

Using this evaluation function, we can play the game by searching to various depths, evaluating the frontier nodes, minimaxing these values up the tree, and moving to the child of the root with the best value. If we search all the way to the terminal nodes, which consist of single squares, minimax plays optimally, winning in positions that are forced wins, and losing otherwise. Since the actual game boards are explicitly generated, we can compute the optimal strategy, and compare moves made by different algorithms against the optimal moves. Thus, we can measure the decision quality of an algorithm as the percentage of times it makes an optimal decision. An optimal decision is one that takes a forced winning position, and moves to a child that is also a forced win, or any move from a forced losing position.

As expected, the decision quality of minimax as a function of search

depth increases with depth, up to a point. Beyond a certain depth, however, the percentage of optimal moves made by minimax search is less than for minimax searching to a shallower depth! This is counterintuitive, but real and reproducible. This counterintuitive result is called game-tree pathology. In a sense, the error propagation due to minimaxing overcomes the additional information derived from searching deeper.

In real games, however, searching deeper almost never results in poorer overall quality of play. The puzzle then is to determine what it is about board splitting that causes it to be pathological, unlike real games. There are several possible explanations.

One reason proposed for why real games are not pathological is that the accuracy of the evaluation function increases as it gets closer to the endgame, and that this effect overcomes the error due to minimaxing over more levels. However, Pearl[74] has shown that in order to overcome the minimaxing error, the accuracy of the evaluation function would have to increase by at least 50% with each level. This seems unlikely for any real game and evaluation function.

An alternative explanation, proposed by Pearl[73], is that in a real game, all terminal nodes are not at the same depth, in contrast to the uniform depth of the board-splitting game tree. For example, in chess, the "fool's mate" is a won position at level four in the tree. Such terminal positions, labelled 'traps' by Pearl, have exact evaluations, and if they occur with sufficient frequency, they increase the overall accuracy of the evaluation function with deeper search. In other words, the benefit of minimax lookahead is to discover traps in the search tree, both premature winning and losing positions for the player to move. Pearl provides analytic evidence that a reasonable fraction of traps in the game tree will overcome minimax pathology.

In fact, Abramson[2] showed that one can add such traps to board splitting, simply by modifying the evaluation function. For example, if the vertical player detects a complete vertical line of winning squares in the remaining board, regardless of what the horizontal player does, vertical can always guarantee a win simply by preserving that vertical slice. Similarly, a horizontal line of wins guarantees a win for the horizontal player, by preserving that line regardless of what vertical does. Thus, we can modify the evaluation function to detect these forced

wi the values of ∞ and $-\infty$, depending on which pl in. If we repeat the pathology experiments with th tion function, the pathology disappears, lending cr ent that the uniform tree depth is the cause of th

Another argument that has been made to explain pathology is that the culprit is the independence of sibling values in board splitting, something that does not occur in real games. To test this hypothesis, Nau[64] created a similar game, but without sibling-node independence. He also used a tree of uniform branching factor and depth, but assigned the terminal values differently. To each edge of the game tree, he assigned either a zero or one, independently with probabilities p and $1-p$. Then, the value of all nodes in the tree are computed as the sum of the edge weights from the root to the node in question. For terminal nodes, this is their exact value, and for interior nodes this is interpreted as their heuristic value. This guarantees that nodes close in the tree will have correlated values, since they will share much of their path to the root, and its corresponding edge weights. He found that these game trees did not exhibit minimax pathology and argued that independence of sibling values was the true cause of pathology.

Yet another feature of board splitting has been identified as the cause of pathology, in particular the uniform branching factor assumption. Michon[59] generated game trees that were identical to board-splitting trees, except that the number of children of each node was a random variable. These trees also failed to exhibit pathology.

Thus, we see that when we remove any of the assumptions of board splitting, including uniform branching factor, uniform depth, or independence of sibling values, the pathology disappears. As a result, it is difficult to argue convincingly that any one of these factors is the cause of pathology. Rather, it is a rather small effect, and easily disappears with any significant modification of the basic model. The real value of game-tree pathology is to remind us that minimaxing of uncertain values is statistically misguided. The virtue of minimax is the efficiency with which the minimax value can be computed, by alpha-beta pruning.

Chapter 11

Learning Two-Player Evaluation Functions

In Chapter 6 we explored techniques for acquiring heuristic evaluation functions for single-agent problems. We now turn our attention to the problem of learning heuristic functions for two-player games.

The most obvious, and still most commonly used technique, is hand-coding by a human domain expert. For example, the Deep Blue chess team employed a grandmaster, Joel Benjamin, to help tune their evaluation function by hand. Schaeffer reports that in spite of progress in machine learning of evaluation functions, hand tuning of Chinook's checkers heuristic was the most effective method[87].

11.1 Samuel's Checker Player

An absolutely pioneering research effort in both computer games and machine learning was Arthur Samuel's checkers program[85], written in the 1950's. In 1962, running on an IBM 7094, the machine defeated R.W. Nealy, a future Connecticut state checkers champion. What was most impressive about Samuel's program, however, was that it was one of the first machine learning programs, introducing a number of different learning techniques.

The first of these is what Samuel referred to as "rote learning". When a minimax value is computed for a position from a lookahead

sea is stored along with its minimax value. If the sa ntered again, there is no need to research it to an r equal to the depth to which is was previously sea ed minimax value can simply be returned. This ca t of a much deeper search. For example, if the stored minimax value of a position is based on a lookahead search to depth ten, and the same position is encountered again at the horizon of a depth-ten search, by using the stored value we get the effect of a depth-twenty search of that part of the tree. Due to memory constraints, all the generated board positions cannot be stored, and Samuel used a set of criteria for determining which positions to actually store.

A much more sophisticated example of machine learning was learning the evaluation function itself. The basic technique employed by Samuel is to compare the static evaluation of a node with the backed-up minimax value from a lookahead search. For example, if a heuristic evaluation function were perfect, then the static value of a node would be the same as the backed-up minimax value of the node based on a lookahead search applying the same evaluation at the frontier nodes. In practice, the difference between the static value and the backed-up value can be used as an error term to modify the heuristic function. For example, if the static value of a position is less than its backed-up minimax value, then those terms in the heuristic function that made a negative contribution to the overall score would have their relative weights reduced, while those terms that contributed positively to the evaluation would have their weights increased. As we will see below, this must be done in an iterative fashion, since the error is based on the current evaluation function, which is not an exact evaluation.

In addition to modifying the relative weights of the different terms in the heuristic function, Samuel's program could also select which terms to actually use, from a library of possible terms. In addition to material, these terms attempted to measure such board features as center control, advancement of the pieces, mobility, and more. Samuel's program computed the correlation between the values of these different features and the overall evaluation score. If the correlation of a particular feature dropped below a certain level, the feature was removed from the evaluation function, and replaced by another feature.

11.2 Lin[illegible]on

While Samuel's r[illegible]g the weights of the different terms in the evaluatio[illegible]ewhat ad hoc, here we describe a more principled v[illegible]is task, due to Jens Christensen[11]. Consider, for example, a checkers evaluation function based just on material. Such a function would be of the form: $c_p p_w + c_k k_w - c_p p_b - c_k k_b$, where p_w is the number of single white pieces on the board, p_b is the number of single black pieces on the board, k_w is the number of white kings on the board, k_b is the number of black kings on the board, c_p is the coefficient or weight assigned to single pieces, and c_k is the weight assigned to kings. In other words, the material evaluation is the weighted sum of white's pieces and kings, minus the weighted sum of black's pieces and kings, assuming that white is the maximizer. Since the game is symmetric with respect to white and black, we assign the same relative weight to white's pieces and kings as to black's pieces and kings. Note that we could simplify this function even further by arbitrarily setting the weight of single pieces to one, and representing the relative weight of kings by a single parameter, but we'll maintain both parameters here for pedagogical reasons.

Given such a class of evaluation functions, an individual function is represented by a particular set of values for the two parameters c_p and c_k. We can represent all such functions by a two-dimensional space with c_p on one axis and c_k on the other axis. Our task is to find the best point in this space, or the best relative weights for pieces and kings.

Assume that we start with an initial approximation of the relative weights, for example that both types of pieces are worth the same amount, which corresponds to the values $c_p = c_k = 1$. Consider now a particular state of the game. For purposes of this evaluation, any state is completely characterized by the number of single pieces and kings on the board for each player. The static heuristic value of this position will equal $c_p p_w + c_k k_w - c_p p_b - c_k k_b$, where the numbers of pieces and kings are replaced by their constant values, and the coefficients are replaced by their initial approximation. From this game state, we can perform a lookahead search as deeply as our computational resources allow. At the frontier, our current evaluation function is applied to the leaf nodes, and these values are minimaxed back up to the root, to determine a

ba he position. In general, this backed-up value will no lue of the node.

on $b = c_p p_w + c_k k_w - c_p p_b - c_k k_b$. If we view c_p and c_k quation represents the set of all possible weight co h the static value of this particular position will equal its backed-up minimax value from the given depth. This is the equation of a line in $c_p - c_k$ space. This particular line, however, is based on only a single game state. If we repeat this process, including the lookahead search, for another game state, we get another line in c_p, c_k space. Each state produces another line. In general, these lines will intersect one another, but not all at the same point.

Given a set of such lines, we can determine the point which most nearly approximates their mutual intersection. The problem of finding the best intersection among a set of lines is the dual of, and equivalent to, the problem of finding the best line through a set of points. Standard mathematical techniques such as linear regression can be applied to solve this problem. The best intersection corresponds to a new point in $c_p - c_k$ space, and hence a different evaluation function.

The entire above process used a particular approximation of the evaluation function, which was applied to the leaf nodes of each minimax search. Thus, the new function that results from running linear regression over the resulting lines must be viewed as simply a different, and hopefully better, approximation. Thus, to get an even better function, we must rerun the entire process again, applying the new approximation to the frontier nodes, to get yet another approximation.

Therefore, we have two loops in this process. The inner loop uses a particular evaluation function and a large number of minimax searches from particular game states to derive a new approximation for the evaluation function. The outer loop then iterates this process over multiple approximations. Hopefully, the approximations generated by the outer loop will eventually converge to a particular function, or at least a small neighborhood of such functions.

11.2.1 Experiments with Chess

As a test of these ideas, Christensen addressed the task of learning the relative weights of the different chess pieces. He constructed an

evaluation functi material, with five parameters, the weights of queen ights, and pawns. He constrained the weights to be layers, and initially set all weights to one. The look mited to two levels deep, and linear regression wa h successive approximation to the evaluation function. Surprisingly, the values eventually converged to a fixed point.

The classical weights for the pieces from the chess literature is queen-9, rook-5, bishop-3, knight-3, and pawn-1. In contrast, his program learned the values queen-8, rook-4, bishop-4, knight-3, and pawn-1. These are not the same, but they are close, and at least have the order of the pieces correct.

Bear in mind, however, that this learning experiment was performed with a purely material evaluation function, and only two-level lookahead. As a further test of these results, he played his learned evaluation function against the classical material function, using only two-levels of lookahead for both players. In that restricted context, the learned function played roughly evenly with the classical function.

Chapter 12

Constraint-Satisfaction Problems

In addition to single-agent path-finding problems and two-player games, the third major application of heuristic search is constraint-satisfaction problems (CSP). The classic example is the N-queens Problem. Given an $N \times N$ chessboard, the task is to place N queens on the board so that no two queens are on the same row, column, or diagonal. Other common examples include graph coloring and boolean satisfiability. Given a graph of nodes and edges, the graph coloring problem is to assign a color to each node, such that no two adjacent nodes are assigned the same color, and the total number of colors is minimized. In boolean satisfiability, we are given a propositional logic formula in conjunctive normal form, such as $(a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c)$, and must assign true or false to each variable so that the entire expression evaluates to true. In this case, assigning a, b, and c all to true satisfies the formula. Graph coloring and boolean satisfiability are NP-Complete problems.

What distinguishes constraint-satisfaction problems from single-agent path-finding problems and two-player games is that in a CSP we are not interested in the moves made to reach a solution, but simply finding a problem state that satisfies all the constraints. In CSPs, the goal is given implicitly, since an explicit goal description is a solution to the problem. Nevertheless, there is a strong similarity among the algorithms employed for all three types of problems, as we will see.

12.1 Representation

A [illegible]on problem can be represented by a set of variables and [illegible]alues for each variable. For example, in the N-queens problem, the variables are the queens, and the values are the different locations they could occupy on the board. In graph coloring, the variables are the nodes, and the values are the different colors they can be assigned. In boolean satisfiability, the variables are the boolean variables, and the values are true and false. In addition, there is a set of constraints on the values of the variables. A unary constraint on a variable specifies a subset of all possible values that can be assigned to that variable. For example, a unary constraint on the color of a node in a graph-coloring problem would restrict the set of legal colors to some subset. A binary constraint between two variables, which is the most common type, specifies which possible combinations of assignments to a pair of variables satisfy the constraint between them. In the N-queens problem, for example, there is a binary constraint between every pair of queens that specifies that they can't both occupy the same row, column, or diagonal. In graph coloring, there is a binary constraint between each pair of adjacent nodes that prevents them from being assigned the same color. A ternary constraint limits the set of values that can be simultaneously assigned to three different variables. For example, in the boolean satisfiability formula given above, each clause constrains the three variables contained in it, such that at least one of the literals in the clause must be assigned the value true.

12.1.1 Dual Representations

Consider the problem of designing a three-by-three crossword puzzle, or equivalently, solving such a puzzle with no clues, meaning any valid words are legal. One representation of this problem as a CSP has six variables, one for each row and column. The set of legal values is all three-letter words. There are nine binary constraints, one between each row and column, which require that the words chosen for each row and column pair must assign the same letter to their intersecting square.

An alternative formulation of the problem consists of nine variables, one for each square. The values are the 26 letters of the alphabet.

There are six te[illegible]ne for each row and column, that the letters in eac[illegible]nust constitute a legal word.

These two re[illegible]uals of each other. In fact, every CSP has a dual [illegible]ch variable in the original representation becom[illegible]he dual representation, and each constraint in the original representation becomes a variable in the dual representation. If the constraints in the original representation are binary, the corresponding variables in the dual representation are assigned ordered pairs of the values from the original representation, that satisfy the constraints of the original representation. The constraints in the dual representation are that the same value must be assigned to each variable from the original representation. As another example, in the dual representation for graph coloring, the variables are the edges of the original graph, the values are ordered pairs of colors, and the constraints are that wherever more than one edge is connected to the same node, they all must assign the same color to that node.

The dual of the dual representation of a CSP is the original representation. Since the choice of representation can have a large effect on the efficiency of solving a problem, the virtue of the dual representation is that the dual may be easier to solve than the original problem. For example, crossword puzzles are easier to solve by picking words, and then checking their consistency where they cross, than by picking individual letters and then checking that they make up words.

12.2 Brute-Force Search

The obvious brute-force approach to solving a constraint-satisfaction problem is to try all possible assignments of values to variables, and reject those assignments that violate any of the constraints. If there are n variables, and each variable can take on any of k different values, then there are k^n different possible assignments of values to variables.

12.2.1 Chronological Backtracking

A better approach to a CSP is called *chronological backtracking*. Select an order for the variables, and an order for the values, and assign

va at a time. Each assignment is made so that all co ny variables already assigned are satisfied. The re constraint is violated, no assignments to the re- ma possibly resatisfy that constraint, allowing large pa e to be pruned. If a variable has no remaining legal assignments, then the last variable that was assigned is reassigned to its next legal value. The algorithm continues until either a complete, consistent assignment is found, resulting in success, or all possible assignments are shown to violate some constraint, resulting in failure. Alternatively, the algorithm can be continued even after a successful assignment is found, in order to find all possible solutions.

Figure 12.1 shows the tree generated by chronological backtracking to find all solutions to the Four-queens problem. In this case, each queen is restricted to a particular row, and the values are the different columns in the row. Since there are four variables and four possible values, a brute-force search of this space would generate $4^4 = 256$ nodes, in contrast to the 16 nodes generated by chronological backtracking. The tree is searched depth-first to minimize memory requirements.

In general, each level of a brute-force search tree instantiates a different variable of the CSP. Thus, the maximum depth of the tree is the number of variables, four in the above example. Each branch at a given node assigns a different value to the variable associated with that node. Thus, the branching factor of a node is the number of legal values for the given variable, which is a maximum of four in the above example. Since these trees have fixed depth, and all solutions are at the maximum depth, simple depth-first search can be used.

12.3 Intelligent Backtracking

We can improve the performance of backtracking with ideas such as variable ordering, value ordering, backjumping, and forward checking, each of which will be discussed in turn.

Figure 12.1: Tree Generated to Solve the Four-queens Problem

12 Ordering

Th ariables are assigned can have a large effect on th tree. In general, variables should be ordered fr l to least constrained[27, 80]. For example, if there is a variable with only one value remaining that is consistent with the previously assigned varriables, that variable should be assigned that value immediately. Variables should be assigned in increasing order of the size of their remaining domains. This can either be done statically at the beginning of the search, or dynamically, by reordering the remaining variables each time a variable is assigned a new value.

12.3.2 Value Ordering

The order in which the values of a given variable are chosen determines the order in which the tree is searched. Since it doesn't affect the size of the tree, it makes no difference if all solutions are to be found. If only a single solution is required, however, value ordering can decrease the time required to find a first solution. In general, values should be ordered from least constraining to most constraining[16, 33].

12.3.3 Backjumping

The idea of backjumping is that when a deadend is reached, instead of undoing the last decision made, the decision that actually caused the failure should be modified[29]. For example, consider a three-variable problem where the variables are assigned in the order x, y, z. Assume that values have been chosen for both x and y, but that all possible values for z conflict with the value chosen for x. In chronological backtracking, the value chosen for y would be changed, and then all the possible values for z would be tested again, to no avail. A better strategy is to go back to the source of the failure, and change the value of x in this case, before trying different values for y. Since every value of x conflicts with z, changing the value of x prunes off dead-end branches, speeding up the search without sacrificing completeness.

12.3.4 For

The idea of forw when a variable is assigned a value, check each rema iable to make sure that there is at least one assign s consistent with all the currently assigned variables. If not, the current variable is assigned its next value.

12.4 Constraint Recording

In a CSP, some constraints are explicitly specified, and others are implied by the explicit constraints. Implicit constraints may be discovered either during backtracking, or in a preprocessing phase. The idea of constraint recording is that once these implicit constraints are discovered, they should be saved so that they don't have to be rediscovered.

Consider the CSP represented by the graph in Figure 12.2, which is called a *constraint graph*. It consists of three variables, x, y, and z, with a unary constraint on each variable, and a binary constraint between each pair of variables. The unary constraints are that variable x can only be assigned the values a or b, variable y can only be assigned c or d, and variable z can only be assigned e or f. The binary constraints are represented by an edge between each pair of variables, and are labelled explicitly by the ordered pairs that are allowed by the constraint. For example, the only assignments to variables x and z that satisfy the constraint between them are $x = b$ and $z = e$, or $x = a$ and $z = f$.

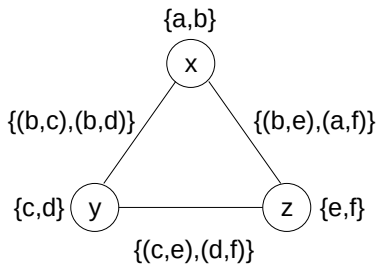


Figure 12.2: Constraint Graph for a Three-Variable CSP

12.4.1 Arc Consistency

A [illegible] onstraint recording is *arc consistency*[27, 57, 61]. Fo[illegible] of the constraint between variables x and y, if va[illegible] a, then there is no consistent value that can be as[illegible] Thus, value a can be deleted from the domain of variable x, since no solution to the problem can assign a to x. In general, for each pair of variables X and Y that are related by a binary constraint, we remove from the domain of X any values that do not have at least one consistent assignment to Y, and vice versa.

Several iterations may be required to achieve complete arc consistency. For example, once a is removed from the domain of x, then if f is assigned to z, there is no consistent assignment to x, resulting in the deletion of f from the domain of z. Similarly, this results in the deletion of d from the domain of y. The algorithm terminates when no additional values are removed by examining all the arcs. In this particular case, arc consistency alone is sufficient to solve the problem, returning the solution $x = b$, $z = e$, and $y = c$.

12.4.2 Path Consistency

Path consistency is a generalization of arc consistency where instead of considering pairs of variables, we examine triples of constrained variables. Returning to Figure 12.2, without performing arc consistency first, consider the combinations of assignments to variables x and y that are allowed by the constraint between them. In particular, if $x = b$ and $y = d$, then there is no possible assignment to variable z that is consistent with these two assignments. Thus, we can remove the ordered pair (b, d) from the set of pairs allowed by the constraint between x and y.

The effect of performing arc or path consistency before backtracking is that the resulting search space can be dramatically reduced. As we saw above, in some cases, preprocessing of the constraints can eliminate the need for search entirely. Arc and path consistency can be generalized to consider larger groupings of variables, called $k - consistency$. The complexity of performing such consistency checks is polynomial in k, the number of variables considered simultaneously. At some point,

higher-order con me less effective than backtracking search. In practi s almost always worth doing, and path consistency orming before backtracking search.

12.5 Heuristic Repair

Backtracking searches a space of consistent partial assignments to variables, in that all constraints among assigned variables are satisfied, looking for a complete consistent assignment to the variables, or in other words a solution. An alternative approach is to search a space of inconsistent but complete assignments to the variables, until a consistent complete assignment is found. This approach is known as heuristic repair[60]. For example, in the N-queens problem, we can place all N queens on the board at the same time, and move the queens one at a time until a solution is found. The obvious heuristic, called min-conflicts, is to move a queen that is in conflict with the most other queens, to a position where it conflicts with the fewest other queens.

What is surprising about this strategy is how well it performs, relative to backtracking. While intelligent backtracking can solve hundred-queen problems, heuristic repair can solve million-queen problems, often with only about 50 individual queen moves, assuming that the initial queen placement minimizes the number of conflicts. This strategy has been extensively explored in the context of boolean satisfiability, where it is referred to as GSAT[89]. GSAT can satisfy difficult formulas with several thousand variables, whereas the best backtracking approach, the Davis-Putnam algorithm with unit propagation[14], can only satisfy difficult formulas with several hundred variables.

The main drawback of this approach is that it is not complete, in the sense that it is not guaranteed to find a solution in a finite amount of time, even if one exists. If there is no solution, heuristic algorithms will run forever, whereas backtracking will eventually discover that a problem is not solvable.

11 rison to Single-Agent and Two-Game Search

W action problems appear somewhat different from sir ng problems and two-player games, there is a strong similarity among the algorithms employed. For example, backtracking can be viewed as a form of branch-and-bound, where a node is pruned when a constraint is violated. In this case, the cost function is the total number of constraints violated so far, the function is monotonic non-decreasing as we go down a branch, and a goal is a state of zero cost. Similarly, heuristic repair can be viewed as a heuristic search with this same evaluation function and goal state.

Chapter 13

Parallel Search Algorithms

Search is a very computation-intensive process, as we have seen. As a result, there is motivation to apply the most powerful computers available to the task. In terms of raw computing cycles, the most powerful machines are parallel processors. Thus, we now turn our attention to parallel search algorithms. These can be divided into three very broad and general classes, parallel node generation and evaluation, parallel window search, and tree splitting algorithms. We consider each in turn.

13.1 Parallel Node Generation

Perhaps the simplest and most obvious approach to parallel search is to parallelize the generation and evaluation of each node. For example, the Deep Blue chess machine uses parallel custom VLSI hardware to generate the legal moves from a given position, and to apply the heuristic static evaluation function to the child nodes.

There are several limitations to this approach, however. The first is that the technique is inherently domain-specific, and a different algorithm must be hand-crafted for each application domain. This may be a difficult and time-consuming process. As a result, there is little that can be said about this approach in general. The second drawback is that the total amount of available parallelism using this technique is limited by the domain. While some domains are more highly parallel than others, there will always be a maximum amount of parallelism that

ca any domain. Beyond that amount, additional pr ed up the search any further.

1 l Window Search

The next approach is called *parallel window search*. It was first proposed in the context of alpha-beta minimax search, and subsequently applied to Iterative-Deepening-A*.

13.2.1 Alpha-Beta Minimax

The first parallel window search algorithm was proposed by Gerard Baudet[3] for parallelizing alpha-beta search. In the standard serial implementation of alpha beta, the initial call of the recursive algorithm is on the root of the tree with a lower bound alpha of $-\infty$, and an upper bound beta of ∞. The algorithm then searches the tree and returns the exact minimax value.

Alternatively, if we search the tree with initial alpha and beta values that are closer together, called an alpha-beta window, then the value of more nodes will fall outside of the bounds, resulting in more pruning, and hence a more efficient search. If the minimax value of the root node falls within the initial alpha-beta window, then the algorithm will return the minimax value. If the value falls outside the window, then the algorithm will simply return that the true minimax value is either less than the lower bound alpha, or greater than the upper bound beta. This is known as failing low or failing high, respectively.

Given multiple processors, we can divide up the total range from $-\infty$ to ∞ into a series of contiguous but smaller alpha-beta windows, and give each processor the entire tree to search from the same root, but with different initial values of alpha and beta. For example, with two processors, we might have one search the tree with an initial alpha value of $-\infty$, and an initial beta value of 0, while the other processor searches the same entire tree, but with alpha equal to 0 and beta equal to ∞. One of these processors will fail high or low, and the other will return the actual minimax value of the root, and faster than if it had started with the entire alpha-beta window.

While this al[illegible]nd extremely simple, its practical usefulness is limi[illegible] that even with a *minimal window* equal to the mir[illegible]ree, meaning that both alpha and beta are equal to[illegible]x value, a search to verify that the minimax value is[illegible]eta still has asymptotic time complexity $O(b^{d/2})$. This is because a strategy for Max must be explored to show that the minimax value is not less than this value, and a strategy for Min must be explored to show that the value is not greater. Thus, even with an arbitrary number of processors, the asymptotic complexity will still be $O(b^{d/2})$. In contrast, good node ordering techniques can approach this complexity for a serial implementation of alpha beta.

13.2.2 IDA*

The idea of parallel window search can also be applied to Iterative-Deepening-A* (IDA*)[18]. In this case, different processors simultaneously search the entire tree, but with different cutoff thresholds.

Given p processors, the first p different cutoff thresholds are assigned to different processors, and the search begins. As soon as a processor completes its assigned iteration, it begins the iteration with the next cutoff threshold that is not yet assigned to a processor. This assumes that the successive cutoff thresholds can be predicted in advance. For example, on the sliding-tile puzzles with the Manhattan distance heuristic function, the successive thresholds will increase by two, starting with the Manhattan distance of the initial state. This is because the g value of a child is always one greater than that of its parent, and the h value of a child is either one greater or one less than that of its parent, therefore the $f = g + h$ value of a child is either the same as its parent, or two greater.

If all the thresholds can't be predicted in advance, then as soon as the first iteration is started, nodes will be pruned, and their costs will represent cutoff thresholds for future iterations, that can be started immediately on other processors. For example, again consider the sliding-tile puzzles with the Manhattan distance function, and assume that we didn't know what the cutoff thresholds would be in advance. Assume that the root has a Manhattan distance of 10 moves. We would start the initial iteration, with a threshold of 10. Almost immediately,

no uld be generated and pruned. Thus, we could st on another processor with a cutoff threshold of 12 uld almost immediately generate nodes of cost 14 te another iteration on another processor with th etc. Of course, parallel window search is not a best-first search, because iterations with larger thresholds begin before all the iterations with smaller thresholds have been completed.

The main drawback of parallel window search applied to IDA* is that the time to complete the algorithm will be dominated by the time of the goal iteration, or the previous iteration if the goal is found very early in the last iteration. The only real savings are due to running of the previous iterations in parallel, saving the time to perform these iterations. Previously we have argued, however, that the time to complete the non-goal iterations will not have a significant impact on the overall complexity. Thus, since each iteration runs on a single processor, the amount of parallel speedup will be strictly limited.

13.3 Tree Splitting

Perhaps the most natural way to parallelize a heuristic search is to have different processors search different parts of the search tree. Due to the small granularity of the basic computational unit of node generation and evaluation, there is at least the potential for enormous speedup, limited only by the number of processors. For example, consider searching a uniform binary tree of depth d with 2^n processors. In $\log_2 n$ time, the 2^n processors could be distributed to the nodes at level n. The rest of the tree could then be searched in parallel, and finally again in $\log_2 n$ time the processors could combine their results to return a single answer. Assuming that the tree is large compared to the number of processors, so that the time of the parallel searches dominates the total time, the speedup of this algorithm would approach 2^n. Note that this scheme applies to any depth-first search algorithm, and requires no communication among the different processors, except at the initial distribution phase and the final combination phase.

The difficulty here is that most search trees of practical interest are not uniform, but highly irregular. This requires some type of load

balancing so that terminating the search of their subtrees early can assist in pruning other subtrees. A number of effective algorithms exist for dynamic load balancing, one of which we describe below.

13.3.1 Distributed Tree Search

Distributed Tree Search (DTS) is a parallel tree-search algorithm designed for effective load balancing on irregular trees [24]. It is designed to scale up to an arbitrary number of processors. As a consequence of this constraint, it does not rely on any centralized control or shared memory, features that are not scalable, but become saturated by a large enough number of processors. Each processor runs the same code. The algorithm is also fairly general, making no commitment to any particular type of tree search, or even to a particular processor allocation strategy.

The best way to think about DTS is to think about processors moving over a search tree. In reality, nodes are sent between processors in messages, but the image of processors moving around a tree makes the algorithm easier to understand. Associated with each node of a search tree is a single process. At any given time, that process may be running on a particular processor, dormant on a processor, or not even created yet.

Initially, the process associated with the root node is started. All of the processors are assigned to this process, but the process itself runs on only one of them. This process expands the root node, generating each of its children, and starts a separate process for each child. Next, it allocates its processors to the child processes, according to some processor allocation strategy. For example, in a breadth-first allocation strategy, the processors would be divided among the child processes as evenly as possible. In a depth-first allocation strategy, all the processors would be assigned to one of the child processes. Even the processor on which the root process is running is assigned to a child process. All of the child processes that were assigned at least one processor start executing, on one of their assigned processors. Next, the root process goes to sleep, awaiting a message from one of its child processes. Note that while the root process resides on some processor, that processor

ha to a child process, so the processor is not idle. processes has an associated node, and a set of pr is to search the subtree rooted at their node, us t of processors. When they complete this task, th of the search, along with their processors, to the parent process that called them, via a wakeup message to the parent process. When the dormant parent process is awakened by a message from one of its child processes, it checks to see if this was the last child process to complete. If so, it computes whatever the value of its node is, and returns this value, along with the returned processors, to its parent process. Otherwise, it takes the processors that were returned by the child and reassigns them to the remaining child processes that have not yet completed their work. If any of those processes are receiving their first processor as result of this reallocation, those processes are started by the parent. Otherwise, if all the child processes receiving more processors are already running, the parent processor sends a message to the child processes, with the additional processors that are being assigned to it. Once all the processors have been reassigned, the parent process goes to sleep again.

All the processes run the same algorithm we have described above, regardless of their place in the search tree. Initially, all the processors are distributed down the tree, and very quickly we reach a situation where each node or process is assigned a single processor.

Consider how DTS behaves when a node has only a single processor assigned to it. The process associated with the node obviously runs on this processor. It expands its node, generating a separate process for each child node. It then assigns its processor to one of its children, and goes to sleep, until the child process completes the search of its subtree, and wakes up the parent process. When the parent process is awakened, it assigns the processor to the next child, and goes to sleep again, until the last child process returns. In the meantime, the child process that is active expands its nodes, generates a process for each child, and assigns the processor to one of its children. This is exactly what a depth-first search does. The dormant parent processes correspond to frames of the recursion stack. Thus, we see that the special case of DTS with a single processor is just depth-first search. For efficiency, at some level of the tree, we would switch to actual serial depth-first search, since simple

recursion is much ... creating multiple processes under most systems.

What does D... large number of processors? In that case, assuming th... signed to the children breadth-first, DTS performs a ... t search of the tree, generating all the nodes at a given level of the tree essentially simultaneously.

13.3.2 Branch-and-Bound Pruning

While DTS performs effective load balancing on an irregular tree, we assumed that the search tree is fixed, in the sense that results from one part of the tree do not have any effect on whether other parts of the tree are searched. This is true of brute-force search, and even heuristic searches such as IDA*. In any non-goal iteration of IDA*, the part of the tree that must be searched is determined by the node costs and the cutoff threshold, and is not affected by search results from other parts of the tree.

However, branch-and-bound algorithms, such as DFBnB or alpha-beta, are different. In this case, whether a branch of the tree is searched at all depends on results from other parts of the tree. This makes these algorithms particularly difficult to parallelize. For example, consider a parallel breadth-first search of a tree, with a very large number of processors. Since the tree is searched breadth-first, we will generate every node down to a given depth, before we have any complete solutions or any bounds on solution quality. While this will provide some parallel speedup, the total work performed will be much greater than a serial branch-and-bound search, since the breadth-first search will not prune any of the tree.

More realistically, however, the number of processors will usually be much smaller than the number of leaf nodes in the tree. In that case, again assuming a breadth-first allocation of p processors, the processors will quickly migrate down to the point where there is one node per processor. Then, p different depth-first searches will proceed in parallel, followed by another brief phase of combining the results.

Single-agent branch-and-bound is relatively easy to parallelize using this scheme. The reason is that the only bound involved is a global bound on the entire tree, equal to the cost of the best complete solution

fou tial solutions that equal or exceed this cost can be enever a processor finds a better solution than th it simply broadcasts the cost of this solution to all s, and they continue their search, using the new pr result, the parallel algorithm is likely to do no more total work than a serial algorithm. It fact, it has been observed that a single-agent parallel DFBnB often generates fewer total nodes than its serial counterpart. The reason is that with p parallel searches proceeding simultaneously, good solutions are found quicker than by the serial algorithm, resulting in lower bounds sooner, and hence more pruning.

An algorithm like alpha-beta minimax is more difficult to parallelize. The reason is that the particular alpha and beta bounds are local to a given subtree, and don't apply to the tree as a whole. Below we analyze the speedup that results from a straightforward parallel implementation of such an algorithm.

13.3.3 Analysis of Parallel Branch-and-Bound

Assume we are searching a uniform tree of branching factor b and depth d, using p processors, with an algorithm such as alpha beta. Assume that the effective branching factor of the serial algorithm is b^x, for an overall serial time of b^{xd}. For example, for alpha beta in the best case, $x \approx 1/2$. The speedup of a parallel algorithm is defined as the time of the best serial algorithm, divided by the time of the parallel algorithm.

Our parallel implementation will proceed in three phases. First, using a breadth-first allocation strategy, the p processors will be distributed down the tree to a level where there are p nodes, one per processor. This will occur at depth $\log_b p$. Since this distribution is done in parallel, at each time step it will proceed one level lower in the tree, and hence will only take $\log_b p$ time.

In the second phase, each processor will search the subtree rooted at its node in parallel, using the serial algorithm. Our assumption here is that the bounds developed in one subtree will not affect the pruning of the other subtrees, implying that the individual searches are independent of each other. The depth of each of these searches is $d - \log_b p$. Since the individual searches use the serial branch-and-bound

algorithm, the ti one, which is the same as the time to perform them

Finally, the r idual searches must be propagated back up the tree. on phase, the time to perform this step in parallel w he total time taken by the parallel algorithm is $b^{x(d-\log_b p)} + 2\log_b p$. We'll assume that the tree is large compared to the number of processors, and hence the total time is dominated by the time to perform the individual parallel searches, or $b^{x(d-\log_b p)}$.

The speedup is the serial time divided by the parallel time, or

$$b^{xd}/b^{x(d-\log_b p)} = b^{xd-x(d-\log_b p)} = b^{xd-xd+x\log_b p} = b^{x\log_b p} = b^{\log_b p^x} = p^x$$

If x is one, then we get a perfect speedup of p on p processors, which is what we expect, since this corresponds to the brute-force case with no pruning. If $x = 1/2$ however, which corresponds to the best-case of alpha beta, for example, then we only get a speedup of $p^{1/2} = \sqrt{p}$, which is quite poor. This means, for example, that with a hundred processors, we only achieve a speedup of ten. The more effective that pruning is in the serial algorithm, the worse that the parallel algorithm performs. As a result, algorithms like alpha-beta minimax are notoriously difficult to effectively parallelize, and this remains an interesting research challenge.

Bibliography

[1] Abramson, Bruce, personal communication, 1985.

[2] Abramson, B., An explanation of and cure for minimax pathology in *Uncertainty in Artificial Intelligence*, L. Kanal and J. Lemmer, eds., North-Holland, 1986, pp. 495-504.

[3] Baudet, G., "The design and analysis of algorithms for asynchronous multiprocessors", Ph.D. dissertation, Dept. Computer Science, Carnegie Mellon University, Pittsburgh, PA., Apr. 1978.

[4] Billings, D., A. Davidson, J. Schaeffer, and D. Szafron, The challenge of poker, *Artificial Intelligence*, Vol. 134, No. 1-2, January 2002, pp. 201-240.

[5] Boddy, M., and T. Dean, Solving time-dependent planning problems, in *Proceedings of the International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August, 1989, pp. 979-984.

[6] Bratko, I., *PROLOG: Programming for Artificial Intelligence*, Addison-Wesley, 1986, pp. 265-273.

[7] Buro, M., Improving heuristic mini-max search by supervised learning, *Artificial Intelligence*, Vol. 134, No. 1-2, January 2002, pp. 85-100.

[8] Campbell, M., A.J. Hoane, and F. Hsu, Deep Blue, *Artificial Intelligence* Vol. 134, No. 1-2, Jan. 2002, pp. 57-83.

[9] Carroll, C.M., *The Great Chess Automaton*, Dover Publications, Inc., New York, NY, 1975

[10] ..., S. Ghose, A. Acharya, and S.C. de Sarkar, ...n restricted memory, *Artificial Intelligence*, Vol. ...ber 1989, pp. 197-221.

[11] ...nd R.E. Korf, "A unified theory of heuristic evaluation functions and its application to learning", in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, 1986.

[12] Condon, J.H., and K. Thompson, Belle chess hardware, in *Advances in Computer Chess III*, Pergamo Press, 1982.

[13] Culberson, J., and J. Schaeffer, Pattern Databases, *Computational Intelligence*, Vol. 14, No. 4, pp. 318-334, 1998.

[14] Davis, M., and H. Putnam, A computing procedure for quantification theory, *Journal of the Association for Computing Machinery*, Vol. 7, 1960, pp. 201-215.

[15] Dechter, R., and J. Pearl, Generalized best-first search strategies and the optimality of A*, *Journal of the Association for Computing Machinery*, Vol. 32, No. 3, July 1985, pp. 505-536.

[16] Dechter, R., Pearl, J. 1988. Network-Based Heuristics for Constraint-Satisfaction Problems, *Artificial Intelligence*, Vol. 34, No. 1, 1987, pp. 1-38.

[17] Dennett, D. C., Can machines think? Deep Blue and beyond, *Journal of the International Computer Chess Association*, Vol. 20, No. 4, Dec. 1997, pp. 215-223.

[18] Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik*, Vol. 1, 1959, pp. 269-71.

[19] Doran, J.E., and D. Michie, Experiments with the Graph Traverser program, *Proceedings of the Royal Society A*, Vol. 294, 1966, pp. 235-259.

[20] Dudeney, H.E., *The Canterbury Puzzles (and Other Curious Problems)*, E.P. Dutton, New York, 1908.

[21] Dunkel, O. ncerning advanced problem 3918, *Amer. Math* 1941, pp. 219.

[22] Ebeling, C. *ves*, MIT Press, Cambridge, MA, 1987.

[23] Felner, A., R. Meshulam, R. Holte, and R. Korf, Compressing pattern databases, *Proceedings of the National Conference on Artificial Intelligence (AAAI04)*, San Jose, CA, July 2004, pp. 638-643.

[24] Ferguson, C., and R.E. Korf, Distributed tree search and its application to alpha-beta pruning, in *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, Minneapolis, MN, August, 1988.

[25] Fikes, R., and N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, Vol. 2, 1971, pp. 189-208.

[26] Frame, J.S., Solution to advanced problem 3918, *Amer. Math. Monthly*, Vol. 48, 1941, pp. 216-217.

[27] Freuder, E.C. 1982. A sufficient condition for backtrack-free search. *Journal of the Association of Computing Machinery* 29(1):24-32.

[28] Garey, M.R., and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[29] Gaschnig, J. *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis. Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.

[30] Ghosh, S., A. Mahanti, and D.S. Nau, ITS: An efficient limited-memory heuristic tree search algorithm, *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, July 1994, pp. 1353-1358.

[31] Ginsberg, M.L., GIB: Imperfect information in a computationally challenging game, *Journal of Artificial Intelligence Research*, Vol. 14, 2001, pp. 303-358.

[33] Mayer, and M. Yung, Criticizing solutions to relds powerful admissible heuristics, *Information* No. 3, 1992, pp. 207-227.

[34] nd G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, Vol. 14, 1980, pp. 263-313.

[34] Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. SSC-4, No. 2, July 1968, pp. 100-107.

[35] Hart, T.P., and D.J. Edwards, The alpha-beta heuristic, M.I.T. Artificial Intelligence Project Memo, Massachusetts Institute of Technology, Cambridge, MA., October 1963.

[36] Johnson, W.A. and W.E. Story, Notes on the 15 puzzle, *American Journal of Mathematics*, Vol. 2, 1879, pp. 397-404.

[37] Karp, R.M., and J. Pearl, Searching for an optimal path in a tree with random costs, *Artificial Intelligence*, Vol. 21, No. 1-2, 1983, pp. 99-117.

[38] Klavzar, S., and U. Milutinovic, Simple explicit formulas for the Frame-Stewart numbers, *Annals of Combinatorics*, Vol. 6, 2002, pp. 157-167.

[39] Knuth, D.E., and R.E. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence*, Vol. 6, No. 4, 1975, pp. 293-326.

[40] Korf, R.E., Toward a model of representation changes, *Artificial Intelligence*, Vol. 14, No. 1, pp. 41-78, October 1980.

[41] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[42] Korf, R.E., Linear-space best-first search, *Artificial Intelligence*, Vol. 62, No. 1, July 1993, pp. 41-78.

[43] Korf, R.E., h algorithms, *Computing Surveys*, Vol. 27, No. 337-339.

[44] Korf, R.E., algorithm for number partitioning, *Artificial In* No. 2, December 1998, pp. 181-203.

[45] Korf, R.E., Finding optimal solutions to Rubik's Cube using pattern databases, *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, July, 1997, pp. 700-705.

[46] Korf, R.E. Linear-time disk-based implicit graph search, *Journal of the A.C.M.*, Vol. 55, No. 6, December 2008, pp. 26-1 to 26-40.

[47] Korf, R.E., and D.M. Chickering, Best-first minimax search, *Artificial Intelligence*, Vol 84, No. 1-2, July 1996, pp. 299-337.

[48] Korf, R.E, and A. Felner, Disjoint pattern database heuristics, *Artificial Intelligence*, Vol. 134, No. 1-2, Jan. 2002, pp. 9-22.

[49] Korf, R.E., and A. Felner, Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, Jan. 2007, pp. 2334-2329.

[50] Korf, R.E., M. Reid, and S. Edelkamp, Time complexity of Iterative-Deepening-A*, *Artificial Intelligence*, Vol 129, No. 1-2, June 2001, pp. 199-218.

[51] Korf, R.E., and P. Schultze, Large-scale, parallel breadth-first search, *National Conference on Artificial Intelligence (AAAI-05)*, Pittsburgh, PA, July, 2005, pp. 1380-1385.

[52] Korf, R.E., and L.A. Taylor, Finding optimal solutions to the twenty-four puzzle, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, Aug. 1996, pp. 1202-1207.

[53] Korf, R.E, W. Zhang, I. Thayer, and H. Hohwald, Frontier search, *Journal of the Association for Computing Machinery*, Vol. 52, No. 5, Sept. 2005, pp. 715-748.

[54] K. Lenstra, A.H.G. Rinnooy Kan, and D.B. *he Traveling Salesman Problem: A Guided Tour Optimization*, John Wiley, New York, 1985.

[55] *atical Puzzles of Sam Loyd*, Selected and Edited by Martin Gardner, Dover, New York, 1959.

[56] Luckhardt, C.A., and K.B. Irani, An algorithmic solution of N-person games, *Proceedings of the National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, August, 1986, pp. 158-162.

[57] Mackworth, A.K., Consistency in networks of relations, *Artificial Intelligence* Vol. 8, No. 1, 1977, pp. 99-118.

[58] Mero, L., A heuristic search algorithm with modifiable estimate, *Artificial Intelligence*, Vol. 23, 1984, pp. 13-27.

[59] Michon, G., Recursive Random Games, Ph.D. thesis, Computer Science Department, University of California, Los Angeles, CA, 1983.

[60] Minton, S., M.D. Johnston, A.B. Philips, and P. Laird, Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence*, Vol. 58, No. 1-3, December 1992, pp. 161-205.

[61] Montanari, U., Networks of constraints: Fundamental properties and applications to picture processing, *Information Science*, Vol. 7, 1974, pp. 95-132.

[62] Moore, E.F., and C.E. Shannon, Reliable circuits using less reliable relays, *Journal of the Franklyn Institute*, Vol. 262, 1956, pp. 191-208.

[63] Muller, M., Computer go, *Artificial Intelligence*, Vol. 134, No. 1-2, January 2002, pp. 185-219.

[64] Nau, D.S., An investigation of the causes of pathology in games, *Artificial Intelligence*, Vol. 19, 1982, pp. 257-278.

[65] Newell, A., A. Simon, Chess playing programs and the pro *IBM Journal of Research and Development*, . 320-335. Reprinted in *Computers and Though* J. Feldman (Eds.), McGraw-Hill, New York,

[66] Newell, A., and H.A. Simon, Computer science as empirical inquiry: symbols and search, *Communications of the Association for Computing Machinery*, Vol. 19, No. 3, March 1976, pp. 113-126.

[67] Newell, A., Reasoning, problem solving, and decision processes: The problem space as a fundamental category, in *Attention and Performance VIII*, R. Nickerson, (Ed.), Erlbaum, Hillsdale, NJ 1980.

[68] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[69] Nilsson, N.J., personal communication, Jan. 1999.

[70] Papadimitriou, C.H., and K. Steglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, N.J., 1982.

[71] Patrick, B.G., M. Almulla, and M.M. Newborn, An upper bound on the complexity of iterative-deepening-A*, *Annals of Mathematics and Artificial Intelligence*, Vol. 5, 1992, pp. 265-278.

[72] Pearl, J., The solution for the branching factor of the Alpha-Beta pruning algorithm and its optimality, *Communications of the Association of Computing Machinery*, Vol. 25, No. 8, 1982, pp. 559-64.

[73] Pearl, J., On the nature of pathology in game searching, *Artificial Intelligence*, Vol. 20, No. 4, 1983, pp. 427-453.

[74] Pearl, J. *Heuristics*, Addison-Wesley, Reading, MA, 1984.

[75] Pemberton, J., and R.E. Korf, Incremental planning on graphs with cycles, *Proceedings of the First International Conference on*

ems (AIPS-92), University of Maryland, College 1992, pp. 179-188.

[76] el's chess player, *Southern Literary Messenger. ar Allen Poe: Essays and Reviews*, Library of rk, NY, 1984.

[77] Pohl, I., Bi-directional search, in *Machine Intelligence 6*, Meltzer, B. and D. Michie (Eds.), American Elsevier, New York, 1971, pp. 127-140.

[78] Powley, C., and R.E. Korf, Single-agent parallel window search, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 13, No. 5, May 1991, pp. 466-477.

[79] Prieditis, A.E., Machine discovery of effective admissible heuristics, *Machine Learning*, Vol. 12, 1993, pp. 117-141.

[80] Purdom, P.W. 1983. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence* 21(1,2):117-33

[81] Rao, V.N., V. Kumar, and R.E. Korf, Depth-first vs. best-first search, *Proceedings of the National Conference on Artificial Intelligence, (AAAI-91)*, Anaheim, CA, July, 1991, pp. 434-440.

[82] Ratner, D., and M. Warmuth, Finding a shortest solution for the NxN extension of the 15-Puzzle is intractable, *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, 1986, pp. 168-172.

[83] http://hedgehog.math.arizona.edu/ reid/Rubik/index.html.

[84] Russell, S., Efficient memory-bounded search methods, *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, Aug., 1992.

[85] Samuel, A.L., Some studies in machine learning using the game of checkers, IBM Journal of Research and Development, Vol. 3, 1959, pp. 211-229. Reprinted in *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963, pp. 71-105.

[86] Sarkar, U.K., P.P. Chakrabarti, S. Ghose, and S.C. DeSarkar, Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, *Artificial Intelligence*, Vol. 50, No. 2, July, 1991, pp. 207-221.

[87] Schaeffer, J., *One Jump Ahead: Challenging Human Supremacy in Checkers*, Springer-Verlag, New York, 1997.

[88] Schofield, P.D.A., Complete solution of the eight puzzle, in *Machine Intelligence 3,* B. Meltzer and D. Michie (eds.), American Elsevier, New York, 1967, pp. 125-133.

[89] Selman, B., H. Levesque, and D. Mitchell, A new method for solving hard satisfiability problems, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, Ca., July 1992, pp. 440-446.

[90] A.K. Sen and A. Bagchi, Fast recursive formulations for best-first search that allow controlled use of memory, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August, 1989, pp. 297-302.

[91] Shannon, C.E., Programming a computer for playing chess, *Philosophical Magazine*, Vol. 41, 1950, pp. 256-275.

[92] Sheppard, B. World-championship-caliber Scrabble, *Artificial Intelligence*, Vol. 134, No. 1-2, Jan. 2002, pp. 241-275.

[93] Simon, H.A., *The Sciences of the Artificial*, 2nd edition, M.I.T. Press, Cambridge, MA, 1981.

[94] Slate, D.J., and L.R. Atkin, CHESS 4.5 - The Northwestern University chess program, in *Chess Skill in Man and Machine*, Frey, P.W. (Ed.), Springer-Verlag, New York, 1977, pp. 82-118.

[95] Stewart, B.M., Solution to advanced problem 3918, *Amer. Math. Monthly*, Vol. 48, 1941, pp. 217-219.

[96] ...d W.M. Tyson, An analysis of consecutively ...st search with applications in automated deduc... *of the International Joint Conference on Artifi... (IJCAI-85)*, Los Angeles, CA, August, 1985, pp.

[97] Taylor, L., and R.E. Korf, Pruning duplicate nodes in depth-first search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-93)*, Washington D.C. July 1993, pp. 756-761.

[98] Tesauro, G., Programming backgammon using self-teaching neural nets, *Artificial Intelligence*, Vols 134, No. 1-2, January 2002, pp. 181-199.

[99] Thompson, K., Retrograde analysis of certain endgames, *Journal of the International Computer Chess Association*, Vol. 9, No. 3, 1986, pp. 131-139.

[100] Von Neumann, J., and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, Princeton, NJ, 1947.

[101] Wah, B.W., and Y. Shang, A comparison of a class of IDA* search algorithms, *International Journal of Artificial Intelligence Tools*, Vol. 3, No. 4, Oct. 1995, pp. 493-523.

[102] Zhang, W., and R.E. Korf, Performance of linear-space search algorithms, *Artificial Intelligence*, Vol. 79, No. 2, Dec. 1995, pp. 241-292.

[103] Zhang, W. Depth-first branch-and-bound vs. local search: A case study, Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), Austin, TX, August 2000, pp.930-935.

[104] Zhou, R., and E. Hansen, Breadth-first heuristic search, *Artificial Intelligence*, Vol 170, No. 4-5, April, 2006, pp. 385-408.