



Branch-and-Bound Search Algorithms and Their Computational Complexity

Weixiong Zhang

USC/Information Sciences Institute

May 1996

ISI/RR-96-443

DTIC QUALITY INSPECTED 2

19960924 050

INFORMATION
SCIENCES
INSTITUTE



310/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Branch-and-Bound Search Algorithms and Their Computational Complexity

Weixiong Zhang

USC/Information Sciences Institute

May 1996

ISI/RR-96-443

REPORT DOCUMENTATION PAGE			FORM APPROVED OMB NO. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimated or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE May 1996		3. REPORT TYPE AND DATES COVERED Research Report
4. TITLE AND SUBTITLE Branch-and-Bound Search Algorithms and Their Computational Complexity			5. FUNDING NUMBERS DARPA: #MDA972-94-2-0010	
6. AUTHOR(S) Weixiong Zhang				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) USC INFORMATION SCIENCES INSTITUTE 4676 ADMIRALTY WAY MARINA DEL REY, CA 90292-6695			8. PERFORMING ORGANIZATION REPORT NUMBER ISI/RR-96-443	
9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 N. Fairfax Drive Arlington, VA 22203-1714			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12A. DISTRIBUTION/AVAILABILITY STATEMENT UNCLASSIFIED/UNLIMITED			12B. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Branch-and-bound (BnB) is a general problem-solving paradigm that has been studied extensively in the areas of computer science and operations research, and has been employed to find optimal solutions to computation-intensive problems. Thanks to its generality, BnB takes many search algorithms, developed for different purposes, as special cases. Some of these algorithms, such as best-first search and depth-first search, are very popular, some, such as iterative deepening, recursive best-first search and constant-space best-first search, are known only in the artificial intelligence area. Because it was studied in different areas, BnB has been described under different formulations. In the first part of this paper, we give comprehensive descriptions of the BnB method and of these search algorithms, consolidating the basic features of BnB. In the second part, we summarize recent theoretical development on the average-case complexity of BnB search algorithms.				
14. SUBJECT TERMS average-case complexity, branch-and-bound, combinatorial optimization, phase transitions, problem solving, search			15. NUMBER OF PAGES 36	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNLIMITED	

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement.

Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."

DOE - See authorities.

NASA - See Handbook NHB 2200.2.

NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.

DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.

NASA - Leave blank.

NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17.-19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Branch-and-Bound Search Algorithms and Their Computational Complexity

Weixiong Zhang *

Information Sciences Institute and
Department of Computer Science
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
Email: zhang@isi.edu

Abstract

Branch-and-bound (BnB) is a general problem-solving paradigm that has been studied extensively in the areas of computer science and operations research, and has been employed to find optimal solutions to computation-intensive problems. Thanks to its generality, BnB takes many search algorithms, developed for different purposes, as special cases. Some of these algorithms, such as best-first search and depth-first search, are very popular, some, such as iterative deepening, recursive best-first search and constant-space best-first search, are known only in the artificial intelligence area. Because it was studied in different areas, BnB has been described under different formulations. In the first part of this paper, we give comprehensive descriptions of the BnB method and of these search algorithms, consolidating the basic features of BnB. In the second part, we summarize recent theoretical development on the average-case complexity of BnB search algorithms.

*The author was supported by DARPA Contract, #MDA972-94-2-0010.

Contents

1	Introduction	3
2	Branch-and-Bound	3
2.1	Branch-and-bound method	3
2.2	An example	4
2.3	State space and the state-space tree	6
2.4	Search strategies	7
2.4.1	Best-first search	7
2.4.2	Depth-first search	9
2.4.3	Iterative deepening	10
2.4.4	Recursive best-first search	12
2.4.5	Constant-space best-first search	14
2.5	Graph vs. tree	16
2.6	Historical remarks and discussions	17
3	Computational Complexity	18
3.1	Incremental random trees	19
3.2	Problem complexity and cost of an optimal goal	20
3.3	Algorithm complexity	21
3.3.1	Best-first search	21
3.3.2	Depth-first search	22
3.3.3	Iterative deepening	23
3.3.4	Recursive and constant-space best-first searches	24
3.4	Summary and complexity transition	25
3.5	Historical remarks and discussions	26
4	Conclusions	27

1 Introduction

Branch-and-bound (BnB) is a principal problem-solving paradigm for finding exact solutions to combinatorial optimization problems in many areas, including computer science and operations research. It has been broadly used to solve computation-intensive, typically NP-hard [14], problems, such as the Traveling Salesman Problem [3], sequencing and scheduling problems [2], and the knapsack problem [40], to name just a few.

BnB's generality allows it to encapsulate many search algorithms as special cases, including the well-known best-first search and depth-first search [32, 37, 47], as well as iterative deepening [29], recursive best-first search [30], and constant-space best-first search [6, 52], which are relatively unknown outside the area of artificial intelligence.

The main goal of this paper is to give a state-of-the-art account of the BnB method, including the algorithms and their complexity. Because BnB was studied in different areas, it has been described using different terminology and formulations. In the first part of this paper (Section 2), we give a comprehensive presentation of BnB, and discuss the BnB search algorithms. In the second part of the paper (Section 3), we summarize recent theoretical results on the computational complexity, particularly the average-case complexity, of BnB search algorithms. Finally, our conclusions appear in Section 4.

2 Branch-and-Bound

2.1 Branch-and-bound method

Formally, a combinatorial optimization problem is a triple $\Phi = \langle D, S, f \rangle$, where D is a set of problem instances, $S \subset D$ is the set of feasible solutions, and $f : D \rightarrow R$ is the cost function, where R is the set of real numbers. A solution $s \in S$ is optimal if $f(s) = \min \{f(s') \mid \forall s' \in S\}$.

The underlying idea of BnB is to take a given problem that is difficult to solve directly, and decompose it into smaller partial problems in such a way that a solution to a partial problem is also a solution to the original problem. This decomposition method is recursively applied to a partial problem until it can be solved directly or is proven not to lead to an optimal solution.

BnB consists of three major components: *branching rules*, *bounding rules*, and *search strategies*. We discuss the first two components below, but postpone the description of search strategies until Section 2.4.

Branching rules are used to decompose a problem into partial problems. This is generally done by including some entities in the solution to a partial problem

and/or excluding some entities from the solution. Because branching rules are problem dependent, developing effective branching rules for a specific problem requires insight into that problem. A good rule of thumb is to decompose a problem in such a way that a partial problem can be generated only once. In other words, the sets of partial problems that can be generated from the immediate partial problems of the original problem must constitute a partition of all possible partial problems. One scheme that we can use is the principle of *inclusion-and-exclusion* [39]. Briefly, this principle requires an entity to be excluded from the solution to a partial problem if the entity has been included in the solution to a previously generated sibling partial problem. This principle is very general and can be applied to most combinatorial optimization problems. Section 2.2 gives an example of this principle.

Bounding rules are used to determine whether a partial problem needs to be decomposed further. These rules must guarantee the completeness of BnB, meaning that an optimal solution must be found if it exists. This guarantee comes from the monotonicity of the cost function used, which is the basis of BnB. A cost function f is called *monotonic* if the cost of a partial problem n' , $f(n')$, is greater than or equal to the cost of its parent problem n , $f(n)$, i.e., $f(n') \geq f(n)$. With a monotonic cost function, a partial problem can be excluded from further consideration if its cost is greater than or equal to the cost of the best solution we have so far, as it cannot lead to a solution of less cost.

The monotonic property is based on the fact that a generated partial problem typically represents a more constrained problem than its parent, and hence costs at least as much. The monotonicity of a cost function can be achieved if the function is *lower bound*. A cost function is called lower bound if it never overestimates the actual cost of a problem. A common practice to derive a lower-bound cost function is to relax the constraints to the original problem [36, 47]. Given a lower-bound function, a monotonic function can be constructed by taking the cost of a partial problem n as the maximum cost of all problems on the path from the original problem to n , guaranteeing that $f(n') \geq f(n)$, where n is the parent of n' .

Before we discuss search strategies, for the purpose of clarity we take the Traveling Salesman Problem as an example to elaborate the branching and bounding rules described above.

2.2 An example

Given n cities, $V = \{1, 2, 3, \dots, n\}$, and a matrix $C = (c_{i,j})$ of intercity costs that defines a cost between each pair of cities, the Traveling Salesman Problem (TSP) is to find a minimum-cost tour that visits each city exactly once and returns to the starting city. Let $\Pi_m = (r_1, r_2, \dots, r_m)$ be a cyclic permutation of the cities $\{r_1, r_2, \dots, r_m\}$, which defines a subtour, $r_1 \rightarrow r_2 \rightarrow \dots \rightarrow r_m \rightarrow r_1$, on V with

size m , where $1 < m \leq n$ and $r_i \in V$. A tour with size n is called a complete tour. The set of problem instances D is then the set of possible subtours defined on V , and the set of feasible solutions S is the set of complete tours. Furthermore, the length of Π_m is defined as $L_m = c_{r_1, r_2} + c_{r_2, r_3} + \cdots + c_{r_m, r_1}$. The cost function f is then the sum of all subtours on V .

When the cost matrix is asymmetric, i.e., the cost of traveling from city i to city j is not necessarily equal to the cost of traveling from city j to city i , the problem is called the Asymmetric TSP (ATSP). The most effective lower-bound cost function for the ATSP is the solution to the *assignment problem* [3, 46]. The assignment problem is to assign to each city i another city j , with $c_{i,j}$ as the cost of this assignment, such that the total cost of all assignments is minimized. The assignment problem is a relaxation of the ATSP, since the assignments need not form a complete tour, allowing collections of disjoint subtours. This relaxation provides a lower bound on the cost of the ATSP tour, which is an assignment of each city to its successor in the tour. If the assignment problem solution happens to be a complete tour, it is the solution to the ATSP as well. The assignment problem is solvable in $O(n^3)$ time [46].

We use an example, illustrated in Figure 1, to introduce the branching rules [3]. We first solve the assignment problem for the given six cities. Assume that the assignment problem solution contains the two subtours shown in the root node of the tree. Since the assignment problem solution contains subtours, we try to eliminate them, one at a time. If subtour $2 - 3 - 2$ is chosen to be eliminated, we have two choices. We may exclude either edge $(2, 3)$ or edge $(3, 2)$, each of which leads to a partial problem with an additional constraint, the excluded edge. We then solve the assignment problems of the partial problems, and further decompose a partial problem if its assignment problem solution is still not a single complete tour.

In order to keep the total number of partial problems generated as small as possible, we should avoid generating duplicate partial problems. This can be realized by including in the current partial problem any edges that were excluded in previous partial problems. In our example, suppose that we generate the first partial problem A by excluding edge $(2, 3)$. The second partial problem B excludes edge $(3, 2)$, but includes edge $(2, 3)$. Therefore, no partial problems generated under A can have edge $(2, 3)$, but all partial problems under B will have edge $(2, 3)$, guaranteeing that the partial problems will be mutually disjoint.

In general, let E denote the set of excluded edges, and I the set of included edges of a partial problem whose assignment problem solution is not a single complete tour. We choose one subtour, the one with minimum number of edges to eliminate. Assume that there are t edges in the subtour, $\{x_1, x_2, \dots, x_t\}$, that are not in I . We then decompose the problem into t children, with the k -th one having excluded

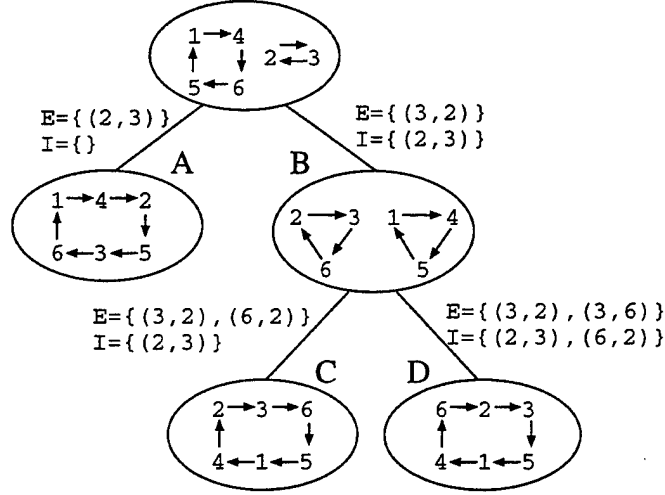


Figure 1: An example of solving the ATSP.

arc set E_k and included arc set I_k , such that

$$\left. \begin{aligned} E_k &= E \cup \{x_k\}, \\ I_k &= I \cup \{x_1, \dots, x_{k-1}\}, \end{aligned} \right\} k = 1, 2, \dots, t. \quad (1)$$

Since x_k is an excluded edge of the k -th partial problem, $x_k \in E_k$, and it is an included edge of the $k+1$ -st partial problem, $x_k \in I_{k+1}$, any partial problems generated from the k -th partial problem cannot contain edge x_k , but all partial problems obtained from the $k+1$ -st partial problem must include edge x_k . Therefore, no duplicate partial problems will be generated, and the state space is a tree of unique nodes.

Briefly, a given ATSP can be solved by taking the original problem as the root partial problem and repeating the following: First, solve the assignment problem for the current partial problem. If the assignment problem solution is not a single complete tour, then select a subtour, and generate all child partial problems by eliminating edges in the subtour. Next, select as the current partial problem a new partial problem that has been generated but not yet expanded. Continue this process until there are no unexpanded partial problems, or all unexpanded partial problems have costs greater than or equal to the cost of the best complete tour found so far.

2.3 State space and the state-space tree

Following the discussion in the previous two sections, the process of solving a problem can be formulated as search in a *state space*. A state space consists of a set of *states* and a collection of *operators*. The states include the problem to be solved

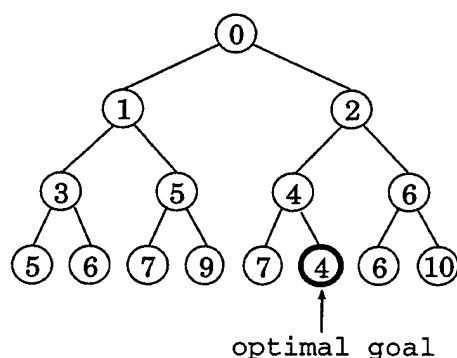


Figure 2: A binary tree with depth three.

and all partial problems that can be generated. The operators, corresponding to the branching rules, are actions that map one state to another. An optimal goal node corresponds to an optimal solution. In general, the structure of a state space is a graph in which nodes represent states, and edges represent operators or state transitions. *Search* in a state space is a systematic exploration of the space in order to find one or more optimal goal nodes, or a path from the initial state to an optimal goal state.

When the principle of inclusion-and-exclusion is followed in decomposing a problem, no duplicate partial problem will be generated. Therefore, the state space is a tree without duplicate nodes, and the leaf nodes in the tree are goal nodes with various costs. In a state-space tree, the number of children of a node is referred to as the *branching factor* of the node.

2.4 Search strategies

We are now in a position to describe search strategies that define sequences of partial problems in which they are decomposed. These different strategies lead to different search algorithms, including best-first search, depth-first search, iterative deepening, recursive best-first search, and constant-space best-first search.

For the sake of simplicity, we adopt the state-space tree in our following description. In Section 2.5, we discuss how state-space graphs are generally handled. To make our explanation concrete, we use a tree, as shown in Figure 2, to illustrate these strategies. It is a uniform binary tree with depth three. The numbers inside the nodes are the node costs.

2.4.1 Best-first search

The first strategy is best-first search (BFS). BFS maintains a partially expanded state-space tree, and at each cycle expands a node of minimum cost, among all

Table 1: Best-first search algorithm.

```

BFS(root)
  open  $\leftarrow$  root
  WHILE (open is not empty)
    n  $\leftarrow$  a node of minimum f cost in open
    IF (n is a goal node) RETURN with success
    DELETE n from open
    EXPAND n, generating all its children
    INSERT all children into open
  RETURN with failure

```

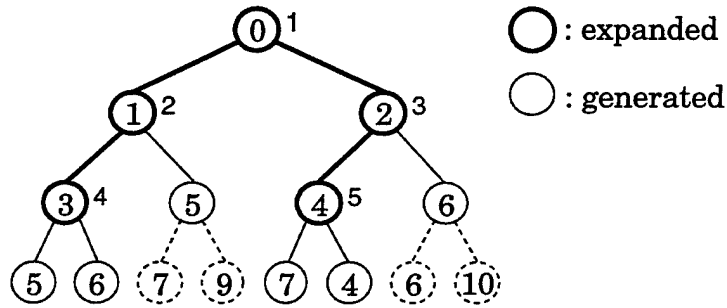


Figure 3: The order of node expansions in best-first search.

frontier nodes that have been generated but not yet expanded, until an optimal goal node is chosen for expansion.

The algorithm is given in Table 1, where the list *open* is used to maintain current frontier nodes. The algorithm starts with $\text{BFS}(\text{root})$. Figure 3 illustrates how BFS works on the tree of Figure 2, with the numbers beside the nodes being the order in which the nodes are expanded.

Thanks to the monotonicity of cost functions and the best-first node selection policy, BFS expands only those nodes with costs less than the optimal solution cost t , plus some nodes with cost equal to t . It can be easily shown that the number of nodes with costs less than t is the minimum number of nodes that any search algorithm needs to expand in order to find an optimal goal [62]. Therefore, BFS is optimal, in terms of the number of nodes expanded, among all algorithms that use the same cost function and guarantee to find an optimal goal node, up to tie breaking among nodes with cost equal to t . It has also been shown that BFS is optimal under many different conditions [11].

BFS is also very general, including many algorithms as special cases. If the

Table 2: Depth-first search algorithm.

```

DFS( $n$ )
  GENERATE all  $k$  children of  $n$ :  $n_1, n_2, \dots, n_k$ 
  EVALUATE them, and SORT them in increasing order of cost
  FOR ( $i$  from 1 to  $k$ )
    IF ( $cost(n_i) < u$ )
      IF ( $n_i$  is a goal node)  $u \leftarrow cost(n_i)$ 
      ELSE DFS( $n_i$ )
    ELSE RETURN
  RETURN

```

depth of the node in a search tree is taken as the cost function, then BFS becomes breadth-first search [47], another well-known but less useful algorithm. On a graph, if the costs between nodes are used as the cost function, then BFS leads to the well-known Dijkstra algorithm [12].

However, in order to expand nodes in the best-first order, BFS must maintain frontier nodes in a priority queue [1]. To this end, BFS typically requires space that is *exponential* in the search depth, making it impractical for most applications. Furthermore, the time to choose a node from all frontier nodes is an increasing function of the number of these nodes. If a heap, an optimal implementation of a priority queue [1], is used to organize the frontier nodes, then the time to select a node for expansion is a logarithmic function of the number of nodes in the heap.

2.4.2 Depth-first search

To reduce the space requirement, we can use depth-first search (DFS). Starting at the root node, and with a global upper bound u on the cost of an optimal goal, DFS always selects the most recently generated node or the deepest node to expand next. Whenever a new leaf node is reached whose cost is less than u , u is revised to the cost of this new leaf. Whenever a node selected for expansion has a cost greater than or equal to u , it is pruned, because node costs are non-decreasing along a path from the root, and all descendants of a node must have costs at least as great as that of their ancestors. In order to find an optimal goal node quickly, the newly generated child nodes should be searched in increasing order of their costs. This is called *node ordering*.

Table 2 is a recursive version of DFS using node ordering. The top-level call is made on the root node, $DFS(root)$, with initial upper bound $u = \infty$. Figure 4 shows how DFS works on the tree of Figure 2. The numbers beside the nodes are

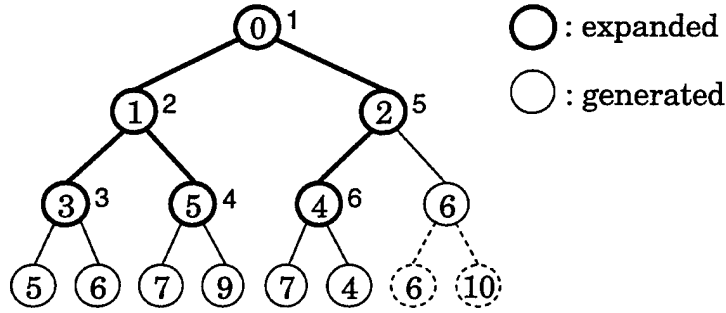


Figure 4: The order of node expansions in depth-first search.

the order in which they are expanded.

DFS runs in space that is only linear in the search depth. It only needs to maintain all the nodes on the path from the root to the node that is currently under consideration, plus their siblings. The search depth is usually linear in the size of the problem to be solved.

Backtracking [4, 47] is a special version of DFS. It generates only one child node of a node at a time. The newly generated node, unless it is a leaf node, is explored immediately. If this node is pruned, the algorithm backtracks to its closest ancestor that still has ungenerated children. Backtracking has mostly been studied on and applied to constraint satisfaction problems [31].

The penalty for DFS to run in linear space is that it expands some nodes that would not be explored by BFS, because their costs are greater than the optimal goal cost. For example, the node with sequence number four (4) in Figure 4 is not expanded by BFS in Figure 3, because its cost (5) is greater than the optimal goal cost (4). In addition, DFS often fails on a graph with cycles, since it may keep expanding the nodes on a cycle. In this case, DFS requires either a finite tree or a cutoff depth in order to guarantee termination.

2.4.3 Iterative deepening

In order to use space that is linear in the search depth, and avoid expanding nodes whose costs are greater than the optimal goal cost, we turn to iterative deepening (ID) [29]. It is a cost-bounded, iterative version of depth-first search.

Using a global variable called the cutoff *threshold*, initially set to the cost of the root, iterative deepening performs a series of depth-first search iterations. In each iteration, it expands all nodes with costs less than or equal to the threshold. If a goal node is chosen for expansion, then the algorithm terminates successfully. Otherwise, the threshold is increased to the minimum cost of all nodes that were generated but not expanded on the last iteration, and a new iteration is begun.

The algorithm, shown in Table 3, starts with $ID(root)$. It repeatedly calls a

Table 3: Iterative deepening algorithm.

<pre> ID(<i>root</i>) <i>threshold</i> $\leftarrow$ <i>cost</i>(<i>root</i>) <i>next_threshold</i> $\leftarrow$ ∞ REPEAT DFS(<i>root</i>) <i>threshold</i> $\leftarrow$ <i>next_threshold</i> <i>next_threshold</i> $\leftarrow$ ∞ </pre>
<pre> DFS(<i>n</i>) FOR (each child <i>n_i</i> of <i>n</i>) IF (<i>n_i</i> is a goal node and <i>cost</i>(<i>n_i</i>) $\leq$ <i>threshold</i>) EXIT with optimal goal node <i>n_i</i> IF (<i>cost</i>(<i>n_i</i>) $\leq$ <i>threshold</i>) DFS(<i>n_i</i>) ELSE IF (<i>cost</i>(<i>n_i</i>) < <i>next_threshold</i>) <i>next_threshold</i> $\leftarrow$ <i>cost</i>(<i>n_i</i>) RETURN </pre>

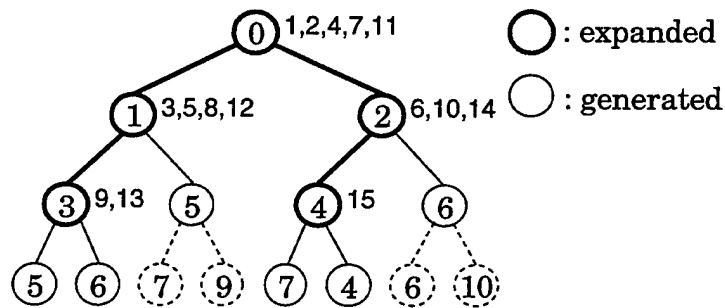


Figure 5: The order of node expansions in iterative deepening.

Table 4: Recursive best-first search algorithm.

```

RBFS( $n, F[n], threshold$ )
  IF ( $cost(n) > threshold$ ) RETURN  $cost(n)$ 
  IF ( $n$  is a goal) EXIT with optimal goal node  $n$ 
  IF ( $n$  has no children) RETURN  $\infty$ 
  FOR (each child  $n_i$  of  $n$ )
    IF ( $cost(n) < F[n]$ )  $F[i] \leftarrow \text{MAX}(F[n], cost(n_i))$ 
    ELSE  $F[i] \leftarrow cost(n_i)$ 
  SORT  $n_i$  and  $F[i]$  in increasing order of  $F[i]$ 
  IF (only one child)  $F[2] \leftarrow \infty$ 
  WHILE ( $F[1] \leq threshold$  and  $F[1] < \infty$ )
     $F[1] \leftarrow \text{RBFS}(n_1, F[1], \text{MIN}(threshold, F[2]))$ 
    INSERT  $n_1$  and  $F[1]$  in sorted order
  RETURN  $F[1]$ 

```

depth-first search procedure for each iteration, with increasing cost thresholds. The global variables *threshold* and *next_threshold* are the node cutoff thresholds for the current and next iterations, respectively. The depth-first search, $\text{DFS}(n)$, does not have to use node ordering and is implemented recursively. Figure 5 shows how ID works on the tree of Figure 2, with the numbers beside the nodes being the order in which they are expanded. In this example, successive iterations have cost thresholds of 0, 1, 2, 3, and 4.

Although iterative deepening will not expand a node whose cost is greater than the cost of the optimal goal node, it may expand some nodes more than once, as shown in Figure 5.

2.4.4 Recursive best-first search

Another algorithm that runs in space that is linear in search depth is recursive best-first search (RBFS) [30]. It always expands unexplored nodes in best-first order, regardless of the cost function, and is more efficient than iterative deepening with a monotonic cost function.

The key difference between iterative deepening and RBFS is that while iterative deepening maintains a single global cutoff threshold for the whole tree, RBFS computes a separate local cutoff threshold for each subtree of the current search path. The local cutoff threshold of a subtree is the minimum cost of all frontier nodes that are not in the subtree. This means that the exploration of the subtree continues as long as the subtree has the best frontier node among all frontier nodes.

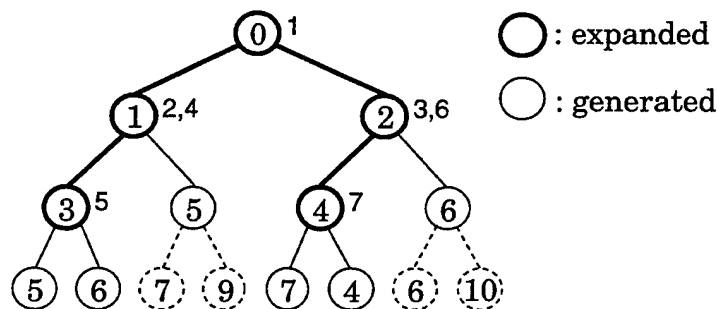


Figure 6: The order of node expansions in recursive best-first search.

The algorithm is given in Table 4, where $F(n)$ is the stored value of node n , and *threshold* is a local upper bound. The initial call is $\text{RBFS}(\text{root}, \text{cost}(\text{root}), \infty)$.

For simplicity, we describe the behavior of the algorithm on the tree in Figure 2. After the root is expanded (see Figure 6), RBFS is called recursively on the left child with an upper bound of 2, the value of the right child. The reason is that the best frontier node will be in the left subtree as long as its value is less than or equal to 2, the best frontier node in the right subtree. When the left child is expanded, both its children's values, 3 and 5, exceed the upper bound of 2, causing RBFS to return to the root and release the memory for the children of the left child of the root node. First, however, it backs up the minimum child value of 3 and stores that as the new value of the left child of the root. After returning to the root, RBFS calls itself recursively on the right child with an upper bound of 3, the value of the best frontier node in the left subtree. When the right child is expanded, both its children's values, 4 and 6, exceed the upper bound of 3, so RBFS backs up the minimum of these values, 4, stores it as the new value of the right child, and returns to the root. It then calls itself recursively on the left child with an upper bound of 4, the value of the best frontier node in the right child. After the left child and its left child are expanded, all frontier nodes in the left child are greater than or equal to 5, so 5 is stored as the new value of the left child of the root, and RBFS is called on the right child with an upper bound of 5. It then proceeds down the right subtree until it chooses to expand the goal node of cost 4, and terminates.

At any moment, RBFS maintains in memory the current search path, along with all immediate siblings of nodes on the current path, together with the costs of the best frontier nodes below each of those siblings. Thus, its space complexity is linear in the search depth. An important feature of RBFS is that with monotonic node costs, it generates fewer nodes than iterative deepening, up to tie-breaking. For example, RBFS expands only seven nodes on the tree in Figure 2 (see Figure 6), but iterative deepening expands fifteen nodes on the same tree (see Figure 5). Like

Table 5: Constant-space best-first search algorithm.

CBFS(<i>root</i>) $F(\text{root}) \leftarrow f(\text{root}); \text{open} \leftarrow \text{root}; \text{used} \leftarrow 1$ WHILE (<i>open</i> is not empty) $\text{best} \leftarrow$ a deepest node of minimum F-cost in <i>open</i> IF (<i>best</i> is a goal node) RETURN with success $\text{child} \leftarrow$ next element of <i>ungenerated_children</i>(<i>best</i>) REMOVE <i>child</i> from <i>ungenerated_children</i>(<i>best</i>) $F(\text{child}) \leftarrow \max\{f(\text{child}), F(\text{best})\}$ IF (all children of <i>best</i> have been generated) BACKUP(<i>best</i>) If (all children of <i>best</i> are in memory) DELETE <i>best</i> from <i>open</i> $\text{used} \leftarrow \text{used} + 1$ IF ($\text{used} > \text{memory_size}$) $\text{worst} \leftarrow$ a shallowest, highest F-cost leaf in <i>open</i> DELETE <i>worst</i> ADD <i>worst</i> to <i>ungenerated_children</i>(<i>parent</i>(<i>worst</i>)) IF (<i>parent</i>(<i>worst</i>) $\notin$ <i>open</i>) INSERT <i>parent</i>(<i>worst</i>) into <i>open</i> $\text{used} \leftarrow \text{used} - 1$ INSERT <i>child</i> into <i>open</i> RETURN with failure
BACKUP(<i>n</i>) IF (<i>completed</i>(<i>n</i>) and <i>n</i> has a parent) $F(n) \leftarrow$ least F-cost of all <i>n</i>'s children IF ($F(n)$ changed) BACKUP(<i>n</i>)

iterative deepening, however, RBFS also suffers from node re-expansion overhead.

2.4.5 Constant-space best-first search

Depth-first search, iterative deepening, and recursive best-first search are linear-space search algorithms [62], since they run in space that is linear in search depth. Best-first search, however, is at the opposite extreme of the space-requirements spectrum: it needs space that is exponential in search depth. The main drawback of a linear-space search is that it usually generates more nodes than best-first search, visiting nodes that are not explored by BFS or expanding nodes more than once. In addition, linear-space search does not fully utilize the total available space during execution. It saves only the nodes on the path from the root to the node that is currently under consideration and their siblings. This is why all linear-space

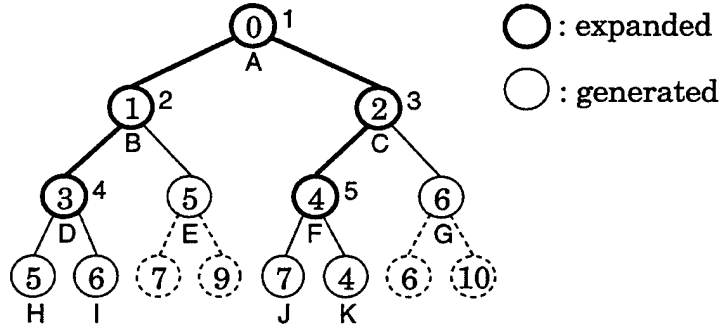


Figure 7: The order of node expansions in constant-space best-first search with 7 memory units.

algorithms can be implemented on a stack.

In practice, although available space is a precious resource, it is usually more than linear in search depth. What we need is an algorithm that can use all of the available space in order to reduce the number of node generations in linear-space search, and still explore new nodes in best-first order. Such an algorithm is MA* [6], which is refined into SMA* [52, 24]. For naming consistency and to reflect its nature, we call this algorithm constant-space best-first search (CBFS).

Given a limited, constant amount of memory, CBFS first runs best-first search until the memory is full. It then prunes the least promising part of the generated search space and reallocates memory to push forward toward the goal. More specifically, CBFS generates one node at a time. If there is no memory available for the new node, it removes the worst leaf node in the current saved search tree. Similar to RBFS, CBFS maintains two costs on a node: static and dynamic node costs. The static cost of a node is the same as the regular node cost. The initial dynamic cost of a node is set to its static cost. After all of its children have been generated or explored, the dynamic cost of the node is revised to the minimum static cost of all frontier nodes under it, although not all of these frontier nodes must be stored in memory. This node pruning plus dynamic cost propagation is called *node retraction*. The best node is defined as the node with the minimum dynamic cost among all nodes in memory that have not been fully expanded. The worst node is defined as the leaf node with the maximum dynamic cost among all leaf nodes in memory.

The algorithm is given in Table 5, where $F(n)$ is the dynamic cost of node n . The initial call is $\text{CBFS}(\text{root})$. Figure 7 shows how CBFS works on the tree of Figure 2 with total memory size of 7, which is the memory required to run RBFS. In Figure 7, the numbers beside the nodes are the order in which they are fully expanded; in this particular example, the second child of an expanded node is generated immediately after its first child. After three full node expansions,

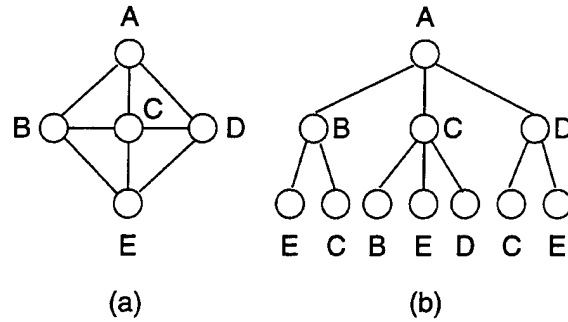


Figure 8: A simple graph and part of its depth-first search tree.

nodes A to G , a total of 7 nodes, are generated and saved in memory. The next node to be expanded is node D . In order to accommodate its first child node H , node G (the current worst leaf node in memory) is removed. To provide room for the second child I of D , node E is removed next, as it is shallower than node H . The next node selected for expansion is node F . Similarly, to accommodate its children (nodes J and K), nodes H and I must be removed. In this example, CBFS expands and generates the same numbers of nodes as best-first search, but with less memory. Compared to RBFS, CBFS uses the same amount of memory and avoids the expansion of some nodes by carefully managing available memory.

2.5 Graph vs. tree

In our previous descriptions of search strategies, we assumed that the structure of the state space is a tree. Sometimes, however, the state space is better represented by a graph, for example in a problem such as a sliding-tile puzzle. Nevertheless, a graph can be represented by a tree with duplicate nodes. For example, the graph in Figure 8(a) can be represented by a tree with many duplicate nodes in Figure 8(b), where node A represents the original problem.

In a state-space graph, search algorithms become complicated, requiring special care to handle the problem of expanding the same node more than once, and search complexity is increased. When there is no memory restriction, re-expansion of a node can be avoided, as is done in AO* [45], the best-first graph search algorithm. The method is to save in memory all of the generated nodes (call them GN) and to check a newly generated node n against GN . Two cases may occur. In the simple case, n does not appear in GN , meaning that n is being visited for the first time. In this case, n is added to GN . In the difficult case, there is a node $n' \in GN$ which is the same as n , meaning that the same node in the state space has been reached via two different paths. If the cost of n , $f(n)$, is greater than or equal to the cost of n' , $f(n')$, n can be discarded since it was discovered through

a longer path. Otherwise, n' must be replaced by n , and the costs of the nodes in GN which are descendents of n' must be updated accordingly. This relationship between n and n' is described by the dominance relation [22]. If we can show that the best descendent of a node n is at least as good as the best descendent of a node n' , then we say n dominates n' .

In general, there is no guarantee that a node in a state-space graph will not be re-expanded if available memory is restricted. On the other hand, if a state space has structures such as grids, the properties of these structures can be learned in a preprocessing phase and encoded in a finite-state machine [57]. This finite-state machine can be consulted during a search to reduce the number of node re-expansions.

2.6 Historical remarks and discussions

The idea of BnB can be traced back to the work of Dantzig, Fulkerson and Johnson [9, 10] on integer programming and the Traveling Salesman Problem [36]. Eastman first systemically investigated this idea [13]. The term “branch-and-bound” was coined by Little, Murty and Karel [38] in their work on the TSP, which led to Little’s algorithm.

In the late 1970s and the early 1980s, formal descriptions of the BnB method were given [21, 23, 28], and the relationships between BnB and the sequential decision process, dynamic programming, AND/OR graph search, and game-tree search were revealed [23, 33, 34]. The main result of this work is that BnB can be considered as a sequential decision process that is closely related to dynamic programming. Furthermore, many different search algorithms, including alpha-beta pruning [27] and SSS* [56] for game-tree search, can be formulated as BnB.

Starting in the mid-1980s, research on heuristic search in the artificial intelligence community focused on the development of search algorithms using restricted memory, aimed at solving large real-world problems that cannot be solved by best-first search. Korf [29] developed the iterative deepening algorithm for single-agent search problems, based on the same prevalent idea used in two-person game playing [54]. Korf also designed recursive best-first search [30]. A similar idea was independently developed by Russell [52]. Chakrabarti et al. [6] first proposed best-first search using restricted, but more than linear, memory. This algorithm was subsequently improved by Russell [52]. The main idea is node retraction for releasing some occupied memory. This idea was also used earlier in Bratko’s implementation of A^* [5]. Several other algorithms also require space more than linear in search depth. Pearl presented three possible BFS-DFS combinations to meet a limited memory requirement [47]. Sen and Bagchi [53] suggested a combination of BFS and iterative deepening.

Many applications in operations research require finding an optimal solution to

a combinatorial optimization problem, such as integer programming, which can be described by a set of equations. Thus, a cost function can be derived by directly relaxing or removing some constraints on the equations. Such a cost function usually does not overestimate the actual cost of the problem, and is thus referred to as a *lower-bound* cost function. Furthermore, BnB is usually formulated as a search in a state space that is an OR graph, in which a solution to a child node is also a solution to its parent node.

Many applications in artificial intelligence, on the other hand, require finding a sequence of operators that map the initial problem, step by step, into an optimal solution. Therefore, a natural way to construct a cost function is to combine the cost of the path from the original problem to the current state and the estimated cost of the path from the current state to a goal state. A simple cost function is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to the current node n , and $h(n)$ is an estimated cost from node n to a goal node. Best-first search using this additive cost function is the well known A^* algorithm [18]. Function h is also referred to as *heuristic function*, and function f is therefore called *heuristic evaluation*. Function h is also called *admissible* if it does not overestimate the actual cost from node n to a goal node. Furthermore, a cost function f is *monotonic* if the cost of a child node is greater than or equal to that of its parent. Clearly, an admissible h implies a lower-bound cost function f . A lower-bound function leads to a monotonic function, by taking the cost of a node as the maximum of all the costs of the nodes on the path from the root to the node [43].

The state space of a problem considered in artificial intelligence may be an AND/OR graph, in which case the solution to an AND node is a combination of the solutions to all of its child nodes. One example of AND/OR graph search is game-tree search. It has been shown that many AND/OR graph search algorithms, including game-tree search algorithms such as alpha-beta pruning [27] and SSS* [56], can be formulated under a BnB framework [34].

3 Computational Complexity

In the worst case, a BnB search algorithm must explore every node in a state space. Thus the worst-case complexity of BnB is the size of a state space. On the other hand, if the lower-bound cost function used is exact, the complexity is linear in the length of the path from the initial node to a goal node. However, these extreme cases rarely occur and thus convey little information about the actual performance of an algorithm. Average-case complexity is therefore a more realistic performance measure. The main purpose of an average-case analysis is to find the relationship between the average-case complexity and the accuracy of lower-bound

cost functions used.

Like many other average-case analyses, a study of the average-case complexity of BnB requires many restrictions and assumptions. In order to make them feasible, almost all average-case complexity analyses of BnB are based on a state-space tree, as opposed to a state-space graph, which is analytically intractable. The state-space tree used is an OR tree, which is much more manageable than an AND/OR tree.

One well-known model used in artificial intelligence is a uniform tree with a constant branching factor b , a single goal node at depth d , and the heuristic estimates h on different nodes being independent and identically distributed random variables [15, 47, 49]. This model is used to understand the relationship between the accuracy of the heuristic estimation h and the complexity of the A^* algorithm. The main conclusion from this analysis is that in most cases, the average-case complexity of the A^* algorithm is exponential in the tree depth. We leave the interested reader to a complete and excellent text [47] on this analysis.

Another model, called *incremental random tree*, has been used to analyze the average-case complexity of best-first search, depth-first search, and iterative deepening. This model is relatively more realistic than the first one. The rest of this section discusses analyses based on this model and summarizes the results.

3.1 Incremental random trees

As stated in Section 2, the nodes in a state-space tree have costs, which are used to decide which node to explore next. The cost of a node, $f(n)$, is defined as either an estimate of the actual cost of the node, $f^*(n)$, which is the cost of solving the problem that includes this node, or an estimate of the cost of the best goal node in the tree underneath the given node.

A state-space tree with node costs can also be treated as if it has edge costs instead. The cost of an edge that connects two nodes is the difference between the cost of the child node and that of the parent. The cost of a node is then the sum of the edge costs on its path to the root, plus the cost of the root. Edge costs are non-negative if node costs are monotonically non-decreasing with depth. An edge cost can also be viewed as the cost of an operator that maps the parent to the child node. This leads to the following model.

Definition 3.1 *An incremental random tree, or random tree, $T(b, d)$, is a tree with depth d , root cost 0, and independent and identically distributed random branching factors with mean b . Edge costs are finite, non-negative, and independently drawn from a common probability distribution. The cost of a node is the sum of the edge costs from the root to that node. An optimal goal node is a leaf node of minimum cost at depth d .*

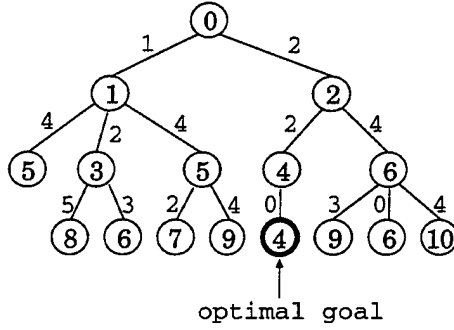


Figure 9: A simple incremental random tree.

Figure 9 shows an example of an incremental random tree, with edges and nodes annotated by edge costs and the resulting node costs, respectively.

Two important features of this model are worth mentioning. First, multiple optimal goal nodes may exist, in contrast to the model in which only one optimal goal is allowed [15, 47, 49]. Second, unlike the conventional assumption that node costs are independent [15, 47, 49], the costs of two nodes are correlated if they share common edges on their paths to the root, with the degree of dependence based on the number of edges they have in common.

In this random tree model, the edge costs are assumed to be independent and identically distributed (the *i.i.d.* assumption). The *i.i.d.* assumption also applies to the branching factors of different nodes. Unfortunately, these assumptions rarely hold in practice. Nevertheless, they are usually introduced to facilitate average-case analyses. In many cases, however, analytic results obtained under *i.i.d.* assumptions are often used to characterize real-world problems where these assumptions do not hold. Examples can be found in [62, 63].

3.2 Problem complexity and cost of an optimal goal

We first consider the problem complexity of finding an optimal goal node, in terms of the total number of node expansions, thus deriving a lower bound on an algorithm complexity.

Theorem 3.1 [62] *On a state-space tree with monotonic node costs, the total number of nodes whose costs are strictly less than the optimal goal cost is a lower bound on the complexity of finding an optimal goal node, and the total number of nodes whose costs are less than or equal to the optimal goal cost is an upper bound on the complexity of finding an optimal goal node.*

This theorem indicates that problem complexity is directly related to optimal goal cost. The optimal goal cost of the random tree has been studied using the tools

of an area of mathematics called branching processes, particularly age-dependent branching processes [17], and first-percolation processes [16, 26].

Let p_0 be the probability that an edge cost is zero. Since b is the mean branching factor, bp_0 is the expected number of zero-cost branches leaving a node, or the expected number of children of a node that have the same cost as their parent. We call these nodes *same-cost children*. It turns out that the expected number of same-cost children of a node determines the expected cost of optimal goal nodes. Intuitively, when $bp_0 > 1$, a node should expect to have one same-cost child, so that the optimal goal cost should not increase with depth. On the other hand, when $bp_0 < 1$, most nodes should not have a same-cost child, causing the optimal goal cost to increase with depth.

Theorem 3.2 [41, 42] *Let C^* be the expected cost of optimal goal nodes of a random tree $T(b, d)$ with $b > 1$. As $d \rightarrow \infty$,*

1. $C^*/d \rightarrow \alpha$ almost surely[†] when $bp_0 < 1$, where α is a constant independent of d ,
2. $C^*/(\log \log d) \rightarrow 1$ almost surely when $bp_0 = 1$,
3. C^* almost surely remains bounded when $bp_0 > 1$.

3.3 Algorithm complexity

We now discuss the average-case complexity of BnB search algorithms, including best-first search, depth-first search, iterative deepening, recursive best-first search, and constant-space best-first search. The complexity measure is the total number of nodes expanded.

3.3.1 Best-first search

The following theorem implies that BFS is optimal, up to tie-breaking, on an incremental random tree, since it has monotonic node costs.

Theorem 3.3 [62] *On a state-space tree with monotonic node costs, best-first search is optimal among all algorithms that use the same node costs and are guaranteed to find an optimal goal node, up to tie-breaking, among nodes whose costs are equal to the optimal goal cost.*

[†]A sequence X_n of random variables is said to converge *almost surely* (with probability one) to X if $P(\lim_{n \rightarrow \infty} X_n = X) = 1$ [51].

It has been shown in [41, 42] that the expected number of nodes expanded by any algorithm that is guaranteed to find an optimal goal node of $T(b, d)$ is exponential in d when $bp_0 < 1$, and the expected number of nodes expanded by BFS on $T(b, d)$ is quadratic in d when $bp_0 = 1$, and linear in d when $bp_0 > 1$. Since BFS is an optimal algorithm for this problem, up to tie-breaking, then we have the following.

Theorem 3.4 [25, 41, 42] *The expected number of nodes expanded by best-first search for finding an optimal goal node of a random tree $T(b, d)$ is*

1. $\theta(\beta^d)^\dagger$ when $bp_0 < 1$, where β is a constant, $1 < \beta < b$,
2. $\theta(d^2)$ when $bp_0 = 1$, and
3. $\theta(d)$ when $bp_0 > 1$,

as $d \rightarrow \infty$, where p_0 is the probability of a zero-cost edge.

BFS expands those nodes whose costs are less than or equal to the cost C^* of an optimal goal node. Intuitively, the average number of nodes whose costs are less than C^* is exponential when $bp_0 < 1$, since C^* grows linearly with depth when $bp_0 < 1$. The extreme case is when no two edges or nodes have the same cost, thus $p_0 = 0$ and $bp_0 = 0$. On the other hand, when $bp_0 > 1$, there are a large number of nodes that have the same cost, and there are many optimal goal nodes as well. The extreme case is when all edges or nodes have cost zero, *i.e.* $p_0 = 1$, and hence every leaf of the tree is an optimal goal node. In this case, to find a leaf node and verify that it is optimal is easy.

3.3.2 Depth-first search

There are two major differences between DFS and BFS. One is that DFS runs in space that is linear in the search depth, whereas BFS usually requires space that is exponential in the search depth. The other difference is that DFS may expand nodes whose costs are greater than the optimal goal cost, whereas BFS does not. The second difference makes the analysis of DFS more difficult, as the mathematical tool of branching processes used for BFS cannot be carried over to DFS directly. One approach to circumvent this difficulty is to determine the relationship between the complexity of DFS and that of BFS, which is stated in the following theorem.

$^\dagger\theta(\chi(x))$ denotes the set of all functions $\omega(x)$ such that there exist positive constants c_1, c_2 , and x_0 such that $c_1\chi(x) \leq \omega(x) \leq c_2\chi(x)$ for all $x \geq x_0$. In words, $\theta(\chi(x))$ represents functions of the same asymptotic order as $\chi(x)$.

Theorem 3.5 [61, 62] *Let $N_B(b, d)$ be the expected number of nodes expanded by best-first search, and $N_D(b, d)$ be the expected number of nodes expanded by depth-first search, on a random tree $T(b, d)$. As $d \rightarrow \infty$,*

$$N_D(b, d) < (b - 1) \sum_{i=1}^{d-1} N_B(b, i) + d$$

This theorem, combined with Theorem 3.4, leads to the following.

Theorem 3.6 [61, 62] *The expected number of nodes expanded by depth-first search for finding an optimal goal node of a random tree $T(b, d)$, as $d \rightarrow \infty$, is*

1. $\theta(\beta^d)$ when $bp_0 < 1$, meaning that depth-first search is asymptotically optimal, where β is the same constant as in Theorem 3.4,
2. $O(d^3)$ when $bp_0 = 1$, and
3. $O(d^2)$ when $bp_0 > 1$,

where p_0 is the probability of a zero-cost edge.

Theorem 3.6 is analogous to Theorem 3.4, showing the expected number of nodes expanded by DFS under different values of the expected number of same-cost children. Note that β in Theorem 3.6 is the same β as in Theorem 3.4. This means that when $bp_0 < 1$ and $d \rightarrow \infty$, DFS expands asymptotically the same number of nodes as BFS. Since BFS is optimal by Theorem 3.3, DFS is asymptotically optimal in this case.

3.3.3 Iterative deepening

Iterative deepening (ID) is a cost-bounded depth-first search. In each iteration, it expands those nodes whose costs are less than or equal to the current cost bound, and expands them in depth-first order.

For iterative deepening, each node cost that is less than the cost of the optimal goal node will generate a different iteration. Thus, the behavior of iterative deepening critically depends on the distribution of edge costs. The following edge-cost distribution plays an important role. A distribution is a *lattice* distribution if it takes values from a finite set $\{a_0\Delta, a_1\Delta, \dots, a_m\Delta\}$ for nonnegative integers $a_0, a_1, \dots, a_m$, and positive constant Δ that is chosen so that the integers $a_0, a_1, \dots, a_m$ are relatively prime [17]. Any distribution on a finite set of integers or a finite set of rational numbers is a lattice distribution, for example. Furthermore, Δ may also be an irrational number, such as $\Delta = \pi$.

Theorem 3.7 [62] *On a random tree $T(b, d)$ with edge costs chosen from a continuous distribution, iterative deepening expands $O((N_B(b, d))^2)$ expected number of nodes, where $N_B(b, d)$ is the expected number of nodes expanded by best-first search. On a random tree $T(b, d)$ with edge costs chosen from a lattice distribution, iterative deepening expands $O(N_B(b, d))$ expected number of nodes as $d \rightarrow \infty$, which is asymptotically optimal.*

This theorem indicates that there is a significant node-regeneration overhead in iterative deepening when edge costs are chosen from a continuous distribution. In this case, node costs are unique with probability one, and each iteration expands only one node that has not been expanded in the previous iteration. Theorem 3.7 also provides an upper bound on the expected complexity of iterative deepening when edge costs are chosen from a hybrid distribution with continuous nonzero values, but an impulse at zero so that the probability of a zero-cost edge is not zero. In this case, Theorem 3.7 implies that iterative deepening expands $O(\beta^{2d})$, $O(d^4)$, and $O(d^2)$ expected number of nodes when $bp_0 < 1$, $bp_0 = 1$, and $bp_0 > 1$, respectively, where β is defined in Theorem 3.4.

On the other hand, when edge costs are chosen from a lattice distribution, node regenerations lead to a constant multiplicative overhead. The reason is that many nodes whose paths to the root have different combinations of edge costs have the same cost. In fact, the total number of iterations is linear in the search depth when edge costs are chosen from a lattice distribution. This means that the number of distinct node costs that are less than the optimal goal cost is also linear in the search depth.

Continuous and lattice distributions are two extreme cases for edge costs. Unfortunately, iterative deepening is not asymptotically optimal when edge costs are chosen from any non-lattice distribution. Consider discrete edge costs that are rationally independent. Two different numbers x and y are rationally independent if there exists no rational number r such that $x \cdot r = y$. For example, 1 and π are rationally independent. When some edge costs are rationally independent, any different edge-cost combinations, $c_1 \cdot 1 + c_2 \cdot \pi$ for instance, will have a total cost that is different from all other combinations. In this case, iterative deepening is no longer asymptotically optimal.

3.3.4 Recursive and constant-space best-first searches

With a monotonic cost function, recursive best-first search (RBFS) generates fewer nodes than iterative deepening, up to tie-breaking [30]. Thus, for a state space with monotonic node costs, the complexity of iterative deepening is an upper bound on the complexity of recursive best-first search. In consequence, RBFS is asymptotically optimal when the edge costs of an incremental random tree are chosen from a lattice distribution.

Table 6: Expected complexity of BFS and DFS.

<i>algorithm</i>	$bp_0 < 1$	$bp_0 = 1$	$bp_0 > 1$
BFS	$\theta(\beta^d)$ optimal	$\theta(d^2)$ optimal	$\theta(d)$ optimal
DFS	$\theta(\beta^d)$ asym. optimal	$O(d^3)$	$O(d^2)$

Table 7: Expected complexity of ID, RBFS, and CBFS.

<i>lattice edge costs</i>		
$bp_0 < 1$	$bp_0 = 1$	$bp_0 > 1$
$\theta(\beta^d)$ asym. optimal	$\theta(d^2)$ asym. optimal	$\theta(d)$ asym. optimal
<i>non-lattice edge costs</i>		
$bp_0 < 1$	$bp_0 = 1$	$bp_0 > 1$
$O(\beta^{2d})$	$O(d^4)$	$O(d^2)$

RBFS is a special case of constant-space best-first search (CBFS), where the constant space available to RBFS is $b_m d_g + 1$, where 1 is included to count the unit storing the root node, b_m is the maximum branching factor, and d_g is the depth of a goal node. With space more than linear in the goal depth, CBFS re-expands fewer nodes than RBFS, and the total number of nodes expanded by CBFS will not be greater than that expanded by RBFS. Thus, the total number of nodes expanded by iterative deepening is also an upper bound of the number of nodes expanded by CBFS.

3.4 Summary and complexity transition

Tables 6 and 7 summarize the average-case complexity of BnB search algorithms, measured by the number of nodes expanded.

The results show that the average-case complexity of a BnB search algorithm on an incremental random tree experiences a dramatic transition, from exponential to polynomial. This phenomenon is similar to a *phase transition*, which is a dramatic change to some problem property as some *order parameter* changes across a critical point. The simplest example of a phase transition is water's change from a solid phase to a liquid phase when the temperature rises from below the freezing point to above that point.

The order parameter that determines the complexity transition of tree search is the expected number of *same-cost children*. If the probability that an edge takes

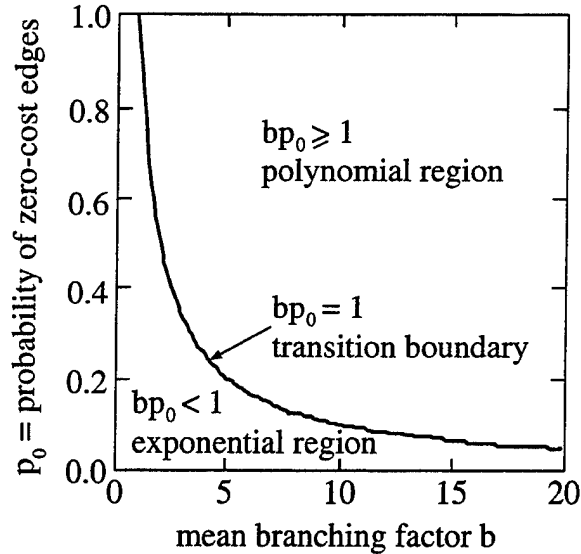


Figure 10: Difficult and easy regions for tree search.

cost zero is p_0 and the mean branching factor is b , then bp_0 is the expected number of same-cost children of a node. When $bp_0 < 1$, a BnB search algorithm expands an exponential number of nodes on average. On the other hand, when $bp_0 \geq 1$, a BnB search algorithm runs in polynomial average time. Therefore, its average-case complexity changes from exponential to polynomial as the expected number of same-cost children increases from less than one to greater than or equal to one. Figure 10 illustrates these two complexity regions and the transition boundary between them.

3.5 Historical remarks and discussions

Researchers who analyze BnB search algorithms can be classified into two groups based on the analytical models that they use. The first group, consisting of Gaschnig [15], Pohl [49], Pearl, and his students [20, 47], focused on the A^* algorithm using a single-goal tree model. This line of research is very well described and summarized in [47].

The second group, consisting of researchers from the areas of artificial intelligence and combinatorial optimization, has analyzed many BnB search algorithms. The basic model used is the incremental random tree described in Definition 3.1. The earliest study was done by Karp and Pearl [25] on a binary random tree with edge costs of zero and one for the complexity of best-first search. In this seminal paper, they first discovered the complexity transition of best-first search. Their results were also suggested earlier by Hammersley [16]. Smith [55] used an incre-

mental random tree with leaf nodes at different depths as goal nodes. He derived equations for the average-case complexity of best-first search and depth-first search. However, no closed-form solutions to these equations were obtained, and thus no direct conclusion was drawn regarding average-case complexity. Wah and Yu [58], used an incremental random tree with a uniform branching factor. They modeled the process of BFS as the behavior of two walls moving toward each other, with one wall as the minimum cost of the currently generated nodes, and the other as the current upper bound. They also compared depth-first search with best-first search, arguing that the former is comparable to the latter when the cost function used is very accurate or very inaccurate. McDiarmid and Provan [41, 42] significantly extended Karp and Pearl's work on best-first search to cover incremental random trees with arbitrary edge-cost distributions, and random branching factors. These results were further extended by Korf and this author to depth-first search and iterative deepening [60, 61, 62]. Their results also imply the average-case complexity of recursive best-first search and variable-space best-first search, as discussed in this section.

In addition, there is a large body of literature on the average-case complexity of backtracking for solving constraint-satisfaction problems. See [50] for a survey of this line of research.

Huberman and Hogg [19] studied complexity transition phenomena in intelligent systems. Cheeseman et al. [7] empirically showed that complexity transitions exist in many NP-hard problems, including Hamiltonian circuits, constraint-satisfaction problems, graph coloring, and the symmetric Traveling Salesman Problem. Complexity transitions in constraint-satisfaction problems have attracted the attention of many researchers [8, 35, 44, 59]. Most recently, Pemberton and this author [48, 64] proposed a method of exploiting complexity transitions to find approximate and optimal solutions to combinatorial optimization problems. On the asymmetric Traveling Salesman Problem, DFS using this method runs faster and finds better solutions than a local search method.

4 Conclusions

Branch-and-bound (BnB) is a general and principal problem-solving method for finding exact solutions to computation-intensive problems. We have given a comprehensive presentation of BnB. We have also discussed many different search algorithms based on BnB, including best-first search, depth-first search, iterative deepening, recursive best-first search, and constant-space best-first search. The key to applying BnB to a given problem is to find a monotonic, underestimated cost function. The efficiency of a search based on such a function depends on the function's accuracy. The more accurate the estimated cost, the easier it is to solve

the problem. Recent analysis reveals: most problems require computation that is exponential in search depth on average; search algorithms using restricted memory are asymptotically optimal on most problems on average.

Acknowledgments

The author is grateful to Richard Korf for discussions and to Sheila Coyazo for editorial assistance.

References

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. John Wiley & Sons, 1974.
- [2] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA J. on Computing*, 3:149–156, 1991.
- [3] E. Balas and P. Toth. Branch and bound methods. In E.L. Lawler *et. al.*, editor, *The Traveling Salesman Problem*, pages 361–401. John Wiley & Sons, Essex, 1985.
- [4] J. Bitner and E. Reingold. Backtracking programming techniques. *Communications of ACM*, 18:651–655, 1975.
- [5] I. Bratko. *The Art of Prolog Programming*. Academic Press, 1986.
- [6] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41:197–221, 1989.
- [7] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, (IJCAI-91)*, pages 331–337, Sydney, Australia, August 1991.
- [8] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 21–27, Washington, D.C., July 1993.
- [9] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

- [10] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. On a linear programming, combinatorial approach to the traveling salesman problem. *Operations Research*, 7:58–66, 1959.
- [11] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A^* . *Journal of ACM*, 32:505–536, 1985.
- [12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1971.
- [13] W. L. Eastman. *Linear programming with pattern constraints*. PhD thesis, Harvard University, 1958.
- [14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, NY, 1979.
- [15] J. G. Gaschnig. *Performance measurement and analysis of certain search algorithms*. PhD thesis, Computer Science Dept. Carnegie-Mellon University, Tech. Rep. CMU-CS-79-124, Pittsburgh, PA, 1979.
- [16] J. M. Hammersley. Postulates for subadditive processes. *Annals of Probability*, 2:652–680, 1974.
- [17] T. Harris. *The Theory of Branching Processes*. Springer, Berlin, 1963.
- [18] T. P. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science and Cybernetics*, 4:100–107, 1968.
- [19] B. A. Huberman and T. Hogg. Phase transitions in artificial intelligence systems. *Artificial Intelligence*, 33:155–171, 1987.
- [20] N. Huyn, R. Dechter, and J. Pearl. Probabilistic analysis of the complexity of A^* . *Artificial Intelligence*, 15:241–254, 1980.
- [21] T. Ibaraki. Theoretical comparisons of search strategies in Branch-and-Bound algorithms. *Computer and Information Sciences*, 5:315–344, 1976.
- [22] T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *Journal of ACM*, 24:264–279, 1977.
- [23] T. Ibaraki. Branch-and-Bound procedure and state-space representation of combinatorial optimization problems. *Information and Control*, 36:1–27, 1978.

- [24] H. Kaindl and A. Khorsand. Memory-bounded bidirectional search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 1359–1364, Seattle, WA, July 1994.
- [25] R. M. Karp and J. Pearl. Searching for an optimal path in a tree with random costs. *Artificial Intelligence*, 21:99–117, 1983.
- [26] J. F. C. Kingman. The first birth problem for an age-dependent branching process. *Annals of Probability*, 3:790–801, 1975.
- [27] D. E. Knuth and R. E. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.
- [28] W. H. Kohler and K. Steiglitz. Characterization and theoretical comparison of Branch-and-Bound algorithms for permutation problems. *Journal of ACM*, 21:140–156, 1974.
- [29] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
- [30] R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.
- [31] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13:32–44, 1992.
- [32] V. Kumar. Search branch-and-bound. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1468–1472. Wiley-Interscience, New York, 2 edition, 1992.
- [33] V. Kumar and L. N. Kanal. A general Branch and Bound formulation for understanding and synthesizing And/Or tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.
- [34] V. Kumar, D. S. Nau, and L. Kanal. A general branch-and-bound formulation for and/or graph and game tree search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 91–130. Springer-Verlag, New York, 1988.
- [35] T. Larrabee and Y. Tsuji. Evidence for a satisfiability threshold for random 3cnf formulas. In *Working Notes of AAAI 1993 Spring Symposium: AI and NP-Hard Problems*, pages 112–118, Stanford, CA, March 1993.
- [36] E. L. Lawler, J. K. Lenstra, A. H. G Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem*. John Wiley & Sons, Essex, 1985.

- [37] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14:699–719, 1966.
- [38] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.
- [39] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.
- [40] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementation*. John Wiley & Sons, West Sussex, England, 1990.
- [41] C. J. H. McDiarmid. Probabilistic analysis of tree search. In G. R. Gummert and D. J. A. Welsh, editors, *Disorder in Physical Systems*, pages 249–260. Oxford Science, 1990.
- [42] C. J. H. McDiarmid and G. M. A. Provan. An expected-cost analysis of backtracking and non-backtracking algorithms. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, (IJCAI-91)*, pages 172–177, Sydney, Australia, August 1991.
- [43] L. Méro. Some remarks on heuristic search algorithms. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence, (IJCAI-81)*, pages 572–574, Vancouver, August 1981.
- [44] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, San Jose, CA, July 1992.
- [45] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.
- [46] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [47] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.
- [48] J. C. Pemberton and W. Zhang. Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems. *Artificial Intelligence*, 81, 1996, in press.
- [49] I. Pohl. Practical and theoretical considerations in heuristic search algorithms. In E. W. Elcock and D. Michie, editors, *Machine Intelligence 8*, pages 55–72. Wiley, New York, 1977.
- [50] P. W. Purdom. A survey of average time analyses of satisfiability algorithms. *Journal of Information Processing*, 13:449–455, 1990.

- [51] A. Rényi. *Probability Theory*. North-Holland, Amsterdam, 1970.
- [52] S. Russell. Efficient memory-bounded search methods. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, 1992.
- [53] A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, (IJCAI-89)*, pages 297–302, Detroit, MI, August 1989.
- [54] D. J. Slate and L. R. Atkin. CHESS 4.5: The Northwestern University chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer, New York, 1977.
- [55] D. R. Smith. Random trees and the analysis of branch and bound procedures. *Journal of ACM*, 31:163–188, 1984.
- [56] G. C Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12:179–196, 1979.
- [57] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 756–761, Washington, D.C., July 1993.
- [58] B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Transactions on Software Engineering*, 11:922–934, 1985.
- [59] C. P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.
- [60] W. Zhang and R. E. Korf. An average-case analysis of branch-and-bound with applications: Summary of results. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 545–550, San Jose, CA, July 1992.
- [61] W. Zhang and R. E. Korf. Depth-first vs. best-first search: New results. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 769–775, Washington, D.C., July 1993.
- [62] W. Zhang and R. E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79:241–292, 1995.

- [63] W. Zhang and R. E. Korf. A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence*, 81, 1996, in press.
- [64] W. Zhang and J. C. Pemberton. Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems – Initial results. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, July 1994.