ELSEVIER

Artificial Intelligence 79 (1995) 241-292

Artificial Intelligence

Performance of linear-space search algorithms *

Weixiong Zhang *, Richard E. Korf

Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90024, USA

Received December 1992; revised June 1994

Abstract

Search algorithms that use space linear in the search depth are widely employed in practice to solve difficult problems optimally, such as planning and scheduling. In this paper, we study the average-case performance of linear-space search algorithms, including depth-first branch-and-bound (DFBnB), iterative-deepening (ID), and recursive best-first search (RBFS). To facilitate our analyses, we use a random tree $T(b,d)$ that has mean branching factor b, depth d, and node costs that are the sum of the costs of the edges from the root to the nodes. We prove that the expected number of nodes expanded by DFBnB on a random tree is no more than bd times the expected number of nodes expanded by best-first search (BFS) on the same tree, which usually requires space that is exponential in depth d. We also show that DFBnB is asymptotically optimal when BFS runs in exponential time, and ID and RBFS are asymptotically optimal when the edge costs of $T(b,d)$ are integers. If bp_0 is the expected number of children of a node whose costs are the same as that of their parent, then the expected number of nodes expanded by these three linear-space algorithms is exponential when $bp_0 < 1$, at most $O(d^4)$ when $bp_0 = 1$, and at most quadratic when $bp_0 > 1$. In addition, we study the heuristic branching factor of $T(b,d)$ and the effective branching factor of BFS, DFBnB, ID, and RBFS on $T(b,d)$. Furthermore, we use our analytic results to explain a surprising anomaly in the performance of these algorithms, and to predict the existence of a complexity transition in the Asymmetric Traveling Salesman Problem.

1. Introduction and overview

Search is a fundamental problem-solving technique. In this paper, we study search algorithms that are widely used in practice for problem solving. In particular, we are

* This research was supported by NSF Grant, #IRI-9119825, and a grant from Rockwell International to the second author, and partially by a GTE Graduate Fellowship (1992-93) and a UCLA Chancellor's Dissertation Year Fellowship (1993-94) to the first author.

* Corresponding author. Current address: USC/Information Sciences Institute, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292, USA. Telephone: (310)822-1511. E-mail: zhang@isi.edu.

SSDI 0004-3702(94)00047-6

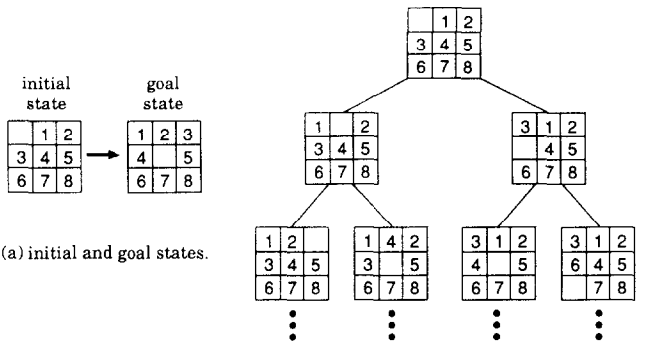

Fig. 1. An Eight Puzzle example.

interested in those algorithms that use space that is linear in the search depth, which we call *linear-space algorithms*. Linear-space algorithms are important because they are the algorithms of choice for large and difficult problems, such as the NP-hard problems [11]. The known linear-space search algorithms include depth-first branch-and-bound, iterative-deepening, and recursive best-first search. The primary goal of this paper is to understand the average-case time complexity of these linear-space algorithms when they are employed to solve difficult problems *optimally*.

1.1. Search problems

We start by discussing two search problems, the sliding-tile puzzles, which have been used extensively as example problems in artificial intelligence, and the (Asymmetric) Traveling Salesman Problem, which is ubiquitous in operations research. These two problems are used as benchmarks for our experiments, and to introduce the analytic model we will use and the search algorithms we will analyze.

We first present the problems themselves, their most effective operators, and the most commonly used or most efficient lower-bound cost functions, and then briefly describe how these problems can be solved. Operators are used to decompose a problem into subproblems if the original one cannot be solved directly. The lower-bound cost functions are used to guide a search algorithm.

1.1.1. Sliding-tile puzzles

A square sliding-tile puzzle consists of a $k \times k$ frame holding $k^2 - 1$ distinct movable tiles, and a blank space (see Fig. 1). Any tiles that are horizontally or vertically adjacent to the blank may move into the blank position. An operator is any such legal move. Given an initial and goal state of a sliding-tile puzzle, we are asked to find a minimum number of moves that transform the initial state into the goal state, which is NP-complete for arbitrary-size puzzles [43].

A commonly used cost function, or heuristic evaluation, is $f(n) = g(n) + h(n)$, where $g(n)$ is the number of moves from the initial state to state n, and $h(n)$ is the Manhattan

distance from n to the goal state. The Manhattan distance is computed by counting, for each tile not in its goal position, the number of moves along the grid it is away from its goal location, and summing these values over all tiles, excluding the blank. Manhattan distance is an underestimate of the minimum number of moves required to solve the problem, since every tile must move at least its Manhattan distance to its goal location, and only one tile can move at a time.

A given sliding-tile puzzle can be solved as follows. Starting at the initial state, the current state is expanded by individually moving each tile that is horizontally or vertically adjacent to the blank. Each such possible move produces a new state, a child of the current state. The cost function is then applied to all new states. A state that has been generated but not yet expanded is then selected as the next current state, and this state-selection and state-expansion process continues until there are no unexpanded states, or all unexpanded states have costs greater than or equal to the cost of the best goal node found so far. How a new state is selected depends on the search algorithms employed, which is discussed in detail in Section 1.3.

1.1.2. The Asymmetric Traveling Salesman Problem

Given n cities, $\{1,2,3,\ldots,n\}$, and a matrix $(c_{i,j})$ 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. Many NP-hard combinatorial optimization problems can be formulated as TSPs, such as vehicle routing, workshop scheduling, computer wiring, etc. [29]. When the cost matrix is asymmetric, i.e. the cost from city i to city j is not necessarily equal to the cost from city j to city i, then the problem is the asymmetric TSP (ATSP).

The most effective lower-bound cost function for the ATSP is the solution to the *assignment problem* [1,36]. 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 single tour, allowing collections of disjoint subtours, and thus 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 single complete tour, it is the solution to the ATSP as well. The assignment problem is solvable in $O(n^3)$ time [36].

We use an example, illustrated in Fig. 2, to introduce the operators [1]. We first solve the assignment problem for the given six cities. Assume that the assignment problem solution contains 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 $2 \rightarrow 3 \rightarrow 2$ is chosen to be eliminated, we have two choices. We may either exclude edge $(2,3)$ or edge $(3,2)$, each of which leads to a subproblem with an additional constraint, the excluded edge. We then solve the assignment problems of the subproblems, and further decompose a subproblem if its assignment problem solution is still not a single complete tour.

In order to keep the total number of subproblems generated as small as possible, we should avoid generating duplicate subproblems. This can be realized by including in

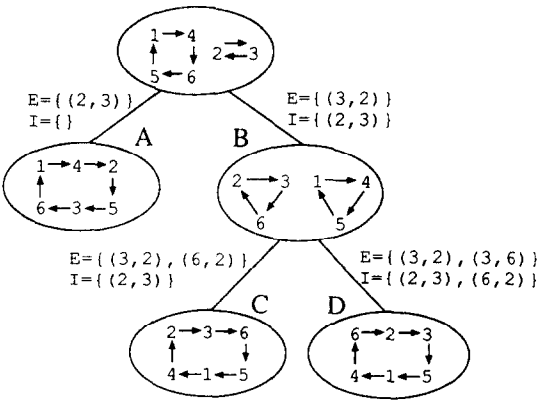

Fig. 2. An example of solving the ATSP.

the current subproblem any edges that were excluded in previous subproblems. In our example, suppose that we generate the first subproblem A by excluding edge $(2,3)$. The second subproblem B excludes edge $(3,2)$, but includes the edge $(2,3)$. Therefore, no subproblems generated under A can have edge $(2,3)$, but all subproblems under B will have edge $(2,3)$, guaranteeing that the subproblems will be mutually disjoint.

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

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

Since x_k is an excluded edge of the kth subproblem, $x_k \in E_k$, and it is an included edge of the $(k+1)$st subproblem, $x_k \in I_{k+1}$, any subproblems generated from the kth subproblem cannot contain edge x_k, but all subproblems obtained from the $(k+1)$st subproblem must include edge x_k. Therefore, no duplicate subproblems 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 subproblem and repeating the following: First, solve the assignment problem for the current subproblem. If the assignment problem solution is not a single complete tour, then select a subtour, and generate all child subproblems by eliminating edges in the subtour. Next, select as the current subproblem a new subproblem that has been generated but not yet expanded. This process continues until there are no unexpanded subproblems, or all unexpanded subproblems have costs greater than or equal to the cost of the best complete tour found so far. How to select the next current subproblem is described in Section 1.3.

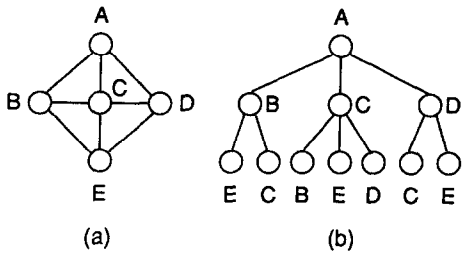

Fig. 3. A simple graph and part of its depth-first search tree.

1.2. State-space tree model

1.2.1. A state space

As described in the previous examples, 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* are configurations of the problem to be solved. The *operators* are actions that map one state to another. In general, 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 goal nodes that have specified properties, or a path from the initial state to a goal state. In this paper, we focus on state-space trees. A *state-space tree* is a state-space graph without cycles, in which leaf nodes are goal nodes, but of varying cost. The number of children of a node is referred to as the *branching factor* of the node.

We adopt the tree model for two reasons. The first is that a tree is a realistic model for problem solving in practice. As discussed in the ATSP example, a problem can often be decomposed into disjoint subproblems by including/excluding some entities in/from the solutions to the subproblems. This is a general principle that can be applied to most combinatorial optimization problems. If the problem is decomposed such that one entity included in one subproblem is excluded from its siblings, then we have a partition of the state space, which is a tree without any duplicate nodes.

The second reason is that a tree is an appropriate model for linear-space algorithms. While a graph with cycles is the most general model of a state space, linear-space algorithms explore a tree, at the cost of generating duplicate nodes. Due to their space restriction, linear-space algorithms cannot in general detect all duplicate nodes in a graph. For example, a brute-force depth-first search of the graph in Fig. 3(a), starting at node A, is shown in Fig. 3(b), in which duplicate nodes appear.

The nodes in a state-space tree have associated costs, which are used by a search algorithm to decide which node to explore next. The estimated cost of a node, $f(n)$, is 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 total cost of the best goal node in the tree underneath the given node.

A cost function is a *lower-bound* if it never overestimates the actual cost of a node, i.e. $f(n) \leqslant f^*(n)$, for all nodes n in the state space. A lower-bound cost function can be obtained by relaxing the original problem [39]. One such example is the assignment

problem for the ATSP. A cost function is *monotonic* if the cost of a child node n', $f(n')$, is always greater than or equal to the cost of its parent node n, i.e. $f(n') \geq f(n)$. The monotonic property comes from the fact that a child node typically represents a more constrained problem than its parent, and hence costs at least as much. The monotonic property is slighter stronger than the lower-bound property, and the former implies the latter, but not vise versa. Given a lower-bound function, a monotonic function can be constructed by taking the cost of a node as the maximum cost of all nodes on the path from the root to the node, guaranteeing that $f(n') \geq f(n)$, where n is the parent of n'.

A particular additive cost function, $f(n) = g(n) + h(n)$, has been widely used in artificial intelligence [16]. $g(n)$ is the sum of the cost of the path from the initial state to the current node n, and $h(n)$ is the estimated cost, or the *heuristic estimate*, from n to a goal node, for example the Manhattan distance. h is called *consistent* [39] if for any child node n' and its parent n, $h(n') + k(n,n') \geq h(n)$, where $k(n,n')$ is the cost of the edge from n to n'. Monotonicity of f and consistency of h are equivalent [23]. This can be simply shown as follows. Given $f(n') \geq f(n)$, or $g(n') + h(n') \geq g(n) + h(n)$, using $g(n') = g(n) + k(n,n')$, we then have $g(n) + k(n,n') + h(n') \geq g(n) + h(n)$, or $k(n,n') + h(n') \geq h(n)$.

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. Edge costs are nonnegative if node costs are monotonically non-decreasing with their depths. An edge cost can also be viewed as the cost of an operator that maps the parent to the child node.

1.2.2. A random tree model

An additional reason that we use a state-space tree model is that a tree is analytically tractable. To facilitate our average-case analyses, we introduce the following model.

Definition 1.1. 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, nonnegative, 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.

Fig. 4 shows an example of an incremental random tree, with the numbers on the edges and in the nodes being the edge costs and the resulting node costs, respectively.

Two important features of this model are worth mentioning. The first is that multiple optimal goal nodes may exist, in contrast to models in which only one optimal goal is allowed [12,39,41]. The second is that, unlike the conventional assumption that the costs of all nodes are independent [12,39,41], 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, which is called the *i.i.d.* assumption. The i.i.d. assumption also applies to the branching factors of different nodes. These assumptions, unfortunately, rarely hold

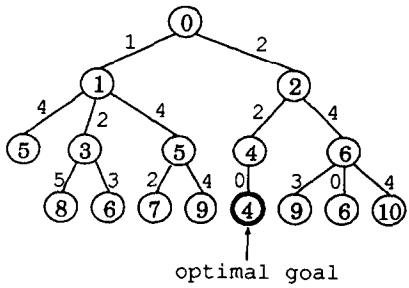

Fig. 4. A simple incremental random tree.

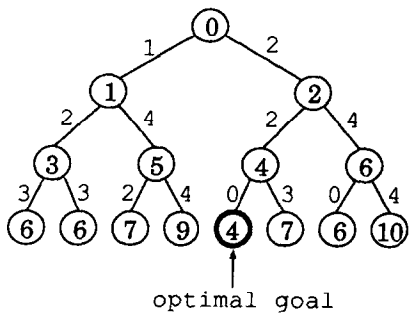

Fig. 5. A binary tree with depth three and edge costs from $\{0, 1, 2, 3, 4\}$.

in practice. Nevertheless, they are usually introduced to facilitate analyses, especially average-case analyses. In many cases, however, analytic results obtained under i.i.d. assumptions often characterize real-world problems where the assumptions do not hold. Examples are presented in Section 4.

Since the branching factor of a node is a nonnegative random variable, it may be zero. Thus, it is possible for a random tree to be finite, meaning that every path from the root in the tree has finite length. In order for a random tree to have an infinite number of nodes with nonzero probability, it is necessary and sufficient that the mean branching factor be greater than one, i.e. $b > 1$ [15]. In the rest of this paper, whenever we mention a random tree $T(b, d)$, we assume that $b > 1$. The reason for this assumption is that most of our results are asymptotic results as the tree depth goes to infinity, and hence we must rule out finite depth trees.

1.3. Search algorithms

A search algorithm is a strategy used to decide which node to explore next. We use a simple random tree to illustrate our algorithms. As shown in Fig. 5, it is a uniform binary tree with depth three, edge costs chosen from $\{0, 1, 2, 3, 4\}$, and an optimal goal node of cost four (4). The numbers on the edges are edge costs, and the numbers in the nodes are the resulting node costs.

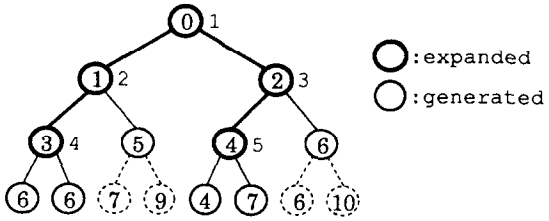

Fig. 6. The order of node expansions of best-first search.

1.3.1. Best-first search

The first algorithm we consider 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 nodes that have been generated but not yet expanded, until an optimal goal node is chosen for expansion. To maintain the partially expanded tree, BFS typically requires space that is *exponential* in the search depth, making it impractical for most applications. Pseudo-code for the algorithm is in Appendix A. Fig. 6 illustrates how BFS works on the tree of Fig. 5, with the numbers beside the nodes being the order in which the nodes are expanded.

A special case of BFS is the A* algorithm [16], which uses the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the sum of 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. An important feature of A* is that for a given consistent heuristic estimate h, it expands the minimum number of nodes among all algorithms guaranteed to find an optimal goal, up to tie-breaking among nodes whose cost equal to the optimal goal cost [7].

1.3.2. Depth-first branch-and-bound

Depth-first branch-and-bound (DFBnB) uses space that is only linear in the search depth. Starting at the root node, and with a global upper bound u on the cost of an optimal goal, DFBnB always selects a most recently generated node, or a 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 is selected for expansion whose cost is greater than or equal to u, it is pruned, because node costs are non-decreasing along a path from the root, and all descendents 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 an increasing order of their costs. This is called *node-ordering*. Throughout this paper, when we refer to DFBnB, we mean DFBnB with node ordering. A recursive version of the algorithm is included in Appendix A. Fig. 7 shows how DFBnB works on the tree of Fig. 5. The numbers beside the nodes are the order in which they are expanded.

The penalty for DFBnB to run in linear space is that it expands some nodes that are not explored by BFS, or some nodes whose costs are greater than the optimal goal cost. For example, the node with sequence number four (4) in Fig. 7 is not expanded by BFS

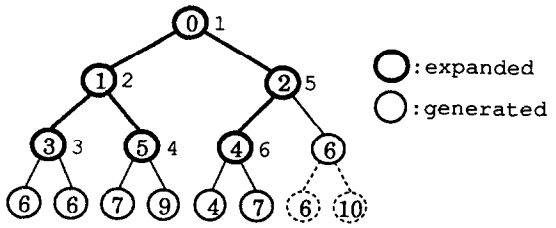

Fig. 7. The order of node expansions of depth-first branch-and-bound.

in Fig. 6, because its cost (5) is greater than the optimal goal cost (4). In addition, DFBnB does not work on a graph with cycles, since it may keep to expand the nodes on a cycle. In this case, DFBnB requires either a finite tree, or a cutoff depth in order to guarantee termination.

1.3.3. Iterative-deepening

In order to use space linear in the search depth, and never expand a node whose cost is greater than the optimal goal cost, we turn to iterative-deepening (ID) [22].

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 it 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 is listed in Appendix A. Fig. 8 shows how iterative-deepening works on the tree of Fig. 5, 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.

Iterative-deepening is a general search algorithm, and includes a number of special cases, depending on the cost function. For example, depth-first iterative-deepening (DFID) uses depth as cost and expands all nodes at a given depth before expanding any nodes at a greater depth. Uniform-cost iterative-deepening, an iterative version of Dijkstra's single-source shortest-path algorithm [8], uses the sum of the costs on the path from the root to a node as the node cost. Iterative-deepening-A* (IDA*) employs the A* cost function $f(n) = g(n) + h(n)$.

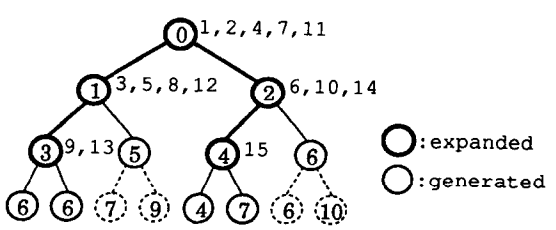

Fig. 8. The order of node expansions of iterative-deepening.

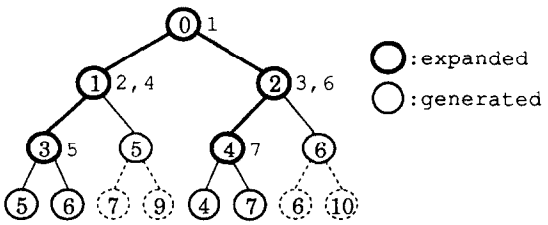

Fig. 9. The order of node expansions of recursive best-first search.

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 Fig. 8.

1.3.4. Recursive best-first search

Another problem with iterative-deepening is that it does not expand new nodes in best-first order when node costs are nonmonotonic. It is also somewhat inefficient even with a monotonic cost function. Recursive best-first search (RBFS) [24] always expands unexplored nodes in best-first order, regardless of the cost function, is more efficient than iterative-deepening with a monotonic cost function, and still runs in linear space. 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. For simplicity, we describe the behavior of the algorithm on the tree in Fig. 5 (see Fig. 9), and leave a formal description of the algorithm to Appendix A, and a full treatment of the algorithm to [24].

After expanding the root (see Fig. 9), 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. After expanding the left child, 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 is called recursively on the right child with a upper bound of 3, the value of the best frontier node in the left subtree. After expanding the right child, 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 expanding the left child and its left child, all frontier nodes in the left child are greater than or equal to 5, 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 point in time, RBFS maintains in memory the current search path, along with all immediate siblings of nodes on the current path, together with the cost 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 only expands seven nodes on the tree in Fig. 5 (see Fig. 9), but iterative-deepening expands fifteen nodes on the same tree (see Fig. 8). Similarly to iterative-deepening, however, RBFS also suffers from node re-expansion overhead.

1.4. A surprising anomaly

In spite of the fact that these search algorithms are widely used in practice, their performance is still not fully understood. For example, the following anomaly cannot be explained by existing results. Understanding the cause of this anomaly was one of the original motivations behind this work.

Fig. 10 shows the performance of DFBnB on trees with different uniform branching factors b, and edge costs uniformly and independently chosen from $\{0, 1, 2, 3, 4\}$. The horizontal axis is the tree depth d, and the vertical axis is the number of nodes generated, on a logarithmic scale. The straight dotted lines to the left show the numbers of nodes in the trees, which is $(b^{d+1} - 1)/(b - 1) = O(b^d)$, growing exponentially with the tree depth, and increasing with increasing branching factor. The curved solid lines to the right represent the average number of nodes generated by DFBnB to find a minimum-cost leaf node, averaged over 1000 trials for each branching factor and search depth.

Fig. 10 displays the following counter-intuitive anomaly. *For a fixed search depth, DFBnB can search the trees with larger branching factors faster.* For example, if the search depth is fixed at 50, then DFBnB generates 1,846,801 nodes on average on a random tree with branching factor two, 110,894 nodes on a tree with branching factor four, 9,076 nodes on a tree with branching factor six, and only 1,276 nodes on a tree with branching factor ten. *Alternatively, for a given amount of computation or total number of node generations, DFBnB can search deeper in the trees with greater branching factors.*

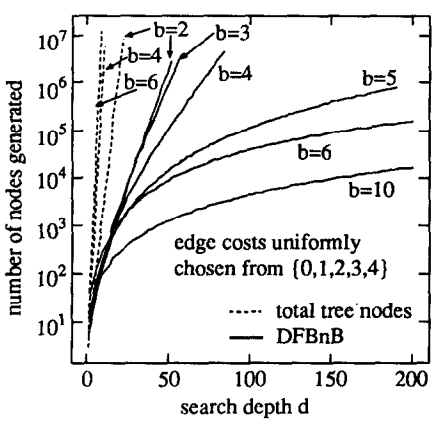

Fig. 10. Anomaly of depth-first branch-and-bound.

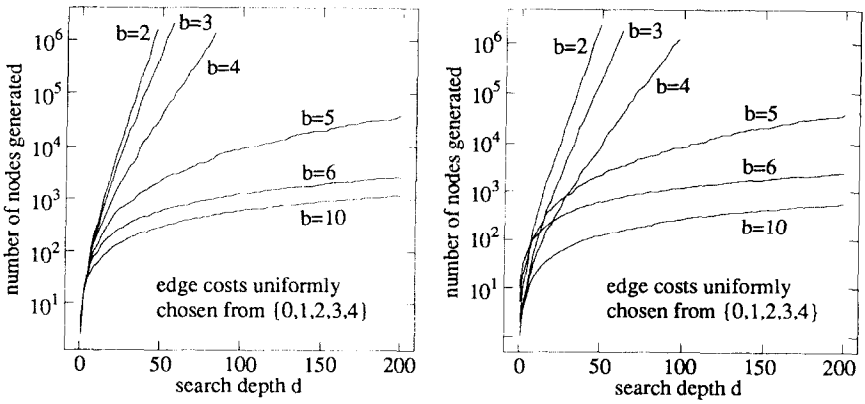

Fig. 11. Anomaly of iterative-deepening and recursive best-first search.

For example, if the total computation is fixed at 100,000 node generations, then DFBnB can reach depth 37 on a binary random tree, depth 50 on a tree with $b = 4$, depth 160 on a tree with $b = 6$, and depth more than 200 on a tree with $b = 10$. Furthermore, this anomaly exists in iterative-deepening and RBFS as well, as illustrated by Fig. 11.

1.5. Overview

All the algorithms mentioned above have to examine every node in a state space in the worst case. As the worst case is usually pathological, a more realistic measure of a search algorithm is its average-case performance. Despite the fact that linear-space search algorithms are used most of the time in practice, their average-case complexity is still unknown.

In this paper, we analyze the average-case complexity of linear-space search algorithms. To facilitate our analyses, we use a random tree $T(b, d)$ that has mean branching factor b, depth d, and a node cost computed as the sum of the edge costs on its path to the root. We analytically show that on a random tree $T(b, d)$, DFBnB expands at most $O(d \cdot N(b, d))$ nodes on average, where $N(b, d)$ is the expected number of nodes expanded by BFS. We also prove that DFBnB is asymptotically optimal when BFS runs in exponential time. We further show that iterative-deepening and RBFS are asymptotically optimal on a random tree with integer edge costs. If p_0 is the probability of a zero-cost edge in $T(b, d)$, then the expected number of children of a node whose costs are the same as that of their parent is bp_0. Overall, we prove that the average number of nodes expanded by these linear-space algorithms is exponential when $bp_0 < 1$, at most $O(d^4)$ when $bp_0 = 1$, and at most quadratic when $bp_0 > 1$. In addition, we study the heuristic branching factor of $T(b, d)$, and the effective branching factor of BFS and linear-space algorithms. These results are presented in Section 2.

The results in Section 2 indicate that there is an abrupt complexity transition, from exponential to polynomial in the search depth, when the average number of same-cost children bp_0 increases from less than one to greater than or equal to one. We examine

this complexity transition in Section 3. In Section 4, we apply the analytic results to explain a previously observed anomaly of DFBnB on sliding-tile puzzles, and predict a complexity transition on the ATSP. We further consider how to select a search algorithm for a given problem in Section 5. Our experimental results show that linear-space algorithms are usually the algorithms of choice for large problems, both in terms of space and running time. We discuss related work in Section 6. Finally, we conclude in Section 7.

Previous results from this research were presented in [57, 58].

2. Complexity of linear-space search

In this section, we analyze the expected complexity of the linear-space search algorithms and their effective branching factors on random trees, and the heuristic branching factor of random trees. The difficulty to directly obtain the expected complexity of the linear-space algorithms is that the mathematical tools used for best-first search cannot be carried over to these algorithms. To circumvent this difficulty, our approach is to characterize the relationship between the expected complexity of linear-space algorithms and that of best-first search.

2.1. Problem complexity and optimal goal cost

To analyze the expected complexity of these algorithms, we first consider the problem complexity of finding an optimal goal node, in terms of the total number of node expansions.

Lemma 2.1. *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.*

Proof. See Appendix B. □

This lemma indicates that the problem complexity is directly related to the optimal goal cost. The optimal goal cost and other properties of the random tree have been studied using the tools of an area of mathematics called branching processes [15], and in particular, age-dependent branching processes [15] and first-percolation processes [14, 21].

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 at least 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, which causes the optimal goal cost to increase with depth.

Lemma 2.2 ([32,33]). *Let C^* be the expected cost of optimal goal nodes of a random tree $T(b,d)$ with $b > 1$. As $d \to \infty$,*

(1) *$C^*/d \to \alpha$ almost surely*[1] *when $bp_0 < 1$, where α is a constant independent of d,*

(2) *$C^*/(\log\log d) \to 1$ almost surely when $bp_0 = 1$,*

(3) *C^* almost surely remains bounded when $bp_0 > 1$.*

Lemma 2.2 means that when $bp_0 < 1$, αd is the dominant term in the optimal goal cost C^*. Similarly, when $bp_0 = 1$, $\log\log d$ is the dominant term in C^*.

We can further show the following monotonic property of the optimal goal cost.

Lemma 2.3. *On a random tree $T(b,d)$ with $bp_0 > 1$, as $d \to \infty$, the expected cost of optimal goal nodes decreases when bp_0 increases for a given b, or a given $p_0 > 0$. In particular, the expected goal cost approaches zero when $p_0 \to 1$ for a given b, or when $b \to \infty$ for a given $p_0 > 0$.*

Proof. See Appendix B. □

2.2. Best-first search

To characterize the expected complexity of the linear-space algorithms, we first consider the expected complexity of BFS.

Lemma 2.4. *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.*

Proof. See Appendix B. □

Lemma 2.4 implies that BFS is optimal, up to tie-breaking, on a random tree as well, since it has monotonic node costs.

It has been shown in [32,33] 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.

[1] A sequence X_n of random variables is said to converge *almost surely* (with probability one) to X if $P(\lim_{n\to\infty} X_n = X) = 1$ [44].

Lemma 2.5. *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) $\theta(\beta^d)$ [2] *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 $d \to \infty$, *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, and to find one and verify that it is optimal is easy.

2.3. Depth-first branch-and-bound

To reiterate, there are two major differences between DFBnB and BFS. One is that DFBnB runs in space linear in the search depth, whereas BFS usually requires space exponential in the search depth. The other difference is that DFBnB may expand nodes whose costs are greater than the optimal goal cost, whereas BFS does not. The second difference makes the analysis of DFBnB more difficult, as the mathematical tool of branching processes used for BFS cannot be carried over to DFBnB directly. To circumvent this difficulty, our approach is to determine the relationship between the complexity of DFBnB and that of BFS. More specifically, we try to answer the question: how many more nodes does DFBnB generate than BFS?

Theorem 2.6. *Let* $N_B(b,d)$ *be the expected number of nodes expanded by best-first search, and* $N_D(b,d)$ *the expected number of nodes expanded by depth-first branch-and-bound, on a random tree* $T(b,d)$. *As* $d \to \infty$,

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

Proof. See Appendix B. □

Theorem 2.6 is one of the two most important analytic results of this paper. It shows the relationship between the expected complexity of DFBnB and that of BFS. Most of the following results on DFBnB are based on this theorem.

[2] $\theta(\chi(x))$ denotes the set of all functions $\omega(x)$ such that there exist positive constants c_1, c_2, and x_0 such that $c_1\chi(x) \leqslant \omega(x) \leqslant c_2\chi(x)$ for all $x \geqslant x_0$. In words, $\theta(\chi(x))$ represents functions of the same asymptotic order as $\chi(x)$.

Corollary 2.7. *On a random tree $T(b,d)$, $N_D(b,d) < O(d \cdot N_B(b,d-1))$, where $N_B(b,d)$ and $N_D(b,d)$ are the expected numbers of nodes expanded by best-first search and depth-first branch-and-bound, respectively.*

Proof. See Appendix B. □

Theorem 2.6, combined with Lemma 2.5, also leads to the following.

Theorem 2.8. *The expected number of nodes expanded by depth-first branch-and-bound for finding an optimal goal node of a random tree $T(b,d)$, as $d \to \infty$, is*

(1) *$\theta(\beta^d)$ when $bp_0 < 1$, meaning that depth-first branch-and-bound is asymptotically optimal, where β is the same constant as in Lemma 2.5,*
(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.

Proof. See Appendix B. □

Theorem 2.8 is analogous to Lemma 2.5, showing the expected number of nodes expanded by DFBnB under different values of the expected number of same-cost children. Note that β in Theorem 2.8 is the same β as in Lemma 2.5. This means that when $bp_0 < 1$ and $d \to \infty$, DFBnB expands asymptotically the same number of nodes as BFS. Since BFS is optimal by Lemma 2.4, DFBnB is asymptotically optimal in this case.

We are also interested in the behavior of DFBnB when the expected number of same-cost children grows.

Corollary 2.9. *On a random tree $T(b,d)$ with $bp_0 > 1$, as $d \to \infty$, the expected number of nodes expanded by depth-first branch-and-bound approaches $O(d)$ when $p_0 \to 1$ for a given b, or when $b \to \infty$ for a given $p_0 > 0$.*

Proof. See Appendix B. □

This corollary means that when $bp_0 > 1$ and bp_0 increases for a given p_0 or a given b, it becomes easier to find an optimal goal node, and verify that it is optimal.

2.4. 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 $\{a_0\Delta, a_1\Delta, \ldots, a_m\Delta\}$ for nonnegative integers $a_0, a_0, \ldots, a_m$, and positive constant Δ that is chosen so that the integers $a_0, a_0, \ldots, a_m$ are relatively prime [15]. 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 2.10. *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 $d \to \infty$, which is asymptotically optimal.*

Proof. See Appendix B. □

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 2.10 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 2.10 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 Lemma 2.5.

On the other hand, when edge costs are chosen from a lattice distribution, node regenerations only 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. The fact that the total number of iterations is linear in the search depth when edge costs are chosen from a lattice distribution, as shown in the proof of Theorem 2.10, 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.

2.5. Recursive best-first search

With a monotonic cost function, recursive best-first search (RBFS) generates fewer nodes than iterative-deepening, up to tie-breaking [24]. Thus, for a state space with

Table 1
Expected complexity of BFS and DFBnB

Algorithm	$bp_0 < 1$	$bp_0 = 1$	$bp_0 > 1$
BFS	$\theta(\beta^d)$ optimal	$\theta(d^2)$ optimal	$\theta(d)$ optimal
DFBnB	$\theta(\beta^d)$ asym. optimal	$O(d^3)$	$O(d^2)$

Table 2
Expected complexity of ID and RBFS

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

monotonic node costs, the complexity of iterative-deepening is an upper bound on the complexity of recursive best-first search. Consequently, the expected complexity of iterative-deepening on a random tree is an upper bound on the expected complexity of RBFS. Thus, RBFS is also asymptotically optimal when edge costs are chosen from a lattice distribution.

2.6. *Summary of tree search complexity*

Tables 1 and 2 summarize the results of this section, i.e. the average number of nodes expanded by best-first search (BFS), depth-first branch-and-bound (DFBnB), iterative-deepening (ID), and recursive best-first search (RBFS) on a random tree $T(b,d)$, asymptotically as $d \to \infty$.

2.7. *Branching factors*

Before we close this section, we discuss three different branching factors. The first one is the average brute-force branching factor b of a tree, which is the average number of children of a node in the tree. b captures the rate of growth of the number of nodes from one depth to the next.

The second is the asymptotic *heuristic branching factor*, or heuristic branching factor B, of a tree [51]. B is the ratio of the average number of nodes with a given cost x, to the average number of nodes with the largest cost less than x, in the limit as $x \to \infty$.

The heuristic branching factor B was introduced to capture the overhead of iterative-deepening over BFS. Iterative-deepening is asymptotically optimal when $B > 1$, because the average number of node expansions increases exponentially with successive iterations. In this case, the overhead of the total average number of nodes expanded by iterative-deepening over those expanded by BFS is a multiplicative constant $(B/(B-1))^2$ as $d \to \infty$ [22]. When edge costs are continuous variables, node costs are unique, which leads to $B = 1$. When $bp_0 > 1$, the optimal goal cost remains a constant as $d \to \infty$, so that B does not exist because it is defined only in the limit of large node costs. When edge costs are chosen from a lattice distribution and $bp_0 \leqslant 1$, the average value of B is greater than one, because iterative-deepening is asymptotically

optimal (Theorem 2.10). As we will see in Theorem 2.11, B approaches a constant in this case.

Theorem 2.11. *Let B be the asymptotic heuristic branching factor of a random tree $T(b,d)$ with $b > 1$. $B = 1$ when edge costs are chosen from a continuous distribution. When $bp_0 < 1$, and edge costs are chosen from a lattice distribution on $\{a_0 = 0, a_1\Delta, a_2\Delta, \ldots, a_m\Delta\}$ with probabilities $p_0, p_1, p_2, \ldots, p_m$, respectively, where $\Delta > 0$ is chosen such that the nonnegative integers $a_1 < a_2 < \cdots < a_m$ are relatively prime, B approaches a constant as $d \to \infty$, which is a solution greater than one to the equation*

$$\sum_{i=0}^{m} \frac{p_i}{B^{a_i}} = \frac{1}{b}.$$

Proof. See Appendix B. □

The third branching factor is the asymptotic *effective branching factor*, or effective branching factor β, of a search algorithm [40]. β of a search to depth d is the dth root of the average number of nodes expanded, as $d \to \infty$. In other words, β measures the relative increase in average complexity due to extending the search depth by one extra level. β in Lemma 2.5 and Theorem 2.8, for example, is the effective branching factor of BFS and DFBnB.

It is evident that the effective branching factor β of a search algorithm on a state-space tree cannot be greater than the brute-force branching factor b of the tree. However, the heuristic branching factor B of a tree can be either greater or less than b, and can be either greater or less than β of an algorithm. We have the following relationship among β, B, and the optimal goal cost C^* when edge costs are chosen from a lattice distribution.

Theorem 2.12. *On a random tree $T(b,d)$ with $b > 1$, when $bp_0 < 1$ and edge costs are chosen from a lattice distribution, best-first search, depth-first branch-and-bound, iterative-deepening, and recursive best-first search all have the same effective branching factor $\beta = B^{\alpha}$ as $d \to \infty$, where p_0 is the probability that an edge has cost zero, B is the asymptotic heuristic branching factor of $T(b,d)$, α is the limit of C^*/d as $d \to \infty$, and C^* is the optimal goal cost.*

Proof. See Appendix B. □

3. Complexity transition of tree search

3.1. Average-case complexity transition

The results of Section 2 show that the average-case complexity of a tree search algorithm on random trees experiences a dramatic transition, from exponential to polynomial. This phenomenon is similar to a *phase transition*, which is a dramatic change to some

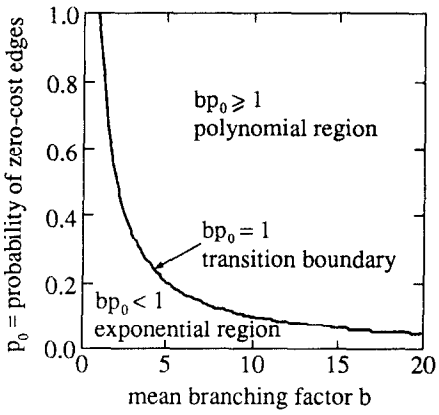

Fig. 12. Difficult and easy regions for tree search.

problem property as some *order parameter* changes across a critical point. The simplest example of a phase transition is that water changes 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 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 $bp_0 < 1$, both BFS and the linear-space algorithms expand an exponential number of nodes on average. On the other hand, when $bp_0 \geq 1$, BFS and the linear-space algorithms run in polynomial average time. Therefore, their 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. Fig. 12 illustrates these two complexity regions and the transition boundary between them.

The condition of $bp_0 > 1$, $bp_0 = 1$ or $bp_0 < 1$ can be estimated by random sampling of the nodes in a state space, counting the average numbers of same-cost children. If the edge costs are independent of each other, this random sampling and our analytic results provide a means to estimate the performance of a search algorithm. Unfortunately, the independence of edge costs often does not hold in practice. Nevertheless, the independence assumption may be a reasonable one for some cases, as discussed in Section 4.

3.2. Finding all optimal goal nodes

The complexity transition exists if only one optimal goal node is desired. To find all solutions, however, is difficult in both regions of $bp_0 \geq 1$ and $bp_0 < 1$, but for different reasons. When $bp_0 > 1$, a node has at least one same-cost child on average. Consequently, the expected number of optimal goal nodes increases exponentially with the goal depth. Therefore, to find *all* optimal goal nodes is difficult because the expected number of optimal goal nodes is exponential. On the other hand, when $bp_0 < 1$, the expected number of nodes that have the minimum cost at a particular depth is small. However, the expected number of nodes whose costs are less than the optimal goal

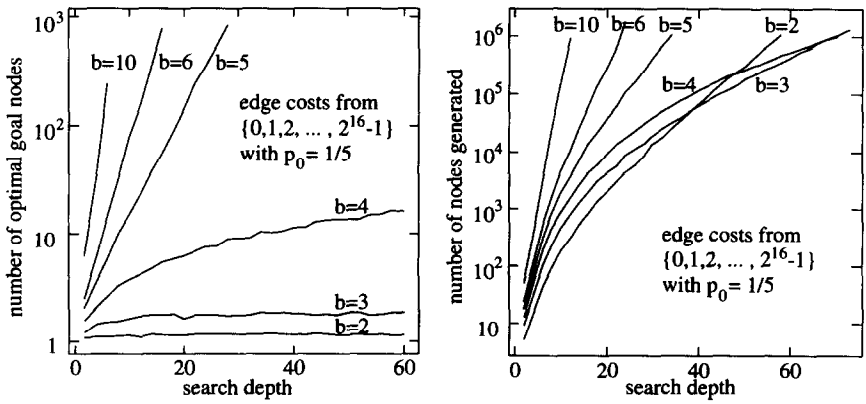

Fig. 13. Finding all solutions is always difficult.

cost increases exponentially with the tree depth. Hence, to find all optimal goal nodes is difficult because there exist only *a few* optimal goal nodes among an exponential number of nodes with lower costs.

Fig. 13 shows an example of finding all optimal goal nodes on random trees with uniform branching factor b and $p_0 = 1/5$ using DFBnB. To emphasize the impact of p_0, we choose the remaining nonzero edge costs uniformly from $\{1, 2, \ldots, 2^{16} - 1\}$. The results of Fig. 13 are averaged over 1000 trials. Fig. 13(a) shows that the average number of optimal goal nodes increases exponentially when $bp_0 > 1$. Fig. 13(b) gives the corresponding average number of nodes generated by DFBnB, indicating that it is exponential for all values of bp_0.

3.3. Meaning of p_0

The previous results indicate that the complexity transition is closely related to the number of edges whose costs are zero. What if there are no zero-cost edges? Assume, for example, that the edges have costs $\{\delta, \delta + 1, \delta + 2, \ldots, \delta + m\}$ with probabilities $p_0, p_1, p_2, \ldots, p_m$, respectively, where δ is a positive number. Does a complexity transition exist in this case?

Consider an example of a random tree with uniform branching factor b and edge costs uniformly chosen from $\{1, 2, 3, 4, 5\}$. This is only different from the example in Fig. 5 of Section 1 by shifting the edge costs by one. At first glance, this should not affect the performance of any algorithm, since the same value is added to every edge in the tree. Fig. 14 is our experimental results of DFBnB, averaged over 1000 runs. Fig. 14 shows that there is no complexity transition when edge costs cannot take value zero!

When edge costs in a tree change, node costs change accordingly. However, the costs of nodes located at different depths are not adjusted by the same amount, even if the same value is added to *every* edge, because the cost of a node is the sum of edge costs on the path from the root to the node. Thus, a larger value is added to a node at a deeper depth than a node at a shallower depth, and the costs of the nodes at depth d, where the

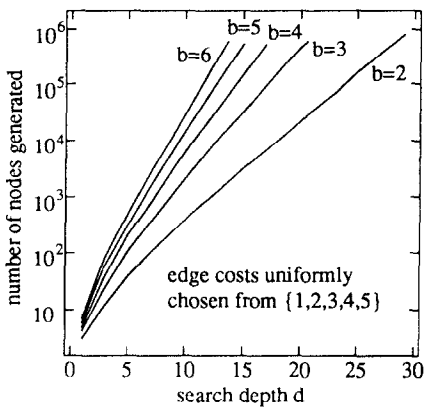

Fig. 14. There is no search anomaly without zero-cost edges.

optimal goal node is located, increase by the maximum amount. In consequence, more nodes will have new costs that are less than the new optimal goal cost, so that searching for an optimal goal node becomes harder. Furthermore, when there is no zero-cost edge, node costs strictly increase with depth. If the minimum edge cost is δ, then the cost of a node at depth d is at least δd, which is a lower bound on the optimal goal cost. There are more nodes that have costs less than or equal to δd on a tree with a larger branching factor than on a tree with a smaller branching factor. This is why there is no search anomaly when edge costs are nonzero, as shown in Fig. 14.

We can use the information of the minimum edge cost to increase search efficiency, however. Consider the scenario when DFBnB is generating a node n at depth j. The node n and the whole subtree below it do not need to be explored if $f(n)+(d-j)\delta \geqslant u$, where $f(n)$ is the cost of node n, d is the goal depth, and u is the current upper bound. This is because the cost of a leaf node below n is at least $f(n)+(d-j)\delta$.

In summary, *the intrinsic meaning of p_0 is the probability that edges take the minimum cost,* and this cost is known *a priori*. In other words, a known minimum edge cost and zero edge cost are equivalent. Given an edge-cost distribution with a known minimum value, we can convert this distribution into one with zero edge costs by subtracting the minimum cost from every edge.

The obstacle we have to overcome in practice is to obtain the minimum cost of all edges in the state-space tree of a real-world problem. Although this minimum edge cost is generally unknown, it can be learned or estimated by sampling the state-space tree. This learned minimum edge cost can then be subtracted from every edge encountered by a search algorithm to improve its efficiency.

p_0 can also be considered as a rough measure of the accuracy of a cost function. The accuracy of a cost function on a given problem can be measured by the difference between the optimal goal cost and the cost of the initial node in a state space. If this difference is zero, then the cost function is exact on the root node. In a random tree, this difference is represented by the optimal goal cost, as the root has cost zero. In a random tree, a larger p_0 gives rise to more same-cost children, and thus reduces the optimal goal cost, according to Lemma 2.2. Therefore, the larger p_0 is, the closer the estimated cost

of a node is to its actual cost, and the more accurate the cost function will be. However, the edge costs in the state space of a real problem are generally dependent upon each other. It is not clear how the dependence of edge costs will impact our results.

4. Applications

Our analyses in Section 2 explain the search anomaly presented in Figs. 10 and 11 in Section 1.4. Since the edge costs in the trees are uniformly chosen from $\{0, 1, 2, 3, 4\}$, the probability p_0 of a zero-cost edge is a constant $1/5$. When the branching factor b is less than 5 ($bp_0 < 1$), the average-case complexity of a search algorithm is exponential in search depth d; it is at most $O(d^4)$ when $b = 5$ ($bp_0 = 1$); and it is quadratic in d when $b > 5$ ($bp_0 > 1$). Thus, when the branching factor b increases, the average-case complexity decreases.

This section presents applications of our analytic results to our benchmark problems, sliding-tile puzzles and the Asymmetric Traveling Salesman Problem.

4.1. Anomaly of DFBnB on sliding-tile puzzles

Given an initial and a goal state of a sliding-tile puzzle, we are asked to find a sequence of moves that map the initial state into the goal state. In a real-time setting, we may have to make a move with limited computation. One approach to this problem, called *fixed-depth lookahead search* [23], is to search from the current state to a fixed depth, then move to the child of the current state that contains the minimum-cost frontier node in its subtree, and take that child as the next current state.

Fig. 15 shows experimental results of lookahead searches on sliding-tile puzzles using DFBnB. The cost function used is $f(n) = g(n) + h(n)$, where $g(n)$ is the total number of moves made from the initial state to node n, and $h(n)$ is the Manhattan distance from n to the goal state. The straight dotted lines to the left represent the total numbers of nodes in the search tree, for the Eight, Fifteen, Twenty-four, and Ninety-nine puzzles. The curved solid lines to the right represent the total numbers of nodes expanded by DFBnB. The results show the following anomaly, originally reported in [23]. *For a given search depth, DFBnB can search the trees of larger puzzles faster. Alternatively, for a given amount of computation, DFBnB can search deeper in the trees of larger puzzles.* For example, if we fix the total computation at one million node expansions, DFBnB can reach depth 35 on the Eight Puzzle, depth 42 on the Fifteen Puzzle, depth 49 on Twenty-four Puzzle, and depth 79 on the Ninety-nine Puzzle, respectively.

An explanation for this anomaly is the following. Moving a tile either increases its Manhattan distance h by one, or decreases it by one. Since every move increases the g value by one, the cost function $f = g + h$ either increases by two or stays the same with each move. Thus, this problem can be approximately modeled by a random tree with edge costs zero or two, if dependence among edge costs is ignored. The probability that the h value either increases or decreases by moving a tile is roughly one half initially, independent of the problem size. Thus, the probability that a child node has the same cost as its parent is approximately one half, i.e. $p_0 \approx 0.5$. In addition, the

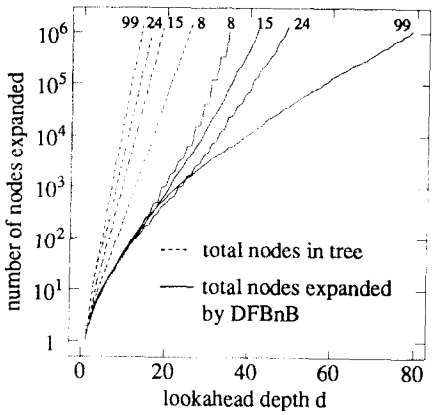

Fig. 15. Anomaly of lookahead search on sliding-tile puzzles.

average branching factors b of the Eight, Fifteen, Twenty-four, and Ninety-nine puzzles are approximately 1.732, 2.130, 2.368, and 2.790, respectively, i.e. b grows with the puzzle size. Thus, bp_0 increases with the puzzle size as well, and lookahead search is easier on larger puzzles following the analyses in Section 2.

This explanation, however, does not imply that solving a larger puzzle is easier than solving a smaller one. First of all, the solution length for a larger puzzle is longer than that of a smaller one, making the larger puzzle more difficult to solve. Secondly, for a particular problem instance, a unique board configuration of the puzzle is specified as the goal state. Although multiple nodes with the same board configuration may be encountered during the search, the number of copies of the goal state is relatively small compared to the number of nodes with minimal cost at a fixed search depth. Finally, the most important reason is that the assumption of independent edge costs is not valid. A sequence of moves that decrease the Manhattan distance makes a move that increases the distance more likely. Specifically, from our experiments on 1000 random initial states of the Eight Puzzle, p_0 is 0.501 initially, and steadily decreases with search depth, making the problem more difficult as the search depth increases.

4.2. *Complexity transition on the Asymmetric TSP*

Consider the node costs in the state-space tree of the ATSP, which are solution costs of the corresponding assignment problems (Section 1.1.2), with intercity costs uniformly chosen from $R = \{0, 1, 2, \ldots, r\}$, for some positive integer r. Let us examine the relationship between the number of distinct intercity costs r, and the edges in the state-space tree which have cost zero. The probability that two sets of n values from R have the same total sum is smaller if r is larger. Thus, the probability that two sets of n edges in the assignment problem solutions to two subproblems have the same total cost decreases as r increases. When r is small compared to the number of cities, the probability that the assignment problem cost of a child subproblem in the search tree is equal to the assignment problem cost of its parent is large. In other words, the

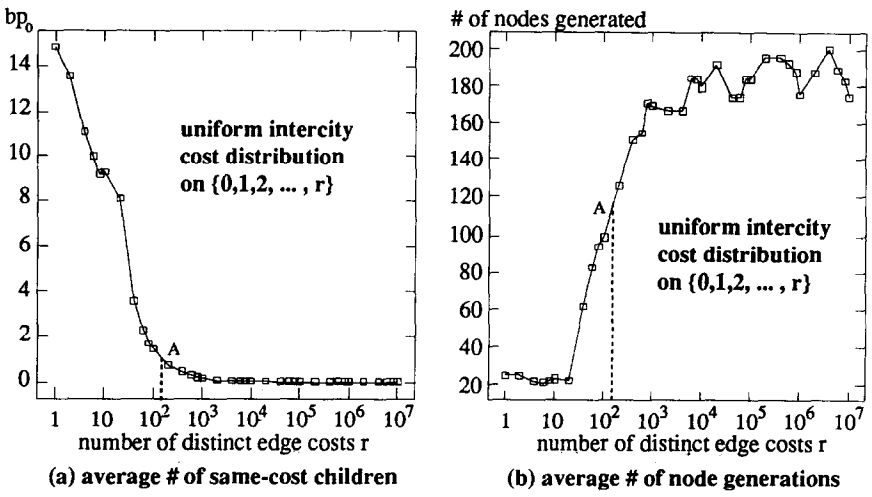

Fig. 16. Complexity transition on 100-city Asymmetric TSP, uniform distribution.

probability p_0 of zero-cost edges in the search tree is larger when r is smaller, so that the average number of same-cost children bp_0 may be greater than one, if b does not significantly decrease when r increases. Therefore, the problem may be easy to solve when r is small. Conversely, the probability that the assignment problem cost of a child is equal to that of its parent is smaller when r is relatively larger, as is the probability p_0 of zero-cost edges in the search tree. Consequently, the problem may be difficult to solve when r is large.

The above argument suggests that there may exist a complexity transition of BnB on the ATSP as r changes, following the complexity transition of random-tree search shown in Fig. 12. We experimentally tested this prediction by experiments on 100-city ATSPs. In our experiment, we randomly generated 1000 problem instances whose initial assignment problem solutions are not single complete tours, solved the problems using DFBnB, and averaged the results from these instances. We examined the relationship between the average number of same-cost children bp_0 of a node in the search tree of the ATSP and the average number of nodes generated by DFBnB. Fig. 16 shows the results, where the horizontal axes are the number r of distinct intercity costs, on a logarithmic scale.

Fig. 16(a) shows that $bp_0 > 1$ when $r \leqslant 130$, and $bp_0 < 1$ when $r > 130$, indicated by point A. Following the complexity transition of random-tree search in Fig. 12, this means that the problem is easy to solve when $r \leqslant 130$, but becomes difficult when $r > 130$. Fig. 16(b) illustrates the average numbers of nodes expanded by DFBnB under different values r of distinct intercity costs, showing a complexity transition. When $r \leqslant 20$ the problem is easy and it is difficult when $r > 1000$. A similar complexity transition has also been found on 200-, 300-, 400- and 500-city random ATSPs.

However, the transition from easy problem instances to difficult ones is not as dramatic as we expected. This is most likely due to the following factors. First, bp_0 decreases

gradually when r increases for $r < 1000$, making the complexity increase slowly. Secondly, the search trees have relatively small depths, while the complexity transitions of random tree search are asymptotic results as tree depth approaches infinity. Finally and more importantly, edge costs in the search tree are not independent from each other, nor are the branching factors of different nodes, so that the i.i.d. assumptions made for random trees are violated.

As suggested by Section 2, if we take the particular value of r such that $bp_0 = 1$ as the transition point, point A in Fig. 16 for instance, then this transition point increases with the problem size, or the number of cities. It remains an open problem, however, to analytically determine the value of the cost range r at which the transition occurs for a random ATSP.

In short, the average-case complexity of the ATSP is primarily determined by the average number of same-cost children in the problem space of the ATSP. Our results showed that difficult ATSP instances can be easily generated by using a large number of distinct intercity costs, which is useful when we need difficult ATSP instances to test a search algorithm.

5. Algorithm selection

The complexity measure of search algorithms used in Section 2 is the total number of nodes expanded or generated. However, space and running time are two more practical and important measures for choosing a search algorithm to find an optimal solution of a given problem. We continue our investigation below by taking into account the space and running time complexity. The purpose is to provide some guidelines for algorithm selection.

First consider space complexity. In order to select one node of minimum cost among all nodes that have been generated but not yet expanded, which we call *active nodes*, BFS has to maintain all the active nodes in memory, so that its space required is usually exponential in the search depth. Given the ratio of memory to processing speed on current computers, BFS typically exhausts the available memory in a matter of minutes, halting the algorithm. For example, if a new active node is generated in one millisecond on a computer with ten megabyte memory, then BFS will fill the memory in less than twenty minutes. Therefore, BFS is not applicable to large problems. Since linear-space algorithms use space that is only linear in the search depth, which is linear in problem size, they are the only algorithms of choice for large problems in practice.

Assuming that memory is not a constraint, given the analytic results in Section 2, it is still not clear which algorithm, BFS, DFBnB, iterative-deepening or RBFS, we should use for a given problem. Although BFS expands fewer nodes than the other algorithms, does it really run faster in practice? It is also not clear from the analytic results which linear-space algorithm runs faster than the others under different conditions. The remaining of this section experimentally compares running time of these algorithms on random trees, sliding-tile puzzles and the Asymmetric Traveling Salesman Problem.

5.1. Comparison on the analytic model

We first compare the average number of nodes expanded by BFS, DFBnB, iterative-deepening, and RBFS, and then consider the running time of these algorithms on random trees.

5.1.1. Node expansions

We use random trees with uniform branching factors, and two different edge-cost distributions. In the first case, edge costs are uniformly distributed among $\{0, 1, 2, 3, 4\}$. We choose this as a representative of discrete edge costs. In the second case, edge costs are set to zero with probability $p_0 = 1/5$, and nonzero edge costs are uniformly chosen from $\{1, 2, 3, \ldots, 2^{16} - 1\}$. The purpose of choosing this hybrid edge-cost distribution is twofold. We set $p_0 = 1/5$ to study the impact of p_0 on the complexity of the search algorithms. We take nonzero costs from a large range to simulate a continuous distribution. Note that this simulation only approximates a continuous distribution for shallow trees. When d is relatively large, this distribution should be treated as a discrete distribution. We choose three different branching factors: $b = 2$ for an exponential complexity case ($bp_0 < 1$), $b = 5$ for the transition case ($bp_0 = 1$), and $b = 10$ for a polynomial complexity case ($bp_0 > 1$).

The algorithms are run to different depths, each with 1000 trials. The average results are shown in Fig. 17. These results are consistent with the analytic results: BFS expands the fewest nodes among all algorithms, and RBFS is superior to iterative-deepening. DFBnB is asymptotically optimal when $bp_0 < 1$ (Figs. 17(a) and 17(d)), and iterative-deepening and RBFS are asymptotically optimal when edge costs are discrete (Figs. 17(a), 17(b) and 17(c)). However, when $bp_0 < 1$ and edge costs are chosen from a large range (Fig. 17(d)), the slope of the iterative-deepening curve is nearly twice the slope of the BFS curve. This is consistent with the analytic result that iterative-deepening expands $O(N^2)$ nodes on average when edge costs are chosen from a continuous distribution, where N is the expected number of nodes expanded by BFS.

Fig. 17 also provides additional information not provided by the analytic results. When $bp_0 > 1$ (Figs. 17(c) and 17(f)), iterative-deepening and RBFS are asymptotically optimal, regardless of the number of distinct nonzero edge costs. Moreover, when $bp_0 \geqslant 1$ and edge costs are discrete (Figs. 17(b) and 17(c)), DFBnB is worse than both iterative-deepening and RBFS. In these cases, the overhead of DFBnB, the number of nodes expanded whose costs are greater than the optimal goal cost, is larger than the re-expansion overheads of iterative-deepening and RBFS. When $bp_0 < 1$ and edge costs are discrete (Fig. 17(a)), however, DFBnB outperforms both iterative-deepening and RBFS. Fig. 17(d) also shows that when $bp_0 < 1$ and edge costs are chosen from a large set of values, RBFS has the same unfavorable asymptotic complexity as iterative-deepening.

In summary, for large problems that can be formulated as a tree with a bounded depth, and require exponential computation ($bp_0 < 1$), DFBnB should be used, and for easy problems ($bp_0 \geqslant 1$) or those with unbounded depth, RBFS should be adopted.

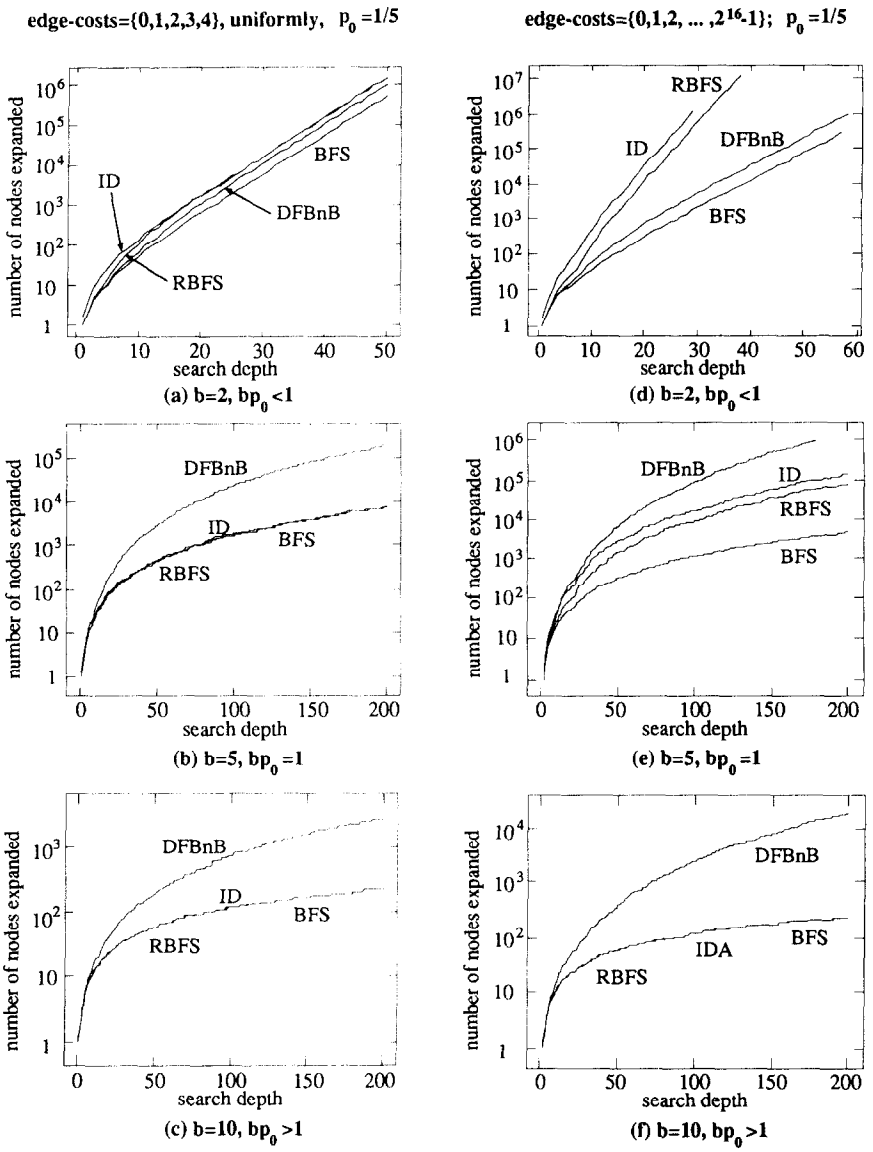

Fig. 17. Average number of nodes expanded on random trees.

5.1.2. *Running times*

We now consider how BFS compares to linear-space algorithms when running time is the main performance measure, assuming that space is not a constraint. Although BFS is optimal in terms of the number of node expansions, that does not mean that it runs faster than the linear-space algorithms.

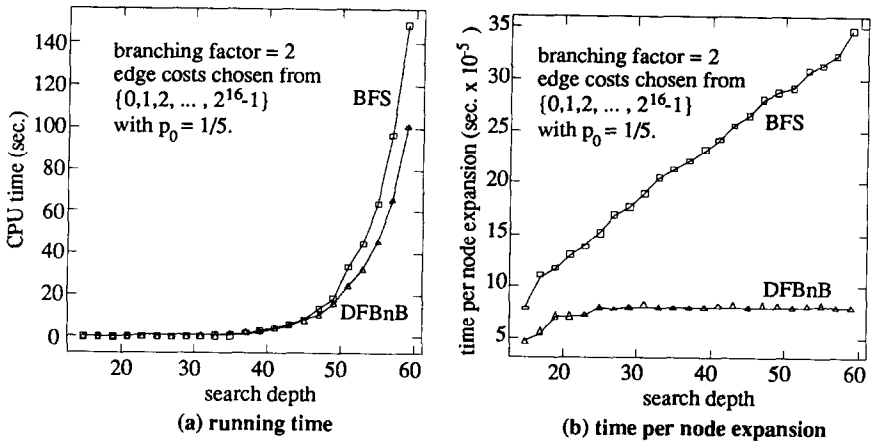

Fig. 18. Running time and time per node expansion on a binary tree.

To expand nodes in best-first order, BFS has to maintain a priority queue to store all the nodes that are generated but not yet expanded. The time to access the priority queue, selecting a node from the queue and inserting new nodes into the queue, depends on the total number of nodes stored, and increases as more nodes are added. If a heap is used, which is an optimal implementation of a priority queue, access time is logarithmic in the total number of nodes in the heap. For a difficult problem that requires node generations exponential in the search depth, $O(\beta^d)$ for some constant $\beta > 1$, the heap access time becomes $\log(\beta^d) = O(d)$. This means that BFS takes time linear in the search depth to expand a node for difficult problems. All linear-space algorithms, on the other hand, can be implemented on a stack; each operation is executed on the top of the stack, and takes constant time.

Fig. 18(a) shows one example on a binary random tree where the running time of BFS increases faster than that of DFBnB. The zero edge costs were chosen with probability $p_0 = 1/5$, and nonzero edge costs were uniformly chosen from $\{1, 2, 3, \ldots, 2^{16} - 1\}$. Fig. 18(b) illustrates the corresponding average time per node expansion for both BFS and DFBnB in this case, which confirms our analysis.

Overall, whether or not BFS runs faster than a linear-space algorithm depends on the problem complexity and its size, which determine the number of nodes generated, and the time for expanding a node and generating all its children.

5.2. *Comparison on actual problems*

5.2.1. *Lookahead search on sliding-tile puzzles*

Consider fixed-depth lookahead search on a sliding-tile puzzle, which searches from the current state to a fixed depth, and returns a node at the given depth whose cost is a minimum among all nodes at that depth. We compared DFBnB, iterative-deepening and RBFS on lookahead search. In our experiments, the cost function used is $f(n) = g(n) + h(n)$, where $g(n)$ is the total number of moves made from the initial state

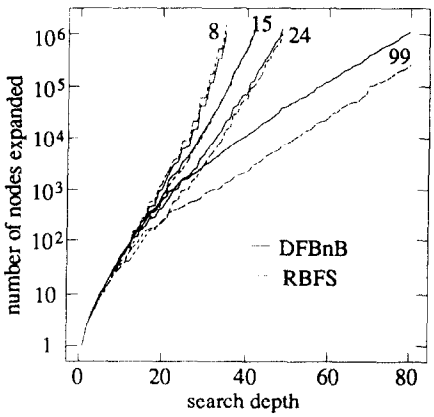

Fig. 19. Lookahead search on sliding-tile puzzles.

to node n, and $h(n)$ is the Manhattan distance from node n to the goal state. Our experiments show that iterative-deepening expands slightly more nodes than RBFS for lookahead search, as expected. Fig. 19 compares DFBnB and RBFS. The horizontal axis is the lookahead depth, and the vertical axis is the number of nodes expanded. The results were averaged over 200 initial states. The curves labeled by 8, 15, 24 and 99 are the results on Eight, Fifteen, Twenty-four and Ninety-nine puzzles, respectively. The results show that DFBnB performs slightly better than RBFS on small puzzles, while RBFS is superior to DFBnB on large ones.

Unfortunately, the state space of sliding-tile puzzles is not a tree, but a graph with cycles. In addition, a shortest complete solution path in such a graph is unknown in advance. Thus, DFBnB cannot be applied in that setting, since it may keep to expand the nodes on a cycle, and does not terminate. Furthermore, although RBFS generates insignificantly fewer nodes than iterative-deepening, RBFS has slightly higher overhead on running time than iterative-deepening. Consequently, iterative-deepening is still the algorithm of choice for finding an optimal solution path to the problem.

5.2.2. The Asymmetric TSP

We ran DFBnB, iterative-deepening and RBFS on the ATSP with intercity costs uniformly chosen from $\{0, 1, 2, 3, \ldots, r\}$, where r is an integer. Figs. 20(a) and 20(b) show our results on 100-city and 300-city ATSPs, averaged over 500 trials each. The horizontal axes are the number of distinct intercity costs r, and the vertical axes are the numbers of nodes generated. When the number of distinct costs r is small or large, relative to the number of cities n, the ATSP is easy or difficult, respectively, following the discussion in Section 4.2. Iterative-deepening cannot compete with RBFS and DFBnB, especially for difficult ATSPs when r is large. RBFS is worse than DFBnB on easy ATSPs, indicating that the overhead of RBFS is larger than that of DFBnB. RBFS does poorly relative to DFBnB on difficult ATSPs as well, since in this case the node costs in the search tree are unique, which causes significant node regeneration overhead.

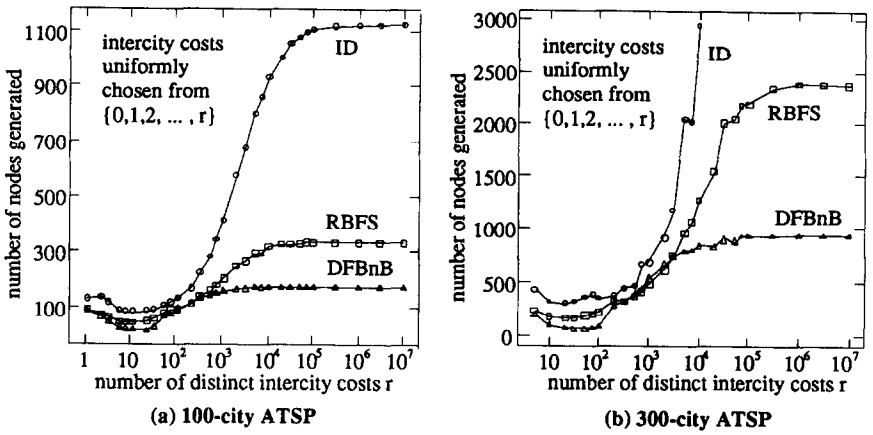

Fig. 20. Performance of linear-space algorithms on the ATSP.

5.3. Summary

Even though linear-space search algorithms expand more nodes than BFS, by exploring nodes that are not generated by BFS, or re-generating a node multiple times, the linear-space algorithms may run faster than BFS. This is typically the case when the problem space is large, and the time for node expansion is relatively small. In addition, the linear-space algorithms take much less space than BFS. For large problems, linear-space methods are often the algorithms of choice.

Among the linear-space algorithms, DFBnB is inapplicable to problems whose state spaces are graphs with cycles, since no cutoff bounds are known in advance. DFBnB is preferable on problems whose search spaces are bounded-depth trees and require exponential computation. RBFS should be applied to problems that cannot be represented by bounded-depth trees, or problems that can be solved in polynomial time.

6. Related work

6.1. Analytic models

Most existing models for analyzing search algorithms are trees. There are at least three different tree models in the literature. The first model assumes a uniform branching factor, with one or more solutions that lie exactly at depth d [13,27,49]. Constraint-satisfaction problems can also be formulated by this model [42].

The second model is a uniform tree with a unique goal node at a given depth, and a node cost function [6, 12, 18, 39, 41]. This cost function distinguishes this model from the first one. Node costs are assumed to be independent and identically distributed random variables. The most important analytic result on this model is that the expected complexity of the A* algorithm [16], a best-first search with $f(n) = g(n) + h(n)$, is exponential in most cases [18,39].

The third model is a random tree with independent edge costs, the cost of a node as the sum of the edge costs on its path to the root, and an optimal goal node being a minimum-cost leaf node at a fixed depth [20, 32, 33, 53, 57, 58], or at variant depths [48]. The idea of assigning costs to edges can be traced back to the work of Fuller et al. [10] on game-tree search. Two variations of this model are used in the literature, one with a uniform branching factor [20, 53, 57], and the other with a random branching factor [32, 33, 48, 58]. This model is typically suitable for combinatorial optimization problems, such as the Traveling Salesman Problem [29], in which an objective function is to be minimized. We adopt this model with random branching factor but fixed goal depth, which we call an incremental random tree.

6.2. *Analyses of branch-and-bound*

BFS and DFBnB can be considered as special cases of branch-and-bound (BnB) [19, 25, 30], a general technique for problem solving. Kumar et al. [26] showed that many graph-search algorithms developed in artificial intelligence can be formulated as BnB. For example, the A* algorithm [16] is a BFS algorithm.

Smith [48] performed an average-case complexity analysis of BnB. The model he used is a random tree with leaf nodes at different depths as goal nodes, and he derived equations for the average-case complexity. No closed-form solutions to these equations were obtained, however, and thus no direct conclusion is drawn on the average-case complexity of BnB.

Another average-case analysis was conducted by Wah and Yu [53], using a random tree with uniform branching factor. The process of BFS is modeled 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. A stochastic process is employed to derive approximate formulas for the average number of nodes expanded. Empirical results verified that the average-case complexity of BFS is exponential when the edge costs have the gamma distribution and the binomial distribution. They also compared DFBnB with BFS, arguing that the former is comparable to the latter when the lower-bound cost function used is very accurate or very inaccurate.

Karp and Pearl [20] delineated polynomial and exponential average-case complexities of BFS on a random binary tree with edge costs of 1 or 0 with probabilities p or $1-p$. The basic mathematical tools used are branching processes [15]. They elegantly showed that the cost of an optimal goal is almost certain to remain bounded when $p < 0.5$, very likely to be near $\log\log d$ when $p = 0.5$, and almost surely proportional to d when $p > 0.5$. They further proved that the average number of nodes expanded by BFS is linear when $p < 0.5$, quadratic when $p = 0.5$, and exponential when $p > 0.5$. These results were also suggested earlier by Hammersley [14].

McDiarmid and Provan [32, 33] significantly extended Karp and Pearl's work to random trees with arbitrary edge-cost distributions, and random branching factors. They proved that the goal cost almost surely grows linearly in depth d when $bp_0 < 1$, approaches $\log\log d$ when $bp_0 = 1$, and remains bounded when $bp_0 > 1$. They showed that BFS finds an optimal goal node in exponential average time when $bp_0 < 1$, runs in quadratic average time when $bp_0 = 1$, and finishes in linear average time when $bp_0 > 1$.

The results of Karp and Pearl, and of McDiarmid and Provan, form the basis of this research. In fact, the properties of the optimal goal cost (Lemma 2.2) and the average-case complexities of BFS (Lemma 2.5) of this paper are from McDiarmid and Provan's results.

6.3. Analyses of iterative-deepening

Patrick et al. [38] first showed that the worst-case of IDA*, which is iterative-deepening using the A* cost function $f(n) = g(n) + h(n)$, occurs when all nodes in the search tree have unique costs, expanding $(N^2+N)/2$ nodes, where N is the number of nodes that are surely expanded by A*. Mahanti et al. [31] discussed the worst-case performance of IDA* on trees and graphs. Vempaty et al. [51] compared DFBnB with IDA*. They argued that DFBnB is preferable when the solution density is high, whereas IDA* should be used when the heuristic branching factor is high.

Patrick [37] also considered the average-case complexity of IDA* using a random tree with uniform branching factor and integer edge costs. However, in computing the expected number of nodes with a particular cost k, Patrick did not take into account the dependence among nodes that share common edges on their paths to the root [37, Chapter 6, page 63]. In other words, the dependent model was treated as an independent one.

6.4. Algorithms using limited space

BFS and the linear-space algorithms stand on opposite ends of the space spectrum. BFS is on the exponential-space end, and the others are on the linear-space end. In between these two extreme cases are algorithms that use as much space as is available on a machine, but no more.

A general method to make use of more than linear space is to combine BFS and a linear-space algorithm. Pearl presented three possible BFS–DFBnB combinations to meet a limited memory requirement [39]. Both the MREC algorithm [47] and the MA* algorithm [3] are combinations of BFS and iterative-deepening. Unfortunately, these two algorithms run slower than iterative-deepening on the Fifteen Puzzle because of memory maintenance overhead, while generating fewer nodes than iterative-deepening [24]. One algorithm that is close to RBFS is iterative expansion [45]. Unlike RBFS, however, iterative expansion does not expand new nodes in best-first order when node costs are not monotonic.

Several algorithms have been developed to reduce the node regeneration overhead of iterative-deepening, including DFS* [51], IDA*-CR [46] and MIDA* [52]. They all try to reduce the number of iterations in iterative-deepening by setting successive thresholds to values larger than the minimum value that exceeded the previous threshold. In order to guarantee finding an optimal goal node, once a goal is found, the algorithms revert to DFBnB to complete the final iteration.

On problems whose state spaces are not trees, the most serious problem with algorithms that use less than exponential space is how to detect or prevent generating

duplicate nodes. Taylor and Korf [50] proposed one scheme of duplicate node pruning using limited memory. The idea is to first learn some of the structure of the state space of a given problem using a small exploratory search, and then to encode that structure in a finite automata, which is used during the problem-solving search to avoid duplicate node expansions.

6.5. *Complexity transitions*

The earliest evidence of complexity transitions was Karp and Pearl's average-case complexity of BFS [20]. Huberman and Hogg [17] discussed complexity transitions in some intelligent systems. Cheeseman et al. [4] empirically showed that complexity transitions exist in many NP-hard problems, including Hamiltonian circuit, constraint-satisfaction problems, graph coloring, and Symmetric Traveling Salesman Problem. The complexity transitions in constraint-satisfaction problems have attracted the attention of many researchers [5, 28, 34, 54].

Most recently, Zhang and Pemberton [56, 59, 60] proposed a method of exploiting complexity transitions to find approximate and optimal solutions to combinatorial optimization problems. Their method allows BFS or DFBnB to find either high quality approximate solutions quickly, or to obtain better solutions sooner than truncated DFBnB [55]. This method is effective on a problem whose state space has a large number of distinct edge costs. On the Asymmetric Traveling Salesman Problem, DFBnB using this method runs faster and finds better solutions than a local search method.

7. Conclusions

We studied search algorithms that use space linear in the search depth, including depth-first branch-and-bound (DFBnB), iterative-deepening (ID) and recursive best-first search (RBFS). Due to their linear-space requirement, all these algorithms expand more nodes than best-first search (BFS), which typically uses space that is exponential in the search depth.

Using a random tree model, we analytically proved that DFBnB expands at most $O(d \cdot N)$ nodes on average, where d is the goal depth and N is the expected number of nodes expanded by BFS. We further showed that DFBnB is asymptotically optimal when BFS runs in exponential time. We also proved that iterative-deepening and RBFS are asymptotically optimal on a random tree with integer edge costs. Overall, the average number of nodes expanded by these linear-space algorithms is exponential in the tree depth when the average number of same-cost children is less than one, and is at most $O(d^4)$ when the average number of same-cost children is greater than or equal to one. Our analytic results successfully explain a previously observed anomaly in the performance of DFBnB, and predict the existence of a complexity transition of search algorithms on the Asymmetric Traveling Salesman Problem. In addition, we studied the heuristic branching factor of a random tree, and the effective branching factor of BFS, DFBnB, iterative-deepening, and RBFS on a tree.

BFS cannot compete with linear-space algorithms on large problems, because of its exponential space requirement. It may even run slower than a linear-space algorithm. Therefore, linear-space algorithms are usually the only algorithms of choice for optimally solving large and difficult problems. Among the linear-space algorithms, DFBnB is inapplicable to problems that cannot be formulated as a tree with a bounded depth, or a state-space graph with known cutoff depth. DFBnB is the best, however, on a problem that requires exponential computation, and whose state space can be represented by a bounded-depth tree. RBFS should be adopted on a problem that cannot be represented by a bounded-depth tree, or a problem that can be solved in polynomial time.

Acknowledgements

We would like to thank Mark Cassorla, Peter Cheeseman, Sheila Greibach, Lars Hagen, Tad Hogg, Judea Pearl, Joe Pemberton, Curt Powley, Greg Provan, Roberto Schonmann, Sek-Wah Tan, and Colin Williams for helpful discussions related to this research. We also want to thank two anonymous reviewers whose comments greatly improved the clarity of this paper, and the anonymous reviewers of our previous papers [57, 58], on which this paper is based.

Appendix A. Search algorithms

This appendix contains pseudo-code descriptions of best-first search (BFS), depth-first branch-and-bound (DFBnB), iterative-deepening (ID), and recursive best-first search (RBFS).

In the following descriptions, *root* represents the root node or initial state of a state-space tree, and $cost(n)$ is the cost of a node n.

A.1. Best-first search

The BFS algorithm for tree search is given in Fig. A.1, where the list *open* is used to maintain current frontier nodes. The algorithm starts with BFS(*root*).

```
BFS(root)
  open ← ∅; n ← root
  WHILE (n is not a goal node)
    EXPAND n, generating and evaluating all its children
    INSERT all its children into open
    DELETE n from open
    n ← a minimum-cost node in open
```

Fig. A.1. Best-first search algorithm.

```
DFBnB(n)
  GENERATE all k children of n: n_1, n_2, ..., 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 ← cost(n_i)
      ELSE DFBnB(n_i)
    ELSE RETURN
  RETURN
```

Fig. A.2. Depth-first branch-and-bound algorithm.

```
ID(root)
  threshold ← cost(root)
  next_threshold ← ∞
  REPEAT
    DFS(root)
    threshold ← next_threshold
    next_threshold ← ∞

DFS(n)
  FOR (each child n_i of n)
    IF (n_i is a goal node and cost(n_i) ⩽ threshold)
      EXIT with optimal goal node n_i
    IF (cost(n_i) ⩽ threshold) DFS(n_i)
    ELSE IF (cost(n_i) < next_threshold)
            next_threshold ← cost(n_i)
  RETURN
```

Fig. A.3. Iterative-deepening algorithm.

A.2. *Depth-first branch-and-bound*

Fig. A.2 is a recursive version of DFBnB using node ordering. The top-level call is made on the root node, DFBnB(*root*), with initial upper bound $u = \infty$.

A.3. *Iterative-deepening*

The iterative-deepening (ID) algorithm is in Fig. A.3, and starts with ID(*root*). It repeatedly calls a 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, DFS(n), does not use node ordering and is implemented recursively.

```
RBFS(n, F[n], u)
  IF (cost(n) > u) RETURN cost(n)
  IF (n is a goal) EXIT with optimal goal node n
  IF (n has no children) RETURN ∞
  FOR (each child n_i of n)
    IF (cost(n) < F[n]) F[i] ← MAX(F[n], cost(n_i))
    ELSE F[i] ← cost(n_i)
  SORT n_i and F[i] in increasing order of F[i]
  IF (only one child) F[2] ← ∞
  WHILE (F[1] ≤ u and F[1] < ∞)
    F[1] ← RBFS(n_1, F[1], MIN(u, F[2]))
    INSERT n_1 and F[1] in sorted order
  RETURN F[1]
```

Fig. A.4. Recursive best-first search algorithm.

A.4. Recursive best-first search

The RBFS algorithm is given in Fig. A.4, where $F(n)$ is the stored value of node n, and u is a local upper bound. The initial call is RBFS(*root*, *cost*(*root*),∞).

Appendix B. Proofs

This appendix contains proofs to the lemmas and theorems in Section 2.

Lemma 2.1. *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.*

Proof. We prove the lower bound by showing that any algorithm that is guaranteed to find an optimal goal node must expand at least those nodes whose costs are less than the optimal goal cost. This is done by contradiction, and is based on [7]. Assume the converse, namely, that there is an algorithm $\mathcal{A}$ that is guaranteed to find an optimal goal node on a tree with monotonic node costs, and that skips a node that has cost less than the optimal goal cost. Let T be a tree with monotonic node costs. Suppose that when $\mathcal{A}$ is applied to T, it does not expand a node n of cost $f(n)$ that is strictly less than the optimal goal cost. We now create another tree T' that is the same as T except that node n in T' has only one child that is also the only optimal goal node of T' with cost $f(n)$. Note that the node costs are monotonic in T'. Since algorithm $\mathcal{A}$ skips node n when it is applied to T, $\mathcal{A}$ must skip node n as well when it is applied to the new tree T'. Consequently, algorithm $\mathcal{A}$ cannot find the only optimal goal node of T', contradicting the optimality assumption for algorithm $\mathcal{A}$.

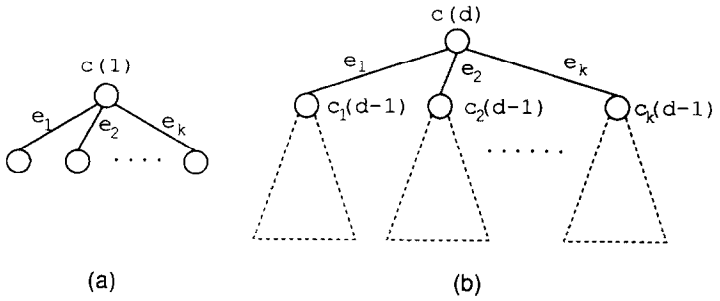

Fig. B.1. Structure of a random tree for proving Lemma 2.3.

We prove the upper bound by showing that there is an algorithm that finds an optimal goal node and expands at most those nodes whose costs are less than or equal to the optimal goal cost. Best-first search is such an algorithm. By the monotonicity of node costs, and the best-first node-selection strategy it uses, BFS always expands a node with cost x before any node with cost greater than x is chosen for expansion. Thus, BFS will not expand any node whose cost is greater than the optimal goal cost. □

Lemma 2.3. *On a random tree $T(b,d)$ with $bp_0 > 1$, as $d \to \infty$, the expected cost of optimal goal nodes decreases when bp_0 increases for a given b, or a given $p_0 > 0$. In particular, the expected goal cost approaches zero when $p_0 \to 1$ for a given b, or when $b \to \infty$ for a given $p_0 > 0$.*

Proof. Let $c(d)$ be the expected optimal goal cost of a random tree $T(b,d)$, as shown in Fig. B.1. We prove the lemma by induction on the depth d. Consider a random tree of depth one, as in Fig. B.1(a). $c(1)$ is the expected cost of the minimum of k edge costs, where k is a random variable with mean b, i.e.

$$c(1) = E[\min\{e_1, e_2, \ldots, e_k\}]. \tag{B.1}$$

It is evident that $c(1)$ decreases as k increases, since the minimum is taken over more independent and identically distributed random variables. Increasing b causes k to increase. Notice that $bp_0 > 1$ implies $p_0 > 0$. Thus, when b increases to infinity, $c(1)$ approaches zero, since each e_i has a nonzero probability of being zero.

As the inductive step, we assume that $c(d-1)$ decreases and approaches zero as b increases with a fixed $p_0 > 0$. Let $c_i(d-1)$ be the optimal goal cost of the subtree rooted at the ith child node of the root node of a random tree $T(b,d)$, as shown in Fig. B.1(b). Then $c(d)$ is computed as

$$c(d) = E[\min\{e_1 + c_1(d-1), e_2 + c_2(d-1), \ldots, e_k + c_k(d-1)\}]. \tag{B.2}$$

When b increases, the random variable $e_i + c_i(d-1)$ decreases and reaches e_i as $b \to \infty$ for $i = 1, 2, \ldots, k$, following the inductive assumption. Thus, when b increases, $c(d)$ decreases, since it is the minimum of more random variables, and each of them decreases as b increases. Similarly, variable $e_i + c_i(d-1)$ has a nonzero probability of

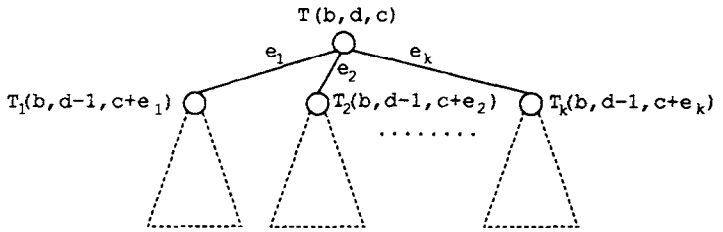

Fig. B.2. Structure of a random tree for proving Theorem 2.6.

being zero when $b \to \infty$, because $p_0 > 0$ and $c_i(d-1) \to 0$ in this case. Consequently, $c(d) \to 0$ when $b \to \infty$, which concludes our claim for the case when $b \to \infty$.

The fact that the optimal goal cost decreases and reaches zero when $p_0 \to 1$ with a given b can be proved similarly. □

Lemma 2.4. *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.*

Proof. This is a corollary of Lemma 2.1. BFS must expand all nodes with costs less than the optimal goal cost. Because of its best-first node-selection strategy, BFS never expands a node with cost greater than the optimal goal cost. □

Theorem 2.6. *Let $N_B(b,d)$ be the expected number of nodes expanded by best-first search, and $N_D(b,d)$ the expected number of nodes expanded by depth-first branch-and-bound, on a random tree $T(b,d)$. As $d \to \infty$,*

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

Proof. For convenience of discussion, denote a random tree with root node cost c as $T(b,d,c)$. By this notation, a random tree $T(b,d)$ with root cost zero is $T(b,d,0)$. For the same reason, let $N_D(b,d,c,u)$ be the expected number of nodes expanded by DFBnB on $T(b,d,c)$ with initial upper bound u.

As shown in Fig. B.2, the root of $T(b,d,c)$ has k children, $n_1, n_2, \ldots, n_k$, where k is a random variable with mean b. Let e_i be the edge cost from the root of $T(b,d,c)$ to its ith child, the root of its ith subtree $T_i(b,d-1,c+e_i)$, for $i = 1,2,\ldots,k$. The children of the root are generated all at once and sorted in nondecreasing order of their costs. Thus $e_1 \leqslant e_2 \leqslant \ldots \leqslant e_k$, arranged from left to right in Fig. B.2. We first make the following two observations.

First, subtracting the root cost from all nodes and the upper bound has no affect on the search. Therefore, the number of nodes expanded by DFBnB on $T(b,d,c)$ with initial upper bound u is equal to those expanded on $T(b,d,0)$ with initial upper bound $u-c$. That is

$$\begin{cases} N_D(b,d,c,u) = N_D(b,d,0,u-c), \\ N_D(b,d,c,\infty) = N_D(b,d,0,\infty). \end{cases} \tag{B.3}$$

Secondly, because a larger initial upper bound causes at least as many nodes to be expanded as a smaller upper bound, the number of nodes expanded by DFBnB on $T(b,d,c)$ with initial upper bound u is no less than the number expanded with initial upper bound $u' \leqslant u$. That is

$$N_D(b,d,c,u') \leqslant N_D(b,d,c,u), \quad \text{for } u' \leqslant u. \tag{B.4}$$

Now consider DFBnB on $T(b,d,0)$. It first searches the subtree $T_1(b,d-1,e_1)$ (see Fig. B.2), expanding $N_D(b,d-1,e_1,\infty)$ expected number of nodes. Let ρ be the minimum goal cost of $T_1(b,d-1,0)$. Then the minimum goal cost of $T_1(b,d-1,e_1)$ is $\rho+e_1$, which is the upper bound after searching $T_1(b,d-1,e_1)$. After the subtree $T_1(b,d-1,e_1)$ is searched, subtree $T_2(b,d-1,e_2)$ will be explored if its root cost e_2 is less than the current upper bound $\rho+e_1$, and the expected number of nodes expanded is $N_D(b,d-1,e_2,\rho+e_1)$. $N_D(b,d-1,e_2,\rho+e_1)$ is also an upper bound on the expected number of nodes expanded in $T_i(b,d-1,e_i)$, for $i=3,4,\ldots,k$. This is because the upper bound can only decrease after searching $T_2(b,d-1,e_2)$ and the edge cost e_i can only increase as i increases, both of which cause fewer nodes to be expanded. Since the root of $T(b,d,0)$ has b expected children, each of which is independently generated, we write

$$\begin{aligned} &N_D(b,d,0,\infty) \\ &\quad \leqslant N_D(b,d-1,e_1,\infty) + (b-1)N_D(b,d-1,e_2,\rho+e_1) + 1, \end{aligned} \tag{B.5}$$

where the 1 is for the expansion of the root of $T(b,d,0)$. By (B.3) and (B.5), we have

$$\begin{aligned} &N_D(b,d,0,\infty) \\ &\quad \leqslant N_D(b,d-1,0,\infty) + (b-1)N_D(b,d-1,0,\rho+e_1-e_2) + 1. \end{aligned} \tag{B.6}$$

Since $\rho+e_1-e_2 \leqslant \rho$ for $e_1 \leqslant e_2$, by (B.4), we rewrite (B.6) as

$$\begin{aligned} &N_D(b,d,0,\infty) \\ &\quad \leqslant N_D(b,d-1,0,\infty) + (b-1)N_D(b,d-1,0,\rho) + 1. \end{aligned} \tag{B.7}$$

Now consider $N_D(b,d-1,0,\rho)$, the expected number of nodes expanded by DFBnB on $T(b,d-1,0)$ with initial upper bound ρ. If $T(b,d-1,0)$ is searched by BFS, it will return the optimal goal node and its expected cost ρ, and expand $N_B(b,d-1)$ nodes on average. When $T(b,d-1,0)$ is searched by DFBnB with upper bound ρ, only those nodes whose costs are strictly less than ρ will be expanded, and these nodes must also be expanded by BFS. We thus have

$$N_D(b,d-1,0,\rho) < N_B(b,d-1). \tag{B.8}$$

Substituting (B.8) into (B.7), we then write

$$
\begin{aligned}
&N_D(b,d,0,\infty)\\
&\quad < N_D(b,d-1,0,\infty) + (b-1)N_B(b,d-1) + 1\\
&\quad < N_D(b,d-2,0,\infty) + (b-1)(N_B(b,d-1) + N_B(b,d-2)) + 2\\
&\quad < \cdots\\
&\quad < N_D(b,0,0,\infty) + (b-1)\sum_{i=1}^{d-1} N_B(b,i) + d. \qquad \text{(B.9)}
\end{aligned}
$$

This proves the lemma since $N_D(b,0,0,\infty) = 0$. □

Corollary 2.7. *On a random tree $T(b,d)$, $N_D(b,d) < O(d \cdot N_B(b,d-1))$, where $N_B(b,d)$ and $N_D(b,d)$ are the expected numbers of nodes expanded by best-first search and depth-first branch-and-bound, respectively.*

Proof. It directly follows Theorem 2.6 and the fact that

$$\sum_{i=1}^{d-1} N_B(b,i) < (d-1)N_B(b,d-1), \qquad \text{(B.10)}$$

since $N_B(b,i) < N_B(b,d-1)$ for all $i < d-1$. □

Theorem 2.8. *The expected number of nodes expanded by depth-first branch-and-bound for finding an optimal goal node of a random tree $T(b,d)$, as $d \to \infty$, is*

(1) $\theta(\beta^d)$ *when $bp_0 < 1$, meaning that depth-first branch-and-bound is asymptotically optimal, where β is the same constant as in Lemma 2.5,*
(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.

Proof. It is evident that $N_B(b,i) < N_B(b, \lfloor d/2 \rfloor + i)$. This allows us to use the asymptotic expected complexity of BFS as $d \to \infty$. By Lemma 2.5 and Theorem 2.6, when $bp_0 < 1$,

$$
\begin{aligned}
N_D(b,d) &< (b-1)\sum_{i=0}^{d-1} N_B(b,i) + d\\
&< 2(b-1)\sum_{i=\lfloor d/2 \rfloor}^{d-1} N_B(b,i) + d\\
&= 2(b-1)\sum_{i=\lfloor d/2 \rfloor}^{d-1} \theta(\beta^i) + d\\
&= \theta(\beta^d).
\end{aligned}
$$

The other two cases directly follow from Lemma 2.5 and Theorem 2.6. □

Corollary 2.9. *On a random tree $T(b,d)$ with $bp_0 > 1$, as $d \to \infty$, the expected number of nodes expanded by depth-first branch-and-bound approaches $O(d)$ when $p_0 \to 1$ for a given b, or when $b \to \infty$ for a given $p_0 > 0$.*

Proof. When $b \to \infty$ for a given edge-cost distribution, or when $p_0 \to 1$ for a given b, the optimal goal cost C^* approaches zero, according to Lemma 2.3. Thus, the upper bound after searching $T_1(b, d-1, e_1)$ approaches e_1 (see Fig. B.2). Since the root of subtree $T_i(b, d-1, e_i)$ has cost e_i, which is no less than the current upper bound e_1, no nodes in $T_i(b, d-1, e_i)$ will be expanded, for $i = 2, 3, \ldots, k$. Then, $N_D(b, d-1, e_2, e_1) = 0$, and (B.7) becomes

$$N_D(b,d,0,\infty) = N_D(b,d-1,0,\infty) + 1, \tag{B.11}$$

where the 1 is for the expansion of the root of $T(b,d,0)$, which leads to

$$\begin{aligned} N_D(b,d,0,\infty) &= N_D(b,d-2,0,\infty) + 2 \\ &= N_D(b,d-3,0,\infty) + 3 \\ &= \cdots \\ &= d. \end{aligned}$$

□

Theorem 2.10. *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 $d \to \infty$, which is asymptotically optimal.*

Proof. When edge costs are chosen from a continuous distribution, all node costs are unique with probability one. Thus, each successive iteration expands one new node, and the ith iteration expands i nodes on average. Consequently, the expected number of iterations is $N_B(b,d)$. Hence the expected number of nodes expanded by iterative-deepening is

$$N_{ID}(b,d) = \sum_{i=1}^{N_B(b,d)} i = O((N_B(b,d))^2).$$

Now consider lattice edge costs chosen from the set $\{a_0\Delta, a_1\Delta, a_2\Delta, \ldots, a_m\Delta\}$, where $\Delta > 0$ is a constant and $a_0 < a_1 < a_2 < \cdots < a_m$ are nonnegative relatively prime integers. To include the situation where edges may have zero cost, we set $a_0 = 0$. It is clear that all node costs are multiples of Δ, and so is the difference of two distinct node costs. This means that the cost threshold increases by at least Δ, which is a constant independent of d, from one iteration to the next.

The last iteration has a cost threshold equal to the optimal goal cost C^* of $T(b,d)$, and expands all nodes whose costs are less than C^* and some nodes whose costs are

equal to C^*. In other words, the expected number of nodes expanded by the last iteration is of the same order as the expected complexity of best-first search. In addition, each iteration before the last one expands fewer nodes than the last iteration.

When $bp_0 > 1$, the optimal goal cost almost surely remains bounded by a constant by Lemma 2.2, and thus the maximum cost threshold will not exceed that constant. Therefore, the total number of iterations is also a constant, since the cost threshold increases by at least a constant Δ from one iteration to the next. A constant number of iterations only contributes a constant multiplicative overhead to the total number of nodes expanded. Hence iterative-deepening is asymptotically optimal in this case.

The number of iterations is not a constant when $bp_0 \leqslant 1$, since the optimal goal cost increases with the depth d. First consider the case when $bp_0 < 1$. The optimal goal cost $C^*(d)$ of $T(b,d)$ grows linearly with d, and the expected number of nodes with costs less than $C^*(d)$ is exponential in d in this case. Intuitively, the expected number of nodes whose costs are less than a particular cost seems to be exponential in that cost, so that the optimality of iterative-deepening should follow. This suggests the following proof.

Consider the iterations that iterative-deepening has to execute to find the optimal goal cost $C^*(d)$ of $T(b,d)$ after obtaining the optimal goal cost $C^*(d-1)$ of a shallower tree $T(b,d-1)$ of $T(b,d)$. For simplicity, name the set of these iterations $\mathcal{I}(d)$, which explore new nodes whose costs are greater than $C^*(d-1)$ and less than or equal to $C^*(d)$. Each iteration of $\mathcal{I}(d)$ increases the cost threshold by at least Δ. It is evident that $C^*(d) \leqslant C^*(d-1) + a_m\Delta$, where $a_m\Delta$ is the largest edge cost and a constant independent of d. This means that there are a constant number of iterations in $\mathcal{I}(d)$, which is at most a_m. The last iteration of $\mathcal{I}(d)$ expands $\theta(\beta^d)$ expected number of nodes as $d \to \infty$, where $\beta > 1$ is a constant, following Lemma 2.5. In addition, the last iteration expands more nodes than any other iterations of $\mathcal{I}(d)$. Therefore, the total expected number of nodes expanded by all iterations of $\mathcal{I}(d)$ is $\theta(\beta^d)$. Furthermore, iterative-deepening executes a sequence of sets of iterations $\mathcal{I}(i)$, for $i = 0, 1, 2, \ldots, d$. Notice that the number of nodes expanded by the iterations in $\mathcal{I}(i)$ is less than the number of nodes expanded by the iterations in $\mathcal{I}(\lfloor d/2 \rfloor + i)$. This allows us to use the asymptotic expected number of nodes expanded by BFS as $d \to \infty$. By definition, $\theta(\beta^i) \leqslant K \cdot \beta^i$ for some $K > 0$ and large i. Thus, the total expected number of nodes expanded to find an optimal goal node of $T(b,d)$ is

$$
\begin{aligned}
N_{\mathrm{ID}}(b,d) &< 2 \sum_{i=\lfloor d/2 \rfloor}^{d} \theta(\beta^i) \leqslant 2K \sum_{i=\lfloor d/2 \rfloor}^{d} \beta^i \\
&< 2K \sum_{i=0}^{d} \beta^i = 2K \frac{\beta^{d+1} - 1}{\beta - 1} \\
&< 2K \frac{\beta}{\beta - 1} \beta^d,
\end{aligned}
$$

which is $O(\beta^d)$. Hence iterative-deepening is asymptotically optimal when $bp_0 < 1$.

The above proof also indicates that the total number of iterations that iterative-deepening executes is linear in the search depth d when $bp_0 < 1$. This directly follows the fact that the total number of iterations that the algorithm has to execute to find $C^*(d)$ after obtained $C^*(d-1)$ is a constant.

When $bp_0 = 1$, best-first search expands $\theta(d^2)$ expected number of nodes in a random tree $T(b,d)$ by Lemma 2.5, which is also the asymptotic order of the expected number of nodes expanded by the last iteration of iterative-deepening. The optimal goal cost $C^*(d)$ of $T(b,d)$ satisfies $\lim_{d\to\infty} C^*(d)/\log\log d = 1$ by Lemma 2.2, which is equivalent to

$$C^*(d) = \log\log d + \phi(\log\log d), \tag{B.12}$$

as $d \to \infty$, for some function $\phi(\log\log d) \in o(\log\log d)$ [3]. Notice that $C^*(d) < 2\log\log d$, because $\phi(\log\log d) < \log\log d$, as $d \to \infty$. Therefore, the total number of iterations of iterative-deepening for searching $T(b,d)$ is $I \leqslant C^*(d)/\Delta = O(\log\log d)$, where the constant Δ is the minimum increment of the cost threshold from one iteration to the next.

Consider a shallower tree $T(b, \lceil \sigma d \rceil)$ of $T(b,d)$, for some constant $0 < \sigma < 1$. For the same reason as above, the optimal goal cost $C^*(\lceil \sigma d \rceil)$ of $T(b, \lceil \sigma d \rceil)$ satisfies

$$C^*(\lceil \sigma d \rceil) = \log\log\lceil \sigma d \rceil + \phi(\log\log\lceil \sigma d \rceil), \tag{B.13}$$

as $d \to \infty$. Function ϕ in (B.13) is the same function ϕ in (B.12), since $T(b, \lceil \sigma d \rceil)$ is within $T(b,d)$ and these problems are structurally the same. In the following, we show that the first time that nodes with costs in the interval $[C^*(\lceil \sigma d \rceil), C^*(d)]$ are chosen for expansion, it is during the same iteration, which also implies that the penultimate iteration cannot reach depth $\lceil \sigma d \rceil$. Following this result and Lemma 2.5, the last iteration, iteration I, expands $\theta(d^2)$ expected number of nodes, and the penultimate iteration, iteration $I-1$, expands *less than* $\theta(\lceil \sigma d \rceil^2)$ expected number of nodes. For the same reason, iteration $I-2$ expands less than $\theta(\lceil \sigma^2 d \rceil^2)$ expected number of nodes, etc. By definition, $\theta(d^2) < K \cdot d^2$ for some $K > 0$ as $d \to \infty$, and $\lceil x \rceil < x + 1$. Thus, the total expected number of nodes expanded is

$$\begin{aligned} N_{\mathrm{ID}}(b,d) &< \sum_{i=0}^{I-1} \theta(\lceil \sigma^i d \rceil^2) < K \sum_{i=0}^{I-1} (\sigma^i d + 1)^2 \\ &= K \left(\sum_{i=0}^{I-1} \sigma^{2i} d^2 + \sum_{i=0}^{I-1} 2\sigma^i d + \sum_{i=0}^{I-1} 1 \right) \\ &< K \left(d^2 \sum_{i=0}^{\infty} \sigma^{2i} + 2d \sum_{i=0}^{\infty} \sigma^i + I \right) = K \left(\frac{d^2}{1-\sigma^2} + \frac{2d}{1-\sigma} + I \right), \end{aligned}$$

which is $\theta(d^2)$, since σ is a constant.

[3] $o(\chi(x))$ denotes the set of all functions $\omega(x)$ such that for all positive constants c there is an x_0 such that $\omega(x) \leqslant c\chi(x)$ for all $x \geqslant x_0$, or equivalently $\lim_{x\to\infty} \omega(x)/\chi(x) = 0$.

We now show that it is during the same iteration that nodes with costs in the interval $[C^*(\lceil \sigma d \rceil), C^*(d)]$ are expanded for the first time as $d \to \infty$. It is sufficient to prove that $\lim_{d\to\infty} \left(C^*(d) - C^*(\lceil \sigma d \rceil)\right) = 0$. By (B.12) and (B.13),

$$C^*(d) - C^*(\lceil \sigma d \rceil) = \log\log d - \log\log\lceil \sigma d \rceil + \phi(\log\log d) - \phi(\log\log\lceil \sigma d \rceil), \tag{B.14}$$

as $d \to \infty$. We first prove that $\log\log d - \log\log\lceil \sigma d \rceil = 0$ as $d \to \infty$. Notice that $\lceil \sigma d \rceil \geqslant \sigma d$ and $\log x$ is an increasing function of x. Thus, $\log\log\lceil \sigma d \rceil \geqslant \log\log(\sigma d)$. Therefore,

$$\begin{aligned} \lim_{d\to\infty} \left(\log\log d - \log\log\lceil \sigma d \rceil\right) &\leqslant \lim_{d\to\infty} \left(\log\log d - \log\log(\sigma d)\right) \\ &= \lim_{d\to\infty} \log\left(\frac{\log d}{\log(\sigma d)}\right) \\ &= \log\left(\lim_{d\to\infty} \frac{\log d}{\log d + \log \sigma}\right) \\ &= \log 1 = 0. \end{aligned} \tag{B.15}$$

In the above calculation, we used $\lim \log(Y(x)) = \log(\lim Y(x))$ because $\log(y)$ is continuous [2]. On the other hand, $\log\log d - \log\log\lceil \sigma d \rceil \geqslant 0$, since $\sigma < 1$. This and (B.15) mean that

$$\lim_{d\to\infty} \left(\log\log d - \log\log\lceil \sigma d \rceil\right) = 0. \tag{B.16}$$

We now prove $\phi(\log\log d) - \phi(\log\log\lceil \sigma d \rceil) = 0$ as $d \to \infty$. We first show that the function $\phi(\log\log d)$ introduced in (B.12) is nondecreasing with d as $d \to \infty$. Consider depths d and $d-1$ in $T(b,d)$. Following (B.12), we write

$$C^*(d) - C^*(d-1) = \log\log d - \log\log(d-1) + \phi(\log\log d) - \phi(\log\log(d-1)). \tag{B.17}$$

Using a similar calculation as in (B.15), we can simply prove

$$\lim_{d\to\infty} \left(\log\log d - \log\log(d-1)\right) = 0.$$

Because $C^*(d) - C^*(d-1) \geqslant 0$, by (B.17) we then write

$$\lim_{d\to\infty} \left(\phi(\log\log d) - \phi(\log\log(d-1))\right) \geqslant 0,$$

meaning that $\phi(\log\log d)$ does not decrease with d, as $d \to \infty$. Therefore,

$$\lim_{d\to\infty} \left(\phi(\log\log d) - \phi(\log\log\lceil \sigma d \rceil)\right) \geqslant 0, \tag{B.18}$$

where $\sigma < 1$. If $\phi(\log\log d)$ does not increase with d either, then

$$\lim_{d\to\infty}\left(\phi(\log\log d) - \phi(\log\log\lceil\sigma d\rceil)\right) = 0. \tag{B.19}$$

If $\phi(\log\log d)$ increases with d, on the other hand, its growth rate cannot be greater than that of the function $\log\log d$, since $\phi(\log\log d) \in o(\log\log d)$. Thus, by (B.16), we have

$$\begin{aligned}&\lim_{d\to\infty}\left(\phi(\log\log d) - \phi(\log\log\lceil\sigma d\rceil)\right)\\ &\quad\leqslant \lim_{d\to\infty}\left(\log\log d - \log\log\lceil\sigma d\rceil\right) = 0.\end{aligned} \tag{B.20}$$

Inequations (B.18) and (B.20) also lead to (B.19). Thus,

$$\lim_{d\to\infty}\left(C^*(d) - C^*(\lceil\sigma d\rceil)\right) = 0,$$

combining (B.14), (B.16) and (B.19). Therefore, iterative-deepening is asymptotically optimal in the case when $bp_0 = 1$. □

Theorem 2.11. *Let B be the asymptotic heuristic branching factor of a random tree $T(b,d)$ with $b > 1$. $B = 1$ when edge costs are chosen from a continuous distribution. When $bp_0 < 1$, and edge costs are chosen from a lattice distribution on $\{a_0 = 0, a_1\Delta, a_2\Delta, \ldots, a_m\Delta\}$ with probabilities $p_0, p_1, p_2, \ldots, p_m$, respectively, where $\Delta > 0$ is chosen such that the nonnegative integers $a_1 < a_2 < \cdots < a_m$ are relatively prime, B approaches a constant as $d \to \infty$, which is a solution greater than one to the equation*

$$\sum_{i=0}^{m} \frac{p_i}{B^{a_i}} = \frac{1}{b}. \tag{B.21}$$

Proof. When edge costs are chosen from a continuous distribution, node costs are unique with probability one. Consequently, the ratio of the number of nodes of a given cost to the number of nodes of any other cost is one, and so is the heuristic branching factor.

The heuristic branching factor of $T(b,d)$ with edge costs chosen from a lattice distribution on $\{a_0 = 0, a_1\Delta, a_2\Delta, \ldots, a_m\Delta\}$ is a corollary of the results of age-dependent branching processes [15]. In an age-dependent branching process, an object born at time 0 has a finite lifetime of random length. At the end of its life, the object is replaced by a random number of similar objects of age 0, which independently behave the same as their parent. This process of generating new objects continues as long as objects are present. A random tree $T(b,d)$ can be mapped to an age-dependent branching process as follows. The root node of the tree corresponds to the original object in the process. An edge cost corresponds to the lifetime of the object represented by the node at the end of the edge. The cost of a tree node corresponds to the absolute time when the corresponding object dies, or when the children of the object are born. One minor discrepancy is that the root node has cost zero, as defined, whereas the lifetime of the original object may not be zero. However, this will not cause a problem in our analysis, since the cost of the root node can also be defined to be a nonzero number, and this number is then added to all internal nodes of the tree.

Edge costs are multiples of Δ, and hence, so are the node costs. Without loss of generality, we divide all edge and node costs by Δ, and thus treat edge and node costs as integers. A node cost K is a linear combination of nonzero edge costs,

$$K = k_1 a_1 + k_2 a_2 + \cdots + k_m a_m, \tag{B.22}$$

for some nonnegative integers $\{k_1, k_2, \ldots, k_m\}$. (B.22) is a linear Diophantine equation that requires integer solutions [35]. In fact, for all integer $K > 2a_{m-1}\lceil a_m/m \rceil - a_m$, (B.22) has a solution in nonnegative integers $\{k_1, k_2, \ldots, k_m\}$ [9], indicating that node costs may include every integer K as $K \to \infty$. Indeed, the expected number of nodes with cost k is exponential in K, which we prove below using the results from age-dependent branching processes.

Let $D(K)$ be the expected number of nodes with cost K, which corresponds to the expected number of objects that die exactly at time K in the corresponding branching process. Furthermore, let $P(K)$ be the expected number of objects that are born (produced) exactly at time K, and $A(K)$ be the expected number of objects that are alive at time K in the branching process. Thus, we write

$$A(K) = A(K-1) + P(K) - D(K), \tag{B.23}$$

which means that the total expected number of currently alive objects $A(K)$ is equal to the total expected number of objects alive at the previous time point ($A(K-1)$) plus the expected number of new borns at time K ($P(K)$), and minus the expected number of the objects died at time K ($D(K)$).

It has been shown in [15] that $A(K)$ grows exponentially in K, i.e. as $K \to \infty$

$$A(K) = c_1 e^{\mu K} + \phi_1(e^{\mu K}), \tag{B.24}$$

where $c_1 > 0$ is a constant, $\phi_1(e^{\mu K}) \in o(e^{\mu K})$, and μ is a constant and the positive root to the equation

$$\sum_{i=0}^{m} \frac{p_i}{e^{\mu a_i}} = \frac{1}{b}. \tag{B.25}$$

Eq. (B.25) has a unique positive root if $b > 1$ [15].

Further notice that $P(K) > 0$, because the expected number of alive objects increases with K by (B.24). This gives $D(K) > 0$ since child objects are born when their parents die. Therefore, by (B.23) and (B.24), we have

$$\begin{aligned} P(K) &= A(K) - A(K-1) + D(K) \\ &> A(K) - A(K-1) \\ &= c_1 e^{\mu K} - c_1 e^{\mu(K-1)} + \phi_1(e^{\mu K}) - \phi_1(e^{\mu(K-1)}) \\ &= c_1(1 - e^{-\mu}) e^{\mu K} + \phi_1(e^{\mu K}) - \phi_1(e^{\mu(K-1)}). \end{aligned} \tag{B.26}$$

It is evident that $P(K) \leqslant A(K)$. This, combining (B.24) and (B.26), gives

$$P(K) = c_2 e^{\mu K} + \phi_2(e^{\mu K}), \tag{B.27}$$

where $c_1(1 - e^{-\mu}) < c_2 \leqslant c_1$, and $\phi_1(e^{\mu K}) - \phi_1(e^{\mu(K-1)}) < \phi_2(e^{\mu K}) \leqslant \phi_1(e^{\mu K})$. Therefore, by (B.23), (B.24) and (B.27), we have

$$\begin{aligned} D(K) &= P(K) - A(K) + A(K-1) \\ &= c_2 e^{\mu K} - c_1 e^{\mu K} + c_1 e^{\mu(K-1)} + \phi_2(e^{\mu K}) - \phi_1(e^{\mu K}) + \phi_1(e^{\mu(K-1)}) \\ &= c_3 e^{\mu K} + \phi_3(e^{\mu K}), \end{aligned} \tag{B.28}$$

where $c_3 = c_2 - c_1(1 - e^{-\mu}) > 0$, and $\phi_3(e^{\mu K}) = \phi_2(e^{\mu K}) - \phi_1(e^{\mu K}) + \phi_1(e^{\mu(K-1)})$. Using (B.28) and by the definition, we finally obtain the heuristic branching factor

$$\begin{aligned} B &= \lim_{K \to \infty} \frac{D(K)}{D(K-1)} \\ &= \lim_{K \to \infty} \frac{c_3 e^{\mu K} + \phi_3(e^{\mu K})}{c_3 e^{\mu(K-1)} + \phi_3(e^{\mu K - 1})} \\ &= \lim_{K \to \infty} \frac{c_3 + \phi_3(e^{\mu K})/e^{\mu K}}{c_3 e^{-\mu} + \phi_3(e^{\mu K - 1})/e^{\mu K}} \\ &= e^{\mu}. \end{aligned} \tag{B.29}$$

Because μ is a positive solution to Eq. (B.25), $B = e^{\mu}$ is a solution greater than one to Eq. (B.21). □

Theorem 2.12. *On a random tree $T(b,d)$ with $b > 1$, when $bp_0 < 1$ and edge costs are chosen from a lattice distribution, best-first search, depth-first branch-and-bound, iterative-deepening, and recursive best-first search all have the same effective branching factor $\beta = B^{\alpha}$ as $d \to \infty$, where p_0 is the probability that an edge has cost zero, B is the asymptotic heuristic branching factor of $T(b,d)$, α is the limit of C^*/d as $d \to \infty$, and C^* is the optimal goal cost.*

Proof. When $bp_0 < 1$ and edge costs are chosen from a lattice distribution, DFBnB, iterative-deepening and RBFS are all asymptotically optimal as $d \to \infty$, according to Theorems 2.8, 2.10 and the fact that RBFS is superior to iterative-deepening [24]. Therefore, they all have the same asymptotic effective branching factor as the asymptotic effective branching factor β of BFS. We now only need to show that $\beta = B^{\alpha}$ as $d \to \infty$.

We again use the results from age-dependent branching processes, and adopt the notations used in the proof of Theorem 2.11. Let lattice edge costs be chosen from the set $\{a_0 = 0, a_1\Delta, a_2\Delta, \ldots, a_m\Delta\}$. Again, we divide all edge and node costs by Δ, and treat them as integers without loss of generality. By Lemmas 2.1 and 2.4, BFS expands all nodes with costs less than the optimal goal cost C^*, plus some nodes with cost C^*, but no nodes with cost greater than C^*. That is,

$$Z(C^* - 1) \leqslant N_B(b,d) < Z(C^*), \tag{B.30}$$

where $Z(K)$ is the expected number of nodes with costs less than or equal to K. $Z(K)$ corresponds to the expected number of all objects that died up to time K. The number

of all dead objects up to time K is greater than or equal to the number of objects that died at time K, but is less than or equal to the total number of all living objects at all time points up to K, where a living object is counted again for each point in time at which it is alive. That is,

$$D(K) \leqslant Z(K) \leqslant \sum_{i=0}^{K} A(i), \tag{B.31}$$

where $D(K)$ and $A(K)$ are the expected number of died objects and living objects at time K. Following (B.24), $A(K) = c_1 e^{\mu K} + \phi_1(e^{\mu K}) < 2c_1 e^{\mu K}$, since $\phi_1(e^{\mu K}) < c_1 e^{\mu K}$ as $d \to \infty$. Thus,

$$\begin{aligned}\sum_{i=0}^{K} A(i) &< 2c_1 \sum_{i=0}^{K} e^{\mu i} \\ &= 2c_1 \frac{e^{\mu(K+1)} - 1}{e^{\mu} - 1} \\ &< 2c_1 \frac{e^{\mu}}{e^{\mu} - 1} e^{\mu K}.\end{aligned} \tag{B.32}$$

By (B.28) in the proof to Theorem 2.11,

$$D(K-1) = c_3 e^{\mu(K-1)} + \phi_3(e^{\mu(K-1)}) = (c_3/e)e^{\mu K} + \phi_3(e^{\mu(K-1)}).$$

From (B.30), (B.31) and (B.32), we then have

$$\frac{c_3}{e} e^{\mu C^*} + \phi_3(e^{\mu(C^*-1)}) \leqslant N_B(b,d) < 2c_1 \frac{e^{\mu}}{e^{\mu} - 1} e^{\mu C^*}. \tag{B.33}$$

This means that $e^{\mu C^*}$ is the dominant function of $N_B(b,d)$ as $d \to \infty$. Therefore, using $C^* = \alpha d + o(d)$ (Lemma 2.2) and by the definition of effective branching factor, we finally have

$$\begin{aligned}\beta &= \lim_{d\to\infty} \sqrt[d]{N_B(b,d)} \\ &= \lim_{d\to\infty} \sqrt[d]{e^{\mu(\alpha d + o(d))}} \\ &= \lim_{d\to\infty} e^{\mu\alpha} \cdot e^{\mu o(d)/d} \\ &= e^{\mu\alpha}.\end{aligned} \tag{B.34}$$

Because $B = e^{\mu}$ by Theorem 2.10, (B.34) is equivalent to $\beta = B^{\alpha}$ as $d \to \infty$. □

References

[1] E. Balas and P. Toth, Branch and bound methods, in: E.L. Lawler et al., ed., *The Traveling Salesman Problem* (John Wiley and Sons, Essex, 1985) 361–401.

[2] K.G. Binmore, *Mathematical Analysis* (Cambridge University Press, New York, 1982).

[3] P.P. Chakrabarti, S. Ghose, A. Acharya and S.C. de Sarkar, Heuristic search in restricted memory, *Artif. Intell.* **41** (1989) 197–221.

[4] P. Cheeseman, B. Kanefsky and W.M. Taylor, Where the really hard problems are, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 331–337.

[5] J.M. Crawford and L.D. Auton, Experimental results on the crossover point in satisfiability problems, in: *Proceedings AAAI-93*, Washington, DC (1993) 21–27.

[6] A. Dechter, A probabilistic analysis of branch-and-bound search, Tech. Report UCLA-ENG-81-39, Cognitive Systems Lab, School of Engineering and Applied Science, University of California, Los Angeles, CA (1981).

[7] R. Dechter and J. Pearl, Generalized best-first search strategies and the optimality of A*, *J. ACM* **32** (1985) 505–536.

[8] E.W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.* **1** (1971) 269–271.

[9] P. Erdös and R.L. Graham, On a linear diophantine problem of frobenius, *Acta Arithmatica* **21** (1972) 399–408.

[10] S.H. Fuller, J.G. Gaschnig and J.J. Gillogly, Analysis of the alpha-beta pruning algorithm, Tech. Report, Carnegie-Mellon University, Computer Science Department Pittsburgh, PA (1973).

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, New York, 1979).

[12] J.G. Gaschnig, Performance measurement and analysis of certain search algorithms, Ph.D. Thesis, Tech. Report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA (1979).

[13] M.L. Ginsberg and W.D. Harvey, Iterative broadening, *Artif. Intell.* **55** (1992) 367–383.

[14] J.M. Hammersley, Postulates for subadditive processes, *Ann. Probability* **2** (1974) 652–680.

[15] T. Harris, *The Theory of Branching Processes* (Springer, Berlin, 1963).

[16] T.P. Hart, N.J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Syst. Sci. Cybern.* **4** (1968) 100–107.

[17] B.A. Huberman and T. Hogg, Phase transitions in artificial intelligence systems, *Artif. Intell.* **33** (1987) 155–171.

[18] N. Huyn, R. Dechter and J. Pearl, Probabilistic analysis of the complexity of A*, *Artif. Intell.* **15** (1980) 241–254.

[19] T. Ibaraki, *Enumerative Approaches to Combinatorial Optimization—Part I*, Annals of Operations Research **10** (Scientific, Basel, Switzerland, 1987).

[20] R.M. Karp and J. Pearl, Searching for an optimal path in a tree with random costs, *Artif. Intell.* **21** (1983) 99–117.

[21] J.F.C. Kingman, The first birth problem for an age-dependent branching process, *Ann. Probability* **3** (1975) 790–801.

[22] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artif. Intell.* **27** (1985) 97–109.

[23] R.E. Korf, Real-time heuristic search, *Artif. Intell.* **42** (1990) 189–211.

[24] R.E. Korf, Linear-space best-first search, *Artif. Intell.* **62** (1993) 41–78.

[25] V. Kumar, Search branch-and-bound, in: S.C. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley-Interscience, New York, 2nd ed., 1992) 1468–1472.

[26] 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, eds., *Search in Artificial Intelligence* (Springer-Verlag, New York, 1988) 91–130.

[27] P. Langley, Systematic and nonsystematic search strategies, in: *Proceedings first International Conference on AI Planning Systems*, College Park, MD (1992) 145–152.

[28] 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*, Stanford, CA (1993) 112–118.

[29] E.L. Lawler, J.K. Lenstra, A.H.G.R. Kan and D.B. Shmoys, *The Traveling Salesman Problem* (John Wiley and Sons, Essex, 1985).

[30] E.L. Lawler and D.E. Wood, Branch-and-bound methods: a survey, *Operations Research* **14** (1966) 699–719.

[31] A. Mahanti, S. Ghosh, D.S. Nau, A.K. Pal and L.N. Kanal, Performance of IDA* on trees and graphs, in: *Proceedings AAAI-92*, San Jose, CA (1992) 539–544.

[32] C.J.H. McDiarmid, Probabilistic analysis of tree search, in: G.R. Gummett and D.J.A. Welsh, eds., *Disorder in Physical Systems* (Oxford Science, 1990) 249–260.

[33] C.J.H. McDiarmid and G.M.A. Provan, An expected-cost analysis of backtracking and non-backtracking algorithms, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 172–177.

[34] D. Mitchell, B. Selman and H. Levesque, Hard and easy distributions of SAT problems, in: *Proceedings AAAI-92*, San Jose, CA (1992) 459–465.

[35] I. Niven and H.S. Zuckerman, *An Introduction to the Theory of Numbers* (John Wiley and Sons, New York, 4th ed., 1980).

[36] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

[37] B.G. Patrick, An analysis of iterative-deepening-A*, Ph.D. Thesis, Computer Science Department, McGill University, Montreal, Que. (1991).

[38] B.G. Patrick, M. Almulla and M.M. Newborn, An upper bound on the complexity of iterative-deepening-A*, *Ann. Math. Artif. Intell.* **5** (1992) 265–278.

[39] J. Pearl, *Heuristics* (Addison-Wesley, Reading, MA, 1984).

[40] J. Pearl, Branching factor, in: S.C. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley-Interscience, New York, 2nd ed., 1992) 127–128.

[41] I. Pohl, Practical and theoretical considerations in heuristic search algorithms, in: E.W. Elcock and D. Michie, eds., *Machine Intelligence* **8** (Wiley, New York, 1977) 55–72.

[42] P.W. Purdom, Search rearrangement backtracking and polynomial average time, *Artif. Intell.* **21** (1983) 117–133.

[43] D. Ratner and M. Warmuth, Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 168–172.

[44] A. Rényi, *Probability Theory* (North-Holland, Amsterdam, 1970).

[45] S. Russell, Efficient memory-bounded search methods, in: *Proceedings ECAI-92*, Vienna, Austria (1992).

[46] U.K. Sarkar, P.P. Chakrabarti, S. Ghose and S.C. DeSarkar, Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, *Artif. Intell.* **50** (1991) 207–221.

[47] A.K. Sen and A. Bagchi, Fast recursive formulations for best-first search that allow controlled use of memory, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 297–302.

[48] D.R. Smith, Random trees and the analysis of branch and bound procedures, *J. ACM* **31** (1984) 163–188.

[49] H.S. Stone and P. Sipala, The average complexity of depth-first search with backtracking and cutoff, *IBM J. Research Development* **30** (1986) 242–258.

[50] L.A. Taylor and R.E. Korf, Pruning duplicate nodes in depth-first search, in: *Proceedings AAAI-93*, Washington, DC (1993) 756–761.

[51] N.R. Vempaty, V. Kumar and R.E. Korf, Depth-first vs best-first search, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 434–440.

[52] B.W. Wah, MIDA*, an IDA* search with dynamic control, Tech. Report UILU-ENG-91-2216 CRHC-91-9, Center for Reliable and High-Performance Computing Coordinated Science Lab, College of Engineering, University of Illinois at Urbana Champaign-Urbana, IL (1991).

[53] B.W. Wah and C.F. Yu, Stochastic modeling of branch-and-bound algorithms with best-first search, *IEEE Trans. Software Engineering* **11** (1985) 922–934.

[54] C.P. Williams and T. Hogg, Extending deep structure, in: *Proceedings AAAI-93*, Washington, DC (1993) 152–157.

[55] W. Zhang, Truncated branch-and-bound: a case study on the asymmetric TSP, in: *Working Notes of AAAI 1993 Spring Symposium: AI and NP-Hard Problems*, Stanford, CA (1993) 160–166.

[56] W. Zhang, Analyses of linear-space search algorithms and applications, Ph.D. Thesis, Computer Science Department, University of California at Los Angeles, Los Angeles, CA (1994).

[57] W. Zhang and R.E. Korf, An average-case analysis of branch-and-bound with applications: summary of results, in: *Proceedings AAAI-92*, San Jose, CA (1992) 545–550.

[58] W. Zhang and R.E. Korf, Depth-first vs. best-first search: New results, in: *Proceedings AAAI-93*, Washington, DC (1993) 769–775.

[59] W. Zhang and J.C. Pemberton, Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems, Tech. Report UCLA-CSD-940003, Computer Science Department, University of California, Los Angeles, CA (1994).

[60] W. Zhang and J.C. Pemberton, Epsilon-transformation: exploiting phase transitions to solve combinatorial optimization problems—initial results, in: *Proceedings AAAI-94*, Seattle, WA (1994).