Informed Search

This lecture expands upon our notion of basic search problems to incorporate more general costs. Accordingly, we generalize the notion of optimality discussed in the previous lecture from minimizing depth (i.e., all edges are of cost 1) to minimizing total cost. We introduce algorithms that search in this extended framework. The first is based only on the cost-so-far plus the cost of traversing one additional edge, while the latter incorporate heuristic functions that estimate the cost of reaching a goal node based on domain knowledge.

1 Blind vs. Informed Search

Blind search (e.g., BFS, DFS, IDS) algorithms forge ahead, without accounting for any potential differences in cost along different edges. But search often involves costs! Take for instance the path planning problem depicted in Figure 1, in which the UTAs are searching for a route from Providence to the White Mountains (in New Hampshire) for a retreat. There are two possible routes: one through Boston (I 93 to Rte 1) and one around Boston (I 95). Driving through Boston is less mileage, but involves two edge traversals. BFS, which minimizes depth (i.e., edges traversed from start node), would find the longer route (I-95)!

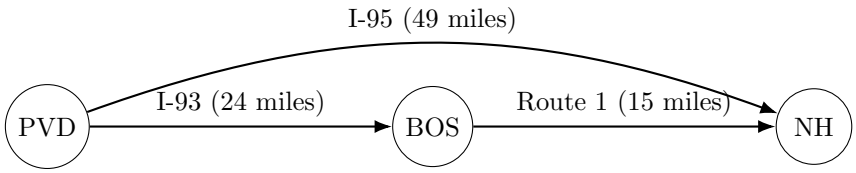

Figure 1: An example path planning problem. PVD is the start state and NH is the goal state.

An alternative approach is to use the available cost information to guide the search. For example, we could use a priority queue to prioritize lower-cost paths. Starting from PVD, this approach would push BOS on the queue before NH, because the former is 24 miles away, while the latter is 49 miles away. Then, after popping BOS off the queue and expanding it, it would arrive at NH a second time, but this time, via a shorter path ($24 + 15 = 39$ miles).

This alternative informed search algorithm is an improvement over blind search, but we can do even better by incorporating domain knowledge to further inform search algorithms. The key idea is to consider not only the costs of reaching all the successors of a state, which become evident when states are expanded, but to further predict the cost of reaching a goal state from each of the successors! Functions that encode this information are called **heuristic** functions. When a heuristic is accurate, the efficiency of search greatly improves, because progress towards the goal is more direct.

2 Applications

Search and optimization problems abound in the modern world! We describe a few prevalent examples here.

Google Maps Whenever you search for directions on Google Maps, Google solves a search problem to find you a route. Google's databases store detailed maps in the form of a graph (with road intersections as vertices and road segments that connecting intersections to one another as edges). A search problem is then created when you enter a start and goal state. Google maps uses informed search to solve this search problem! We note two interesting features of how Google solves your search problems:

- Google does not simply return the shortest path. On the contrary, it returns a few choices, each of which is best according to a different objective function. Different objectives include fastest (accounting for traffic), most fuel efficient (for driving directions), most scenic, etc., and combinations of these features. Since Google does not know your utility function (i.e., how you tradeoff among these features), it allows you to select among its top choices. Note also that Google uses a different set of objective functions depending on whether you are traveling by car, by public transit, by bicycle, or on foot.
- Even when optimizing explicitly for time, Google does not always return the fastest path. Waze, a competitor of Google Maps (also owned by Google!), sometimes find paths that are faster than Google Maps'. But if you have used Waze, you might have noticed that its routes can be more complicated than those provided by Google Maps. This is because Google Maps also optimizes for simplicity, by searching for paths with fewer road changes.

  There is another game-theoretic reason why Google does not always return the fastest path. Imagine there is traffic on the highway, so that the fastest path involves getting off and taking surface roads for a stint. If Google Maps were to tell all its users to exit the highway and take this route, these surface roads would become congested, and would no longer be the fastest path!

Video Games Most video games with Non-Player Characters (NPCs) that move about on their own use informed search algorithms to plan paths for those NPCs. Finding paths for characters in Dragon Age 3, Baldur's Gate, StarCraft, and Warcraft is so important that these problems actually serve as a common benchmark for pathfinding algorithms. Finding paths for NPCs needs to be *fast*! Optimality is sacrificed, because lag in video games is undesirable. Heuristics are designed to prioritize the speed of search.

Multi-Agent Path Finding (MAPF) In automated warehouses, like those run by Amazon, hundreds of robots navigate within the same space, picking up and dropping off packages at specified locations. MAPF is the problem of finding multiple paths for multiple agents navigating within the same environment, such that they all achieve their separate goals without colliding. If each agent were to plan its own path individually (i.e., in a decentralized fashion), the agents would run the risk of colliding. Instead, MAPF algorithms are central planners. A single state stores all the agents' locations, and the successor function returns states where all the agents take an action and none of them collide. Informed search can be used to solve MAPF.

Large Language Models Language Models are AI models that take in a sequence of tokens (e.g., words) and output a probability distribution over the next token (i.e., a probability for each possible next word). Although language models only produce probabilities for the very next possible token, they can be run generatively and repeatedly, each time sampling from the output distribution to generate an output token, which is then appended to the input sequence, before invoking them again. Any such sequence of output tokens, generated in response to an input sequence, is called a **continuation**.

One thing we may want from a language model is the continuation of output tokens with the highest joint probability. Finding the highest probability continuation is a search problem! Unfortunately, there are way too many possible continuations to generate them all. A popular approach to tackling this challenge is to use **beam search**, an informed search algorithm that maintains a non-exhaustive list of promising continuations, meaning a few that seem likely to be of high probability. In this way, beam search balances finding high-quality solutions against memory usage and compute time.

3 Search Problem

A *search problem* is a 5-tuple $\langle X, S, G, \mathcal{T}, c \rangle$, where

- $\langle X, S, G, \mathcal{T} \rangle$ is a basic search problem
- $c : X \times X \to \mathbb{R}$ is a cost function

Given a state x, $c(x, y)$ denotes the cost of reaching y from x, where $y \in \mathcal{T}(x)$ is a successor state of x. Now given path $\{n_0, \ldots, n_i, n_{i+1}, \ldots, n_k\}$, where $n_0 \in S$, $n_k = n$, and $n_{i+1} \in \mathcal{T}(n_i)$ for all $0 \leq i \leq k$, $g(n)$ denotes the *total* cost of reaching node n along the given path:

$$g(n) = \sum_{i=0}^{k} c(n_i, n_{i+1}) \tag{1}$$

Examples of cost functions include: $g(n) = \text{depth}(n)$ and $g(n) = \text{distance}(n)$.

Note that search problems can be stated in terms of cost, with $c(x) \geq 0$ for all $x \in X$, in which case the problem is one of minimization, or value with $c(x) \leq 0$ for all $x \in X$, in which case the problem is one of maximization. In either case, total costs (or values) are monotonic in depth, and bounded below when nondecreasing, and above when nonincreasing.[1]

4 Best-First Search

The main idea of the best-first search class of algorithms is to expand the lowest-cost node on the fringe, according to some evaluation function $e : X \to \mathbb{R}$.

BFS is the special case of best-first search in which the evaluation function $e(n) = \text{depth}(n)$ for node n. Therefore, the complexity of best-first search in the worst-case is at least that of BFS: exponential in the depth of the goal for both time and space. Best-first search visits nodes in depth-first search order, when the evaluation function e dictates the following of paths until the algorithm dead ends. Best-first search is not necessarily complete; nor is it necessarily optimal.

5 Best-g Search

The main idea of best-g search is to expand the lowest-cost node on the fringe, according to cost function g, defined in Equation 1. Best-g is complete, except in search spaces that contain infinitely many nodes n with $g(n) < g^*$ (e.g., an infinite path with finite cost), where g^* is the optimal cost. Best-g is also optimal: it is guaranteed to find the lowest-cost goal, whenever g is a monotonically, nondecreasing function of depth.

Figure 2 depicts two search trees. In both spaces, S is the start state, Y and Z are goal nodes, and Z is optimal. On the LHS, the search tree contains an infinite path of finite cost. Best-g search never reaches either goal node. On the RHS, the search tree contains an edge of negative cost. Best-g search proceeds directly to the suboptimal goal node Y.

[1] Note that values are often shifted to be positive rather than negative, e.g., they may lie in the range of $[0, 1]$, but they must remain bounded above. When costs (or values) are not bounded, it may be possible to traverse a graph forever, perpetually decreasing costs (or accruing value).

BEST-FIRST	
Inputs	search problem $\langle X, S, G, \mathcal{T}, c \rangle$ evaluation function e
Output	(path to) goal node
Initialize	$O = S$ is the list of open nodes

`while` (O is not empty) `do`

1. delete node $n \in O$ *s.t.* $e(n)$ is minimal
2. if $n \in G$, return (path to) n
3. for all $m \in \mathcal{T}(n)$
   (a) compute $e(m)$
   (b) insert m into O with priority $e(m)$

`fail`

Table 1: Best-First Search. Best-g search is the special case of best-first search in which $e = g$. Best-h search is the special case of best-first search in which $e = h$. A* search is the special case of best-first search in which $e = f = g + h$.

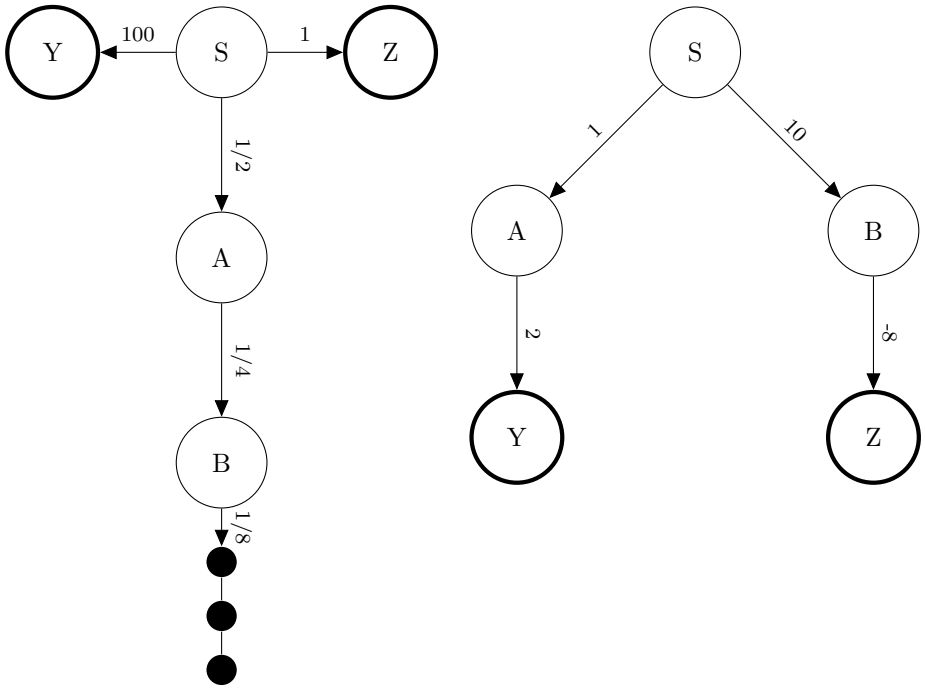

Figure 2: (LHS) A search space that contains an infinite path of finite cost. (RHS) A search space that contains an edge of negative cost. In both search spaces, S is the start state, Y and Z are goal nodes, and Z is optimal.

Language Modeling: Finding the Most Likely Continuation Large Language Models (LLMs) can be augmented with search algorithms to improve their performance. First, let's cover the fundamentals of

the language modeling problem. Language modeling starts with some initial *context* (e.g., the beginning of a sentence) and seeks to predict the most likely *token* to follow. [ERIC: *In general, tokens are parts of words.*] To generate new text, models repeat a two step process: 1) predict probabilities for the next token given a context and 2) append the token to the context. We call the newly generated text a *continuation*. One thing we may wish to find with a language model is the *most likely continuation*, or the continuation with the highest joint predicted probability by the language model. We can compute the likelihood of an entire continuation by multiplying the probability of each generated token.[2] Most language models work by selecting the token with the highest probability at every step of the generation process (with some additional added randomness based a parameter called *temperature*). However, we can easily find examples where selecting the highest probability output at every step does not yield the highest likelihood continuation.

Consider a language model that has been prompted with "The weather today is" and we want to find the most likely 2-word continuation. At each step, the model outputs a probability distribution over possible next words. Figure 3 shows the probability tree for this example.

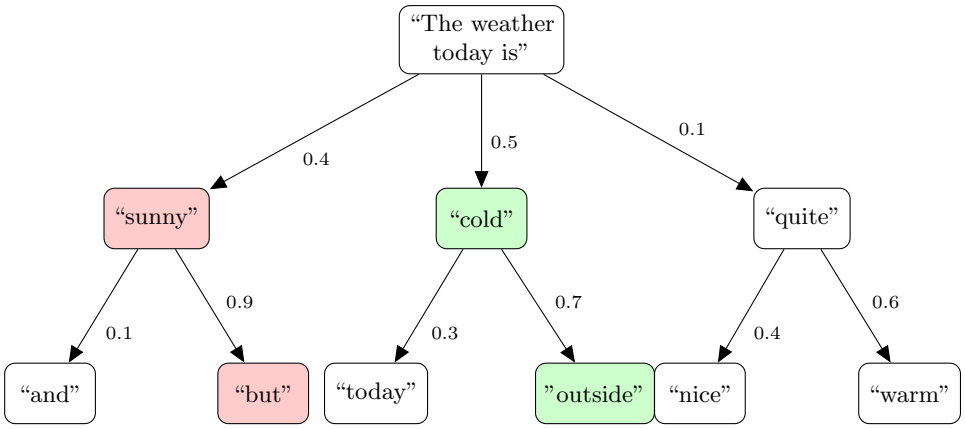

Figure 3: Probability tree for language model continuation. Each edge is labeled with $P(\text{word}|\text{context})$. Green shading shows the greedy path, red shading shows the optimal continuation. The green path selects the edge with highest probability and achieves a continuation with likelihood $0.5 \cdot 0.7 = 0.35$. If the red path had been chosen instead, the model would have achieved a higher likelihood of $0.4 \cdot 0.9 = 0.36$.

To find the highest probability continuation, we need to find the path that maximizes the joint probability. However, there are two problems we must solve before applying our search algorithms. First, our problem asks us to **maximize** an objective, but our search algorithms are designed to **minimize** a cost function. This is easily remedied by taking the negative of our objective, converting maximization to minimization. The second problem is more subtle: probabilities are not monotonically increasing as we move farther from the root node. In fact, since we multiply probabilities at each step (and each probability is at most 1), the joint probability can only decrease or stay the same as we extend the path.

Taking the logarithm elegantly solves both issues simultaneously. Since $\log(ab) = \log(a) + \log(b)$, we can convert the product of probabilities into a sum of log probabilities. Then, by taking the negative log, we transform our maximization problem into a minimization problem where costs are additive. Specifically, we minimize:

$$\text{cost} = -\sum_{i=1}^{k} \log P(w_i|\text{context}) \tag{2}$$

[2]We will cover Bayesian probability thoroughly later in the semester. This statement is true by the chain rule of probability. For a given context c and generated tokens $t_1, \ldots, t_k$, the likelihood of a continuation $P(t_1, \ldots, t_k|c)$ can be found by $P(t_1, \ldots, t_k|c) = P(t_1|c)P(t_2|c, t_1) \ldots P(t_k|c, t_1, \ldots, t_k)$.

This formulation satisfies the requirements of our search algorithms: we have a minimization objective with costs that accumulate (sum) along paths, making the problem well-suited for algorithms like Best-g search.

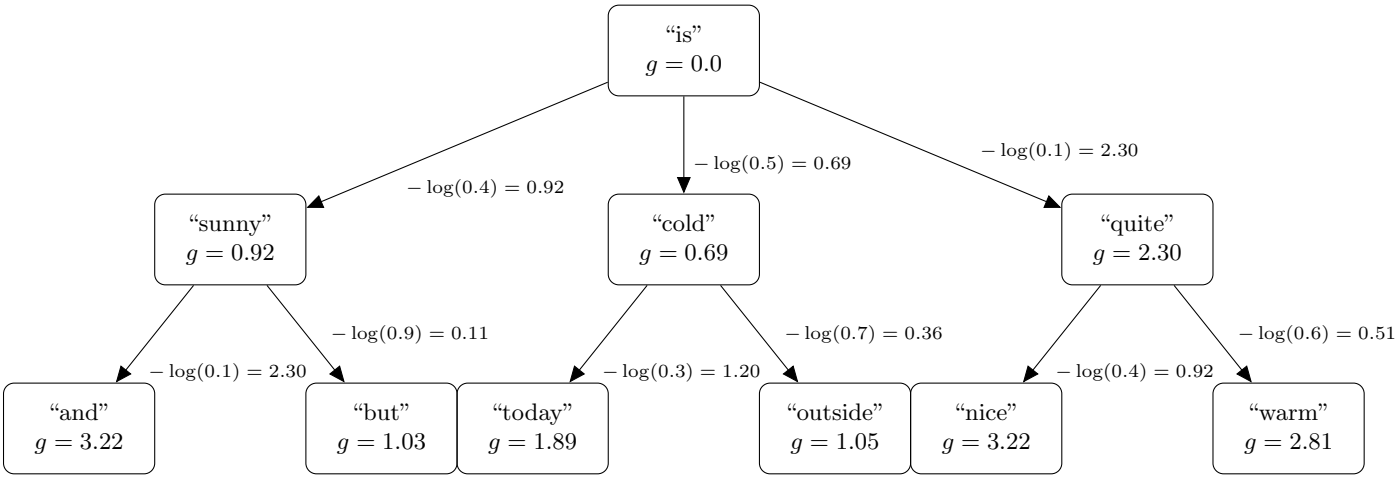

Figure 4: The probability tree for a language model transformed with a negative log operation. Best-g search can be run on this tree without any other modifications and find the most likely continuation.

6 Best-h Search

The main idea of best-h search is to expand the lowest-cost node on the fringe, according to some heuristic function $h : X \to \mathbb{R}$. The degree of optimality of best-h search depends on the quality of the heuristic function.

A heuristic function $h : X \to \mathbb{R}$ computes an estimate of the distance from node n to a goal node. Heuristics are used to guide the search process. In the sliding tiles puzzle, one heuristic function $h_1(n)$ is simply the number of misplaced tiles. A second heuristic function $h_2(n)$ is the Manhattan distance: i.e., the number of moves required to place each tile correctly, summed over all misplaced tiles.

1	3	5
7	2	4
6	8	

1	2	3
4	5	6
7	8	

Figure 5: (LHS) Start State. (RHS) Goal State. $h_1(n) = 6$ and $h_2(n) = 10$.

Figure 5 depicts an arbitrary state n and the goal of the 8-puzzle—the sliding tiles puzzle with 8 tiles. In this state n, there are 6 misplaced tiles, and the Manhattan distance evaluates to 10.

Exercise Give other examples of heuristics for the sliding tiles puzzle.

7 A* Search

Let $f(n) = g(n)+h(n)$, where $g(n)$ is the cost of reaching node n from the start state and $h(n)$ is an heuristic estimate of the distance from node n to the nearest goal node. The main idea of A* search is to expand the lowest-cost node on the fringe, according to the evaluation function f. Like best-g and best-h searches, A* is a special case of the best-first search algorithm. Nonetheless, we present the A* algorithm in its entirely in Table 2.

A* SEARCH	
Inputs	search problem $\langle X, S, G, \mathcal{T}, c\rangle$
	heuristic function h
Output	(path to) optimal goal node
Initialize	$O = S$ is the list of open nodes

`while` (O is not empty) `do`

1. delete node $n \in O$ *s.t.* $f(n)$ is minimal
2. if $n \in G$, return (path to) n
3. for all $m \in \mathcal{T}(n)$
   (a) compute $h(m)$
   (b) $g(m) = g(n) + c(n, m)$
   (c) $f(m) = g(m) + h(m)$
   (d) insert m into O with priority $f(m)$

`fail`

Table 2: A* Search.

A* search is optimal, assuming the heuristic function h is **admissible**.

8 Admissible Heuristics

Let $h^*(n)$ be the true cost from node n to the nearest goal node. A heuristic function $h(n)$ is said to be *admissible* iff $h(n) \leq h^*(n)$, for all nodes n. In other words, admissible heuristics are optimistic: in minimization problems, admissible heuristics never overestimate the distance to a goal; in maximization problems, admissible heuristics never underestimate the value of a goal.

The sample heuristics h_1 and h_2 in the sliding tiles puzzle are both admissible. The heuristic function h_1 is admissible since it requires at least one move to move each misplaced tile to its correct position. The heuristic function h_2 is admissible since, more accurately, it requires at least the Manhattan distance to move each misplaced tile to its correct position.

The most useful admissible heuristics are those which most closely approximate $h^*(n)$ *without going over*. An admissible heuristic h *dominates* an alternative admissible heuristic h' iff $h(n) \geq h'(n)$ for all nodes n. Intuitively, a dominant heuristic is more informed than the heuristic it dominates. For example, the Manhattan distance h_2 dominates h_1.

Exercise Given two admissible heuristics h' and h'', it need not be the case that one dominate the other. In this case, one can construct *composite* heuristics of the form $h(n) = \max\{h'(n), h''(n)\}$ for all n. The new

heuristic h is admissible and it dominates the individual heuristics h' and h''. Prove this claim.

One "heuristic" for constructing admissible heuristics is to remove one or more of the problem's constraints. In the sliding tiles puzzle, moves are constrained in three ways: a tile can only be moved into the blank space; a tile must be moved along the grid; and, a tile can only be moved into an adjacent cell. If we relax only the first constraint, this yields the Manhattan distance (h_2). If we relax the first and the second constraints, this yields another heuristic function—Euclidean distance—call it h'. If we relax all three constraints, this yields the heuristic function h_1. Clearly, h_2 dominates h' dominates h_1, since h_2 enforces more constraints than h'; and, h' dominates h_1, since h' enforces more constraints than h_1.

9 IDA* Search

Iterative deepening A* (IDA*) is an optimal search algorithm with the performance properties of A^*—it is complete and optimal—and the space requirements of DFS—(essentially) linear in depth. The main idea of iterative deepening A* is to repeatedly search in depth-first fashion, over subgraphs with f-cost less than α, less than 2α, less than 3α, and so on, until a goal is found, where α is a lower bound on the cost between nodes and their successors throughout the search space: i.e., $\alpha \leq c(n, m)$, for all $n, m \in \mathcal{T}(n)$.

Recall that the space complexity of ID is $O(bd)$, where d is the depth of the goal node. Similarly, the space complexity of IDA* is $O(bg^*/\alpha)$, where g^* is the optimal cost. The time complexity of IDA*, however, can exceed that of A*. In particular, in search spaces where the f-cost is different at every state, only one additional state is expanded during each iteration. In such a search space, if A* expands n nodes, IDA* expands $1 + \ldots + N = O(N^2)$ nodes. The typical solution to this problem is to fix an increment $\beta > \alpha$ such that several nodes n have cost $f_i < f(n) \leq f_i + \beta$, where f_i is the ith incremental value of the f-cost. This strategy reduces search time, since the total number of iterations is proportional to $1/\beta < 1/\alpha$, and returns solutions that are at worst β-optimal: i.e., if the algorithm returns m^*, then $g(m^*) < g^* + \beta$.

IDA* Search

Inputs	search problem $\langle X, S, G, \mathcal{T}, c \rangle$
	heuristic function h
Output	(path to) optimal goal node
Initialize	$i = 0$ is the cutoff f-value
	$O = S$ is the list of open nodes

`while` (1) `do`

1. `while` (O is not empty) `do`
   (a) delete *first* node $n \in O$
   (b) if $n \in G$, return (path to) goal n
   (c) for all $m \in \mathcal{T}(n)$
      i. compute $h(m)$
      ii. $g(m) = g(n) + c(n, m)$
      iii. $f(m) = g(m) + h(m)$
      iv. if $f(m) \leq i$, insert m in *front* of O
2. increment i by β, $O = S$

Table 3: Iterative Deepening A*.

10 Examples

Best-g Search The tree shown in Figure 6 has cost function $g(n) = \text{depth}(n)$. Best-g on this search space is precisely BFS: it finds the optimal goal node G. Nodes are expanded as follows: A, BCD, CDEF, DEFG, EFG, FG, GHI, GOAL!

Best-h Search The tree depicted in Figure 7 has cost function $h(n)$. Best-h search returns the suboptimal goal node H in this example. The priority queue is maintained as follows: A, BCD, EFCD, FCD, HICD, GOAL!

A* and IDA* Search The tree depicted in Figure 8 has cost function $f(n) = g(n) + h(n)$. A* search returns the optimal goal node G in this example. Nodes are expanded as follows: A, BCD, ECFD, CFD, GFD, GOAL! Or, if ties are broken otherwise, nodes could be expanded in an alternative order: A, BCD, CEDF, EGDF, GDF, GOAL! Since h is admissible, A* is optimal. IDA* expands nodes as follows, for $\beta = 1$: $f = 0$: A; $f = 1$: AB; $f = 2$: ABECG, GOAL!

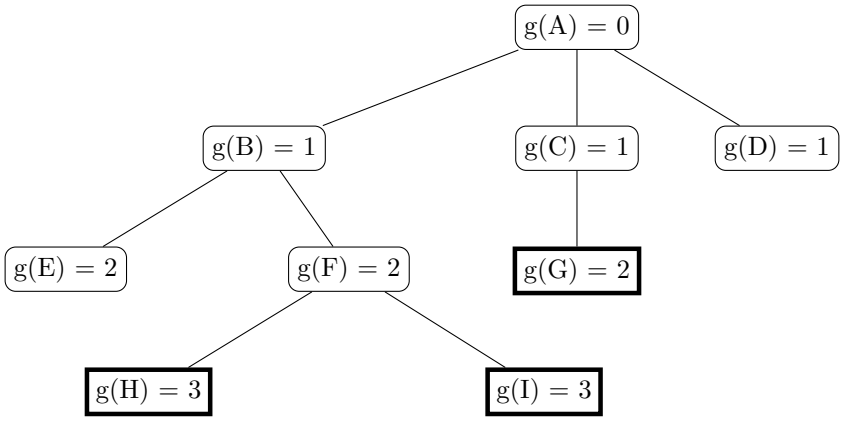

Figure 6: Sample search tree, labeled with costs g. Boxes indicate goal nodes. Best-g returns the optimal goal node G.

11 Summary

Criteria	Best-g
Time	$O(b^d)$: BFS, if $g = \text{depth}$
Space	$O(b^d)$: BFS, if $g = \text{depth}$
Completeness	YES, if there do not exist ∞-many nodes n *s.t.* $g(n) < g^*$
Optimality	YES, if g is monotonically nondecreasing in depth

Criteria	Best-h
Time	$O(b^d)$: BFS, if $h = \text{depth}$
Space	$O(b^d)$: BFS, if $h = \text{depth}$
Completeness	NO, if nodes are visited in DFS order
Optimality	NO, if nodes are visited in DFS order

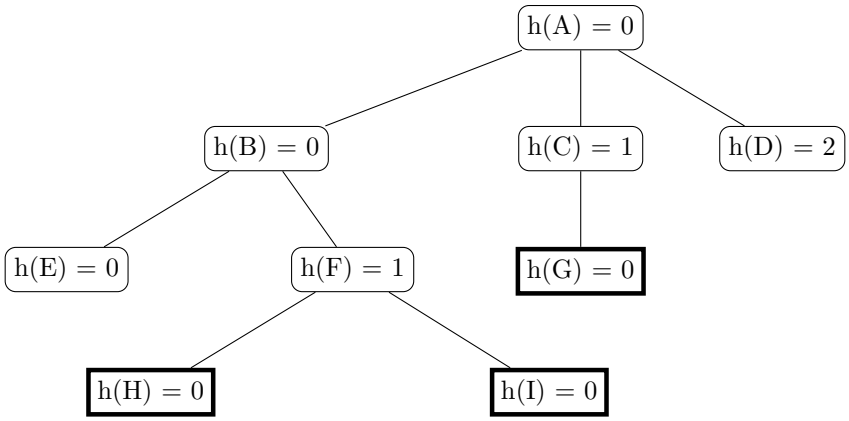

Figure 7: Sample search tree, labeled with heuristic values h. Boxes indicate goal nodes. Best-h search returns the suboptimal goal node H.

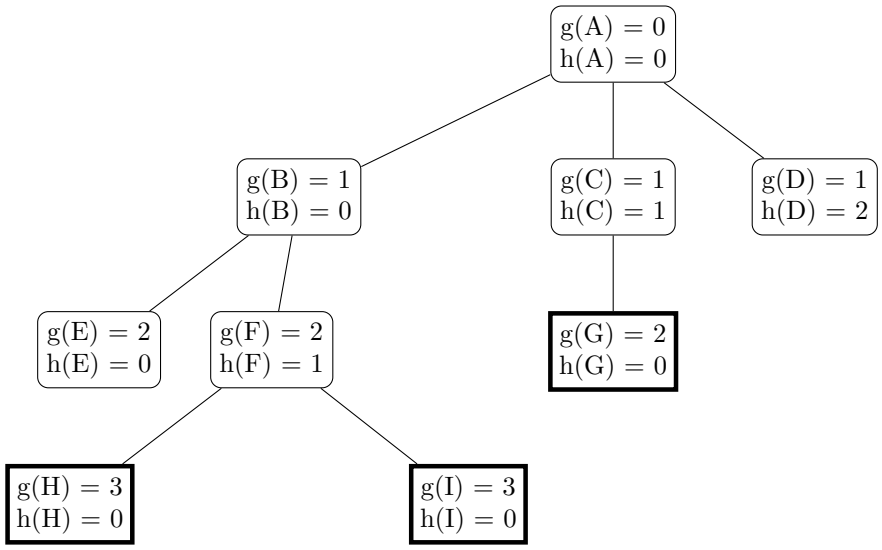

Figure 8: Sample search tree, labeled with costs g and heuristic values h. Boxes indicate goal nodes. A* search returns the optimal goal node G.

Criteria	A*
Time	$O(b^d)$: BFS, if $g =$ depth and $h = 0$
Space	$O(b^d)$: BFS, if $g =$ depth and $h = 0$
Completeness	YES, if there do not exist ∞-many nodes n *s.t.* $f(n) < f^*$
Optimality	YES, if h is admissible and g is monotonically nondecreasing in depth

Criteria	IDA*
Time	$O(N^2)$, if f-costs differ at all states and A* expands n nodes
Space	$O(bg^*/\beta)$, if f is monotonically nondecreasing in depth and if g^* optimal is the optimal cost
Completeness	YES, if there do not exist ∞-many nodes n *s.t.* $f(n) < f^* + \beta$
β-Optimality	YES, if h is admissible and g is monotonically nondecreasing in depth