CS221 Lecture notes #3

Search

Previously, we showed how discretization techniques, such as grids, visibility graphs, and probabilistic roadmaps, could be used to convert a continuous motion planning problem into one on a discrete graph. How can we search graphs like these efficiently? In this set of notes, we will present algorithms to solve search problems on discrete graphs. We will first discuss **blind search** (also called **uninformed search**) algorithm, where we know nothing except for the nodes and edges which make up the graph. We will then describe **heuristic search**, in which we will use knowledge about the problem to greatly speed up the search. These search algorithms are quite general, and are widely used many areas of AI.

Throughout much of these notes, our motivating example will be a toy problem known as the 8-puzzle, shown in Figure 1.

1 Search formalism

A discrete graph search problem comprises:

- **States**. These correspond to the possible states of the world, such as points in configuration space, or board positions for the 8-puzzle. We typically denote individual states as s, and the set of all states as S.
- **(Directed) edges**. There is a directed edge from state s_1 to state s_2 if s_2 can be reached from s_1 in one step. We assume directed edges for generality, but an undirected edge can be represented as a pair of directed edges. We typically use e to denote an edge, and E the set of all edges.
- **Cost function**. A non-negative function $g : E \mapsto \mathbb{R}_0^+$. (This notation means g is a function mapping from the edges E into the set of non-negative real numbers $\mathbb{R}_0^+$.)

(a) (b)

Figure 1: The 8-puzzle. There are 8 tiles, numbered 1 through 8, which slide around on the board. In the initial state, the tiles are scrambled. The goal is to put the numbers in order, with the blank square in the lower right hand corner. (a) An initial state. (b) The goal state.

- **Initial state**. Usually a single state $s \in S$.
- **Goal**. For generality, we represent the goal with a **goal test**, which is a function that tells us if any particular state s is a goal state. This is because our task may be to get to any state within some "goal region," rather than to a very specific, single, goal state. For instance, in motion planning, the goal test might be "end of finger presses the elevator button." This cannot be described with a single goal state, since many different configurations of the robot's joints are probably consistent with the goal. In problems where we are interested in reaching one particular goal state, the goal test will be a function which returns true for a state s if and only if it is the goal state.

Given this definition of a search problem, there are two ways that we can represent the search space:

- **Explicitly**. In this case, we explicitly construct the entire graph of the search space in computer memory or on disk. Thus, we would create a list of all the states, all the edges, and all the costs. The goal test will also be explicitly represented as a list of all the goal states. This explicit representation only works for fairly small graphs. Consider, for instance, our example of the 8-puzzle. The total number of possible board configurations is $9! = 1 \times 2 \times \cdots \times 9$, and the total number of edges is larger still. For the larger 15-puzzle (on a 4x4 grid rather than 3x3 grid), the number of states is $16! \approx 2 \times 10^{13}$. Clearly, we can store the graph in memory only for the smallest problem sizes.
- **Implicitly**. To represent a search problem implicitly, we will use some data structure for representing individual states. For example, if the

states comprise a 2D grid, the data structure might be an (i, j) pair. The edges and costs are also represented implicitly. Edges are usually described in terms of **operators**, which are functions $o : S \mapsto S$ that map from states to states.[1] In a 2D grid, our operators may be N, S, E, and W, corresponding to movements in the four directions. In the 8-puzzle, we would have some data structure that represents board positions, as well as four operators, each corresponding to moving one of the four adjacent tiles into the empty hole. Each operator is assigned a positive real-valued cost.[2] The **successor states** of a state are the states reachable by the application of a single operator.

2 Basic search

As we develop different search algorithms, we'll be interested in questions of whether they are complete, whether they are optimal, and in their computational efficiency. A search algorithm is **complete** if, whenever there exists a path from the initial state to the goal, it will find one. It is **optimal** if any path it finds to the goal is a minimum cost path.

We first consider the case where we are given nothing except the nodes, edges, and costs of the search space. This is referred to as **blind**, or **uninformed** search. We will discuss several different search algorithms for discrete graphs (represented either implicitly or explicitly). Our search algorithms will, conveniently, all follow the same basic search template:

```
Queue q;
q.insert(initialState);
while (!q.isEmpty()){
  node = q.remove();
  if (goalTest(node)) return node;
  foreach (n in successors(node, operators))
    q.insert(n);
```

[1] We will be slightly loose in our terminology, and not specify the result of operators which cannot be applied in a given position, such as what happens if the N operator is applied when we're at the uppermost row of the grid and can't move any further north. There are many options: the operator could be deemed inapplicable to that state, it could return the same state, and so on.

[2] We could allow the cost also to depend on the state itself. For instance, it might cost a robot dog more to move north in rugged terrain than smooth terrain. This is a reasonably straightforward generalization that would require only a small change to the algorithms we describe below.

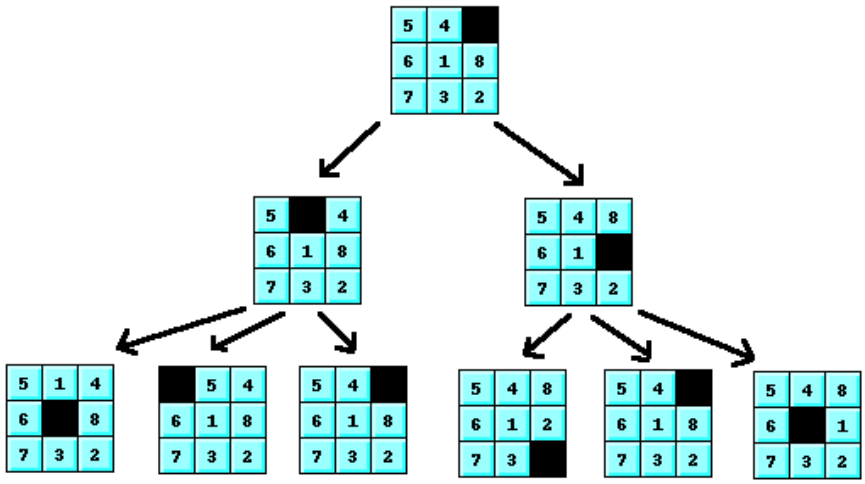


Figure 2: A portion of a search tree for the 8-puzzle. The root node is the initial state, and each of the children of a given node corresponds to one of the four possible operators. In this example, we did not eliminate repeated states (which you should usually do); so, for example, the third and fifth nodes in the bottommost row are repeats of the root.

```
}
return FAIL;
```

For now, we will leave exactly what sort of queue we use unspecified. Depending on this choice, we will get different search algorithms.

In the inner loop of the algorithm, we remove a state, and check if it's a goal. If it's not, we put all of its successors on the queue, to be examined later. We say a node is **expanded** when it is taken out of the list and checked for goalness and its successors are inserted into the queue.

Note that the order of operations is important. In particular, we check for goalness when the node is expanded, *not* when it is being added to the queue. This will become important later on when we prove optimality of various algorithms.

This process implicitly defines a search tree, like the one shown in Figure 2. The search tree contains **nodes** n. Each node is labeled with a state s. Each node also corresponds to a specific *sequence of states* (specifically, the sequence of states you pass through as you start from the root, and traverse the tree until you reach n). Because this is a tree, the path from the root to a node n is always unique. Note that even though nodes in the tree are labeled with states (board positions), nodes and states are two distinct concepts.

We will use "states" only to refer to states in the original search graph, and "nodes" only to refer to nodes in the search tree. For example, in the original search graph, there can sometimes be multiple paths from the initial state to some other state s. Thus, as shown in the figure, the search tree can also contain multiple nodes n that are labeled with the same state s. (Later in these notes, we'll also discuss how to deal with repeated states, but we'll ignore this issue for now.)

The search tree is defined as follows. When the search algorithm is first run and the initial state is placed on the queue, think of this as constructing a tree with just a root node, which is labeled with the initial state. At each point in our search, the nodes in the queue correspond to the "fringe" of the search tree, and nodes which have already been expanded correspond to the interior nodes. Each time we expand a node on the queue, that corresponds to taking a node on the fringe and adding children to it, and then putting these new children (which are now on the fringe) onto the queue.

Note that this search tree is a mathematical concept used to help us reason about the process of running the search algorithms, and is implicitly defined by the running of our search algorithm. The search tree isn't actually explicitly constructed anywhere.

Now we discuss particular search algorithms which follow this template. We give only a very brief overview of each algorithm; some additional details can be found in Section 3.4 of the course textbook.

2.1 Depth-first search

When our queue is a LIFO queue (stack), we get **depth-first search (DFS)**. DFS keeps recursively expanding nodes until it reaches a goal state or encounters a state with no successors. When a state has no successors, DFS backtracks.

In an undergraduate programming/data structures classes, you may have seen DFS written as follows, where the stack was implicitly maintained through the program's stack frame:

```
State DepthFirstSearch(node){
  if (goalTest(node)) return node;
  foreach (n in successors(node, operators)){
    result = DepthFirstSearch(n);
    if (result != FAIL) return result;
  }
  return FAIL;
```

```
}
```

Using a LIFO queue in the search template given previously, the stack is now explicitly maintained through our own queue, but it results in the same search process as this.

Under mild assumptions, DFS is usually complete. Unfortunately, DFS is generally not optimal.

2.2 Breadth-first search

When our queue is a FIFO queue, we get **breadth-first search (BFS)**. Intuitively, nodes will be expanded in order by the length of the path (number of edges). (You should convince yourself of this.)

BFS is a complete search algorithm under very reasonable assumptions. It is also optimal in the particular case where all of the operators have a cost of 1 (because it then becomes a special case of uniform-cost search).

2.3 Uniform-cost search

When our queue is a priority queue ordered by the total cost of the path to a node, we get **uniform-cost search (UCS)**. Note that by the definition of a search tree, the path from the root node to any given node n is always unique (even if, in the original search graph, there may be multiple paths from the initial state to the state s associated with the node n). Thus, the priority of n is simply the total cost of all the edges between the root node and the node n. We will denote this cost $g(n)$.

Assuming that we handle repeated states (described in Section 4.1, this algorithm is also commonly known as Dijkstra's shortest-path algorithm, which is a complete and optimal algorithm. (You can find a proof of this in the textbook.) The gist of the optimality proof is as follows: nodes are expanded in order according to the total cost to reach that node. Out of all paths to a goal, the optimal path is (by definition) the shortest, and so the node in the search tree corresponding to that path will be expanded before all of the others. This proof works for all finite graphs, as well as infinite graphs under mild assumptions.

3 Heuristic search

All of the algorithms presented above are examples of blind (uninformed) search, where the algorithm has no concept of the "right direction" towards

the goal, so that it blindly stumbles around until it happens to expand the goal node. We can do much better if we have some notion of which is the right direction to go. For instance, if a robot is trying to get from the first floor to the basement, then going up in the elevator is unlikely to help. One way to provide this sort of information to the algorithm is by using a **heuristic function**, which estimates the cost to get from any state to a goal. For instance, if our goal is to get to the basement, a heuristic might assign a higher estimated cost to the fourth floor than to the first. More formally, a heuristic is a non-negative function $h : S \mapsto \mathbb{R}_0^+$, that assigns a cost estimate to each state.

For example, consider the grid in Figure 3. We know it is likely to be more productive to move right than to move left, because the goal is to the right. We can formalize this notion with a heuristic called **Manhattan distance**, defined as the total number of N/S/E/W moves required to get from the current state to the goal, *ignoring obstacles*. The name "Manhattan distance" comes from the observation the in much of Manhattan island in New York, the city blocks are our laid along the compass directions, out so that you can only drive in compass directions, rather than diagonally.

In constructing heuristics, we often face a tradeoff between the accuracy of a heuristic and how expensive it is to compute it. For instance, here are two examples of trivial heuristics:

- The constant heuristic $h(s) = 0$. This heuristic requires no computation, but provides no useful information about the problem.
- The heuristic equal to the minimum cost from s to a goal. If we knew this heuristic function, the search problem would be trivial. (Why?) However, computing this requires actually solving the problem, and so it is no good in practice.

The most useful heuristics will almost always lie between these two extremes.

So far, we have defined a heuristic function as taking input a state. Given a node n in a search tree, we also define $h(n)$ to be the value obtained by applying the heuristic function h to the state s that the node n is associated (labeled) with.

Let us now consider some particular algorithms which make use of the heuristic function. As in the section on uninformed search, we will follow the search template given previously.

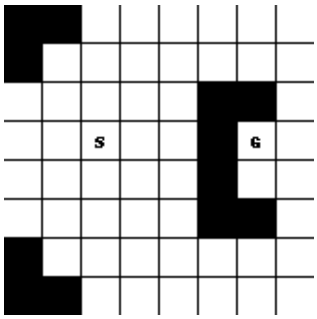


Figure 3: A search problem on a grid, with obstacles in black.

3.1 Best-first search

As in uniform-cost search, we will take our queue to be a priority queue, but this time the priority is given by h itself. The resulting algorithm, **best-first search** (or **greedy search**), greedily expands the node n with the smallest value of $h(n)$, i.e., the one with the smallest estimated distance to the goal. Thus, it repeatedly picks whichever node on the fringe is estimated to be closest to the goal, and expands that node.

Unfortunately, best-first search is not optimal, as is demonstrated in Figure 4. Often, the node which is seemingly closest to the goal is already the result of a long search path. For example, let's suppose we have a choice between expanding two nodes m and n, where the path to get to m is of length 8, and the heuristic value is 3, while the path to n is of length 19 and the heuristic value is 2. Best-first search would choose to expand n, because it looks closer to the goal. However, any resulting path must be of length at least 21, while expanding m might lead to a path of length 11. The problem with best-first search is that it only considers the expected distance to the goal, and this causes it not to be optimal.

By modifying the algorithm to also take into account the cost already incurred along a path, we can obtain an algorithm that is optimal (in the sense of finding minimum cost paths to the goal).

3.2 A^* search

We now describe the A^* **search** algorithm. This algorithm is due to Nils Nilsson, and is one of the most widely used algorithms in all of AI. It uses uses a priority queue like uniform cost search and best-first search, but the priority of a node n is now given by

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the total cost of the path from the root to node n, and $h(n)$ is the heuristic as before.

In other words, A^*'s priority value for a node will be our total estimated cost of getting from the initial state to n to the goal. Thus, the priority takes into account two costs:

- The cost to get to the node n, given by $g(n)$.
- The cost to get from n to a goal, which is estimated by $h(n)$.

Uniform cost search took into account only the first term, and being a blind search algorithm, can be very slow. Best-first search took into account only the second terms, and was not optimal.

For the sake of completeness, here is pseudocode for the A^* algorithm:

```
PriorityQueue q;
q.insert(initialState, h(initialState));
while (!q.isEmpty()){
  node = q.remove();
  if (goalTest(node)) return node;
  foreach (n in successors(node, operators))
    q.insert(n, g(n) + h(n));
}
return FAIL;
```

Let us now prove some key properties of A^* search For convenience, we have summarized most of the notation we'll use in this set of notes in Figure 5.

4 Optimality of A^* search

One basic question to ask about any search algorithm is whether it is **complete**. Recall that a search algorithm is complete if it is guaranteed to find some path to a goal state whenever such a path exists. Under fairly mild assumptions, we can show that A^* is complete. We will not present the proof here, but the interested reader is referred to Chapter 4 of the textbook.

A more interesting question is **optimality**. A search algorithm is optimal if it is guaranteed to find the least-cost path to a goal state, provided a path to the goal exists. To prove the optimality of A^* search, we first need a definition.

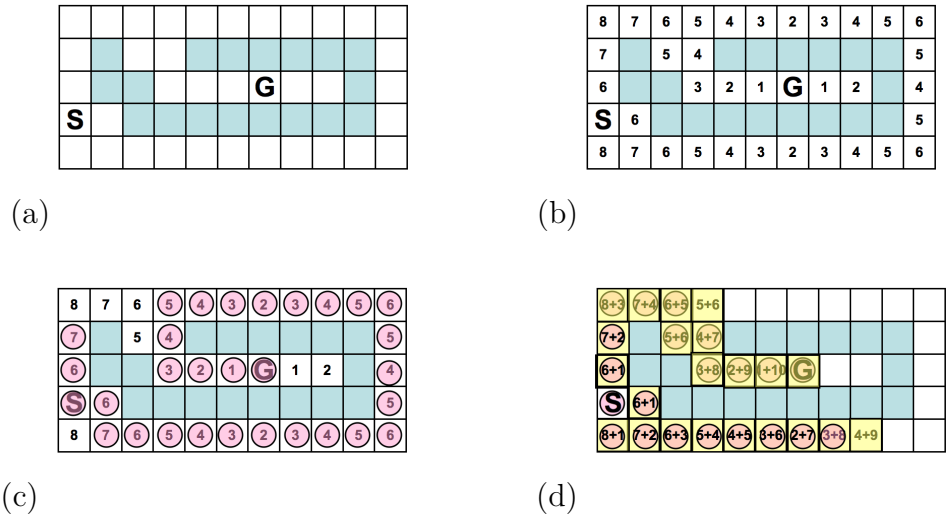


Figure 4: (a) An example of a search problem where the goal is to get from S to G. (b) The search space, labeled with the Manhattan distance heuristic. (c) The set of nodes expanded by best-first search. The solution is clearly suboptimal. (d) The set of nodes expanded by A^* search. (Note: the pair of numbers in each square represent the heuristic value $h(n)$ and the distance already traveled $g(n)$.)

s	a state in search space
n	a node in the search tree
n_g	A goal node
$g(n)$	The cost of the path to node n
$h(n)$	The heuristic function evaluated at n
$h^*(n)$	The actual cost of the least-cost path from n to a goal state
$f(n)$	The estimated cost of the least-cost path that goes from the root node through n to a goal, $f(n) = g(n) + h(n)$
$f^*(n)$	The actual cost of the least-cost path that goes from the root node through n to a goal, $f^*(n) = g(n) + h^*(n)$
$Pa(n)$	The parent of node n in the search tree.

Figure 5: Notation used in this lecture.

Definition. A heuristic function h is **admissible** if it never overestimates the true cost to get to the goal; in order words, if for any state s, we have $h(s) \leq h^*(s)$.

Theorem 4.1: *If h is an admissible heuristic function, then A^* search with h is optimal.*

Proof: Our overall strategy is to take n_g, the first goal node to be expanded, and show that it represents an optimal path. Since A^* returns the first goal node expanded, this implies the optimality of A^*.

Suppose h is an admissible heuristic, and n_g is the *first* goal node to be expanded. Since n_g is a goal and h is an admissible heuristic, we have

$$\begin{aligned} 0 &\leq h(n_g) && \text{since heuristic functions are nonnegative} \\ &\leq h^*(n_g) && \text{since } h \text{ is admissible} \\ &= 0 && \text{since } n_g \text{ is a goal} \end{aligned}$$

Hence, $h(n_g) = 0$, and therefore

$$f(n_g) = g(n_g) + h(n_g) = g(n_g). \tag{1}$$

We need to guarantee that the path to n_g is no longer than any potential path through some other unexpanded node. Suppose that (by expanding more nodes) it is possible to reach some other goal node n'_g other than the one n_g chosen by A^*. We will prove that $g(n'_g) \geq g(n_g)$.

There is some unique path from the root to n'_g. Let n be the *first* node on this path that has not yet been expanded. Note therefore that n's parent must have been expanded previously, and therefore n is on the fringe of our search tree at the instant that A^* expanded n_g.

Let $f^*(n)$ denote the minimum cost path to a goal that passes through n. Since n'_g is a goal node, and the path to n'_g passes through n, we must have

$$f^*(n) \leq g(n'_g).$$

So, by admissibility, we have

$$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f^*(n) \leq g(n'_g). \tag{2}$$

Furthermore, n is on the fringe, and was therefore on the priority queue (along with n_g) at the instant A^* expanded the goal node. But A^* chose

to expand n_g rather than n. Since A^* chooses nodes to expand in order of priority, this implies that

$$f(n_g) \leq f(n). \tag{3}$$

Putting together equations (1-3), we get

$$\begin{aligned} g(n_g) &= f(n_g) && \text{by Equation (1)} \\ &\leq f(n) && \text{by Equation (3)} \\ &\leq g(n'_g) && \text{by Equation (2)} \end{aligned}$$

This proves that $g(n'_g) \geq g(n_g)$ for any goal node n'_g other than the one chosen by A^*. This therefore shows that A^* finds a minimum cost path to the goal.

If this proof made sense to you, then as a studying technique to make sure you really mastered this, one thing you might consider trying is covering up the proof, and proving the theorem from scratch by yourself without referring to these notes. (This is a common studying technique for mastering proofs, and I still use it a lot when learning about new proof methods.)

4.1 Repeated states

In this section, we address a technicality regarding repeated states. As you saw in Figure 2, search trees can have repeated states, in which multiple nodes in the tree are labeled with the same state. In the most straightforward implementation of A* and other search algorithms, we would therefore end up carrying out searches from the same state multiple times, which is very inefficient.

In some problems (example in one of the questions in Problem Set 1), it is possible to formulate the search space so that there are no repeated states. I.e., each state in the discrete graph search problem is reachable only via a unique path from the initial state. In problems such as the 8-puzzle and in various "maze" or "grid search" problems (such as in Figure 3), repeated states are unavoidable, since the nature of the problem is that there are intrinsically many different paths to the same state.

Dealing with repeated states requires only a small change to our search algorithm (as stated in Section 2 and Section 3.2). Specifically, when previously the algorithm would insert a node n onto a queue, we would now modify the algorithm to consider three cases:

- If the node n is labeled with a state s that was previously expanded (i.e., if the search tree contains a previously expanded node n' that was

also labeled with the same state s), then discard n and do not insert it onto the queue.

- If the node n is labeled with a state s that is already on the queue—i.e., if the queue/fringe currently contains a different node n' that is also labeled with the same state s—then keep/add on the queue whichever of n or n' has a lower priority value, and discard whichever of n or n' has a higher priority value.[3]

- If neither of the cases above hold, then insert n onto the queue normally.

If you are familiar with Dijkstra's shortest paths algorithm, you can also check for yourself that making this change to uniform cost search results in exactly Dijkstra's algorithm.

If you are implementing one of these search algorithms for a problem that does have repeated states, then making the change above to the algorithm will usually make it *much* more efficient (since now it would expand each state at most once), and this should pretty much always be done.

4.2 Monotonicity

To hone our intuitions about heuristic functions, another useful property of heuristic functions is **monotonicity**. A heuristic function h is monotonic if (a) for any nodes m and n, where n is a descendant of m in the search tree, $h(m) - h(n) \leq g(n) - g(m)$, and (b) for any goal node n_g, $h(n_g) = 0$. (The second condition is necessary, because otherwise, we could add an arbitrary constant to the heuristic function.) We can rephrase condition (a) into something more intuitive: if n is a descendant of m, then $f(n) \geq f(m)$. In other words, the heuristic function is not allowed to decrease as we follow a path from the root.

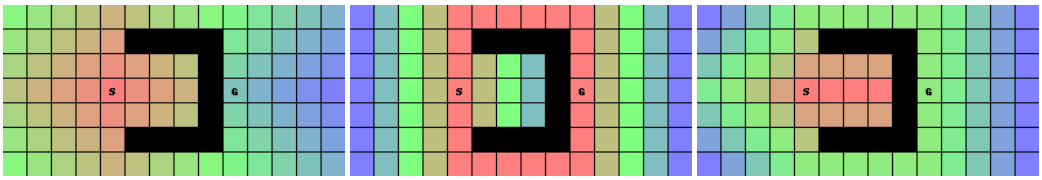
Monotonicity is a stronger assumption than admissibility, as we presently show.

Theorem 4.2: *If a heuristic function h is monotonic, then it is admissible.*

Proof: Let n_g be an arbitrary goal node, and let n be a node on the path to n_g. Then we have

$$\begin{aligned} g(n) + h(n) &= f(n) \\ &\leq f(n_g) \quad \text{by condition (a)} \\ &= g(n_g) \quad \text{by condition (b).} \end{aligned}$$

[3]For BFS and DFS which use FIFO and LIFO queues rather than a priority queue, we can discard n and keep n' on the queue.



(a) (b) (c)

Figure 6: Some examples of the value of $f(n)$ for different nodes during A^* search, when different heuristic functions are used. The value of $f(n)$ is shown via a colorscale, where small values are red, intermediate values are green, and large values are purple. (a) $h(n) = 0$, e.g. blind search. (b) $h(n) = h^*(n)$. (c) Manhattan distance.

By subtracting $g(n)$ from both sides, we get

$$h(n) \leq g(n_g) - g(n).$$

This must hold true for any n_g which is a descendant of n, and in particular the shortest one. Hence, $h(n) \leq f^*(n) - g(n) = h^*(n)$.

The converse is not true, however, and it is possible to create admissible heuristics that are not monotonic. But this difference is largely a formal one, and in practice, admissibility and monotonicity almost always coincide. Therefore, the intuitions we develop for monotonic heuristic functions can generally be applied to admissible functions.

Monotonicity is useful, because when it holds, it implies that A^* expands all of the nodes in increasing order by $f(n)$. This follows easily from the definition; when a node n is expanded, all of its descendants (by property (a)) must have larger values of f. Hence, it is impossible for f to decrease. This gives a convenient intuitive picture of the behavior of A^* search, as shown in Figure 6. A^* first expands the red squares, followed by the orange ones, followed by the green ones, and so on.

5 Heuristic functions

Up to this point, we have treated heuristic functions as a black-box concept which we feed into our algorithms. But how do we actually come up with good heuristics? This is often highly nontrivial, and there have been many research papers published that introduce clever heuristics for given domains. Finding heuristics is hard because we are trying to balance several desiderata:

- *Admissibility.* If we are truly interested in finding the least-cost path, we want only admissible heuristics. (In practice, if we're willing to settle for a suboptimal path, we often use inadmissible heuristics to hopefully get better computational efficiency.)

- *Accuracy.* We want our heuristics to estimate the distance as accurately as possible. When working with admissible heuristics, this means we want the values to be as large as possible (but no larger than h^*).

- *Computational efficiency.* If we come up with a very predictive heuristic, it is still no good if we can't compute it efficiently. This is particularly important, since heuristic functions are usually evaluated deep in the inner loop of a search algorithm.

These different criteria lead to a tradeoff when we design heuristic functions. Two extreme (admissible) examples are the zero heuristic $h(n) = 0$ and the oracle heuristic $h(n) = h^*(n)$, which returns the actual least-cost path to the goal. The former is trivial to compute, but provides no information; using it is equivalent to blind, uniform cost search. On the other hand, if we knew $h^*(n)$, the problem would be trivial because we could head straight to the goal. Computing it, however, is as hard as solving the search problem itself. Clearly, any useful heuristic will have to lie somewhere in between these extremes. Figure 6 shows an intuitive picture of how different heuristics behave in a least-cost-path problem.

Constructing heuristics is not always easy, but there is a very useful way of thinking about the problem that often generates good heuristics. Specifically, we can often construct heuristics by thinking about solving a **relaxed problem**. In other words, we *eliminate one or more constraints* from the problem formulation. Our heuristic $h(n)$ will then be the minimum cost of getting from n to a goal, under the assumption that we do not have to obey the eliminated constraint(s). Such a heuristic will always be admissible. (Why?)

As a concrete example, recall our definition from the previous lecture of **Manhattan distance**, which measures the city-block distance from a state to the goal, *ignoring obstacles*. Basically, we eliminated the constraint that our path could not pass through an obstacle. In other words, $h(n)$ is the *exact* cost of getting from n to the goal if we were to eliminate the "you can't go through obstacles" constraint. Note that whereas computing costs $f^*(n)$ in the original problem was hard (and required performing a search), computing optimal costs when this constraint has been eliminated is easy. Analogously,

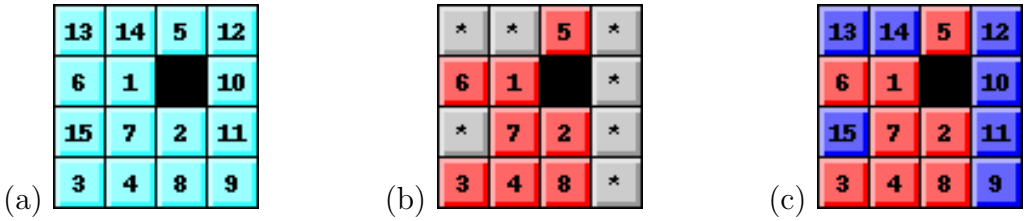


Figure 7: The heuristic for the 15-puzzle, defined as the number of moves required to move a subset of pieces into the correct positions. The number of possibilities to consider is small enough that they can all be cached in memory. (a) A possible board configuration. (b) The subset of tiles considered to compute the heuristic. Asterisks denote "don't care" tiles for the purpose of computing this heuristic. (c) An even better heuristic is if we take the maximum of the times required to get either the red tiles or the blue tiles into their goal positions.

if we are searching in a continuous space, eliminating the constraint that we can't pass through obstacles gives us the **Euclidean distance** heuristic.

More generally, even if we have an NP-hard or combinatorially large problem, eliminating one or more constraints often makes the problem trivial to solve exactly, which allows us to compute a heuristic value very quickly. For a more involved example, let us turn to the 15-puzzle (the 4x4 version of the 8-puzzle from last lecture). Recall that we can move a tile from A to B if (i) A and B are adjacent, and (ii) B is blank. Here are two examples of heuristics we can generate by eliminating constraints:

- If we eliminate (ii), we get the total number of moves the tiles have to make, if we could move them all independently without any tile getting into another tile's way. In other words, we get the sum of the Manhattan distances of all of the tiles to their final locations.
- If we eliminate both (i) and (ii), we get the number of tiles out of place. Note that, because we've eliminated more constraints, the value of this heuristic will be strictly smaller than the previous one. Since the heuristics are admissible, this is equivalent to saying it is always less accurate.

In the case of the 15-puzzle, it turns out to be possible to create an even better heuristic than these. Specifically, we will choose a subset of tiles, and pre-compute the minimum number of moves needed to get this subset of tiles from any initial configuration to their goal states. Our heuristic will ignore

all the tiles except for this subset. (Thus, it corresponds to eliminating the constraint that the tiles not in this subset must also be moved to the goal state.) This is illustrated in Figure 7. Unlike Manhattan distance, this heuristic takes into account interactions between tiles—i.e., that you might need to get one of the tiles (in the subset) "out of the way" before you can move a different tile to its goal. Note also that, no matter which subset of the squares we choose to compute this heuristic on, we get an admissible heuristic. Thus, we can actually partition the tiles into two subsets, say the red and blue subsets show in Figure 7c, and compute a heuristic value for the red subset, and a heuristic value for the blue subset. In other words, we pre-compute the minimum number of moves needed to get the red tiles into their goal positions (ignoring all other tiles); and we also pre-compute the minimum number of moves needed to get the blue tiles into their goal positions. The final heuristic value that we'll give to A^* is then the *maximum* of these two values.[4] (Why not the sum? Why not the minimum?) This approach, called **pattern databases heuristics**, works very well for the 15- (and 24-) puzzle. It also gave the first algorithm that could find optimal solutions to randomly generated Rubik's cube puzzles. (In the Rubik's cube problem, we would precompute the minimum number of moves needed to get different subsets of the Rubik's cube facets into the goal positions).

Note the tradeoff here between the cost of computing the heuristic function and how much it saves us in the number of nodes we need to search. The approach described above is implemented by generating all possible board positions where only the selected subset of tiles (say the red tiles) are treated as distinct, finding the shortest paths for all configurations of this subset of tiles to their goal positions,[5] and caching the results. By including larger subsets of the tiles in this set, we could make the heuristic more accurate (closer to h^*), but this would increase the computational cost of computing the heuristic, as well as the memory requirements of storing it. We could also use a smaller subset of tiles, which would give a less accurate heuristic, but be cheaper to compute and store. More generally, there is often a tradeoff, where we can either find a more expensive, better heuristic (which would cause A^* to need to perform fewer node expansions before finding the goal), or use a heuristic that's cheaper to compute (but result in A^* needing to do more of the work in the sense of expanding more nodes). The heuristics $h(n) = 0$ and $h(n) = h^*(n)$ represent two extremes of this tradeoff, and it'll

[4]More generally, for other search problem as well, if we have several admissible heuristics, we can take the maximum to get an even better admissible heuristic.

[5]It turns out this can be done efficiently by running one big breadth first search algorithm backwards from the goal state.

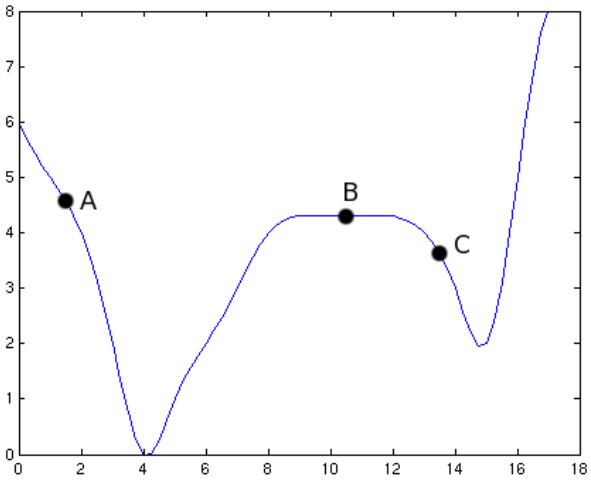


Figure 8: A cartoon example of greedy hill-climbing. The horizontal axis corresponds to the set of states. Starting from point A, the algorithm would take tiny steps downhill until it reaches the global minimum. Starting from the plateau at B, it would not know where to go because the terrain is flat. Finally, starting from C, it would move downhill and get stuck in a local optimum.

usually be something in-between that performs best.

6 Greedy hill-climbing search

Best-first search, or greedy search, expanded nodes in order of the heuristic $h(n)$. It was not optimal, because it was too greedy. Actually, it turns out that there is an even greedier algorithm, that's even more aggressive in terms of trying to get to the goal as quickly as possible. We now describe **greedy hill-climbing search** (called hill-climbing in the textbook). This algorithm throws caution to the winds and simply tries to minimize h and head straight to the goal as quickly as possible, with *no backtracking*. In other words, from a state s, we will choose whichever successor has the lowest heuristic value, and simply "move" there. Figure 8 shows a cartoon example of greedy hill-climbing search. (Given that we're actually going downhill, perhaps this is better called "hill-descending" search, but the terminology we're using is quite standard.)

Note the difference between best-first search and greedy hill-climbing

search: while best-first search backtracks, greedy hill-climbing search does not. In other words, the search tree is really more of a long "chain"-structured graph. Generally, we don't even think of greedy hill-climbing as building a tree; rather, we think of it as just hopping from one state to another. The fact that the algorithm only has to "know" about the current state and its immediate successors is often referred to as **locality**. This property gives the algorithm great power, since it can be used even when some of our earlier assumptions for path-finding are violated. For example, it works for robot motion planning even if we don't know the locations of all of the obstacles in advance, or if the obstacles can move around.

There are two major problems commonly associated with hill-climbing search:

- **Local optima**. This is where we get to a state where all of the possible moves increase the heuristic function (make it worse). Because the algorithm doesn't backtrack, it gets stuck. Because of local optima, hill-climbing search is usually not complete.

- **Plateaus.** The algorithm reaches a plateau when a move in any direction causes the heuristic function to remain the same. Thus, the heuristic offers no information locally about what's a good direction to head in, and we may wander around for a long time on a large plateau before we get lucky and find a way downhill again. As a concrete example, suppose that in the 15-puzzle, we define $h(n)$ to be the number of tiles out of place. There will then often be no single move (or even any short sequence of moves) which changes the value of h. Thus, the heuristic offers no indication of where to go next, and it's hard to make progress towards the goal.

A cartoon illustration of these problems is shown in Figure 8.

Don't be fooled by Figure 8 into thinking that plateaus are easy to deal with. Many local search problems can involve search spaces in thousands of dimensions, and therefore plateaus can be exponentially large. Even if the plateau does have a way out, it might take so long to explore the plateau that we never find it.

6.1 Potential fields

Since we're using greedy hill-climbing search, we've already given up on optimality. Therefore, there's no need to make sure the heuristic h is admissible. In fact, there's no need to even choose h to be an estimate of the cost to the

goal—it can be anything you want, so long as going downhill on h tends to move you closer to the goal.

In motion planning, there's a particularly effective class of methods called **potential fields**, pioneered by Oussama Khatib. Concretely, we will define an **attractive** potential (in configuration space) that tends to pull us towards the goal, and a set of **repulsive** potentials that tend to push us away from the obstacles. Our robot will then perform local greedy hill-climbing search on an overall potential function that's given by the sum of all of these attractive and repulsive potentials, to try to get to areas of lower and lower potential, and hopefully all the way to the goal. An example of this is given in Figure 9.

Potential fields are extremely easy to implement, and are very space efficient because they don't need to maintain a priority queue. Furthermore, while even A^* eventually faces exponential blowup for large enough problem sizes, potential fields can sometimes scale up to very large problem sizes, because all we need to keep track of is one current state. Finally, it applies even when the obstacles may move and we don't in advance how or where they'll move. For example, in class we saw a robot soccer demo, implemented with potential fields, where the robot was attracted to the ball, and repulsed by the opposing team's robots. Even though we don't know where the opponent robots will move next, at any instant in time, all we have to do is place a repulsive potential around where they are currently, place an attractive potential centered on the ball, and take a local greedy hill-climbing step; this will tend to cause our robot move towards the ball while avoiding the other robots.

We also saw in the soccer demo that by adding other terms to the overall potential field, we could quite easily get the robots to exhibit fairly complex behavior. For example, we could use an attractive potential to the ball that's strongest for the player nearest it (so that essentially only one player at a time goes after the ball); we also saw "player position fields" that cause the players to be attracted to different areas of the soccer field, that correspond to various soccer positions, such as defender, right field, center, forward, left field; we also had a "kicking potential field" that causes the robot to be attracted to the appropriate side of the ball so as to kick it towards the opponent goal (and not towards our own goal); and so on. The combination of these relatively simple potential fields led to very rich, effective, soccer playing behavior, with each robot automatically taking into account many factors such as where the ball is, where the teammates are, where the opponent is, etc., and coordinating with its teammates to play soccer impressively well.

One disadvantage of potential fields is that, like other hill climbing search algorithms, we can get stuck in local optima. Sometimes, you might code up

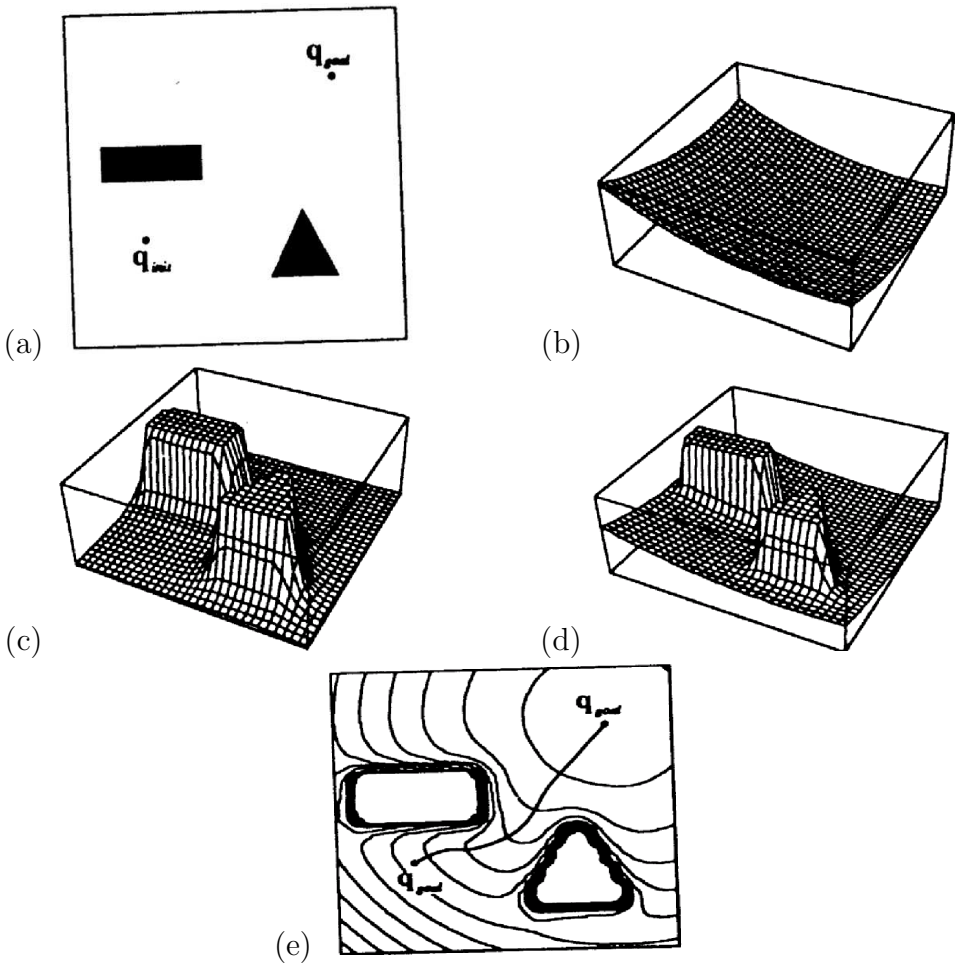


Figure 9: An example of potential fields. (a) The configuration space in a path planning problem. (b) The attractive potential which pulls the robot towards the goal. (c) The repulsive potential which keeps the robot away from obstacles. (d) The total potential. (e) The path the robot winds up following. Images courtesy of Jean-Claude Latombe.

a potential field, hand-tune it slightly and it'll give you exactly the behavior you wanted (as in the soccer example). Other times, it's also possible that if the robot keeps getting stuck in local optima (or keeps running into some other problem), and you may end up spending weeks tuning the potential functions by hand (fiddling with the strength of the attractive vs. repulsive potentials, etc.) and still not get the robot to do what you want. Thus, one disadvantage of potential fields is that if an implementation using a complex combination of potential fields does not work, it can sometimes be extremely difficult to figure out how to modify things to make it work.

Nonetheless, their scalability to huge problems, and ease of implementation, makes potential fields a very good choice for many problems. In class, we also saw a hybrid application of potential fields (together with PRMs) to plan the motions for a character in a computer graphics animation (this was the video of an animated character playing chess with a robot). That example had an extremely large state space—it used 64 degree of freedom, with each axis discretized to 100 values—but was still amenable to a potential fields solution.

6.2 Optimization search

Up until now, we have been considering a search formulation where we are interested in finding a low-cost path from the initial state to a goal. We now describe a different class of search problems, where we don't care about the cost of the path we take, but only about the final state we wind up in.

In an **optimization search** problem, we have some set of states that we search over. Furthermore, we have a cost function we want to minimize (or a goodness function we want to maximize). Usually the states will all correspond to valid solutions to some problem, and our goal is to find a good solution; in other words, a state with minimum possible cost. The solution is therefore defined *implicitly* as the lowest-cost state. In fact, sometimes it's possible for an algorithm to have already found the minimum cost state, but to have no way to realize it. Furthermore, we also don't care about the path we take to get to this state; we care only about the final state we find.

Local search methods work quite well for many optimization search problems, especially in cases where we are willing to settle for a "pretty good" (in the sense of minimizing cost), but not necessarily optimal, solution. Let us consider some examples.

6.2.1 Traveling salesperson problem

The traveling salesperson problem (TSP) is one of the classic NP-hard problems. Consider a salesperson who starts at a given city, and has a list of cities to visit. His task is to visit all of these cities, and then return to the starting city. Furthermore, he is a dishonest salesperson, and therefore can't return to the same city twice, lest he face his angry customers. The problem is to find a circuit of these cities, or **tour**, that has minimum cost.

More formally, we are given a graph G, which we assume is complete (but may possibly have edges with infinite cost). To each of the edges in G is assigned a cost. We are interested in finding the minimum-cost tour that visits each vertex exactly once, and returns to the city where we started. Since TSP is NP-hard, we shouldn't expect to find a polynomial time algorithm which guarantees finding the optimal solution. Nevertheless, local search methods often work well for finding good solutions.

Note that this problem can be expressed as a path planning problem and solved exactly using A^* search. More specifically, the states would correspond to partial tours, and each operator would extend the tour by one city, with the cost function giving the cost of the edge we just added. It is even possible to even define good admissible heuristics. Since A^* is optimal, this will eventually find the optimal solution. Since the problem is NP-hard, however, this suggests that A^* will be exponential in the worst case. In practice, we don't necessarily need to find the *best* tour, but would rather get a good tour quickly. We will use greedy local search to do this, by posing an optimization search problem.

To formulate the TSP as an optimization search problem, we need to define states, operators, and a cost function. There are many possible formulations, but here is one:

- *States:* complete tours (that visit all the cities).
- *Operators:* taking two adjacent cities in the tour and flipping the order in which they are visited.
- *Cost:* the sum of the edge costs of the edges in the tour.

Applying greedy hill-climbing search in this search space to minimize the cost function gives a nice intuitive picture, where the tour is incrementally tweaked (swapping two cities) until it gets to a local optimum—a point where the solution can't be made better by swapping two more cities.

6.2.2 Other examples

There are many other problems that can be posed as optimization search.

- **Machine Learning.** Many machine learning algorithms are formulated by posing an optimization problem, and solving it. For example, suppose we would like to have a learning algorithm estimate the prices of various houses, based on properties of the houses such as its size, etc. In the **linear regression** model (which we'll see in detail in about a week), we will write down a cost function that corresponds to how accurate our algorithm's estimate of housing prices are. We then apply a greedy hill-climbing search algorithm to minimize the errors in our algorithm's predictions.

- **8 queens.** One famous toy problem is the 8 queens puzzle. In the game of chess, a queen can attack any piece which shares a row, column, or diagonal with it. The goal is to place 8 queens on an 8×8 chessboard such that no queen can attack any other queen. Local search methods have been shown to work surprisingly well for this problem. In one possible formulation, each state corresponds to an assignment of all of the queens to squares on the chess board, and the cost function is defined as the number of queens under attack. Our operators can take one of the queens, and move it elsewhere on the board.

- **Building an automobile.** In order to build a car in an automobile plant, we must decide on the order in which parts will be fabricated, finished, painted, combined, and so on. We must schedule jobs on different machines, each of which needs to work on different parts for a different amount of time. The aim is to maximize the output of a factory, so this is an optimization search problem. By applying simulated annealing (a kind of local search) to the car design problem, an engineer at General Motors found a way to save $20 per automobile.

6.3 Propositional satisfiability

One problem which has received a huge amount of attention is **propositional satisfiability**, also known as **SAT**. This is an NP-hard problem — in fact, the original NP-hard problem — but can often be solved efficiently in practice. Here, we are given a sentence in **propositional logic**; in other words, a sentence built out of a set of propositional variables $A_1, \ldots, A_n$, each of which can be either true or false, and the propositional connectives

$\wedge$ (and), $\vee$ (or), and $\neg$ (not). For example, the following sentence is true if and only if A and B are either both true or both false:

$$(A \wedge B) \vee (\neg A \wedge \neg B)$$

Given a sentence in propositional logic, our goal is to determine if there is a **satisfying assignment**, meaning an assignment of true or false to each of the variables which causes the sentence to evaluate to true.

SAT is an important problem because many other problems in AI, such as planning, can be solved by turning it into a huge SAT problem. In fact, one effective way to solve many other AI problems is by converting them into a big SAT problem, and feeding it to a state-of-the-art SAT solver. This takes advantage of the immense effort which has already been spent engineering efficient SAT solvers, and has the added benefit that we get a free performance boost when newer and better SAT solvers are released.

When we discuss SAT algorithms, we consider a particular type of propositional sentence called **conjunctive normal form** (CNF). This form is popular because it significantly simplifies the process of designing SAT algorithms, yet there exists a straightforward procedure to translate any propositional sentence into CNF.

To define CNF, we first introduce some terminology:

- A **literal** is a single variable, possible negated. For example, A_3 or $\neg A_5$.
- A **clause** is a disjunction ("or") of literals. In other words, it is true if any one of the literals is true. For example, $A_3 \vee \neg A_7 \vee A_8$.

A CNF sentence is a conjunction ("and") of clauses. Therefore, a CNF sentence evaluates to true if and only if at least one of the literals is satisfied in each clause. Here is an example CNF sentence:

$$(A_1 \vee A_3 \vee \neg A_4) \wedge (\neg A_1 \vee A_2 \vee A_3) \wedge A_5$$

Since our goal is for the number of unsatisfied clauses to equal zero, we can formulate it as a local search problem in the following way:

- *States:* complete assignments to all of the variables
- *Operators:* flipping the truth value of a single variable
- *Cost:* the number of unsatisfied clauses

Greedy hill-climbing search for SAT, also known as GSAT, iteratively flips the variable which causes the greatest net gain in the number of satisfied clauses. Ties are broken randomly. Despite its simplicity, GSAT is a surprisingly effective SAT algorithm. There're also many other algorithms (such as WalkSAT) which implement various improvements to GSAT, and work even better. The current state-of-the-art can solve often problems with tens of thousands of variables, which is quite impressive for an NP-hard problem.