

Lectures Notes on Search

Alexander Rush

Contents

1 Introduction

A core component of reasoning is making choices in constrained worlds, and in particular deciding on the best action to make based on current information. In AI, we **model** these scenarios by creating abstract worlds and formally defining the possible actions that may occur. These worlds allow us to explore various **algorithms** for selecting actions in these environments. The two components mutually inform each other. The model provides the framework over which the algorithm functions; and the availability of algorithms informs our construction of the model itself.

Search will be our core focus for the first several lectures and a central tool for the first half of this course, and it exemplifies this split. We will begin by defining an abstract representation of our world, utilizing states and actions, and defining explicit goal. Once we have transformed our problem into this form, we can treat search as a blackbox set of algorithms for answering queries about this world. Even better if we can provide further information about the structure of the problem we will be able to improve these algorithms by providing heuristics. Search will also be an important general tool for the next three topics: game-playing, constraint satisfaction, and logical inference.

2 A Running Example

Our running example for search will be path-finding over a graph. This algorithm is central to many consumer applications, such as Google Maps, as well as commercial routing systems.¹ In the path finding problem, we assume that we start at a position (in this case A) and have to reach a goal (in this case C) by traversing the edges in the graph, each of which have an associated length. The best path will be the shortest one to the goal.

This represents one instance of a search problem, but we will see many others throughout the class. We start by defining the general representation we will use for each of these problems.

¹If you have taken CS 124, this setup will look very familiar, although the notation and vocabulary for AI is slightly different.

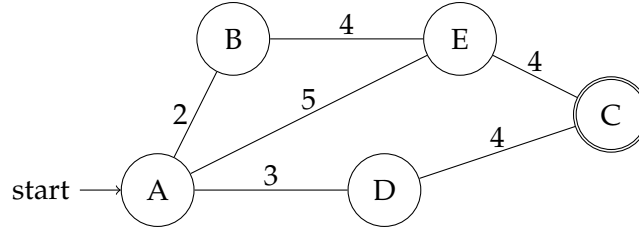


Figure 1: A small path-finding problem on a graph.

3 A Model for Search

We use the term **model** informally to mean an abstract representation of the world. Practically this can be thought of like an interface for which we act on the world. In doing so we filter specific details from the underlying problem, but keep the necessary structure.

3.1 Search Model

We define the search model using 6 elements. Throughout course we will repeatedly return to this model for many other problems:

Name (AIMA)	Type	Description
State space	\mathcal{S}	The set of all possible states of the world.
Action space	\mathcal{A}	The set of all possible actions of the world.
Action model	$\text{ACTIONS} : \mathcal{S} \mapsto 2^{\mathcal{A}}$	Specifies which actions are applicable at a given state.
Transition model	$\text{RESULT} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$	Specifies the new state resulting from taking an action at a state.
Initial state	$s_0 \in \mathcal{S}$	The starting state of the world.
Goal test	$\text{GOAL} : \mathcal{S} \rightarrow \{0, 1\}$	Predicate indicating if a state is a goal.

How do we use this model? Well, we start from state s_0 , at this state we are allowed to take any action $a \in \text{ACTIONS}(s)$, doing so changes the state to $s_1 \leftarrow \text{RESULT}(s, a)$. We proceed until we reach a state with $\text{GOAL}(s) = 1$. We refer to this sequence as a path, formally:

Definition 1 A *path* consists of a sequence of state-action pairs beginning from the initial state s_0 , i.e.:

$$p = (s_0, a_0), (s_1, a_1), \dots, (s_{n-1}, a_{n-1}).$$

Where $s_i = \text{RESULT}(s_{i-1}, a_{i-1})$ and the depth of the path to be n . We define the last state of the path to be

$$p_{\text{last}} \triangleq \text{ACTIONS}(s_{n-1}, a_{n-1}).$$

Define the set of all paths as \mathcal{P} .

Definition 2 A path is said to be a **solution** if it reaches a goal state, that is if $\text{GOAL}(p_{\text{LAST}}) = 1$. Define the set of solutions as:

$$\mathcal{Q} = \{p \in \mathcal{P} : \text{GOAL}(p_{\text{LAST}}) = 1\}$$

3.2 Search with Costs

We will often want to extend the search definition to include **costs** associated with each state-action pair.

Step cost $c : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}_+$ Gives the positive cost of taking an action at a state.

Path cost $g : \mathcal{P} \mapsto \mathbb{R}_+$ Gives the cost of a path, i.e. $g(p) = c(s_0, a_0) + c(s_1, a_1) + \dots + c(s_{n-1}, a_{n-1})$.

Generally we hope to find the lowest cost, or optimal, solution. Define an **optimal solution** as:















$$p^* = \arg \min_{p \in \mathcal{Q}} g(p)$$

where \mathcal{Q} is the set of all solution paths. We define $C^* = g(p^*)$ as the cost of the optimal solution.

3.3 Example: Back to Path Finding

Consider the graph problem in Figure 1. We assume that our goal is to get from state A to the goal state C. Here's how this problem looks in terms of the search formalism.

Question 1 What is the search model for path finding problem?

Name	Value	Description
State space		
Action space		
Actions		
Transition model		
Initial state		
Goal test		
Step cost		

3.4 Properties of a Search Problem

A search problem has several associated properties that will help us quantify the difficulty of finding a solution or the optimal solution.

Name (AIMA)	Type	Description
Branching Factor	b	The maximum number of actions from any state.
Min Depth	d	The solution with the smallest depth (number of actions).
Max Depth	m	The solution with the largest depth (number of actions).
Optimal Cost	C^*	The path cost of the optimal solution.

Note that the solution with min-depth may not be optimal (if all costs are not 1). Also for many problems we will not know the true optimal cost, d or m , or even b .

Question 2 What are the search properties b, d, m, C^* for the path finding problem? Shallowest goal? Deepest goal? Branching factor?

Name	Type	Description
Branching Factor	b	⊛
Min Depth	d	⊛
Max Depth	m	⊛
Optimal Cost	C^*	⊛

3.5 Example 2: Machine Translation

Path-finding over graphs is one of the key uses of search, but it is by no means the only one. A major benefit of the search abstraction is that it can be used as a common language for many problems in AI.

Consider for example, a very different problem: finding the best translation of a Spanish sentence into English. The following figure shows an example of this problem.

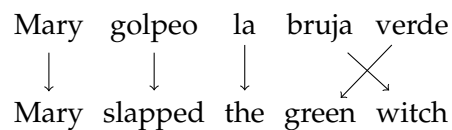


Figure 2: Example Spanish-English translation problem and the expected English output

For this problem we use the following (informal) abstraction:

Name	Value	Description
State space	S	Which words have been translated and the current English translation.
Action space	\mathcal{A}	Translate a word .
Actions	ACTIONS	All possible ways to translate any untranslated word.
Transition model	RESULT	Add a new word to the English translation and remove the translated word from the source.
Initial state	s_0	Translation is blank, all words are untranslated.
Goal test	GOAL(s)	Any state with all words translated.
Step cost	c	How good a translation is this action? How good is the translated output?

Figure 3 shows example states from the machine translation search world.

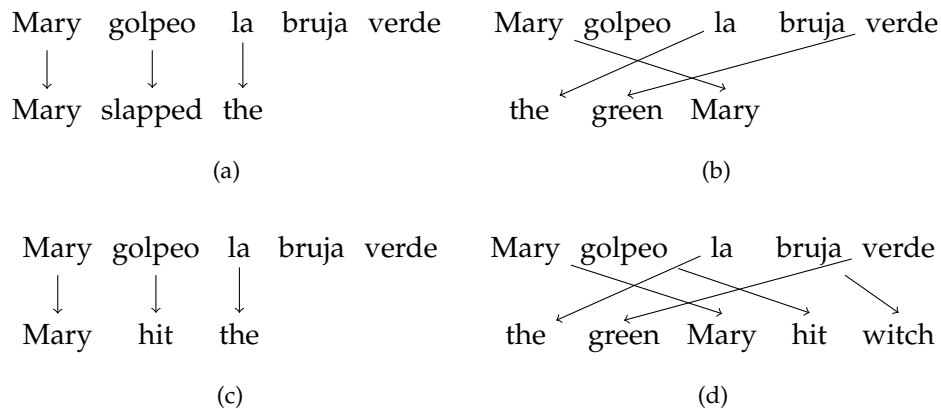






Figure 3: Four different states of the translation search problem. (a) is partially correct, (b) has the wrong order, (c) makes an incorrect translation, (d) is a non-optimal solution.

Question 3 What are the search properties b, d, m for the machine translation problem? Shallowest goal? Deepest goal? Branching factor?

Name	Type	Description
Branching Factor	b	
Min Depth	d	
Max Depth	m	
Optimal Cost	C^*	

4 Search Algorithms

Once we have described our problem using the search model, we can apply a search algorithm to find a solution. We will consider two types of search algorithms, uninformed and informed. Both make use of the same algorithmic skeleton which we discuss in this section.

A search algorithm explores the set of paths \mathcal{P} based only on the abstract problem structure. At each stage the algorithm has subset of available paths, called the **frontier**. The algorithm will expand this frontier to find new paths until a goal state is expanded. However, we will see that the ordering of this process greatly affects which solutions are found and how long it takes to find them.

4.1 Review: Abstract Containers

Just a quick review. We will define a container as an abstract data structure with three operations: `push`, `pop`, `empty`. Informally `push` adds an object to the container, `pop` removes and returns an object, and `empty` tells us if there are any objects remaining. We say it is **abstract** because we do not specify the order elements are popped or the internal representation.

One container we will look at is called a stack. Briefly here is how it looks in Python. It implements `push` by adding an element to the back of a list, and `pop` by removing and returning the last element. Because of this behavior it is called a Last-in First-out (LIFO) container.

```
class Stack:
    def __init__(self): self.data = []
    def push(self, element): self.data.append(element)
    def pop(self): return self.data.pop()
    def empty(self, element): return bool(self.data)
```

4.2 The Algorithms

All search algorithms utilize the following algorithm:

The algorithm start with s_0 in the frontier. At each stage it expands one path from the frontier and pushes it children onto the frontier. It ends when a goal state is found.

Crucially we are using **frontier** as an abstract container. By changing the definition of `push` and `pop` of this container we will get different ordering strategies. This leads to very different search algorithms.




```

1: procedure TREESEARCH(frontier)
2:   frontier.push(Path( $s_0, \epsilon$ ))
3:   while frontier is not empty do
4:      $p \leftarrow$  frontier.pop()
5:      $s \leftarrow p_{\text{last}}$ 
6:     if GOAL( $s$ ) then return  $p$ 
7:     for  $a \in \text{ACTIONS}(s)$  do
8:        $s' \leftarrow \text{RESULT}(s, a)$ 
9:       frontier.push( $p$  extended by  $(s, a)$ )

```

Note that for search on problems like path finding, the search algorithm may end up having paths that wrap around to previous states. We can prevent this by adding some memory to prevent the algorithm from looping. This modification leads to an algorithm called graph search:

```

1: procedure GRAPHSEARCH(frontier)
2:   frontier.push(path starting with  $s_0$ )
3:   
4:   while frontier is not empty do
5:      $p \leftarrow$  frontier.pop()
6:      $s \leftarrow p_{\text{last}}$ 
7:     if GOAL( $s$ ) then return  $p$ 
8:     
9:     for  $a \in \text{ACTIONS}(s)$  do
10:       $s' \leftarrow \text{RESULT}(s, a)$ 
11:      if  then frontier.push( $p$  extended by  $(s, a)$ )

```

GRAPHSEARCH will be necessary for some search problems. However, remembering previous states can lead to a larger cost in terms of memory, so TREESEARCH is also used.

We can make properties of the search algorithms, like memory, more explicit. For every search algorithm we discuss, we will consider the following four properties.

- Completeness; Is the algorithm guaranteed to find a solution?
- Optimality; Is the algorithm guaranteed to find the optimal solution?
- Time Complexity; What is the worst-case running time of the algorithm?
- Space Complexity; What is the worst-case memory usage of the algorithm?

5 Uninformed Search

Uninformed search algorithm assume that we are given only the search model and no further information. We will consider four variants of uninformed search: breadth-first search, depth-first search, depth-limited search, and uniform-cost search. Each corresponds to a different instantiation of the frontier data structure and leads to very different search algorithms.

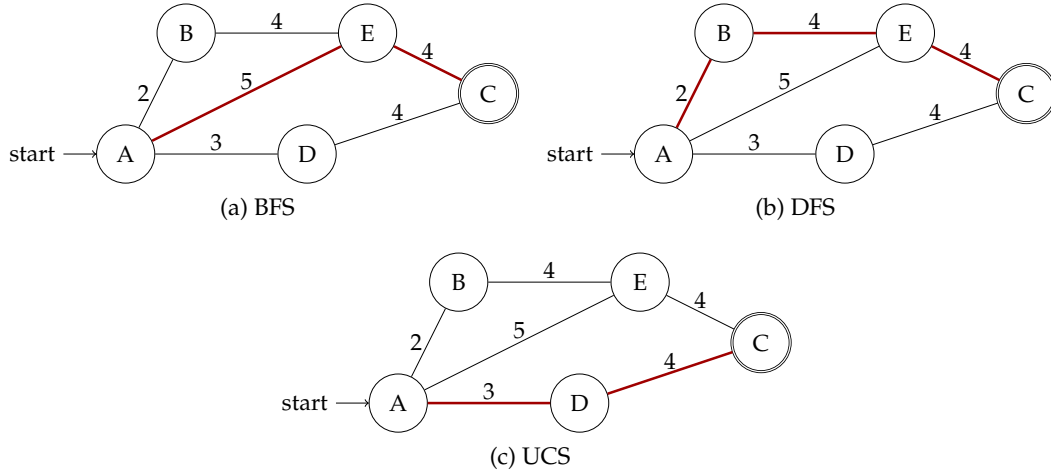


Figure 4: Paths returned by running three different search algorithms.

5.1 Breadth-First Search

Breadth-first search (BFS) is the most cautious search algorithm. It fully explores each layer of states before moving on to any deeper states. Breadth-first search does this by defining the frontier as a first-in first-out (FIFO) queue. This means that paths added to the queue will wait until every other path is expanded before they are expanded.

Let's consider running BFS on the graph in Figure 1, which will lead to discovering the goal path shown in Figure 4a. The following is the process of the algorithm at each iteration.

iter	p	s	frontier by p_{last}	explored
0	-	-	⊛	⊛
1	A	A	⊛	⊛
2	A:B	B	⊛	⊛
3	A:E	E	⊛	⊛
4	A:D	D	⊛	⊛
5	A:B:E	E	⊛	⊛
6	A:E:C	C	-	-

BFS explores every path of each depth in order. How many paths are there up to depth l ? Well in the worst-case there are $O(b^l)$. If the shallowest goal node is at depth d , BFS will search $O(b^d)$ states.

In terms of the properties of the algorithm, this means:

- **Completeness:** Yes. Finds a solution after $O(b^l)$ steps.
- **Optimality:** No. Optimal solution may be deeper than d . However BFS is optimal when costs are all 1.
- **Time-Complexity:** $O(b^d)$
- **Space-Complexity:** $O(b^d)$ (size of frontier at depth d).

5.2 Depth-First Search

Depth-first search (DFS) takes the opposite tack. Whenever possible, it explores a deeper path than in last iteration. This results in a greedy approach that is always hoping to find a goal quickly. DFS utilizes a last-in first-out (LIFO) stack for the frontier. The next path expanded always comes from the last path that had available actions. This leads to exploring deeper and deeper paths.

Consider again the graph in Figure 1. Running DFS will lead to the path in Figure 4b. As with BFS here is the run of the algorithm:

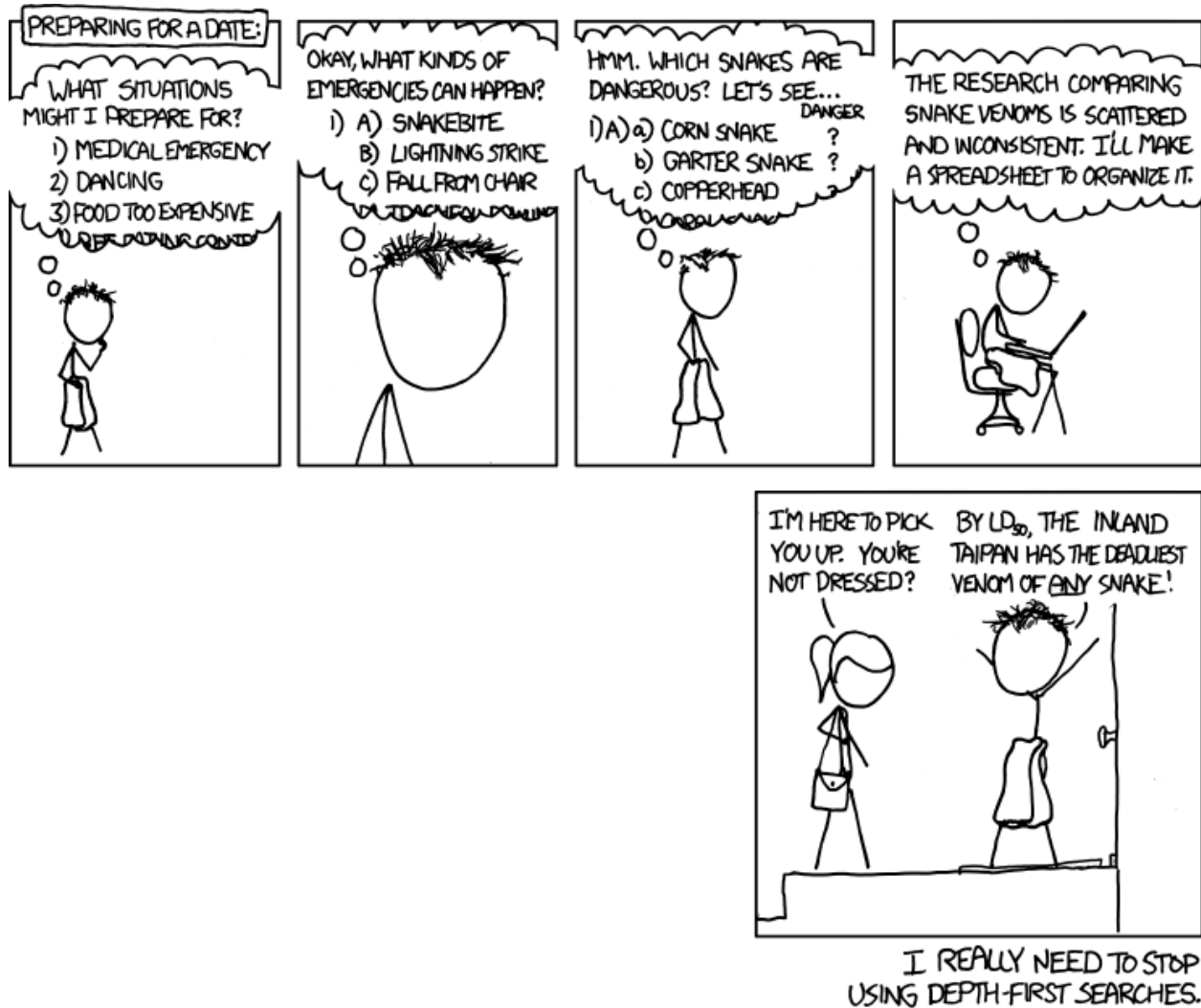
iter	p	s	frontier by p_{last}	explored
0	-	-	⊗	⊗
1	A	A	⊗	⊗
2	A:B	B	⊗	⊗
3	A:B:E	E	⊗	⊗
4	A:B:E:C	C	-	-

DFS has pretty poor search properties. If there is an infinite path it can go down it forever. Also it can find paths that are much deeper than m since it doesn't explore in depth order. And we have seen in the example that it is non-optimal.

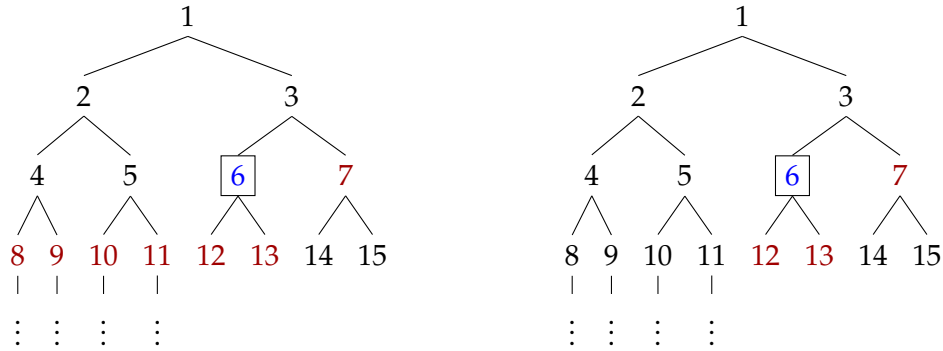
In terms of the properties of the algorithm, this means:

- Completeness: No.
- Optimality: No.
- Time-Complexity: $O(b^m)$
- Space-Complexity: $O(b^m)$ for graph search but only $O(bm)$ for tree search.

XKCD puts these issues into perspective.



Despite all these issues, we will very often elect to use DFS! The main advantage is its very good space complexity for tree search versus BFS. Consider the following example with $b = 2$. The red nodes indicate the current frontier after expanding the blue node (6). For BFS (left), every node at this depth is being added to the frontier. For DFS (right), we at most have b nodes on the frontier for each depth so far. This means that DFS with Tree-Search requires far less memory than BFS, making it more practical in many settings.



5.3 Variants of DFS

Depth-Limited Search The main issue with DFS is that it can go down paths much deeper than the deepest solution m . Depth-limited search (DLS) is identical to DFS, except that we modify the stack delete any paths beyond a fixed depth-limit l .

By doing this we limit the space of paths it can possibly explore to $O(b^l)$ and thus the complexity is improved. However, if $l < d$ the algorithm is still not complete.

- Completeness: No.
- Optimality: No.
- Time-Complexity: $O(b^l)$
- Space-Complexity: $O(b^l)$ for graph-search but only $O(bl)$ for tree-search.

Note that we will implement a variant depth-limited search in the next section on game playing.

Iterative Deepening Search However, we can take this approach one step further. In iterative deepening search we repeatedly run DLS with $l = 1, \dots$. By doing so, l will eventually reach d and we will find a solution.

- Completeness: Yes.
- Optimality: No. (Like BFS, optimal when all costs are 1)
- Time-Complexity: $O(b^d)$
- Space-Complexity: $O(b^d)$

Question 4 Why is the time-complexity for IDS so low if we have to run it some many times?



5.4 Uniform-Cost Search

The failure case for the optimality of BFS is when there is a deep node that has a lower cost than a shallower node. BFS fails to be optimal in this case, because it order paths by depth. Uniform-cost Search (UCS) fixes this issue by ordering paths by cost instead.

To do this we use a **priority queue** where 'pop' gives the next path based on a priority function. Define the priority function as: $f : \mathcal{P} \mapsto \mathbb{R}$ to select the next path. For UCS, we set this function to be $f(p) \triangleq g(p)$, i.e. the cost of the partial path.

Here again is the algorithm applied to our example:

iter	p	s	frontier (p)	explored
0	-	-	☀	☀
1	A	A	☀	☀
2	A:B	B	☀	☀
3	A:D	D	☀	☀
4	A:E	E	☀	☀
5	A:D:C	C	-	-

Because UCS expands paths in cost order and costs are non-negative, each path it expands must be at least as costly as all previous paths. This means that if UCS finds a goal node it must be optimal. With a few further assumptions (see AIMA) we can also show it is complete. AIMA also includes a description of the time- and space-complexity that uses the optimal score and smallest path cost ϵ .

- Completeness: Yes.
- Optimality: Yes.
- Time-Complexity: $O(b^{1+\lceil C^*/\epsilon \rceil})$
- Space-Complexity: $O(b^{1+\lceil C^*/\epsilon \rceil})$

6 Informed Search

Up to this point we assumed that we have no further knowledge into the nature of the search model. With this requirement there is no hope to find an optimal solution without first expanding all path with cost $< C^*$ (as in UCS). However in practice we often have more insight into the structure of the problem, for instance roughly how close we are to a goal state. When we have this information we can instead run informed search.

6.1 Heuristic Functions

A **heuristic** function provides an estimate of the cost from any state to a goal state.

Consider the path finding problem. If we know that we are in two-dimensions and the path costs correspond to distances, we can compute the **straight-line** distance to the goal from each state. Figure ?? shows the path finding problem with these values overlaid. Of course in practice

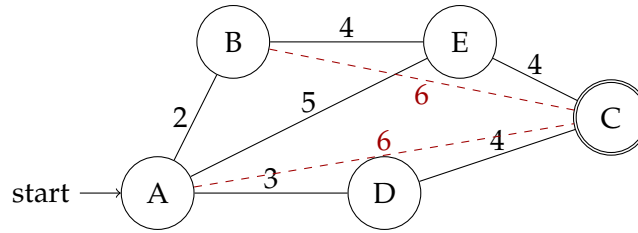


Figure 5: An example heuristic: straight-line distance.

we cannot travel directly from A to C or from B to C, but we will see that this extra information can be used to help us find a solution.

Formally, a heuristic function is just any estimation of this cost from the state itself.

Heuristic $h : \mathcal{S} \mapsto \mathbb{R}$ Estimate of the cost from state s to a goal state.

6.2 Greedy Best-First Search

In greedy best-first search, we use the same algorithm as UCS, but instead use a priority function that substitutes in the heuristic value of the last state of the path, that is:

$$f(p) = h(p_s)$$

In this algorithm we always expand the node that the heuristic thinks has the smallest cost to go to a goal. In a way if UCS is conservative like BFS, this algorithm is greedy like DFS.

The benefit of this method is that with a good heuristic it can find a solution very quickly, since it doesn't waste time refining shallower paths. The downside is that we have no guarantee about the cost of the solution is. In fact the expansion order does not even take account the cost of the path that is expanded.

- Completeness: No.
- Optimality: No.
- Time-Complexity: $O(b^m)$
- Space-Complexity: $O(b^m)$

6.3 A* Search

The lesson from greedy best-first search is that the heuristic can be useful for directing expansion, but it should be combined with some notion of the cost of the path so far.

This is the insight behind A* search, which is one of the most important algorithms we will discuss in this class. Like Greedy Best-First Search A* uses a priority queue but with the modified priority function.

$$f(p) \triangleq g(p) + h(p_s)$$

Let's consider plugging this cost into our path-finding problem.

iter	p	s	frontier (p)	explored
0	-	-	☀	☀
1	A	A	☀	☀
2	A:D	D	☀	☀
3	A:D:C	C	-	-

Note that A* not only finds the optimal solution, but it does so in a smaller number of iterations than the other algorithms. In fact for this problem it uses the minimal number of expansions.

6.4 A* Properties

Now let us consider using A* with a heuristic h satisfying the following properties.

Definition 3 An *admissible* heuristic never overestimates the cost to a goal state, i.e.

$$g(p) + h(p_s) \leq g(\hat{p})$$

where $\hat{p} \in \mathcal{Q}$ is any solution path with p as a partial path.

Definition 4 A *consistent* heuristic obeys the property that for any state s ,

$$h(s) \leq c(s, a) + h(\text{RESULT}(s, a))$$

for all actions $a \in \text{ACTIONS}(s, a)$.

Consistency is a stronger condition and implies admissibility. However, there are cases where just admissibility is sufficient so we define both.

This leads us to the crucial theorem for informed search.

Theorem 5 A* ordering with a consistent heuristic yields an optimal search algorithm.

Proof:

We first prove that the A* estimated cost increase at each expansion, and then use this to show that the first expansion of any state is optimal. This implies that the first time we expand a goal state is optimal.

1. The values of expanded paths are non-decreasing.

Assume we have expanded a path p to a path p' with action a , we have:



Where we have directly used the definition of consistency for the inequality.

2. Whenever A^* expands a path, the optimal path to that node has already been found.

We will prove this by contradiction.

Say we have expand path p before the optimal path to this state q . This means that $p_{\text{last}} = q_{\text{last}}$ and $g(q) < g(p)$, this implies $f(q) < f(p)$.

Since we expanded p first, this also mean there is some partial path of q called q' in the frontier that currently has $f(p) \leq f(q')$ (or else it would have been expanded first).

However by Property (1) we know that as a partial path of q , $f(q') \leq f(q)$.

Together this gives a contradiction:



□

Property (2) implies that whenever A^* expands the goal state it must be the optimal path to the goal.

- Completeness: Yes.
- Optimality: Yes.
- Time-Complexity: $O(b^\Delta)$ (depends on the absolute error of h , see AIMA3e, p.98)
- Memory-Complexity: $O(b^\Delta)$ (graph search)

6.5 Heuristics for Path-Finding

Now let's return to the heuristics we discussed above. How can we check whether they satisfy the properties necessary for A^* search? First consider straight-line distance. We would like to show that for any state s and valid action a :

$$h(s) \leq c(s, a) + h(\text{RESULT}(s, a))$$

and let's call the resulting state $s' = \text{RESULT}(s, a)$.

We have defined our problem such that the cost of the action is the distance between s and s' , i.e. $d(s, s')$. And we have defined our heuristic $h(s)$ as the distance to our goal $h(s) = d(s, C)$. So for consistency we need to show that:



6.6 Comparing Heuristic Functions

While all consistent heuristics will find an optimal solution, not all consistent heuristics are equal. As you will discover first hand in the homework, a better heuristic will find the optimal solution much quicker, and in fact the time and memory complexity of A^* depend on the quality of the

heuristic. A better heuristic will produce a smaller absolute error δ , where delta is the difference from the completing path \hat{p}^* .

$$\delta(p, s) = g(\hat{p}^*) - (g(p) + h(p_s))$$

We can use this to compare different heuristics. Basically we would like the heuristic that produces the highest estimate while still maintaining consistency.

Definition 5 Given two consistent heuristics h and h' , we say h **dominates** h' if for all $s \in \mathcal{S}$, $h(s) \geq h'(s)$.

In the simplest case, if we have two heuristics h_1 and h_2 , we can construct an admissible heuristic that is at least as good by taking $h(s) = \max\{h_1(s), h_2(s)\}$.

Question 6 Show that this heuristic h is still admissible.

In section we will further explore various.

6.7 Generating Heuristic Functions

A harder question though is how to generate consistent heuristics for arbitrary problems. How do we come up with heuristic for real problems? And how do we show that they satisfy consistency?

One method for doing is known as **relaxation**. Generally the approach works like this:

- Write down the explicit constraints that are required for the problem.
- Select a subset of these constraints to “relax”.
- Calculate the optimal solution without these constraints and use as heuristic.

Oftentimes when the constraints are dropped the problem becomes much more efficient to solve, and it leads to an admissible heuristic since we are making it strictly easier to reach any goal.

For path-finding we relaxed the problem by assuming we did not have to follow any of the paths directly. This allows get an underestimate of the solution. For translation, our problem assumed that each word was translated exactly once. Our heuristic is to relax this constraint and allow any remaining word translated multiple times. We can compute this heuristic cost very efficiently, and it gives us an underestimate of the optimal.

6.8 Relaxation Example: 8-Puzzle

In practice, coming up with good relaxations is the key for finding efficient, optimal search algorithms. In your homework, you will construct various heuristics for difficult maze problems. To help with this, in section, we will go over examples of relaxations. In particular we will look at the various heuristics developed for the 8-Puzzle problem below:

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State