Artificial Intelligence **(Harvard CS182, Fall 2015)**

Lectures Notes on Search

Alexander Rush

1 Summary Board

- Search Model
- Paths and Costs
- Example 1
- Example 2
- Review: Containers
- Search Algorithms

2 Introduction

The structure of the lectures:

- Models
- Algorithms

Future Topics of search:

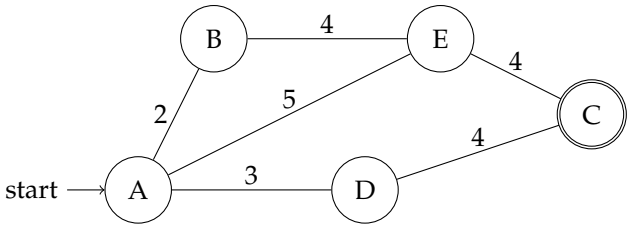
- Pure Search, Game-Playing, Constraint Satisfaction, Logical Inference

Note on Notation

- Please stop me this class for notational questions...

3 Board 1

- MapQuest, google/apple maps
- Commercial routing



4 Board 2

Search Model

Name (AIMA)	Type	Description
State space	$\mathcal{S}$	Set of states of world.
Action space	$\mathcal{A}$	Set of actions in world.
Action model	$\text{ACT} : \mathcal{S} \mapsto 2^{\mathcal{A}}$	Actions applicable at a state
Transition model	$\text{RES} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$	Result of taking action at state (stop an explain).
Initial state	$s_0 \in \mathcal{S}$	Starting State of world.
Goal test	$\text{GOAL} : \mathcal{S} \to \{0, 1\}$	Is state a goal?

(Stop and make clear the function notation)

- Start: s_0,
- Take any action $a \in \text{ACT}(s_0)$
- New state $s_1 \leftarrow \text{RES}(s_0, a)$
- Proceed until we reach $\text{GOAL}(s) = 1$

5 Board 3

Paths and Costs

Definition 1 *A **path** is a sequence of state-action pairs starting at s_0:*

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

Where $s_i = \text{RES}(s_{i-1}, a_{i-1})$ for all i and n is the depth. Let the set of all paths as $\mathcal{P}$.
Let the last state of the path to be

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

Definition 2 *A **solution** is a path that reaches a goal state, that is if* $\textsc{Goal}(p_{\textsc{last}}) = 1$.
Define the set of solutions as:

$$\mathcal{Q} = \{p \in \mathcal{P} : \textsc{Goal}(p_{\textsc{last}}) = 1\}$$

(Stop and make clear the set notation)

6 Board 4

Costs: example length of the graph

Step cost	$c : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}_+$	Cost of action at state.
Path cost	$g : \mathcal{P} \mapsto \mathbb{R}_+$	Cost of a path, i.e. $g(p) = \sum_{i=0}^{n-1} c(s_i, a_i)$.

Goal: Find the **optimal solution** or lowest-cost path:

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

where $\mathcal{Q}$ is the solution set.

7 Board 5

(Pull back down example board)
Example 1

Name	Value	Description
State space	$\mathcal{S} = \{\text{A}, \text{B}, \text{C}, \text{D}, \text{E}\}$	Locations
Action space	$\mathcal{A} = \{\textsc{Go}(\text{A}), \textsc{Go}(\text{B}), \textsc{Go}(\text{C}), \dots$	Move along edge.
Actions	$\textsc{Act}$	Which are possible. For instance, $\textsc{Act}(\text{A}) = \{\textsc{Go}(\text{B}), \textsc{Go}(\text{D}), \textsc{Go}(\text{E})\}$.
Transition model	$\textsc{Res}$	Moves to new location. For instance, $\textsc{Res}(\text{A}, \textsc{Go}(\text{E})) = \text{E}$
Initial state	$s_0 = \text{A}$	The initial state A.
Goal test	$\textsc{Goal}(s)$	Gives 1 if state s is C, 0 otherwise.
Step cost	c	For instance $c(\text{A}, \textsc{Go}(\text{E})) = 5$, distances.

8 Board 6

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

Name	Symbol	Desc
Branching Factor	b	Max actions at a state.
Min Depth	d	Shallowest solution depth.
Max Depth	m	Deepest solution depth.
Optimal Cost	C^*	Cost of the optimal solution $g(p^*)$.

- Min Depth Solution may *not* be optimal
- We may not know any of these properties exactly.

9 Board 7

Question 1 *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^*	

9.1 Board 8

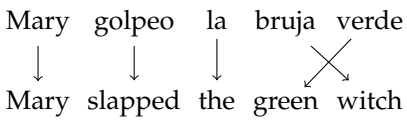
Example 2

- And now for something very different.

(Pause, read story about machine translation)

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.

Where do the scores come from?



- Translation Dictionary: Cost of translating each Spanish word. i.e. golpeo =¿ slapped : 1, golpeo =¿ hit : 2, golpeo: slaps: 20
- Language Model: Cost of two English words being next to each other. i.e. "the =¿ witch": 1, "witch =¿ the": 5, "the =¿ the": 100

9.2 Board 9

Ask people.

Name	Value	Description
State space	$\mathcal{S}$	The English translation AND which words have been translated.
Initial state	s_0	Translation is blank, all words are untranslated.
Action space	$\mathcal{A}$	Translate a word .
Actions	ACT	All ways to translate any unused word.
Transition model	RES	Add word to translation and remove word from source.
Goal test	GOAL(s)	Any state with all words translated.
Step cost	c	Sum of translation and language model.

9.3 Board 10

Question 2 *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^*	

10 Talk

Search Algorithms
(Riff on algorithms versus models)

10.1 Board 12

Abstract Containers

- `push`,
- `pop`,
- `empty`

Abstract: many possible implementations
(Maybe implement)
* 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)
```

11 Board 13

The Core Algorithm
All search algorithms utilize the following algorithm:

```
1:  procedure TreeSearch(frontier)
2:      frontier.push(Path(s_0, ϵ))
3:
4:      while frontier is not empty do
5:          p ← frontier.pop()
6:          s ← p_last
7:          if Goal(s) then return p
8:
9:          for a ∈ Act(s) do
10:             s' ← Res(s, a)
11:
12:             frontier.push(p extended by (s, a))
13:
```

12 Board 14

(Show example of wraparound)

```
1:  procedure GRAPHSEARCH(frontier)
2:      frontier.push(path starting with s_0))
3:      explored ← {}
4:      while frontier is not empty do
5:          p ← frontier.pop()
6:          s ← p_last
7:          if GOAL(s) then return p
8:          explored ← {s} ∪ explored
9:          for a ∈ ACT(s) do
10:             s' ← RES(s, a)
11:             if s ∉ explored then frontier.push(p extended by (s, a))
```

13 Board 15

- 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?

14 Board 16

Uninformed Search

- Breadth-First Search
- Depth-First Search
- Depth-Limited Search
- Iterative-Deepening Search
- Uniform-Cost Search

15 Board 17

15.1 Breadth-First Search

- Conservative-algorithm
- Explores each depth layer completely.
- LIFO Queue

- Push: adds to back
- Pop: removes from front

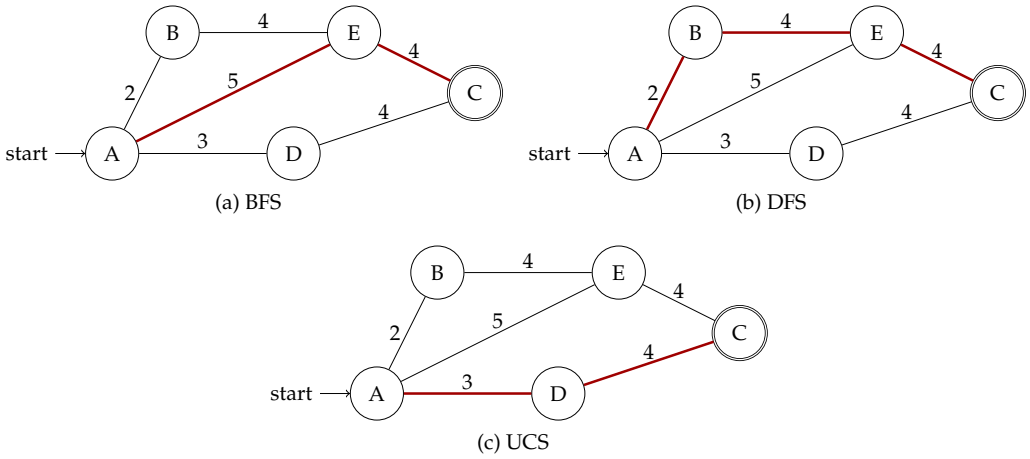


Figure 1: Paths returned by running three different search algorithms.

16 Board 18

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

17 Board 19

Properties of Breadth-First search

(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.)

- Completeness: Yes. Finds solution after $O(b^l)$ steps.
- Optimality: No. Optimal solution may not be at depth d. However BFS optimal when costs are all 1.
- Time-Complexity: $O(b^d)$
- Space-Complexity: $O(b^d)$ (size of frontier at depth d).

18 Board 20

18.1 Depth-First Search

- Greedy algorithm
- Explores further depth each time

- FIFO Queue

- Push: adds to front
- Pop: removes from front

19 Board 21

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

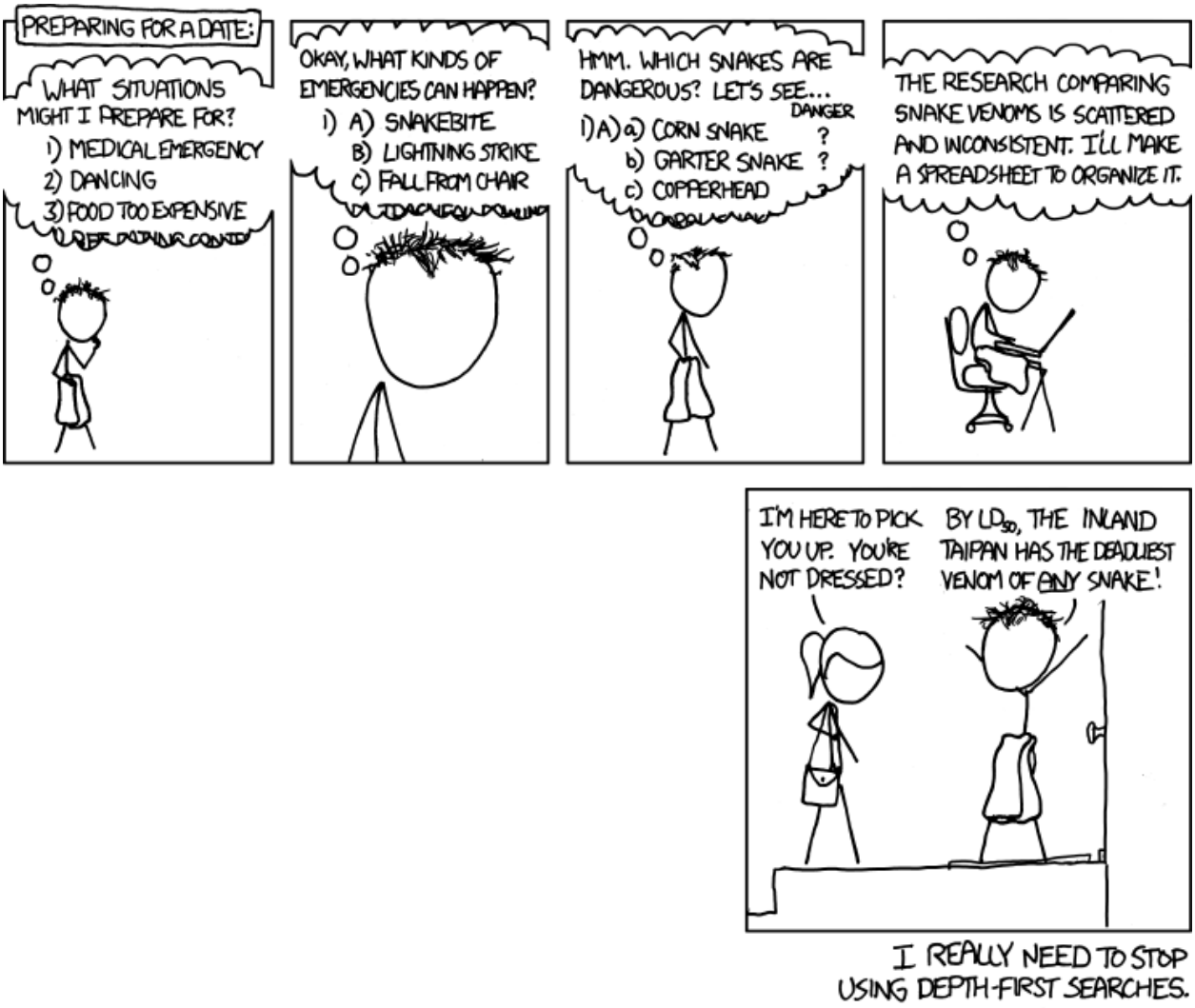
20 Board 22

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

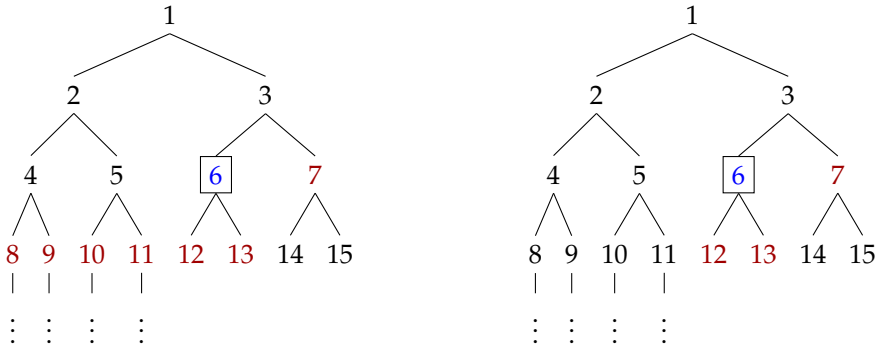
DFS has pretty poor search properties. If there is an infinite path it can go down it forever. Also it can find paths that are much deeper than m since it doesn't explore in depth order. And we have seen in the example that it is non-optimal.

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

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.



20.1 Variants of DFS

Depth-Limited Search The main issue with DFS is that it can go down paths much deeper 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 is 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 $l = 1, \ldots$. 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 3 *Why is the time-complexity for IDS so low if we have to run it some many times?*

✪ We may have to run the algorithm at most d times to find a solution. In the worst case this gives $O(b^1 + \ldots + b^d)$ time. However, in big O notation that becomes just $O(b^d)$ since the last run dominates.

20.2 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	-	-	✪ [A:0]	✪ {}
1	A	A	✪ [A:B:2, A:D:3, A:E:5]	✪ {A}
2	A:B	B	✪ [A:D:3, A:E:5]	✪ {A, B}
3	A:D	D	✪ [A:E:5, A:D:C:7]	✪ {A, B, D}
4	A:E	E	✪ [A:D:C:8]	✪ {A, B, D, E}
5	A:D:C	C	-	-

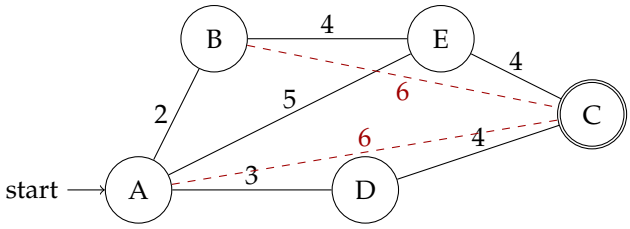


Figure 2: An example heuristic: straight-line distance.

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: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- Space-Complexity: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

21 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 it 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.

21.1 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 2 shows the path finding problem with these values overlaid. Of course in practice 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.

21.2 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: $O(b^m)$

21.3 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	-	-	✪ [A=0+6]	✪ {}
1	A	A	✪ [A:D=3+4, A:B=2+6, A:E=5+4]	✪ {A}
2	A:D	D	✪ [A:D:C:7+0, A:B=2+6, A:E=5+4]	✪ {A, 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.

21.4 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,*

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

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

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 4 *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:

   $$\begin{aligned} f(p') &= g(p') + h(p'_s) = g(p) + c(p_s, a) + h(p'_s) \\ &\geq g(p) + h(p_s) = f(p), \end{aligned}$$

   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:

   $$f(q) < f(p) \leq f(q') \leq f(q)$$

□

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)

21.5 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:

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

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

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 $h(s) = d(s, \text{C})$. So for consistency we need to show that:

$$d(s, \text{C}) \leq d(s, s') + d(s', \text{C})$$

However this is just the triangle inequality! Since this holds for Euclidean distance, we have consistency for this problem.

21.6 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 5 *Show that this heuristic h is still admissible.*

In section we will further explore various.

21.7 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:

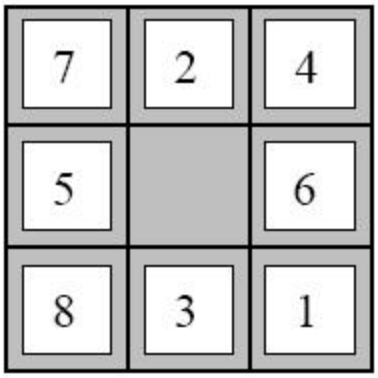
- Write down the explicit constraints that are required for the problem.
- Select a subset of these constraints to "relax".
- Calculate the optimal solution without these constraints and use as heuristic.

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.

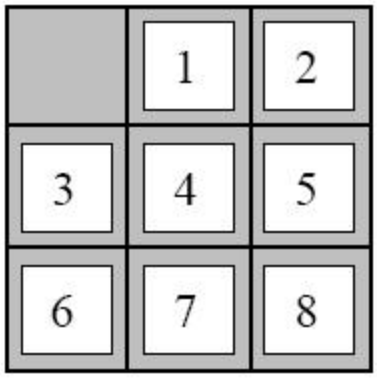
For path-finding we relaxed the problem by assuming we did not have to follow any of the paths directly. This allows get an underestimate of the solution. For translation, our problem assumed that each word was translated exactly once. Our heuristic is to relax this constraint and allow any remaining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

21.8 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:



Start State



Goal State