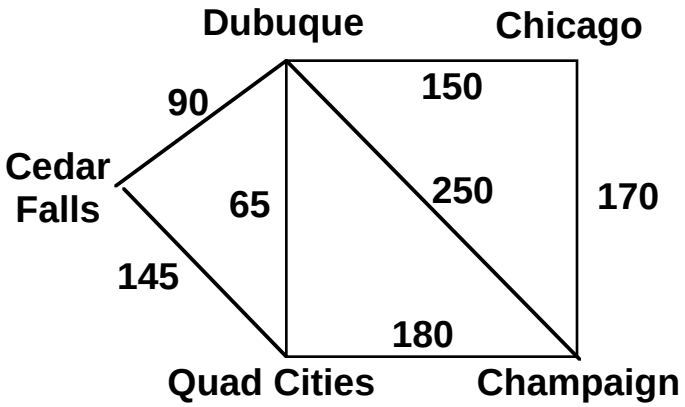
Using Knowledge of the Problem to Improve Search Performance

How can we use knowledge to improve search?

- To change the order in which we consider the states.
- To discard states from the list of states to consider.

- To formulate the problem better.
- To select the best search algorithm.

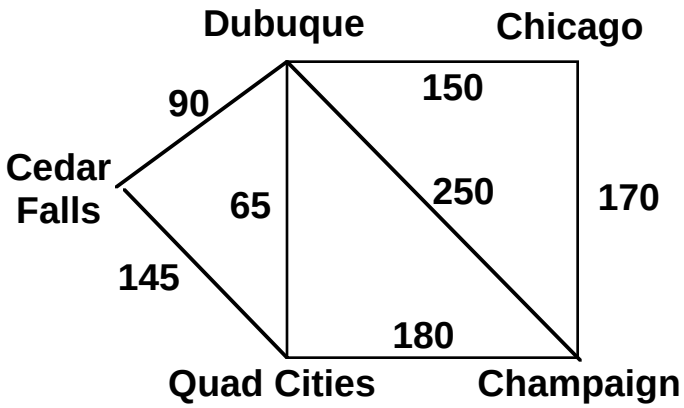
One kind of knowledge useful in search is the *cost of applying an operator*. If my goal is to be in Champaign, Illinois, then there are costs associated with traveling to Cedar Rapids and Waverly. As we apply operators in sequence, we can compute the **path cost** associated with the sequence.



Strategies that use knowledge specific to a particular problem are called **informed**.

Using Path Cost to Improve Search Performance

In **Uniform Cost Search** (UCS), we use a strategy that sorts the list in ascending order on path cost. The path cost of a node n is usually called g(n).



(Cedar Falls)
(Dubuque^{CF}, Quad Cities^{CF})
(Quad Cities^{CF}, ~~Quad Cities^{CF-D}~~, Cedar Falls^{CF-D}, Chicago^{CF-D}, Champaign^{CF-D})
(Cedar Falls^{CF-D}, *Dubuque^{CF-QC}*, Chicago^{CF-D}, ~~*Cedar Falls^{CF-QC}*~~, *Champaign^{CF-QC}*, ~~Champaign^{CF-D}~~)
(Dubuque^{CF-QC}, Chicago^{CF-D}, Champaign^{CF-QC}, Quad Cities$^{CF-D-CF}$)
(Chicago^{CF-D}, Champaign^{CF-QC}, *Cedar Falls$^{CF-QC-D}$*, Quad Cities$^{CF-D-CF}$)
(Champaign^{CF-QC}, Cedar Falls$^{CF-QC-D}$, Quad Cities$^{CF-D-CF}$, *Dubuque$^{CF-D-Chi}$*)

Uniform Cost Search

UCS implements the same idea as BFS: explore states closer to the start state first, but it uses a more practical definition of "close".

Newly-expanded states will usually enter the list of states to consider near the back, with older nodes to the front. But a path of low real cost, even if it requires more steps, can leap-frog ahead of a path with fewer operators.

How does UCS stack up against our evaluation criteria?

completeness	guaranteed to find a solution if one exists
time	generally expensive for the same reasons as BFS, but often an improvement
space	generally expensive for the same reasons as BFS, but often an improvement
optimality	only if the "cost" of each operator > 0

The problem with UCS is that it is too much like BFS. We aren't using much domain knowledge, and its effect on ordering the states is small.

Other Ways to Use Path Cost

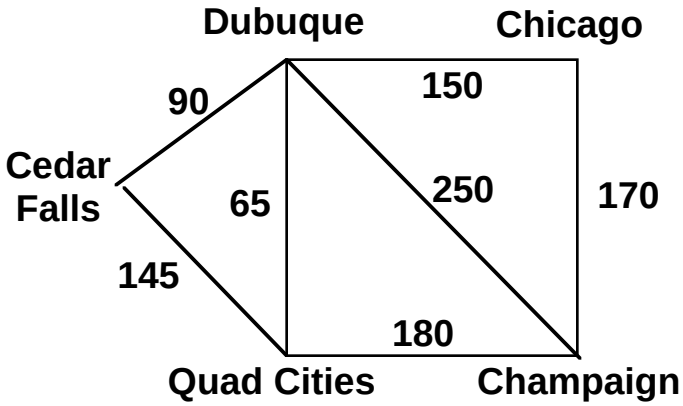
If my goal is to be in Champaign, Illinois, then there are **two costs** associated with traveling to the Quad Cities, and **two costs** associated with traveling to Dubuque. The Quad Cities are farther away from Cedar Falls than is Dubuque, *but they are closer to Champaign*.

UCS uses g(n) to guide its search. g(n) measures the cost expended in getting from the initial state to the current state, the sunk cost of taking the path.

What about looking at what seems to be a more important measure: what will it cost to get from the current state to the goal?

Using Future Path Cost to Improve Search Performance

Greedy search orders the unexplored states in ascending order of their *expected future path cost*



(Cedar Falls)
(Quad Cities^{CF}, Dubuque^{CF})
(*Champaign*$^{CF\text{-}D}$, Dubuque^{CF}, ~~*Dubuque*$^{CF\text{-}QC}$~~, *Cedar Falls*$^{CF\text{-}QC}$)

Using Future Path Costs

Of course, we usually don't have a way to know the **exact** future cost. (In most cases, the knowledge needed to compute future cost will compute the solution!) For this reason, greedy search uses a heuristic function, called `h(n)`, to estimate future cost.

Quick Exercise

Work in a group of three or four to...

- ... identify at least two different ways that we can estimate the future cost of a state in our map search process.

- ... describe how each of your ways of estimating future cost relates to the **actual** future cost.

Greedy Search

Greedy search doesn't look like uninformed methods any more, because they use information that is *very* problem-specific.

A greedy search is usually quite efficient, but it generally is not optimal:

- h(n) is an estimate, so it can be wrong.
- Greedy search considers only future cost, but not past cost.

How does greedy search stack up against our evaluation criteria?

completeness	no, for the same reasons as DFS, but usually a big improvement
time	quite efficient, for the same reasons as DFS
space	quite efficient, for the same reasons as DFS
optimality	no

The A* Algorithm

Both UCS and GS can be improved. Improving them in the same direction leads to the same result:

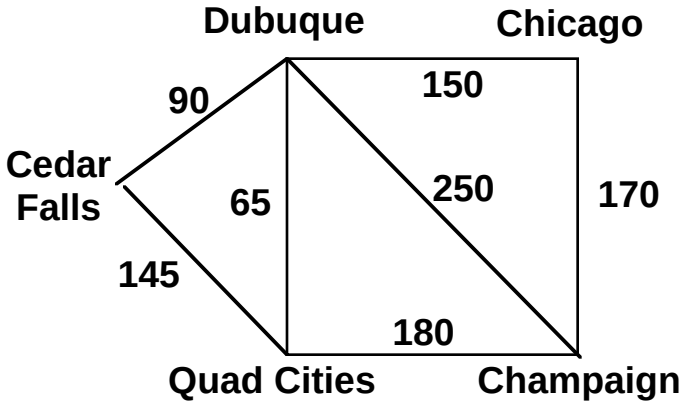
$$f(n) = g(n) + h(n)$$

The result is **A***.

A* sorts the list of states in ascending order on $f(n)$.

A* = UCS (optimal and complete, but inefficient)
\+ GS (efficient, but not optimal nor complete).

The result is an efficient algorithm *that finds optimal solutions efficiently!*



The A* Algorithm

Of course, $h(n)$ is an estimate, because it is almost always difficult to know the cost of a path to a goal without searching the space below the node.

What kinds of estimates might we have?

- An overestimating $h(n)$ makes A* inefficient, perhaps even moreso than vanilla UCS, because it prevents us from extending good paths. Bad overestimates result in even more inefficient search.

- An underestimate is, surprisingly, not so bad. If an underestimate points us down a bad path, the actual cost we discover by expanding the node will correct the search—usually soon. Bad underestimates are, of course, worse than close underestimates, but even still A* will “right itself” as soon as it encounters a bad $g(n)$ value.

More on A*

Optimistic h(n) are usually denoted h′(n).
When we use:

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

we discover that A* is both *optimal* and *as quick as possible*!!!

We will maintain the assumption that the cost of applying every operator is > 0. So...

f(n) increases at each step of the search. In terms of function maximization, you can think of A* as scaling a mountain, where f(n) is the altitude. The reason that A* is so efficient is that it walks up the sharpest contour of the "function". And, since f(n) always takes you up, A* is complete.

A good h′(n) results in more elongated contours (a feature of DFS) toward the goal.

Quick Exercise

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

What happens if, for all n, ...

- $h'(n) = 0$
- $h'(n) = h(n)$
- $h'(n) = -g(n)$

Finding Good Heuristics

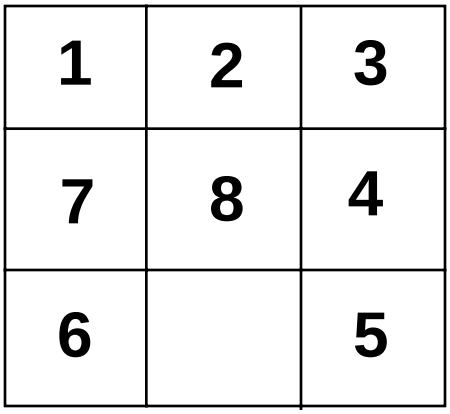
So, if we take A* as our search algorithm of first choice, then the key to doing good search is finding good h'(n) functions. This is where the real domain knowledge comes in:

> What do we know about the problem domain that will allow us to evaluate the closeness of a state to a goal?

We would like h'(n) to be as accurate as possible (”Better is better”) without *ever* overestimating the future cost of a state. If you have to err, err on the side of badly underestimating...

Another Quick Exercise

Create two different h′(n) functions for the 8-puzzle game. All three must use knowledge of the game to make their estimates.



This is a game where a good h′(n) makes all the difference. The typical solution has depth d = 20 and branching factor b = 3.

An exhaustive search expands 3^20 = **3.5 * 10^9** states...

... but there are only 9! = **362,880** unique states!

Finding Good Heuristic Functions

Two possible h′(n) are:

- h1 = number of tiles out of place.

  This is *admissible*, because each tile moves at least one time on its way to its target location.

- h2 = sum of Manhattan distances for each.

  This, too, is *admissible*, because it assumes that the we move each tile only once, to put it into its home immediately.

One way to compare heuristic functions is to compare the number of states expanded using each. We could run an experiment using random initial states and average the cost of each approach over these trials.

…

It turns out that h2 <= h1 at all depths, so h2 is always better. In such a case, we say that **h2 dominates h1**.

Finding Good Heuristic Functions

But how do we invent candidate heuristics in the first place? One useful technique is to relax the constraints on operators and then imagine a solution.

For the 8-puzzle, there is only one operator:

A tile can move from a to b
if adjacent(a, b) and b is empty.

This results in at least three possible relaxations:

- a tile can move from a to b if adjacent(a, b) h2
- a tile can move from a to b if b is empty ??
- a tile can move from a to b, always h1

Does “h3” make any sense?