Search Algorithm

Develop an algorithm to systematically conduct a search.

Analyze how well the algorithm performs.

Optimize the algorithm:

- find the "best" solution (i.e., minimum path length)
- by considering as few cases as possible.

Last Week's Algorithms

Graph Search Algorithm:

- Initialize **visited set**
- Initialize **agenda** (list of paths to consider)
- Repeat the following:
  - Remove one path from the agenda
  - For each child (of that path's terminal vertex):
    - ⋆ If it satisfies the goal condition, return a path.
    - ⋆ Otherwise, if it is not already in the visited set:
      - ▷ Add the new path to the agenda
      - ▷ Add the new vertex to the visited set

  until **goal is found** or **agenda is empty**

Properties of BFS/DFS

BFS

- Always returns a shortest path to a goal vertex, if a goal vertex exists in the set of vertexs reachable from the start vertex.
- May run forever in an infinite domain if there is no solution.
- Generally requires more memory than depth-first search.

DFS

- Always returns **a** path to a goal vertex, if a goal vertex exists in the set of vertexs reachable from the start vertex **and the search domain is finite**.
- May run forever in an infinite domain.
- Doesn't necessarily find the shortest path.
- Efficient in the amount of space it requires to store the agenda.

16 Lines!

```
def search(successors, start_vertex, goal_test, dfs=False):
    if goal_test(start_vertex):
        return [start_vertex]
    else:
        agenda = [(start_vertex, None)]
        visited = {start_vertex}
        while len(agenda) > 0:
            parent_path = agenda.pop(-1 if dfs else 0)
            for child_vertex in successors(parent_path[0]):
                child_path = (child_vertex, parent)
                if goal_test(child_vertex):
                    return flatten_path(child)
                if child_vertex not in visited:
                    agenda.append(child)
                    visited.add(child_vertex)
        return None
```

Today: Adding Costs

BFS is guaranteed always to return the shortest possible path (in terms of total number of edges traversed) to the goal.

Suppose that some edges are more beneficial than others to traverse, though. BFS's sense of optimality does not take that information into account.

Today, we'll develop a new algorithm that takes into account a sense of how desireable each edge is.

What is a Graph?

Set V of vertices (or "states")

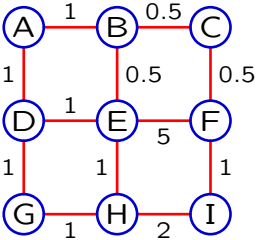
Set E of edges connecting vertices

Set W of edge costs (or "weights")

New goal: rather than finding the *shortest* path, we want to find a path with minimum total *cost*

Example

Find path $E \to I$, *minimizing total cost*



Uniform-Cost Search

Consider searching for **least-cost** paths instead of *shortest* paths. Instead of popping from agenda based on when nodes were added, pop based on of the cost of the paths they represent.

By considering paths in order of increasing cost, we guarantee that we return a path to the goal that has minimal cost.

Uniform-Cost Search

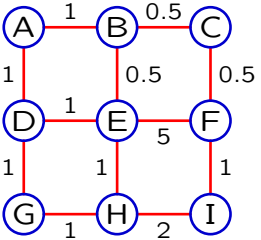
Uniform Cost Search:

- Initialize **expanded set**
- Initialize agenda (list of paths to consider)
- Repeat the following:
  - Remove one path from the agenda
  - **If its terminal vertex is in the expanded set, ignore it.**
  - **If its terminal vertex satisfies the goal condition, return a path.**
  - **Add its terminal vertex to the expanded set**
  - Add each child to the agenda if its terminal vertex is **not already in the expanded set** itemize

  until **goal is found** or **agenda is empty**

Example

Find path $E \to I$, *minimizing total cost*



More Examples

More Examples

Heuristics

So far, we have sorted our agenda based on:

$g(n)$ = the path cost from the start to n

Thus, any values with the same $g(n)$ will be considered at roughly the same time, regardless of whether they are moving "in the right direction" or not.

We can do better by including another function, which we'll call a **heuristic**:

$h(n)$ = an estimate of the remaining cost from n to the goal

Since we want to minimize total path cost, we can instead sort by:

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

$f(n)$ = an estimate of total cost from start to goal through n

Heuristics

Including a heuristic function makes this an "informed" search: it uses information not only about the path so far, but about the nature of the goal, to focus its attention toward the goal. The resulting algorithm (UC search with a heuristic) is often referred to as A* ("A-star").

This can have a dramatic effect on the amount of the search space we explore.

Heuristics and Optimality

In order to guarantee that we still get an optimal path, our heuristic needs a couple of properties:

- **admissibility**: $h(n)$ never overestimates the actual cost of the lowest-cost path from n to the goal
- **consistency**: roughly, $h(n)$ does not increase as n gets closer to the goal.

The ideal heuristic should be

- as close as possible to actual cost (without exceeding it)
- easy to calculate

Check Yourself!

Consider searching in a 4-direction grid, and let (r0,c0) and (r1,c1) represent the current and goal locations.

Which of the admissible heuristics minimizes the number of paths expanded?

1. `abs(r0-r1)+abs(c0-c1)` (i.e. Manhattan distance)
2. `min(abs(r0-r1),abs(c0-c1))`
3. `max(abs(r0-r1),abs(c0-c1))`
4. `2*min(abs(r0-r1),abs(c0-c1))`
5. `2*max(abs(r0-r1),abs(c0-c1))`

Compare

Heuristic	Paths Expanded
`abs(r0-r1) + abs(c0-c1)`	42
`min(abs(r0-r1), abs(c0-c1))`	114
`max(abs(r0-r1), abs(c0-c1))`	60
`2*min(abs(r0-r1), abs(c0-c1))`	72

Heuristics: Eight Puzzle

Consider three heuristic functions for the "eight puzzle":

a. $h(n) = 0$

b. $h(n)$ = number of tiles out of place

c. $h(n)$ = sum over tiles of Manhattan distances to their goals

Let M_i = # of moves in the best solution with heuristic i

Let E_i = # of paths expanded during search with heuristic i

Which of the following is/are true?

1. $M_a = M_b = M_c$
2. $E_a = E_b = E_c$
3. $M_a > M_b > M_c$
4. $E_a \geq E_b \geq E_c$

Compare

Heuristic	Expanded	Solution Cost
a	84,516	22
b	8,329	22
c	1,348	22