6.009: Fundamentals of Programming

Week 2 Lecture: Informed Search

Overflow Room: 32-044

- Review: BFS and DFS
- Finding Minimum-cost Paths (UC Search)
- Heuristics (A*)

adam hartz
hz@mit.edu

14 February 2022 

6.009: Goals

Our goals involve helping you develop your programming skills, in multiple aspects:

- **Programming:** analyzing problems, developing plans
- **Coding:** translating plans into Python
- **Debugging:** developing test cases, verifying correctness, finding and fixing errors

So we will spend time discussing (and practicing!):

- high-level design strategies
- ways to manage complexity
- details and "goodies" of Python
- a mental model of Python's operation
- testing and debugging strategies

Growth, not Perfection

Logistics

Lecture Overflow Room

If 26-100 is full when you arrive, please go to room 32-044 (Stata Basement) instead, where there will be a live feed from the lecture.

Additional Open Lab Hours

We will be staffing additional open lab hours on Tuesdays and Thursdays, 9am-noon in 34-501.

Logistics: Checkoffs

- Starting with lab 2, only a subset of students will be asked to complete an in-person checkoff. There is an indication on each lab page of whether you need to complete an in-person checkoff or not.

- All other students should submit responses to the checkoff discussion questions online (a random subset will be graded asynchronously).
  - Indicating that you are ready to be graded (by submitting the last question) lets us know you are ready.
  - You may still change your code submission and/or your answers to the checkoff questions, but you need to submit the last question again for us to notice those changes.
  - The timestamp associated with the last question is used for measuring checkoff lateness.

Last Week: Path Finding

Given a graph:

- Some set V of vertices (sometimes called states)
- Some collection E of edges connecting vertices

Our goal:

Develop an algorithm to systematically find paths through graphs.

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.

Pathfinding Algorithm

Pathfinding Algorithm:

- Initialize **visited set** (all vertices that have been added to the agenda)
- 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**

Breadth-first Search

Order Matters!

By changing the order in which we consider the paths on the agenda, we were able to notice dramatic differences (both in our result and in how we "explored" the graph).

Order Matters!

BFS

- Pull and pop from the different sides of the agenda (**f**irst **i**n, **f**irst **o**ut)
- Always returns a shortest path to a goal vertex, if a goal vertex exists in the set of vertices 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

- Pull and pop from the same of the agenda (**l**ast **i**n, **f**irst **o**ut)
- Always returns **a** path to a goal vertex, if a goal vertex exists in the set of vertices 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.

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.

Next, 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")

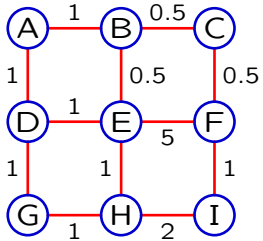
Set E of edges connecting vertices

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

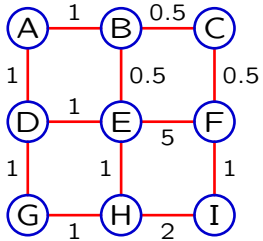
Uniform Cost Search:

- Initialize **expanded set** (all vertices that have been *removed* from the agenda)
- 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**

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

Example

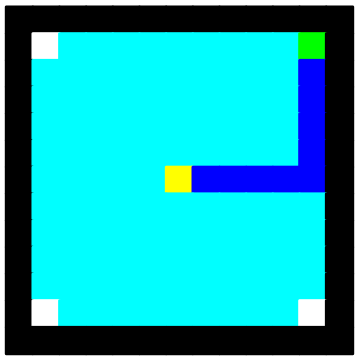
Find path $E \to I$, *minimizing total cost*



Problem?

So far, our searches have radiated outward from the starting point.

We only notice the goal when we stumble upon it. Wasting time searching on the **wrong side of the goal**.



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**: when traversing a link, $h(n)$ does not change by more than the cost of that link.

The ideal heuristic should be

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

Heuristics

The net effect of the heuristic is that we consider paths in order of nondecreasing *estimated total path length*, rather than nondecreasing cost of the path we've uncovered so far.

$h(n)$ is consistent if $\forall$ vertices n, $\forall$ children n' of n, $h(n) \leq c(n \to n') + h(n')$.

Thus, when adding a new path ending with n', we have:

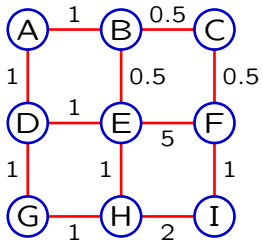
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n \to n') + h(n') \\ &\geq g(n) + h(n) = f(n) \end{aligned}$$

Thus, $f(n')$ (the estimated total cost of the new path) is greater than or equal to $f(n)$ (the estimated total cost of the parent path). The values of $f(\cdot)$ along any path are nondecreasing, which implies that the sequence of paths expanded by A* is in nondecreasing order of $f(\cdot)$.

Since we have $f(\text{goal}) = g(\text{goal})$, we are guaranteed that the first path we find to the goal (sorting by $f(\cdot)$) is also the optimal path in terms of $g(\cdot)$.

Example

Find path $E \to I$, A*, heuristic: $h(s) = M(s, I)/2$

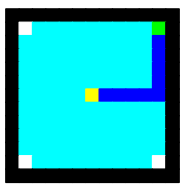


Check Yourself!

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

How many of the following heuristics are **admissible**?

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))`



Check Yourself!

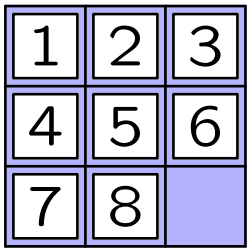
Consider searching in a 4-direction grid, and let (r0,c0) and (r1,c1) represent the current and goal locations.

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

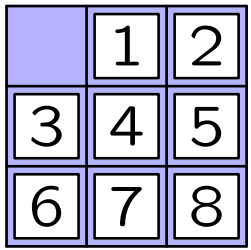
1. `abs(r0-r1)+abs(c0-c1)` (i.e. Manhattan distance)
2. `min(abs(r0-r1),abs(c0-c1))`
3. `max(abs(r0-r1),abs(c0-c1))`
4. `2*min(abs(r0-r1),abs(c0-c1))`
5. `2*max(abs(r0-r1),abs(c0-c1))`

Example: Eight Puzzle

Start



Goal



Rearrange board by sequentially sliding tiles into the free spot.

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. $E_a \geq E_b \geq E_c$

End