Finding Minimum-Cost Paths

While you are welcome to implement things however you like, we introduced a particularly useful algorithm for this lab in the week 2's lecture. For those of you who want a refresher from lecture, we're including this page as a brief outline of the approach introduced there.

Graph Search (BFS/DFS)

In lecture 2 and recitation 2, we looked at methods for finding paths through graphs, which had roughly the following form (in "pseudocode" rather than Python):

- Initialize an "agenda" (containing paths to consider)
- Initialize "visited" set (set of vertices we've ever added to the agenda) to contain only the starting vertex
- Repeat the following:
    - Remove one path from the agenda
    - If this path's terminal vertex satisfies the goal condition, return that path (hooray!)
    - For each of the children of that path's terminal vertex:
        - If it is in the visited set, skip it
        - Otherwise, add the associated path to agenda and add its vertex to visited set

  until agenda is empty (search failed)

We also saw that this approach (or slight variations thereof) could be used to solve a wide variety of problems, and we looked at how some subtle changes to the algorithm (such as changing the order in which we remove paths from the agenda) could have big effects on the way we explore the possibilities, and on the result that we ultimately return.

A Weighty Problem

The additional wrinkle we add here is that, sometimes, some paths are more desirable than others. We'll encode information about the desirability of certain edges by assigning each edge a "cost" (sometimes called a "weight"), and we'll try to focus our attention specifically on finding a path from one point to another whose total cost is minimal.

At least at the start of this lab, we'll use the notion of physical distance as a representation of cost, and we'll try to return paths of minimal distance.

We can achieve this through a slight change to our approach, resulting in something that is sometimes referred to as *uniform-cost search* (or "UC search" for short), which is similar to the above but differs in some important ways.

The high-level overview is:

- Initialize an "agenda" (containing paths to consider, as well as their costs).
- Intialize **empty** "expanded" set (set of vertices we've ever **removed from the agenda**)
- Repeat the following:

- Remove **the path with the lowest cost** from the agenda.
- If this path's terminal vertex is in the expanded set, ignore it completely and move on to the next path.
- If this path's terminal vertex satisfies the goal condition, return that path (hooray!). Otherwise, **add its terminal vertex to the expanded set.**
- For each of the children of that path's terminal vertex:
  - If it is in the expanded set, skip it
  - Otherwise, add the associated path (and cost) to the agenda

until the agenda is empty (search failed)

As we discussed in lecture, following this process will guarantee that, the first time we expand any vertex, we do so in the process of considering the lowest-cost path ending at that vertex (and, so, this algorithm is guaranteed to return the lowest-cost path from the start to the goal).

A quick note for this lab specifically: you might be nervous about how to implement removing the path with the lowest cost efficiently. While that is certainly a good thing to be thinking about, for this lab, it should be fine to implement that piece without worrying too much about efficiency (we are not expecting you to optimize that part of the process).