P1: Navmesh Pathfinding

Introduction

In this programming assignment you will need to implement in Python a bidirectional A* search algorithm to solve the problem of finding paths in navmeshes created from user-provided images. The program to build the navmeshes from images as well as a template code containing the prototype of the function you have to implement are provided. The input parameter is a navmesh and the output is an image showing the path from a source and destination. Both the source and the destination are defined interactively through a given python program. The next section shows an example on how to define the source and destination points as well as how to visualize the output of your algorithm.

Example *(Assuming you are in the /src folder)*

In order to test your pathfinder you need to execute the following command (assuming you are in the /src folder):

```
$ python nm_interactive.py ../input/homer.png ../input/homer.png.mesh.pickle 2
```

Where the parameters represent an image file to display (must be a PNG), a binary file containing the navmesh representation of the given image (.mesh.pickle) and a subsampling factor. The subsampling factor is an integer used to scale down large images for display on small screens. When you run this code you should see a new window showing the image you gave as parameter. You can interact with this window using your cursor. If you click on any region of the image, it should appear a red circle. This first circle represents the source point. You can click on the image once again in order to define the destination point. Figure 1 shows what you should see after each of these steps.

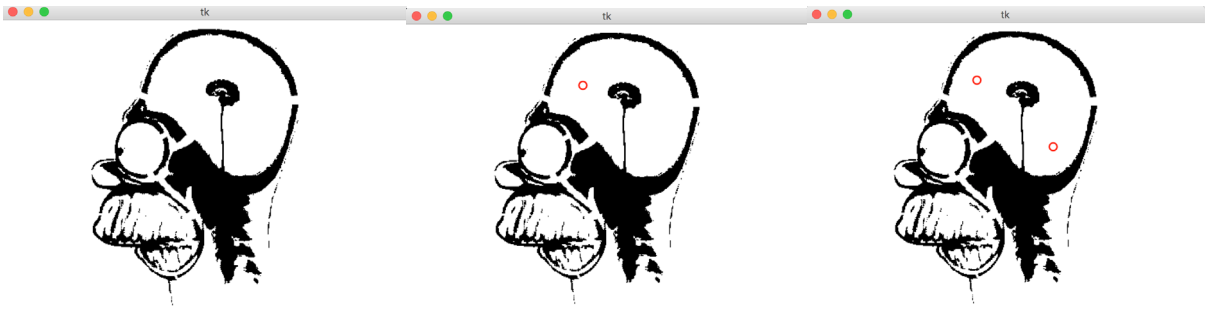


Figure 1: Screenshots of the execution of `nm_interactive.py`. *Before the first click, after the first click and after the second click, respectively*

After the second click, `nm_interactive.py` *calls the* `find_path` function that is defined in the `nm_pathfinder.py` file. Since this function is not yet implemented, `nm_interactive.py` can't calculate the path and hence can't show it. **Your job is to implement an A* bidirectional algorithm**

inside this function. When you finish your implementation, you should see the following image after the second click:

Base Code Overview

The provided base code is composed by input data (/*input*) and three python files (src/), which are described as follows:

- src/nm_interactive.py

This is the main program of this project and it is the one you have to run to test your solution. It takes three command line arguments: an image file to display (must be a PNG), the filename of a pickled mesh data structure (.mesh.pickle), and a subsampling factor. The subsampling factor is an integer used to scale down large images for display on small screens.

- src/nm_meshbuilder.py

This program can build navmeshes for user-provided images. This is the program that produces the '.mesh.pickle' files used by nm_interactive.py. "Pickle" is the name for Python's serialized binary data format. See the OPTIONAL steps at the end of this document for how to make your own maps.

- src/nm_pathfinder.py

This file contain only one function called "*find_path"*, which is the one you have to implement. The parameters and the return values are described in comments inside the function. Note that this function is being called in nm_interactive.py, so to test your solution you just have to execute nm_interactive.py as described in Section "Example".

- ### input/homer.png.mesh.pickle

This is a binary data file created by nm_meshbuilder.py. Once unpickled (this is done for you in nm_interactive.py), this file yields a dict. The mesh dict has two keys: 'boxes' and 'adj': the former is a list of non-overlapping white rectangular regions in homer.png. Think of these as the nodes in a graph. Boxes are defined by their bounds: (x1, x2, y1, y2). From your perspective, (x1,y1) is the top left corner and (x2,y2) is the bottom right. 'adj' is a dict the maps boxes to lists of boxes. Think of these as the edges in a graph. Although there is a geometric relationship between boxes, a distance value is not given in the mesh (because we don't know which point on the border of a box you'll be using to enter or leave). It may be the case that a box has no neighbors. However, for the example homer map, every box has at least one neighbor in 'adj'.

Suggested Workflow

Identify the source and destination boxes.

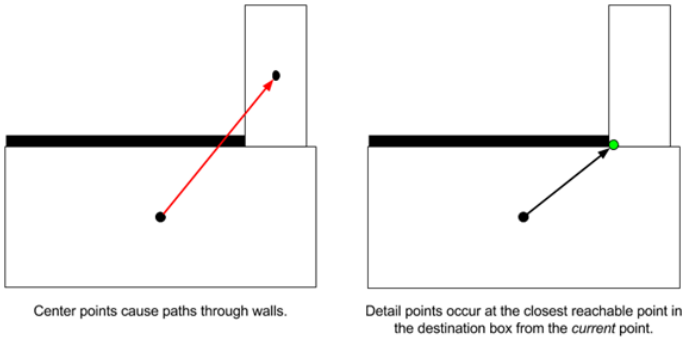
- Scan through the list of boxes to find which contains the source point. Do this as well for the destination point. Instead of returning a list indicating that you haven't visited any boxes, return a list of these two boxes.

Implement the simplest *complete* search algorithm you can.

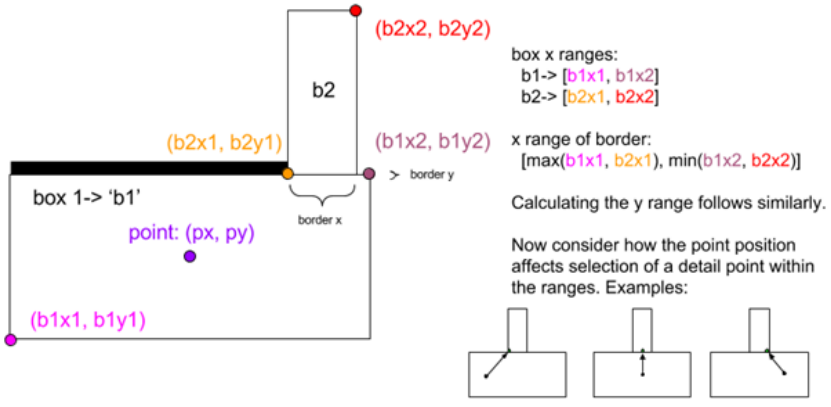
- Starting with the source box, run breadth-first search looking for a sequence of boxes that reaches the destination box. If no such path can be found, make sure your program can report this correctly (such as by printing out "No path!" in the console). To earn the point, the "No path!" message needs to only show when there really isn't a path. *Complete* search algorithms buy us the ability to report this confidently. You can also use this to evaluate your A* outputs!

Modify your simple search to compute a legal list of line segments demonstrating the path.

- Instead of doing your search purely at the box level, add an extra table (dict) to keep track of the precise x,y position within that box that your path with traverse. In the solution code, we call this table 'detail_points', a dictionary the maps boxes to (x,y) pairs. This diagram illustrates why midpoints of boxes will not work for this assignment:



- When considering a move from one box to another, copy the x,y position within the current box and the constrain it (with mins and maxes) to the bounds of the destination box.



Note: In the coordinate system of this problem set, the origin is the top left corner, while x and y grow towards the bottom right corner.

- Use the Euclidean distances between these two points as the length of the edge in the graph you are exploring (not the distance between the midpoints of the two boxes).
- When the search terminates (assuming it found a path), construct a list of line segments to return by looking up the detail points for each box along the path. In this assignment, the order of the line segments is not important. What matters is that the line is legal by visual inspection of the visualization it produces.

Modify the supplied Dijkstra's implementation into an A* implementation.

- **We have supplied an implementation of Dijkstra's forward search as a starting point together with a maze environment and an example maze for testing it.**
- Find the part of the code that puts new cells/boxes into the priority queue.
- Instead of using the new distance (distance to u plus length of edge u--v) as the priority for insertion, augment this with (add to it) an estimate of the distance remaining. If you already produced a helper function to measure the Euclidean distance between two detail points, you can use this function to measure the distance between the new detail point and the final destination point.
- When you are dequeuing boxes from the priority queue, remember that their priority value is not a distance. You'll have to recover the true distance to the just-dequeued box by looking it up in your distance table.
- To make sure A* is implemented correctly, try to find a path along a straight vertical or horizontal hallway. The A* algorithm should mostly visit boxes between the two points. Dijkstra's however, will also explore in the opposite direction of the destination point up to the radius at which it found the destination. In the example Homer map, there is a nice vertical hallway just outside of the circular chamber at the top-right.

Modify your A* into a bidirectional A*.

- Make a copy of the code you have working now for reference.
- Where you had tables recording distances and previous pointers ('dist' and 'prev'), make copies for each direction of the search ('forward_prev' and 'backward_prev'). Don't, however, duplicate the queue.
- Find the part of your code where you put the source box into the priority queue.
- Instead of just enqueuing the source box, also enqueue the destination box (which will be used to start the backwards part of the bidirectional search). In order to distinguish the two portions of the search space, instead of just enqueuing boxes, you should also indicate which goal you are seeking.
- Example:
    - heappush((0, source_box, 'destination'))
    - priority, curr_box, curr_goal = heappop(queue)
- Modify the rest of your queue operations to use this extra representation that keeps track of the goal. Use the goal to decide which set of 'dist' and 'prev' tables to check and update.
- If you are implementing bidirectional A*, change which point you are using as an estimate based on the stated goal you dequeued. This strategy is called front-to-back bidirectional A*.

(Front-to-front bidirectional A* measures the distance to the closest point on the opposite frontier -- it's more complex than you need here.)

- Instead of terminating the search when you dequeue the destination box, stop EVEN EARLIER when the just-dequeued node is known in the table of previous pointers for the other search direction. In other words, stop when either direction of search encounters territory already seen by the other direction.
- Adjust your final path reconstruction logic to build segments from both parts of the path your algorithm discovered.

Requirements

- Implement a function to compute a path from a source to destination point in a navmesh using A* bidirectional algorithm.
- Test your implementation in a new navmesh created from an image provided by you.

Grading Criteria

- **Are the mesh boxes containing the source and destination points identified?** So long as these both show up in the set of visited boxes that your algorithm returns, you are good. Beyond this, whether the set of visualized boxes represents boxes you have actually dequeued or the larger set of boxes you have enqueued is up to you.

- **Does the program behave properly in the following cases:**
  - When there is no path, report it via message in the console
  - When the path is degenerate, draw a line between the start and the destination
  - When the path is between only two adjacent cells (of the navmesh)
  - When the source and destination cells are separated by one additional cell

- **Where there is a path, is it found and drawn in a legal manner?** Legal means forming a single connected polyline from the source to destination point that never veers outside of the bounds of mesh box contained within the set of visited boxes.

- **Is the A* algorithm implemented correctly?**

- **Is a bidirectional search algorithm (or better) implemented correctly?**

Submission Instructions

Submit via Canvas a zip file named in the form of "Lastname-Firstname-P1.zip" containing:

- The *nm_pathfinder.py* file implementing the function find_path.
- An image file named *test_image.png* containing a new image you tested your solution with
- A mesh representation of that image named *test_image.mesh.pickle*

Also, put your name and your partner's name in the Comment field of the submission form

Creating a Custom Map

Here's how to create your own test map:

STEP 1: Find some image that you think will be easy to turn into a black-and-white occupancy map. Save it as a PNG file. nm_interactive can only display PNG files.

ucsc_banana_slug-orig.png

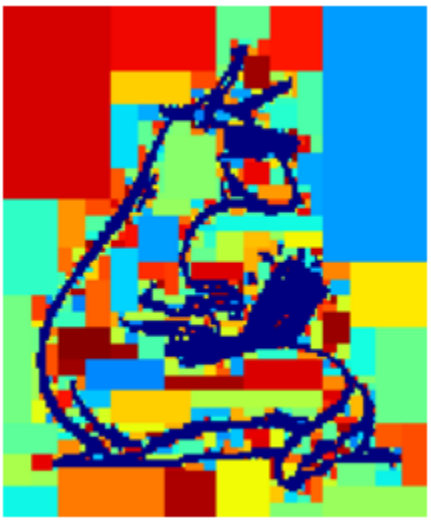
STEP 2: In your favorite photo editor, create a black-and-white version (e.g. by desaturating and then applying brightness and contrast operators). Save this in a lossless format like PNG.

ucsc_banana_slug.png

STEP 3: Run the navmesh builder program. You must have SciPy (http://www.scipy.org/) installed for this program to work.

$ python nm_meshbuilder.py ucsc_banana_slug.png
Built a mesh with 711 boxes.

This will produce two files. The first is the mesh as a pickled Python data structure: *ucsc_banana_slug.png.mesh.pickle*. The second is a visualization of the rectangular polygons extracted by the navmesh builder:



STEP 4: Run your pathfinding program giving the *original* PNG file, the pickled mesh data, and some subsampling factor. A factor of 1 displays the image at original size, while a factor of 4 will scale the image down by a factor of four in each dimension for display (pathfinding will still be done at the full resolution).

$ python nm_interactive.py ucsc_banana_slug-orig.png ucsc_banana_slug.png.mesh.pickle 1