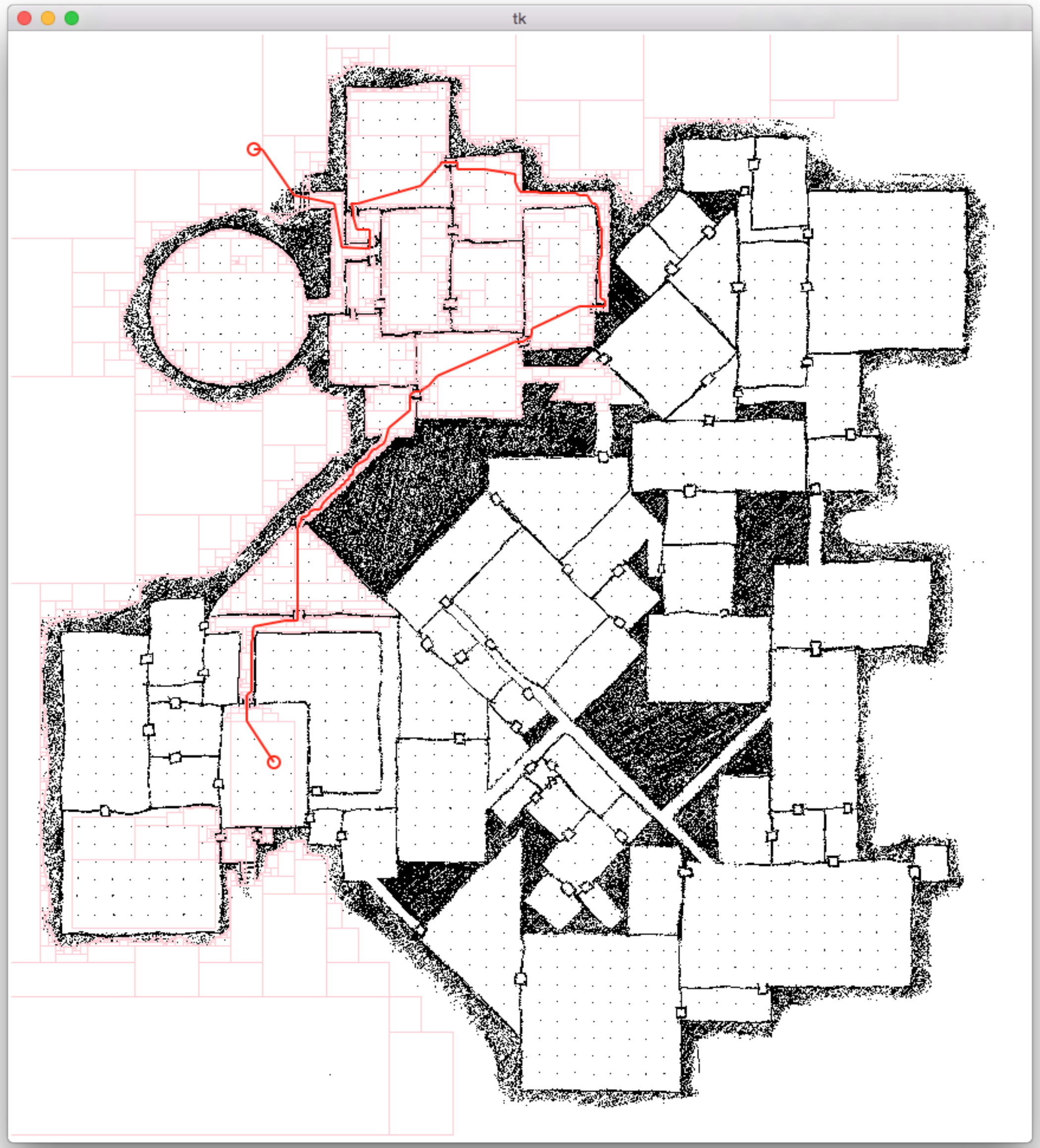
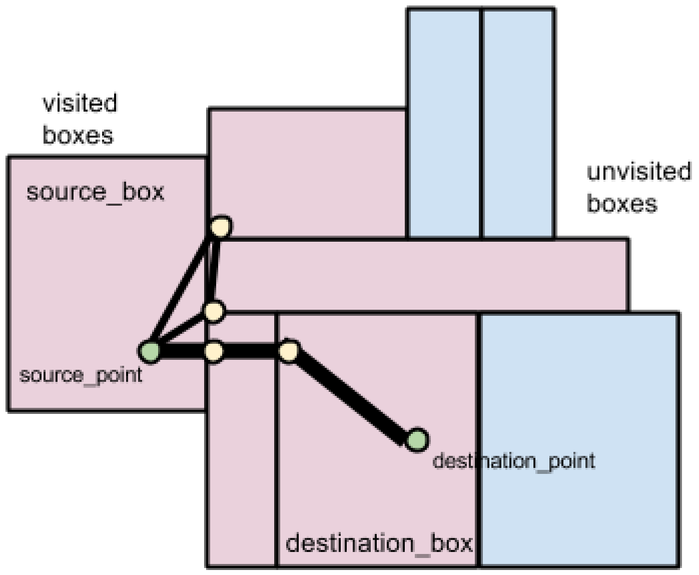
# P3: Navmesh Pathfinding



Example of working solution program.



Visualization of adjacent navmesh boxes and position of detail points within boxes.

## Base Code Overview

You can find the base code here:
[https://drive.google.com/a/ucsc.edu/folderview?id=0B-PPiU3Ga8Z7fk5OWU40WXM0cXlMMmJZUWhyd0NhRXI4Q0ZNWnNQSHFIN2hfbnhzTGQ3WWs&usp=sharing](https://drive.google.com/a/ucsc.edu/folderview?id=0B-PPiU3Ga8Z7fk5OWU40WXM0cXlMMmJZUWhyd0NhRXI4Q0ZNWnNQSHFIN2hfbnhzTGQ3WWs&usp=sharing)

### p3_interactive.py

This is the interactive Tkinter driver program for this assignment. It takes three command line arguments: an image file to display (must be a GIF), 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.

```
$ python p3_interactive.py dungeon-orig.gif dungeon.png.mesh.pickle 8
```

### p3_pathfinder.py

*This file is not given. You need to implement it yourself.*

It should define a function called "find_path". This function should take three arguments:

- source_point: (x,y) pair
- destination_point: (x,y) pair
- mesh: a mesh data structure (a fancy graph, described below)

The function should return two values:

- path: a list of points like ((x1,y1),(x2,y2))
- visited_nodes: a list of boxes explored by your algorithm identified by their bounds (x1,x2,y1,y2) (this is the same as in the mesh format)

In p3_interactive.py, this is how your function will be called:

*path, visited_nodes = p3_pathfinder.find_path(source_point, destination_point, mesh)*

### p3_meshbuilder.py

*You do not need to run this program unless your Python version cannot load the dungeon.png.mesh.pickle file we provide or you are creating a custom map.*

This program can build navmeshes for user-provided images. This is the program that produces the '.mesh.pickle' files used by p3_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.

### dungeon.png.mesh.pickle

- This is a binary data file created by p3_meshbuilder.py. You don't need to regenerate it.
- Once unpickled (this is done for you in p3_interative.py), this file yields a dict.
- The mesh dict has two keys: 'boxes' and 'adj'.
- 'boxes' is a list of non-overlapping white rectangular regions in dungeon.png. Think of these as the nodes in a graph. Unlike in P1, your could *should* examine the structure of these node identifiers.
- Boxes are defined by their bounds: (x1,x2,y1,y2)
- '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).
- For student-created maps, it may be the case that a box has no neighbors. However, for the example dungeon map, every box has at least once neighbor in 'adj'.

## Suggested Workflow

- **Identify the source and destination boxes. (earns 1 point)**
  - Implement a trivial version of find_path() that simply returns an empty list for the path (the first returned value) and an empty list for the collection of visited 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. (earns 1 point)**
  - Starting with the source box, run breadth-first search looking for a gain 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.

- **Modify your simple search to compute a legal list of line segments demonstrating the path. (earns 1 point)**
    - 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.
    - When considering a move from one box to another, copy the x,y position within the current box and the constraint it (with mins and maxes) to the bounds of the destination box. 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 (see below).
- **Modify your simple search to implement Dijkstra's algorithm. (earns 1 point)**
    - If you have already implemented an algorithm better than Dijkstra's (such as A* or bidirectional Dijkstra's or bidirectional A*, *skip this step*.
    - If you were previously using a distance-ignoring algorithm like BFS, upgrade it to Dijkstra's now. As in P1, this will probably involve using the heapq Python module. Alternatively, you can use the PriorityQueue code that some students have discovered referenced in the A* reading by Amit Patel.
    - After this change, your program should be producing near-optimal paths. The paths will not be precisely optimal because the structure of the navmesh structurally disallows certain paths. For the example dungeon map, however, your algorithm should always be choosing the right path at the scale of rooms and hallways. Because of the box decomposition, the not every small passageway will be actually passable (check the region atlas dungeon.png.mesh.png to see how the boxes were formed).
- **Modify your Dijkstra's implementation into an A* implementation. (earns 1 point)**
    - Find the part of your code where you are putting new 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 (add to) this with 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 dungeon map, there is a nice vertical hallway just outside of the circular chamber at the top-right.
- **Modify your Dijkstra's (or A*) into a bidirectional Dijkstra's (or bidirectional A*). (earns 1 last point)**
    - 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.

**Modify your bidirectional Dijkstra's into bidirectional A*. (no extra points)**

- If you haven't already, try combining bidirectional search with A*. If you have this version of your program running, you'll be able to demonstrate all aspects of your program at once without having to swap which search algorithm you are using.
- Note: You can earn full credit without doing this. Further, if you aim for this variation too early on, you might get lost. The suggested workflow above has lots of safe places where you can be sure you've earned points without losing them later.

## Grading Criteria

Each bulleted item receives equal weight:

- **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.
- **When there is no path, can the program identify this state and report it?** Printing text in the command line is fine.
- **Where there is a path, is it found an 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 Dijkstra's algorithm (or better) implemented correctly?**
- **Is the A* algorithm (or better) implemented correctly?**
- **Is a bidirectional search algorithm (or better) implemented correctly?**

We will still accept your algorithms if they make non-optimal choices on the last step or the step that joins the two directions of bidirectional search. Terminating the search as soon as you reach (dequeue) the destination node / a node from the opposite direction of search is sufficient. We mainly care that your algorithm does not visit boxes unnecessarily, so small errors in precise path length don't matter.

## Creating a Custom Map (OPTIONAL)

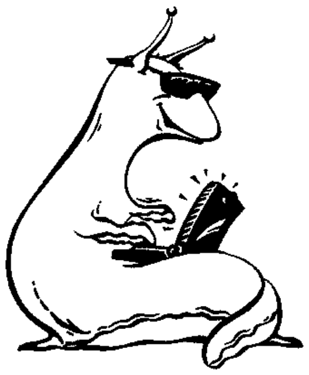
Here's how to create your own test map:

**STEP 1:** Find some image that you think will be easy to turn into a black-and-white occupancy map. Save it as a GIF file. p3_interactive can only display GIF files.

*ucsc_banana_slug-orig.gif*

**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*

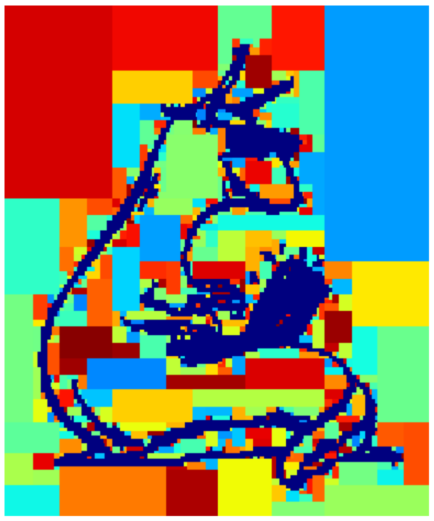


**STEP 3**: Run the navmesh builder program. You must have SciPy ([http://www.scipy.org/](http://www.scipy.org/)) installed for this program to work.

```
$ python p3_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:

*ucsc_banana_slug.png.mesh.png*.



**STEP 4:** Run your pathfinding program giving the *original* GIF 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 p3_interactive.py ucsc_banana_slug-orig.gif \
        ucsc_banana_slug.png.mesh.pickle 1
```