P2: 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

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

*$ python3.5 p2\_interactive.py ../input/homer.gif ../input/homer.gif.mesh.pickle 2*

Where the parameters represent an image file to display (must be a GIF), 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 represent 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.

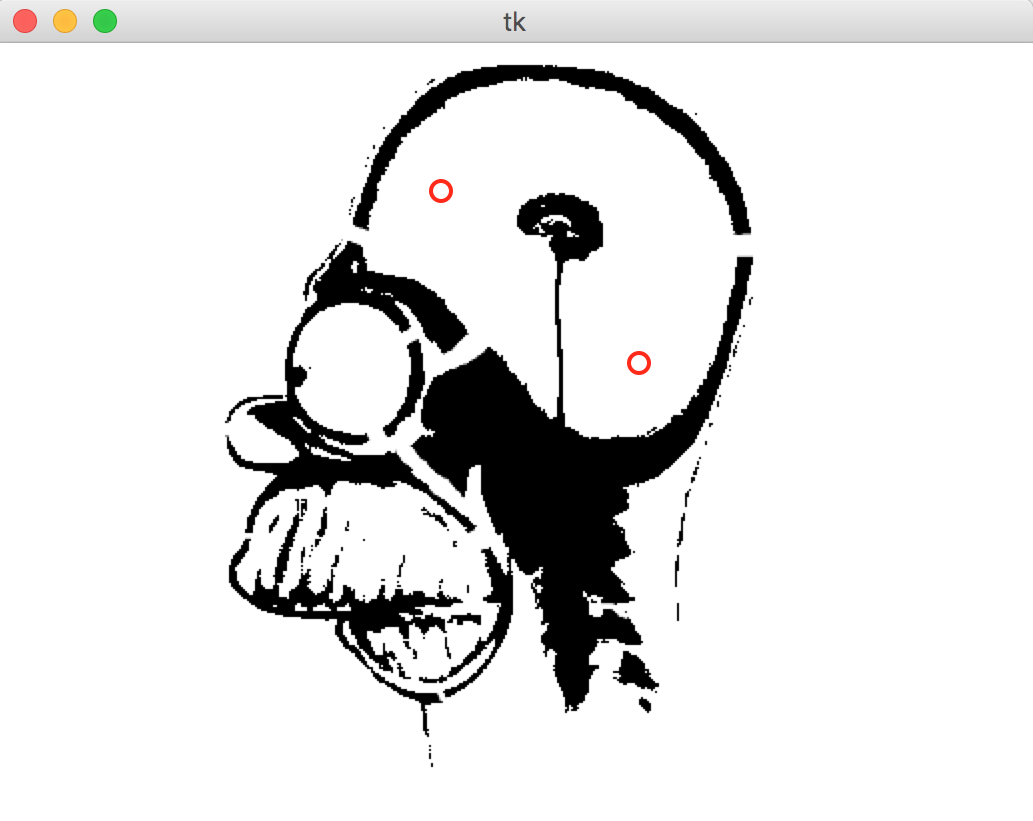
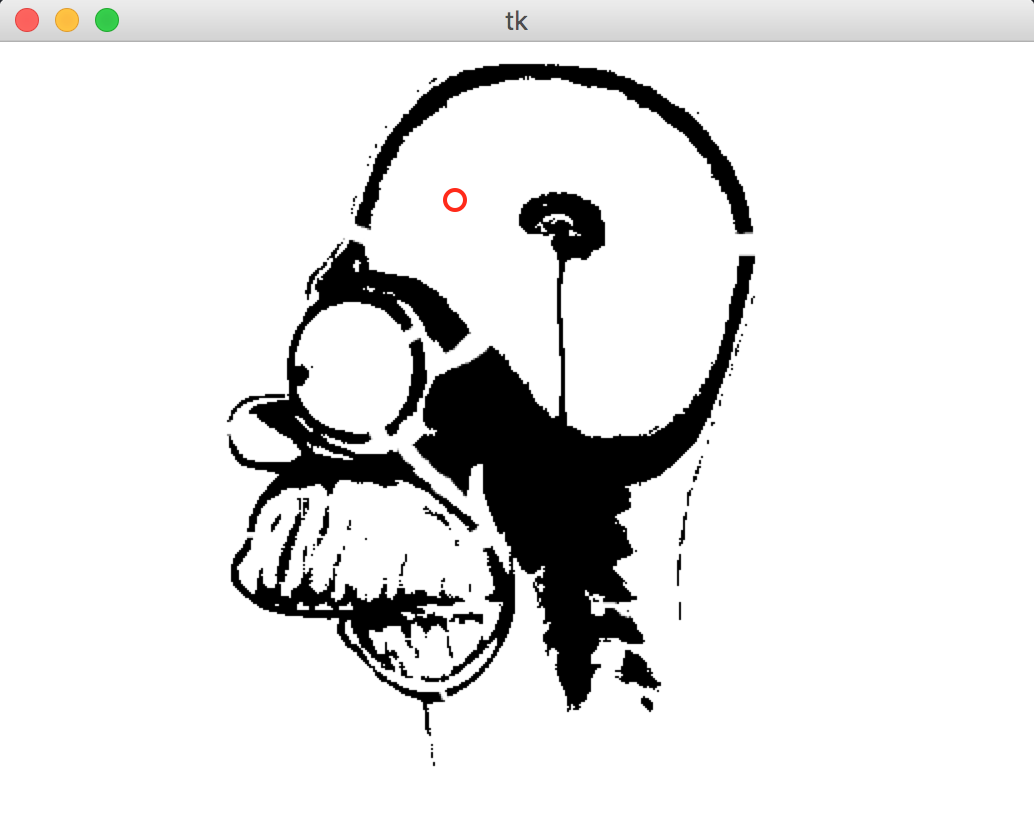
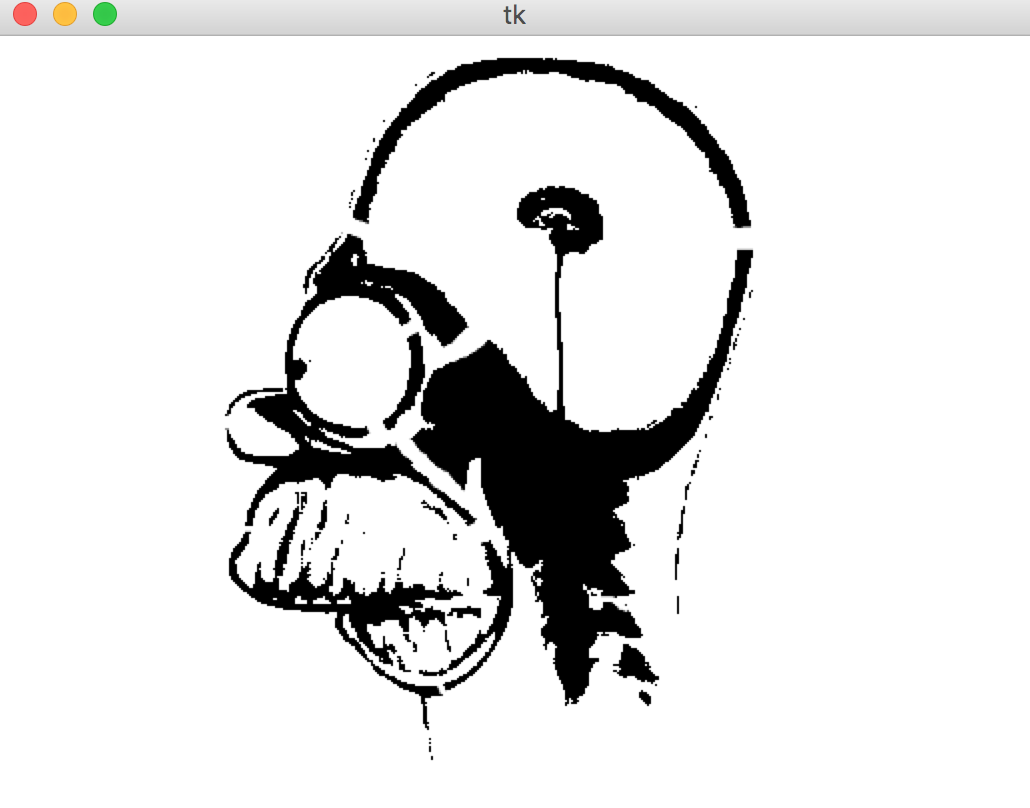


Figure 1: Screenshoots of the execution of *p2\_interactive.py* before the first click, after the first click and after the second click, respectively

After the second click, *p2\_interactive.py calls the find\_path* function that is defined in the *p2\_pathfinder.py* file*.* Since this function is not yet implemented, *p2\_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 navmesh and the path from source to destination , as shown in Figure 2.

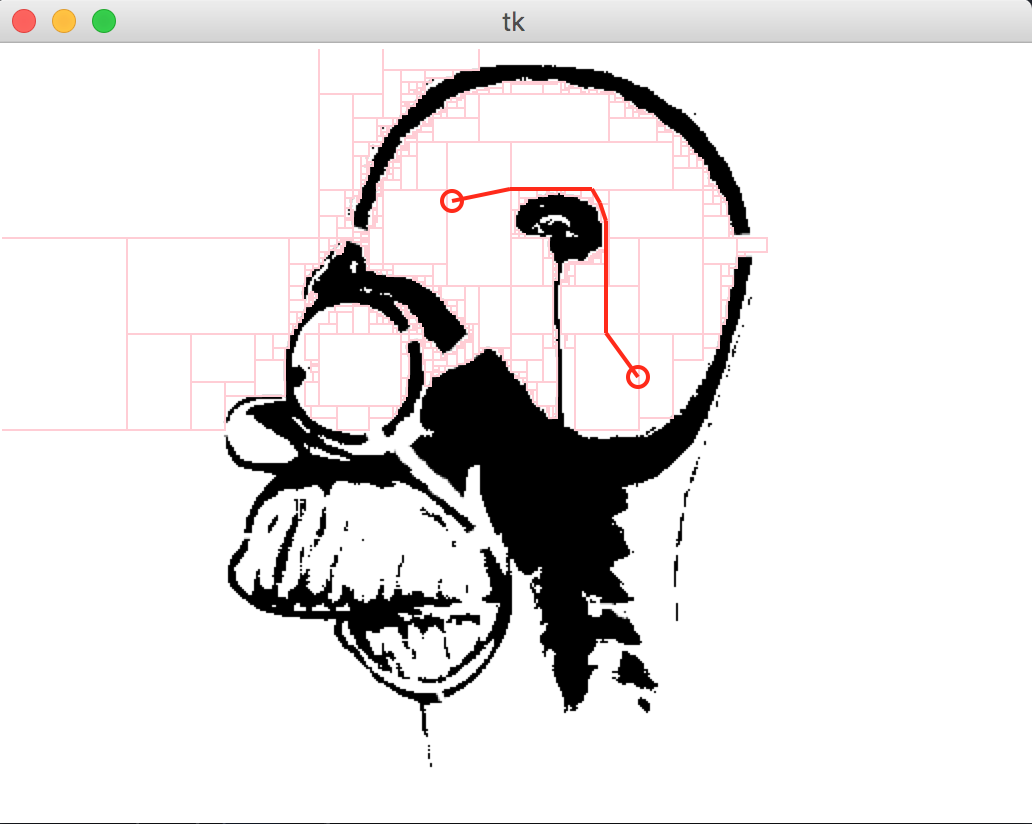


Figure 2: Screenshoots of the execution of *p2\_interactive.py* after the second click when the find path function is implemented correctly.

## Base Code Overview

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

**- src/p2\_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 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.

**- src/p2\_meshbuilder.py**

This program can build navmeshes for user-provided images. This is the program that produces the '.mesh.pickle' files used by p2\_interactive.py. "Pickle" is the name for Python's serialized binary data format.

**- src/p2\_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 p2\_interactive.py, so to test your solution you just have to execute this program as described in Section "Example".

* **input/homer.png.mesh.pickle**

This is a binary data file created by p2\_meshbuilder.py. Once unpickled (this is done for you in p2\_interative.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 and the latter is a dict that maps a box to a list of its adjacent boxes . **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.**

Finding the Detailed Path (within and between Navmesh boxes)

Because a Navmesh cell represents a convex region of open space (boxes, in this case), you must calculated detailed paths between navmesh cells. You will build detailed paths out of line segments that connect a box’s point of entry to the closest point on the boundary of the next box. The point of entry is usually on a box boundary, but it is internal to the very first box of a path. The full, detailed path from the source to the destination point is composed of the points of entry that connect the enclosing navmesh boxes.

Given the x,y coordinate of a box’s entry point, you will need to compute the x,y coordinate of the exit point. It is important to highlight that the exit point of the current box is the entry point of the next one. Hint: the elegant calculation uses *max* and *min* operations to constrain the x,y coordinates of the entry with the edge points of the gateway. See the illustrations below.

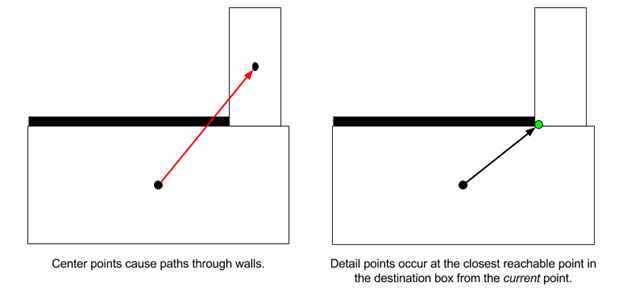


Figure 3: A straight line path between the center of adjacent cells is often illegal.

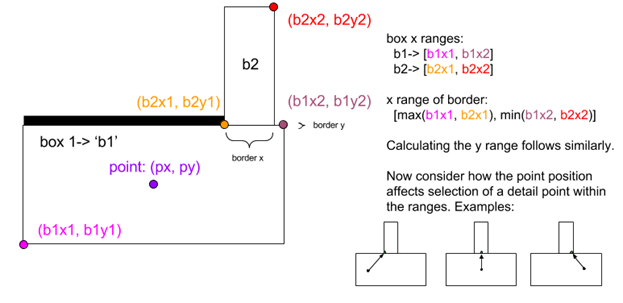


Figure 4: Determining the x coordinate of the nearest point on the boundary.

Use the Euclidean distance between the start and boundary location you have just found as the cost for traversing this box. The A\* algorithm should calculate the total pathcost so far as the sum of the costs for traversing all box until now.

You will need to store the entry point of each box for the forward and backward searches. Use a table (dict) that maps a box to an (x,y) pair. When the search terminates (assuming it found a path), construct a list of entry points to return by looking up each box along the path.

Building A\* from Dijkstra’s Forward Search

The A\* can be seen as a simple variant of Dijkstra’s Forward Search (DFS) and hence you can implement it based on the DFS implementation. We have supplied an implementation of DFS, or you may use our own solution from P1. You can adapt it via the following steps:

1. Find the part of your code where you are putting new cells into the priority queue.
2. Instead of using the new distance as the priority for insertion, extend this with (add to it) an estimate of the distance remaining. A good estimation here is the euclidian distance between the current cell to new and the final destination point.
3. When finding paths in a navmesh you don't queue cells, but boxes. 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.
4. 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.

Bidirectional A\*

Bidirectional A\* is a variant of the A\* algorithm that simultaneously searches from the goal to the start, and from the start to the goal, stopping when the two searches connect. It has certain efficiency advantages over A\* alone. Bidirectional A\* can be implemented by modifications in the queue and process strategy behind A\*, as follows:

1. 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. Remember that the same cell may have two entry points in a bidirectional search - you will need to adapt or duplicate the dict that maps boxes to entry points.
2. Find the part of your code where you put the source box into the priority queue.
3. 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. For example:

heappush((0, source\_box, 'destination'))

priority, curr\_box, curr\_goal = heappop(queue)

1. 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.
2. 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.)
3. (Optional) 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.
4. Adjust your final path reconstruction logic to build segments from both parts of the path your algorithm discovered.

## Requirements

* For partial credit (80%), implement a function to compute a path from a source to destination points in a navmesh using classic A\*.
* For full credit, update your classic A\* to behave as an 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 implemented correctly?**
* **Was the program tested in a new navmesh?**

## Submission Instructions

Submit a zip file named in the form of “Lastname1-Lastname2-P2.zip” containing:

* The *p2\_pathfinder.py* file implementing the function find\_path.
* An image file named *test\_image.gif* containing a new image you tested your solution with
* A mesh representation of that image named *test\_image.mesh.pickle*

[Submission Link](https://goo.gl/zI7uVq)

Creating a Custom 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. p2\_interactive.py can only display GIF files.

**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.

**STEP 3**: Run the navmesh builder program as described below (assuming you are in the /src folder). You must have SciPy (<http://www.scipy.org/>) installed for this program to work.

*$ python3.5 p2\_meshbuilder.py ../input/ucsc\_banana\_slug.png*

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

**STEP 4:**Run your pathfinding program giving the *original* GIF file, the pickled mesh data, and some subsampling factor.

*$ python3.5 p2\_interactive.py ../input/ucsc\_banana\_slug-orig.gif ../input/ucsc\_banana\_slug.png.mesh.pickle 1*

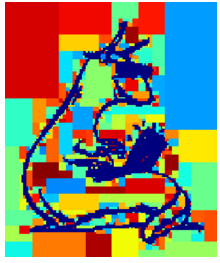
   

Figure 5: From the left to the right, these images are what you can see after the end of each step described to create a custom map.