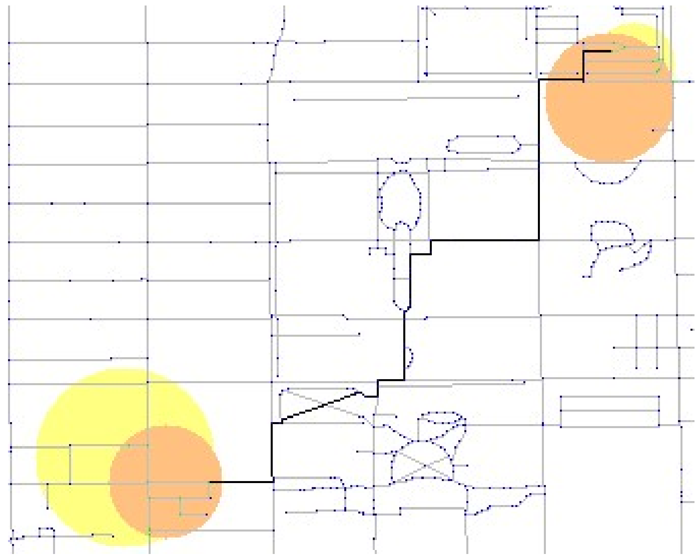
ECE220: Computer Systems and Programming **Spring 2018 Honors Section**
Machine Problem 9 **due: Thursday 29 March, 11:59:59 p.m.**

Mieber: Walk Me There



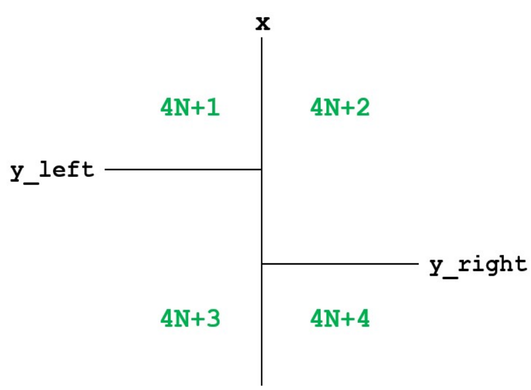
Your task this week is to implement a request matching and pathfinding subroutines for a tool that helps people to find walking partners.

In particular, you must write C subroutines that identify possible starting and ending points and that find the shortest path between any pair of starting and ending points. For this purpose, you will make use of a 'pyramid tree' and write code to implement a heap for use by Dijkstra's single-source shortest-paths algorithm.

The objective for this week is for you to gain experience with array-based data structures in C.

Background

The screenshot above is generated automatically based on the output of your routines. As you might guess, the input data are taken from OpenStreetMap data for the Champaign-Urbana area, and the image generation is provided by your MP5 code. In the image, the yellow and orange circles represent the starting and ending locales for two different people. Light grey lines represent roads, and blue dots represent nodes not in the intersection of the yellow and orange circles. Green dots represent nodes in the intersection of the yellow and orange circles (either the starting locales or the ending locales). Green dots are thus possible starting and ending points for the shared walk. The black line is then the chosen shortest path between any pair of green dots, assuming that the path must follow the roads.



Pyramid tree node structure for node N. Leaf nodes (with no children: 4N+1 ≥ # of nodes) represent graph vertices as (x,y_left). Nodes with children subdivide space into up to four parts. Note that children in any quadrant can be located on the lines (equality is allowed in both directions).

In addition to a copy of the graph itself with vertex positions (x,y), neighbors, and distances between neighbors, you are provided with a pyramid tree containing all vertices in the graph. A pyramid tree enables one to quickly locate nodes with a specified geographic area. The pyramid tree consists of a fixed number of nodes fit into a single array as shown to the right. The number of leaf nodes is equal to the number of nodes in the graph, and each vertex in the graph occupies one leaf node. Internal nodes in the pyramid tree divide space into (up to) four quadrants (one node may have fewer children). For example, if one examines an internal node at array index N, array index 4N+1 is a subtree in which all nodes have x values no greater than the x value of the internal node at array index N and y values no greater than the y value of the internal node at array index N.

Pieces

Your program will consist of a total of three files:

`mp9.h`	This header file provides type definitions, function declarations, and brief descriptions of the subroutines that you must write for this assignment. **You should read through the file before you begin coding.**
`mp9.c`	The main source file for your code. A version has been provided to you with placeholders for the subroutines described in this document. You will need to implement several heap-related subroutines—please place them in this file as well.
`mp9match.c`	The source file for your implementation of the **`match_requests`** function.

Four other files are also provided to you:

`graph`	The map data. Feel free to test with your own graphs as well.
`Makefile`	A file that simplifies the building and visualization process. See the section below on Compiling and Executing Your Program.
`mp9main.c`	A source file that interprets commands, calls your subroutines, implements some game logic, and provides you with a few helper functions (described later). You need not read this file, although you are welcome to do so. You may want to read the headers of the helper functions before using them.
`requests`	The requests used to generate the image at the start of this document. The file consists of two requests (one per line). Each request consists of a starting locale and an ending locale, and each locale consists of an X position, a Y position, and an acceptable range (distance) from that center point.

Finally, we have included slightly modified copies of **`mp5.h`** and **`mp5main.c`** for visualization purposes. In order to visualize your results, you must first add your own **`mp5.c`** solution file to your **`MP9H`** directory. Only **`draw_line`** and **`draw_circle`** are used from your code, so as long as those functions are reasonably correct, the visualization should work.

The Task

The total amount of code needed in my version of this assignment was under 200 extra lines.

The primary function for this MP has the following signature:

```
int32_t match_requests (graph_t* g, pyr_tree_t* p, heap_t* h,
                        request_t* r1, request_t* r2,
                        vertex_set_t* src_vs, vertex_set_t* dst_vs,
                        path_t* path);
```

The function provides you with a copy of the graph, a pyramid tree containing all vertices in the graph, a "blank" heap for your use with Dijkstra's algorithm, and two requests for walking partners. Each request consists of two locales, a starting point and an ending point. Your code must identify up to **`MAX_IN_VERTEX_SET`** graph vertices within range of the starting point for both requests—these should be written into the **`src_vs`** argument. Similarly, your code must identify graph vertices within range of the ending point for both requests—these should be written into the **`dst_vs`** argument. Finally, your code must use a slightly modified version of Dijkstra's single-source shortest path algorithm to find the shortest path between any node in the source set and any node in the destination set. A forward path, including both the initial and final nodes, should then be written into the **`path`** argument. If the source node set or the destination node set is empty, or if the path requires more than **`MAX_PATH_LENGTH`** nodes (counting both the starting and ending nodes), your function should return 0, in which case all outputs are ignored. Otherwise, your function should return 1.

As a first step, you should implement a function to walk the pyramid tree and find any nodes within range of a specified locale:

```
void find_nodes (locale_t* loc, vertex_set_t* vs, pyr_tree_t* p, int32_t nnum);
```

The **`find_nodes`** function should start at array index **`nnum`** and walk the pyramid tree recursively in order to fill the vertex set **`vs`** with the array indices of any graph vertices found to be in range of locale **`loc`**. The count of vertices in the vertex set should be initialized to 0 before calling **`find_nodes`**. You do not need to recurse optimally, but you do need to be reasonably efficient, so use the splitting information in internal pyramid nodes as best you can to avoid recursion. You must use the following function to check whether a leaf node (a graph vertex) is in range of the given locale:

```
int32_t in_range (locale_t* loc, int32_t x, int32_t y);
```

Next, implement a function to remove any graph vertices that are not in range of a locale from a vertex set:

```
void trim_nodes (graph_t* g, vertex_set_t* vs, locale_t* loc);
```

Together, these two functions will enable your **`match_requests`** function to identify the possible starting and ending graph vertices for the given pair of requests.

The last required function must implement Dijkstra's single-source shortest path algorithm (**`https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`**):

```
int32_t dijkstra (graph_t* g, heap_t* h,
                  vertex_set_t* src, vertex_set_t* dest, path_t* path);
```

The function should return 1 if a path is found that can fit into the **`path`** structure. You are encouraged to make use of the heap provided to you to implement the algorithm. You will need to write several additional heap-related subroutines to do so, including heap initialization, removing the closest unvisited graph vertex from the heap, and reducing the distance to a vertex still in the heap. You have seen heaps in MP7. Here, a parent node at array index N should be smaller (nearer, with smaller **`from_src`** distance) than both of its children. Fields have been provided in the graph vertex structure to help you implement the algorithm. You will need to modify the algorithm slightly (rather than calling it once for each possible starting point) in order to obtain full credit. Your Dijkstra routine should write the shortest path found between any pair of vertices in the **`src`** and **`dest`** vertex sets (respectively) into the **`path`** parameter. You may also want to use the **`MY_INFINITY`** preprocessor constant to represent infinity in the algorithm.

Specifics

Be sure that you have read the type definitions and other information in the code and header file as well as descriptions of Dijkstra's algorithm (and, if necessary, heaps) before you begin coding.

- Your code must be written in C and must be contained in files named **`mp9.c`** and **`mp9match.c`**. We will NOT grade files with any other name. **You may add fields to `vertex_t` if desired for your implementation of Dijkstra's algorithm, but you may not make other changes to other files except for debugging purposes.** Track any such changes with care, and make sure to test without them. If your code does not work properly without such changes, you are likely to receive 0 credit.

- You must implement the **`match_requests`**, **`find_nodes`**, **`trim_nodes`**, and **`dijkstra`** functions correctly.
- You may NOT make any assumptions about the values of preprocessor constants in **`mp9.h`**. You MUST use their symbolic names for full credit. We may choose to test your code with modified versions of **`mp9.h`**.
- You may assume that the parameter values passed into your **`match_requests`** function are valid (the output parameters will contain bits, of course). You must ensure that your routines are then passed valid parameters. We may test the individual routines mentioned with parameters other than those provided to you as examples. You may, however, also assume that both vertex sets passed into **`dijkstra`** contain at least one vertex.
- Your routine's return values and outputs must be correct.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook, and be sure to add function headers containing the information that has been provided for you in previous assignments (inputs, outputs, return value, and any side effects, as well as a brief description).

Compiling and Executing Your Program

When you are ready to compile, type:

```
make
```

Warnings and debugging information are turned on in the **`Makefile`**, so you can use **`gdb`** to find your bugs (you will have some).

If compilation succeeds, you can then execute the program by typing, "**`./mp9 graph requests`**" (no quotes). You can also specify other graph files or other request pairs, but you will have to create such files yourself (you may share them with other students for testing purposes). If your **`match_requests`** function returns 1, visualization information will be written into a file called **`result.c`**.

After creating the file, you can visualize the given result by typing:

```
make image
```

which will produce the file **`image.png`**. Be sure to put a copy of your **`mp5.c`** implementation into the **`MP9H`** directory before trying to make an image. If you make the image without executing **`mp9`**, the **`Makefile`** will execute **`mp9`** for you with the default arguments (the **`graph`** file and the **`requests`** file).

To clean up, type "**`make clean`**" (no quotes), or to really clean up, type "**`make clear`**" (as usual, no quotes).

Call for Help

In order to make this tool useful, we need to have a reasonable model for request generation. If you want to contribute, please feel free to identify geographic regions of the map that are more likely to serve as sources and destinations (such as dorms, residence halls, dining halls, places that people like to study, and so forth). We may also need a hub-and-spokes model (like the airlines) in order to get enough aggregation, so identifying likely safe congregation points for cross-campus walks would also be useful.

Grading Rubric

Functionality (60%)

- 15% - **`match_requests`** function works correctly
- 15% - **`find_nodes`** function works correctly
- 5% - **`trim_nodes`** function works correctly
- 25% - **`dijkstra`** function works correctly

Style (20%)

- 5% - **`find_nodes`** is reasonably efficient in recursing down the pyramid tree (no worse than the gold version)
- 5% - **`trim_nodes`** compresses the vertex array in place by removing out-of-range vertices
- 5% - heap implemented well for Dijkstra's algorithm
- 5% - uses only one execution of Dijkstra's algorithm to find shortest path from any starting vertex to any ending vertex

Comments, Clarity, and Write-up (20%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 5% - function headers are complete for all implemented functions (including those for heap or other support functions)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points.