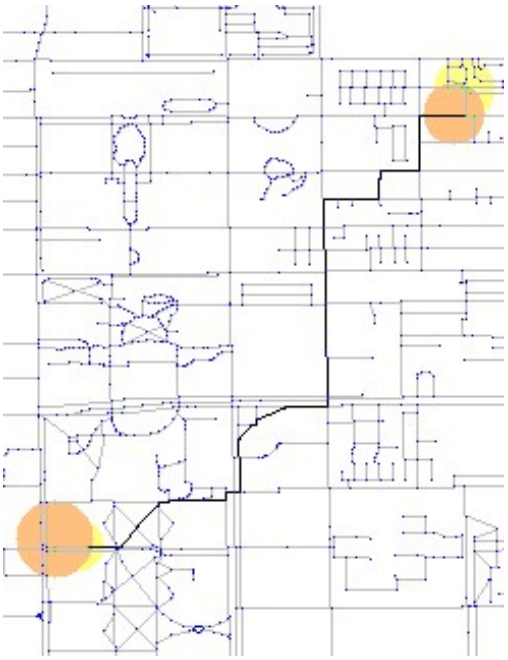
ECE220: Computer Systems and Programming **Spring 2018 Honors Section**
Machine Problem 10 **due: Saturday 7 April, 11:59:59 p.m.**

Mieber: Walk Me There, Part II



Your task this week is to extend last week's program to make use of dynamic allocation in several ways and to attempt to pair up requests from a list of many requests. For this purpose, you must also implement and use structures designed to help improve performance.

In particular, you must write C subroutines that manage dynamic allocation of vertex sets and paths, subroutines that enable, calculate, and use "mini maps" based on a high level of a pyramid tree, and a subroutine that attempts to find a partner for a request among a linked list of unpaired requests. Successful pairings must then be moved into a list of paired requests.

The objective for this week is for you to gain experience with dynamic allocation and singly-linked lists in C.

Background

The screenshot shown here was again generated automatically based on the results of the program. As before, 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.

This week's program reads a file consisting of many requests. For each request, your code must attempt to find a matching request from among those requests that have not already been matched. If a match is found, your code must find a path as before and record the trip information in a list of matched requests. If no match is found, your code must add the request to the list of unmatched requests.

In order to speed up comparisons, you must make use of a "minimap" bit vector based on a high level of the pyramid tree—the level with 64 nodes. For vertex sets and paths, each vertex is associated with an ancestor node at that level of the pyramid tree, and the minimap is the 64-bit value in which each of the 64 pyramid tree nodes is represented with a 1 if any of its children nodes appear in the vertex set (or path) and a 0 if none of them appear. By checking to see whether the minimaps overlap (with AND), one can quickly rule out overlap between distant locales and thus skip over any requests that cannot possibly match.

Pieces

Your program will consist of a total of five files:

`mp10.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. Changes from `mp9.h` are marked as such.**
`mp10.c`	The source file for your implementation of minimap and vertex set merging functions. A version has been provided to you with placeholders for the subroutines described in this document.
`mp10alloc.c`	The source file for your implementation of some dynamic allocation subroutines.
`mp10match.c`	The source file for your implementation of the **`handle_request`** function.
`mp9.c`	Your solution to MP9, which requires some modifications for MP10.

Six 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.
`mp10main.c`	The main program for MP10: reads the graph and requests files, builds the pyramid tree, allocates a heap for your code, and creates a file to illustrate the results of your code. You need not read this file, although you are welcome to do so.
`mp10graph.c`	A source file that contain functions for reading a graph file and freeing a graph. This file is used both by mp10 and by the request generation tool. You need not read this file, although you are welcome to do so.
`genreq.c`	The main program for a tool used to generate requests. You need not read this file, although you are welcome to do so.
`requests`	A file of requests to be matched one at a time against all unmatched requests. As before, 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. The version given to you was generated with the **`genreq`** tool (again, see Compiling and Executing Your Program) with parameters 100 12345 1055000 480000 1225000 650000 4000 8000 **on Cygwin**. My implementation matches 6 pairs among the 100 requests.

As in MP9, 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 250 extra lines.

You should first copy your **mp9.c** solution into your **MP10H** directory. Edit the file and change the **mp9.h** include at the top to include **mp10.h** instead. At the same time, add an include line for **stdlib.h**, which you need for dynamic allocation calls.

You need to modify two of the subroutines that your developed in MP9: **find_nodes** and **dijkstra**. These routines must now use dynamic allocation for the **id** fields (the array of graph vertex ids) of the vertex set and the path, respectively.

For **find_nodes**, you must modify your code in two ways. First, the `id` array is no longer defined as a fixed-size array. Instead, you should use dynamic resizing to grow the array as necessary. If **malloc** or **realloc** fails, simply skip insertion of that vertex' id (as did the earlier version when it ran out of space in the array). The second modification needed is to build the array of vertex ids in increasing order (in other words, sorted from smallest to largest). Sorting the arrays makes intersecting two such arrays faster.

For **dijkstra**, you should allocate an array of appropriate size after you have determined the length of the path (including the starting and ending nodes, just as before), but before writing the graph vertex ids into the array. If allocation fails, the function should also fail by returning 0.

You may next want to implement dynamic allocation calls for vertex sets and paths in **mp10alloc.c**. These routines should be used to create and free vertex sets and paths. Allocation calls should return pointers to the new objects, or **NULL** on failure. Be sure to initialize at least the **count** and **id** fields for vertex sets, and at least the **id** field for paths. Note that the **id** field of either type may be **NULL** when either type is freed, so be careful not to free a **NULL** pointer. The signatures for the four routines are:

```
vertex_set_t* new_vertex_set ();

void free_vertex_set (vertex_set_t* vs);

path_t* new_path ();

void free_path (path_t* path);
```

After the allocation routines, we suggest implementing the minimap functionality in **mp10.c**. Begin by writing a function to mark each vertex in a graph with a bit indicating which node in the fourth level of a pyramid tree is the ancestor of the pyramid tree node corresponding to the vertex. The **vertex_t** structure has been extended with an **mm_bit** field to hold this information, but the field for each vertex must be filled in by your function. These bit numbers will be used to implement the new 64-bit minimap fields of the vertex sets and the paths. Note that the fourth level of a pyramid tree has 64 nodes. The signature for the function is:

```
int32_t mark_vertex_minimap (graph_t* g, pyr_tree_t* p);
```

If the graph has 64 vertices or fewer, simply use each vertex' array index for the vertex' **mm_bit** field. Otherwise, for each graph vertex, your function must identify the pyramid tree node corresponding to each vertex, then find the ancestor node with index from 21 to 84 (these are the nodes at the fourth level of the pyramid tree). Finally, set the vertex' **mm_bit** to the ancestor's node index minus 21. Note that the description here is chosen for clarity in the definition; you may want to use a more efficient approach to finding the necessary information.

Once you have the vertices marked, you can implement the following two functions to build minimaps for the vertex sets and paths:

```
void build_vertex_set_minimap (graph_t* g, vertex_set_t* vs);

void build_path_minimap (graph_t* g, path_t* vs);
```

Note that the graph contains the minimap bit information for each vertex, so you can calculate the vertex set (or path's) minimap field by ORing together the bits for all ids in the vertex set (or the path). Use **`1ULL`** to obtain a 64-bit constant before shifting; if your code shifts a 32-bit constant by more than 31 bits, the results are undefined.

The last function in **`mp10.c`** merges two vertex sets into a third set. This function is used to check for overlap between the vertex sets of two requests. The function has the following signature:

```
int32_t merge_vertex_sets (const vertex_set_t* v1, const vertex_set_t* v2,
                           vertex_set_t* vint);
```

This function finds the intersection of two vertex sets **`v1`** and **`v2`** and writes the results into **`vint`**. The function should assume that the **`id`** array in **`vint`** has enough space to hold the intersection of the sets (as mentioned below, your handle_request function must guarantee this property). If the resulting intersection is non-empty, the function returns 1. Otherwise, it returns 0.

The primary function for this MP appears in **`mp10match.c`** and has the following signature:

```
int32_t handle_request (graph_t* g, pyr_tree_t* p, heap_t* h, request_t* r);
```

This function processes a single request, allocating structures as necessary, then comparing it with all unpaired requests handled previously. If a match is found, the function moves the new request and the matching request into a separate list.

Two file-scope lists are used for recording requests: **`available`** is the singly-linked list of unpaired requests. Any unmatched request is added to this list. The **`shared`** list is the list of paired requests. Initially, of course, both lists are empty.

The **`available`** list is a singly-linked list connected by the **`next`** field of the **`request_t`** structure. The **`partner`** field of all requests in this list should be **`NULL`**. The source and destination vertex sets should be dynamically allocated vertex sets that correspond to the request. The **`path`** field should be **`NULL`**.

The **`shared`** list is also a singly-linked list connected by the **`next`** field of the **`request_t`** structure. When a request is matched to a second request in the **`available`** list, the second request should be removed from the **`available`** list and referenced through the **`partner`** field of the first request. The **`partner`** and **`next`** fields of the second request should then be set to NULL, and the first request should be added to the **`shared`** list (using the first request's **`next`** field). For pairs of requests in the **`shared`** list, both requests should point to the same two vertex sets and to the same path structure. These vertex sets should correspond to the intersection of the two requests, and the path should be that found by the **`dijkstra`** routine.

Here's a little more detail on what your handle_request routine must do:

- Allocate vertex sets for the request and for use with matching.
- Allocate a path for use with matching.
- If any allocation fails, clean up (free any successful allocations) and return 0.
- Fill in the vertex sets for the request. If either is empty, the function should return 0. Be sure to free any vertex sets and path allocated by the function before returning.
- Create minimaps for the request's source and destination vertex sets.

- Allocate **id** arrays for the vertex sets to be used for matching. Make these large enough that they can hold any intersection with the request's corresponding vertex set. Note that you must clean up and return 0 if allocation fails.
- Try to find a matching request in the **available** list. Skip any requests that have no overlap in the source vertex set minimaps or no overlap in the destination vertex set minimaps. For those with overlap, use **merge_vertex_sets** to check for non-empty intersections, then call **dijkstra** to find a path (if both are non-empty). If everything succeeds, pair the two requests.
- If no match is found, add the request to the **available** list and return 1 for success. Again, be sure to free any dynamically allocated structures (specifically, those used for testing matches).
- If a matching request is found, remove the matching request from the **available** list, free both requests vertex sets (these should be replaced with the intersected vertex sets), and make both requests point to the intersected vertex sets and the path filled in by **dijkstra**. Be sure to build minimaps for both intersected vertex sets and for the path. Link the two requests as described previously into the **shared** list and return 1 for success.

Specifics

Be sure that you have read the type definitions and other information in the code and header file before you begin coding.

- Your code must be written in C and must be contained in files named **mp9.c**, **mp10.c**, **mp10alloc.c**, and **mp10match.c**. We will NOT grade files with any other names. **If you added fields to vertex_t for MP9, you must merge them into mp10.h yourself. As with MP9, 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 modify the **find_nodes** and **dijkstra** functions correctly.
- You must write the **new_vertex_set**, **free_vertex_set**, **new_path**, **free_path**, **mark_vertex_minimap**, **build_vertex_set_minimap**, **build_path_minimap**, **merge_vertex_sets**, and **handle_request** functions correctly.
- You may assume that the parameter values passed into your **handle_request** function are valid. 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.
- You may assume that only dynamically allocated vertex sets are passed into **free_vertex_set**, and that only dynamically allocated paths are passed into **free_path**, provided that your code does not violate these assumptions.
- You may assume that the pyramid tree passed with the graph was built from that graph.
- Note that you should assume that the vertex sets passed into **merge_vertex_sets** are sorted, but you are responsible for guaranteeing that property by modifying **find_nodes**.
- 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, "**./mp10 graph requests**" (no quotes). You can also specify other graph files or other requests files, but you will have to create such files yourself (you may share them with other students for testing purposes). The program will print a list of matched and unmatched requests (their identifiers are the line numbers in the requests file, starting with 0 instead of 1).

If you want to visualize a particular matched pair or unmatched request, execute with an extra argument: the id of the desired request (either for a matched pair). The program will then also create the **result.c** file. 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 **MP10H** directory before trying to make an image. If you make the image without executing **mp10**, the **Makefile** will execute **mp10** for you with the default arguments (the **graph** file and the **requests** file and id 0).

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

To make the request generation tool, type:

```
make genreq
```

You can then run the tool to generate new requests files at random. Run the tool without arguments for instructions. Some helpful values: 1st St. is around X=970,000. Vine St. is around X=1,310,000. University Ave is around Y=480,000. Pennsylvania Ave. is around Y=650,000. Feel free to identify more yourselves and to share such findings with others. The units are 10 millionths of degrees of latitude/longitude (about 1.11 meters / unit), so you should be able to make calculations, too…roughly. The aspect ratios aren't quite right.

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 (65%)

- 8% - **`find_nodes`** function works correctly
- 3% - **`dijkstra`** function works correctly
- 4% - **`new_vertex_set`** function works correctly
- 3% - **`free_vertex_set`** function works correctly
- 4% - **`new_path`** function works correctly
- 3% - **`free_path`** function works correctly
- 10% - **`mark_vertex_minimap`** function works correctly
- 3% - **`build_vertex_set_minimap`** function works correctly
- 3% - **`build_path_minimap`** function works correctly
- 6% - **`merge_vertex_sets`** function works correctly
- 18% - **`handle_request`** function works correctly

Style (15%)

- 5% - **`mark_vertex_minimap`** avoids searching pyramid tree for each graph vertex
- 5% - **`merge_vertex_sets`** uses fact that **`id`** arrays are sorted to avoid quadratic running time
- 5% - **`handle_request`** uses minimaps to avoid checking matches when possible

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 any support functions that you create)
- 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.