Assignment 4 Report
Surfin' U.S.A.

By Jason Zhang
Prof. Veenstra
CSE 13S

Due May 21, 2023

Purpose

This program uses graph theory to solve a version of the traveling salesman problem. Data structures will be in graph.c, stack.c, and path.c. These functions will then be accessed through one .c file tsp.c which takes in command line options and implements a harness to combine the algorithms to solve the traveling salesman problem by finding the shortest Hamiltonian cycle to the directed or undirected graph.

How to Run

Run the program from command line using a Makefile:

```
make
```

To create a version for debugging:

```
make debug
```

The program can then be ran directly:

```
./tsp
```

Or with an input txt file:

```
./tsp < input.txt > output.txt
```

Possible command line options for sorting.c include:

```
./sorting -i -o -d -h
```

Of which,

-i : Sets the file to read from (input file). Requires a filename as an argument. The default file to read from is stdin

-o : Sets the file to write to (output file). Requires a filename as an argument. The default file to write to is stdout

-d : Treats all graphs as directed. The default is to assume an undirected graph.

-h : Prints a help message to stdio.

Program Design

The source code for the tsp program is found in 4 files. Of which, `graph.c, stack.c, path.c,` contain the data structures used. These structures are mainly composed of many functions that help add, remove, modify, etc data stored. The `tsp.c` file contains a `main()` function that the program runs with. In addition, the program uses existing libraries from C such as `stdio.h` and `unistd.h`.

Data Structures

The program uses customary `Graph, Stack, Path` structures to store data, as well as `int` for indexing into the arrays. Pseudocode as follows:

```
typedef struct graph {
	uint32_t vertices;
	bool directed;
	bool *visited;
	char **names;
	uint32_t **weights;
} Graph;

typedef struct stack {
	uint32_t capacity;
	uint32_t top;
	uint32_t *items;
} Stack;

typedef struct path {
	uint32_t total_weight;
	Stack *vertices;
} Path;
```

Algorithms

The tsp program utilizes these following steps:

1. Path Validity Check: The algorithm ensures that the chosen path allows for traversal to all desired destinations, adhering to defined criteria.
2. Return Route Evaluation: After finding a valid path, the algorithm checks if the destination node has a connection or path back to the starting node. It may evaluate additional factors such as fuel efficiency.
3. Optimal Path Determination: The algorithm compares alternative paths encountered during the traversal, updating the current best path if a more efficient option is found. The final result is the most optimal path based on defined criteria.

Function Descriptions

Each data structure algorithm performs steps below:

Graph

- **Graph *graph_create(uint32_t vertices, bool directed)**
  - input: int32_5 vertices, bool directed
  - output: Graph *
  - ```
    def Graph *graph_create(uint32_t vertices, bool directed):
                calloc() memory for Graph struct
                graph -> vertices
                graph -> directed
                graph -> visited = calloc() memory
                graph -> weights = calloc() memory
                populate graph -> weights with calloc()
                return graph
    ```
- **void graph_free(Graph **gp)**
  - input: Graph **gp
  - output: void
  - ```
    def graph_free(gp):
          if gp is None or gp[0] is None:
          return
          g = gp[0]
          # Free memory for visited array
          free g['visited']

          # Free memory for names and weights arrays for each vertex
          for i in range(g['vertices']):
                free g['names'][i]
                free g['weights'][i]

          # Free memory for weights and names arrays
          free g['weights']
          free g['names']

          # Free memory for the graph structure
          free g

          # Set the graph pointer to None
          gp[0] = None
    ```
- **uint32_t graph_vertices(const Graph *g)**

    - input: const Graph *g
    - output: uint32_t
    - `Finds the number of vertices in a path.`
- **void graph_add_vertex(Graph *g, const char *name, uint32_t v)**
    - Gives the city at vertex v the name passed in, makes a copy of the name and stores it in the graph object.
    - ```
      def graph_add_vertex():
            if graph->name already exists:
                  remove name
            set graph->name to copy of name
      ```
- **const char* graph_get_vertex_name(const Graph *g, uint32_t v)**
    - Gets the name of the city with vertex v from the array of city names.
- **char **graph_get_names(const Graph *g)**
    - Gets the names of every city in an array.
- **void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)**
    - Adds an edge between int start and int end with weight to the adjacency matrix of the graph.
    - ```
      def graph_add_edge():
            if directed:
                  graph->weightsp[start][end = weight
            otherwise:
                  g->weights[start][end] = weight
                  g->weights[end][start] = weight
      ```
- **uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)**
    - Returns the weight of the edge between start and end.
- **void graph_visit_vertex(const Graph *g, uint32_t v)**
    - Adds the vertex v to the list of visited vertices.
- **void graph_unvisit_vertex(Graph *g, uint32_t v)**
    - Removes the vertex v from the list of visited vertices.
- **bool graph_visited(Graph *g, uint32_t v)**
    - Returns true if vertex v is visited in graph g, false otherwise.
- **void graph_print(const Graph *g)**
    - Prints the graph, helps for debugging.
    - ```
      def graph_print():
            print all names in graph
            print all weights in graph using:
            printf("%2u ", weights[i][j])
      ```
      ```

Stack

- **Stack *stack_create(uint32_t capacity)**
  - input: uint32 capacity
  - output: Stack *
  - ```
    def create_stack(capacity):
        s = {}  # Create an empty dictionary for the stack
        # Allocate memory for the stack
        s['capacity'] = capacity
        s['top'] = 0
        # Allocate memory for items array
        s['items'] = [0] * s['capacity']
        # Return the created stack
        return s
    ```
- **void stack_free(Stack **sp)**
  - input: Stack **sp
  - output: void
  - ```
    def free_stack(sp):
        if sp is not None and sp[0] is not None:
        # Free memory for the array of items first
            if sp[0]['items']:
                free(sp[0]['items'])
                sp[0]['items'] = None
        # Free memory allocated for the stack
        free(sp[0])
        if sp is not None:
            # Set the pointer to None to avoid double free
            sp[0] = None
    ```
- **bool stack_push(Stack *s, uint32_t val)**
  - input: Stack *s, uint32_t val
  - output: bool
  - ```
    def stack_push():
        Add val to top of stack
        increment counter
        return true if successful
        return false if stack full
    ```
- **bool stack_pop(Stack *s, uint32_t *val)**
  - Sets the integer pointed to by val to the last item on the stack, and removes the last item on the stack. Returns true if successful, false otherwise.
  - ```
    def stack_pop():
        exit if stack is empty
        set val pointer value to stack->items[top-1]
    ```

```
top -= 1
return true
```

- **bool stack_peek(const Stack *s, uint32_t *val)**
  - Sets the integer pointed to by val to the last item on the stack, but does not modify the stack. Returns true if successful, false otherwise.
  - ```
    def stack_peek():
        exit if stack is empty
        set val pointer value to stack->items[top-1]
        return true
    ```
- **bool stack_empty(const Stack *s)**
  - Returns true if the stack is empty, false otherwise.
  - `return whether or not top == 0`
- **bool stack_full(const Stack *s)**
  - Returns true if the stack is full, false otherwise
  - `return whether or not top == capacity`
- **uint32_t stack_size(const Stack *s)**
  - Returns the number of elements in the stack.
- **void stack_copy(Stack *dst, const Stack *src)**
  - Overwrites dst with all the items from src.
  - ```
    def stack_copy():
        exit if dst and src capacities are different
        iterate through src_stack->items:
            copy each item from src to dst
        set dst->top to stc->top
        return
    ```

Path

- **Path *path_create(uint32_t capacity)**
  - input: uint32_t
  - output: Path *
  - ```
    def path_create():
        stack_create(capacity, weight=0)
        total_weight = 0
        return
    ```
- **void path_free(Path **pp)**
  - Frees a path, and all its associated memory.
  - ```
    def path_free():
        exit if either pointers are NULL
        stack_free path->vertices
    ```

```
free *pp
*pp = NULL
```

- **uint32_t path_vertices(const Path *p)**
  - Finds the number of vertices in a path.
- **uint32_t path_distance(const Path *p)**
  - Finds the distance covered by a path.
- **void path_add(Path *p, uint32_t val, const Graph *g)**
  - Adds vertex val from graph g to the path.
  - ```
    def path_add():
        if path->vertices is empty:
            path->total_weight = 0
        else:
            weight = graph_get_weight(graph, last vertex, val)
            path->total_weight += weight
        stack_push(path->vertices, val)
        return
    ```
- **uint32_t path_remove(Path *p, const Graph *g)**
  - Removes the most recently added vertex from the path.
  - ```
    def path_remove():
        exit if path->vertices is empty
        else:
            stack_pop(path->vertices, last vertex)
            if path has more than 1 vertices:
                stack_peek(p->vertices, last last vertex)
                distance = graph_get_weight(last last vertex,
    last vertex)
                path->total_weight -= distance
            else (if path has only 1 vertex):
                path->total_weight = 0
        free pointers
        return
    ```
- **void path_copy(Path *dst, const Path *src)**
  - Copies a path from src to dst.
  - ```
    def path_copy():
        exit if bad pointers
        set dst->total_weight to src->total_weight
        stack_copy(dst->vertices, src->vertices)
        return
    ```
    ```

tsp

- **main**
  - uses getopt to process arguments
  - set infile and outfile to stdin and stdout unless user options prompt otherwise
  - uses fscanf to read input files and create Graph *g accordingly
  - runs dsf()
  - if bestPath after dsf() has 0 vertices, we know that there is not a hamiltonian cycle in the Graph, so write to outfile: “No path found! Alissa is lost!”
  - otherwise, path_print() the bestPath
  - free associated paths and graphs
- **void dsf(uint32_t n, Graph *g, uint32_t start, Path *curr, Path *best)**
  - input: int current_node/vertex, Graph g, int starting_node, Path current_path, Path best_path
  - output: since current_path and best_path are pointers, those Path structures are directly modified. The function itself returns void.
  - Performs a depth first search (DFS) graph search
  - ```
    def dsf():
        visit(current vertex)
        add current vertex to currPath
        if (all vertices are visited):
            if (current vertex has path back to start):
                get dist total_distance of full cycle
                if (bestPath is empty) or (dist < bestPath.dist):
                    path_add(curr->start)
                    path_copy(curr->best)
                    path_remove(curr)
        for (i in all adjacent vertices):
            if (there is a path to i):
                if (i is not visited yet):
                    dsf(i, g, start, curr, best)
        unvisit(current vertex)
        remove current vertex from currPath
    ```

Error Handling

`getopt()` includes error handling where in the event that the user’s input option does not exist, an error message will be printed to stderr. In addition, when using -h, a help message appears on stdout. Functions for the custom data structures also contain error checking to make sure pointers passed in aren’t NULL. As per pseudocode above, when invalid pointers are passed into the functions, they will exit and display an error. Destructive functions for data structures involving heap memory allocation are thoroughly tested through valgrind to ensure no memory leaks.
    ```

Result

Program works as intended. Valgrind: All heap blocks were freed -- no leaks possible. No false positives reported by scan-build. No known bugs or errors.

What I learned:

- From running clique 9-13.graph from maps, it seems that each subsequently larger graph takes exponentially longer times to run. I learned that depth-first-search has a rather high complexity and thus runtime.
- I also found an interesting online tool to plot graphs from adjacency matrices: https://graphonline.ru/en/ (see figure 1).
- I've gained significantly more knowledge on how to use `Valgrind`.
- I learned more about input streams from reading and writing to files.
- Attempted `lldb` debugging, with some useful outcome.

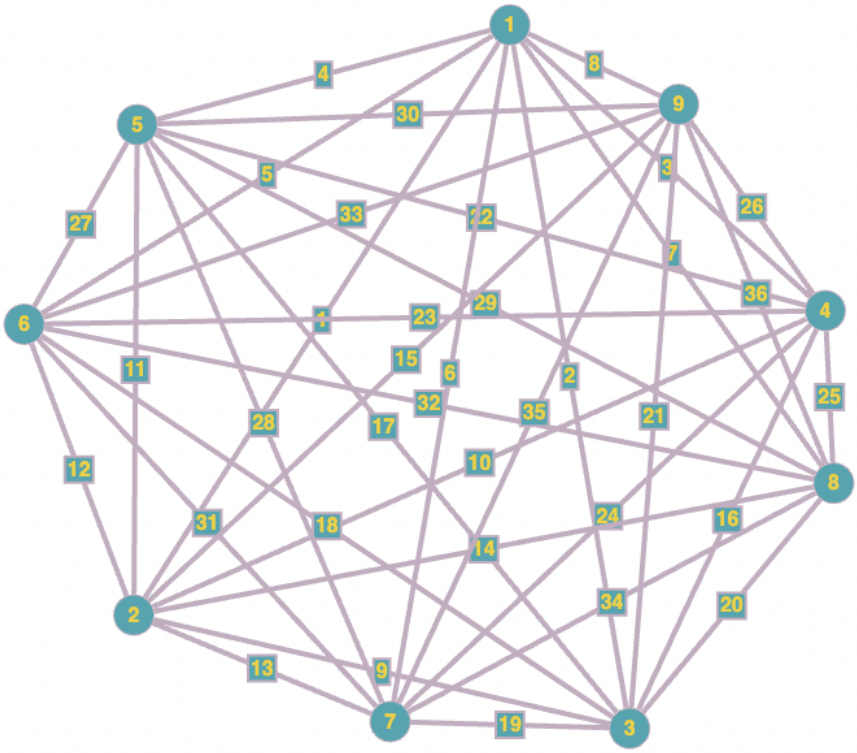

Figure 1. Screenshot of online graphing tool

```
==28505== All heap blocks were freed -- no leaks are possible
==28505==
==28505== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jasonz@vm-cse13:~/desktop/cse13s/asgn4$ ./tsp -i maps/bayarea.graph
Alissa starts at:
Santa Cruz
Half Moon Bay
San Mateo
Daly City
San Francisco
Oakland
Walnut Creek
Dublin
Hayward
San Jose
Santa Cruz
Total Distance: 203
jasonz@vm-cse13:~/desktop/cse13s/asgn4$ ./tsp -i maps/bayarea.graph -o out.txt
jasonz@vm-cse13:~/desktop/cse13s/asgn4$ ls
graph.c       Makefile  path.o       stack.c       tsp              vertices.h
graph.h       maps      path_test.c  stack.h       tsp-arm
graph.o       out.txt   path_test.o  stack.o       tsp.c
graph_test.c  path.c    README.md    stack_test.c  tsp.o
graph_test.o  path.h    report.pdf   stack_test.o  valgrind-out.txt
jasonz@vm-cse13:~/desktop/cse13s/asgn4$ cat out.txt
Alissa starts at:
Santa Cruz
Half Moon Bay
San Mateo
Daly City
San Francisco
Oakland
Walnut Creek
Dublin
Hayward
San Jose
Santa Cruz
Total Distance: 203
```

Figure 2. Screenshot of program running