Assignment 5 – Surfin' U.S.A.

Sebastian Avila

CSE 13S – Fall 2023

Purpose

The purpose of this program is to find a route that visits each city using the smallest amount of gas for a set of cities in a graph.

How to Use the Program

To use this program you will need to have several files in the same directory. The files are `tsp.c\,`, `graph.c`, `graph.h`, `stack.c`, `stack.h`, `path.c`, `path.h`, `vertices.h`, and `Makefile`. With these files in the same directory, use command `make clean` to remove any preexisting object files. Then, use command `make format` to format all the header and source files. Then the same directory use command `make` to compile the program. Then run the program using `./tsp` with command line argument(s). The possible command line arguments and their meanings are as follows:

- `-i` : Sets the input file. It requires a file name as an argument. The default file to read from is `stdin`.
- `-o` : Sets the output file. It requires a file name as an argument. The default out file is `stdout`.
- `-d` : Tells the program to treat all graphs as directed. The default rule is to assume an undirected graph.
- `-h` : Prints a help message to `stdio`.

The program uses several optional compiler flags:

- `-Wall`: This flag enables all warning messages.
- `-Werror`: This flag turns all warnings into errors.
- `-Wextra`: This flag enables extra warning flags that are not enabled by `-Wall`.
- `-Wstrict-prototypes`: This flag warns if a function is declared or defined without specifying the argument types.
- `-pedantic`: This flag issues all the warnings demanded by strict ISO C and ISO C++.
- `-lm`: This flag links the math.h library. This allows the program to access and use the functions from the math.h library.

Program Design

The program begins by creating boolean variables to represent whether different command line arguments have been entered. It then checks the command line arguments and updated values as needed. It then designates a file pointer to stdin. This file pointer is overwritten if the command line argument `i` is used. The same happens with the out file and the command line argument `o`. The program then creates a graph data structure and parses the input file for input. If an error occurs a message is printed and the program terminates. It then creates a two path data structures and runs a depth frist search to find the shortest path. Once it traverses every possible paths it either prints the shortest path or if there are no paths it prints a message that alyssa is lost. It then returns 0.

Data Structures

Graph

This program creates a type `Path`. It is a struct with a 32 bit integer variable, a boolean variable, a pointer to a boolean array, a double pointer to an array of strings, and a double pointer to a 32 bit integer:

- `vertices` : This 32 bit integer represents how many vertices the graph has.
- `directed` : This boolean variable represents whether the edges are directed (true) or not (false).
- `*visited` : This is an array of boolean variables that represents whether the vertexes have been visited (true) or not (false).
- `**names` : This is an array of strings. They are the names of the vertices on the graph.
- `**weights` : This is an adjacency matrix used to represent the edges and their weights.

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

Stack

This program creates a type `Stack`. It is a struct with two 32 bit integer variables and one pointer to a 32 bit integer. These are:

- `capacity` : This is the maximum number of elements of the stack
- `top` : This variable tracks where the value of the top of the stack is in the computers memory.
- `*items` : This is a pointer to an array of elements that make up the stack.

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

Paths

This program creates a type `Path`. It is a struct with a 32 bit integer variable and a pointer to a stack:

- `total_weight` : This 32 bit integer represents the weight of the path.
- `*vertices` : This is a pointer to a stack of vertices.

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

Algorithms

Depth First Search

This algorithm travels all possible paths in a graph. It was given in the assignment instructions. [1] Its specific implementation for this program is seen in the descriptions.

```
def dfs(node n, graph g):
       mark n as visited
       for every one of n’s edges:
           if (edge is not visited):
               dfs(edge, g)
       mark n as unvisited
```

Function Descriptions

TSP

- `void dfs(Graph *g, Path *current_path, Path *shortest_path, uint32_t shortest_distance)` : This function takes in a pointer to a graph `g`, two pointers to paths `current_path` and `shortest_path`, and a 32 bit integer `shortest_distance` . It does not return anything. This function implements a depth first search which will traverse every possible path of a graph and stores the shortest path in `shortest_path`.

```
void dfs(Graph *g, Path *current_path, Path *shortest_path, uint32_t shortest distance)
    current_vertex = path_vertices(current_path)

    if current_vertex = graph_vertices(g) then
        if (graph_get_weight(g, path_remove(current_path, g), path_vertices(current_path)) +
             path_distance(current_path) < shortest_distance) then
            path_copy(shortest_path, current_path)
            shortest_distance = path_distance(shortest_path)
        return

    for i, 0 to i < graph_vertices(g)
        if (!graph_visited(g, i) then
            graph_visit_vertex(g, i)
            path_add(current_path, i, g)

            dfs(g, current_path, shortest_path)

            graph_unvisit_vertex(g, i)
            path_remove(current_path, g)
```

Graph

- `Graph *graph_create(uint32_t vertices, bool directed)` : This function takes in a 32 bit integer `vertices` and a boolean variable `directed`. It returns a pointer to a graph. Its purpuse is to create a new graph struct and return a pointer to it. It initializes all items in the `visited` array to `false`. This function was given in the assignment instructions. [1]

```
Graph *graph_create(uint32_t vertices, bool directed)
    *g = memory for 1 item with size of graph
    g veritces = verices
    g directed = directed

    g visited = memory set to 0 for # of vertices with size of bool
    g names = memeory set to 0 for # of vertices with size of character pointer
    g weights = memory for # of vertices with size of g weights[0]

    for i, 0 to i < vertices
        g weights[i] = memory for # of vertices with size of g weights[0][0]

    return g
```

- `void graph_free(Graph **gp)` : This function takes in a double pointer to a graph `gp`. It does not return anything. Its purpose is to free all the memory used by the graph.

```
void graph_free(Graph **gp)
    if gp not NULL and *gp not NULL then
        if *gp visited not NULL then
            free(*gp visited)
            *gp visited = NULL
        while *gp names not NULL
            free(*gp *names)
            *gp names++
        while *gp weights not NULL
            free(*gp *weights)
            *gp weights++

    if gp not NULL
        *gp = NULL
```

- `uint32_t graph_vertices(const Graph *g)` : This function takes in a pointer to a constant graph `g`. It returns a 32 bit integer. Its purpose is to find the number of vertices in a graph and return the value.

```
uint32_t graph_vertices(const Graph *g)
    v = g vertices
    return v
```

- `void graph_add_vertex(Graph *g, const char *name, uint32_t v)` : This function takes in a graph pointer `g`, a constant character string `name`, and a 32 bit integer `v`. It does not return anything. Its purpose is to give the city at vertex `v` the name passed in. This function is given in the assignment instructions. [1]

```
void graph_add_vertex(Graph *g, const char *name, uint32_t v)
    if g names[v] then free g names[v]
    g names[v] = strdup(name)
```

- `const char* graph_get_vertex_name(const Graph *g, uint32_t v)` : This function takes in a pointer to a constant graph `g` and a 32 bit integer `v`. it returns a constant character string. It purpose is to get the name of the city with vertex `v` and return it.

```
const char* graph_get_vertex_name(const Graph *g, uint32_t v)
    return g names[v]
```

- `char **graph_get_names(const Graph *g)` : This function takes in a pointer to a constant graph `g`. It returns a double pointer - an array of strings.

```
char **graph_get_names(const Graph *g)
    return g names
```

- `void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)` : This function takes in a pointer to a graph `g`, and three 32 bit integers: `start`, `end`, and `weight`. It does not return anything. Its purpose is to add an edge between `start` and `end` with a weight `weight` to the adjacency matrix of the graph.

```
void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)
   if g directed false then
      g weights[end][start] = weight
   g weights[start][end] = weight
```

- `uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)` : This function takes in a pointer to a constant graph `g` and two 32 bit integers: `start` and `end`. It returns a 32 bit integer. Its purpose is to look up the weight of the edge between `start` and `end` and return it.

```
uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)
   return g weights[start][end]
```

- `void graph_visit_vertex(Graph *g, uint32_t v)` : This function takes in a pointer to a graph `g` and a 32 bit integer `v`. It does not return anything. Its purpose is to add the vertex `v` to the list of visited vertices.

```
void graph_visit_vertex(Graph *g, uint32_t v)
   g visited[v] = true
```

- `void graph_unvisit_vertex(Graph *g, uint32_t v)` : This function takes in a pointer to a graph `g` and a 32 bit integer `v`. It does not return anything. Its purpose is to remove the vertex `v` from the list of visited vertices.

```
void graph_unvisit_vertex(Graph *g, uint32_t v)
   g visited[v] = false
```

- `bool graph_visited(Graph *g, uint32_t v)` : This function takes in a pointer to a graph `g` and a 32 bit integer `v`. It returns either true or false. Its purpose is to return true if vertex `v` is visited in graph `g` and return false otherwise.

```
bool graph_visited(Graph *g, uint32_t v)
   return g visited[v]
```

- `void graph_print(const Graph *g)` : This function takes in a pointer to a constant graph `g`. It does not return anything. Its purpose is to print a human-readable representation of a graph.

```
void graph_print(const Graph *g)
   print(num of vertices : g vertices)
   print(is directed: g directed)

   length = size of g names / size of g names[0]
   for i, 0 to i < length
      print(names[i])

   length = size of g visited / size of g visited[0]
   for i, 0 to i < length
      print(visited[i])

   length = size of g weigths / size of g weights[0]
   for i, 0 to i < length
      print(weights[i])
```

Stack

- `Stack *stack_create(uint32_t capacity)` : This function takes in a 32 bit integer `capacity`. It returns a pointer to a stack that it creates. It purpose is to create a stack, dynamically allocate space for it, then return a pointer to it. This function was given in the assignment pdf. [1]

```
Stack *stack_create(uint32_t capacity)
    stack pointer s = memory of size(sizeof(Stack)) cast to type Stack
        s capcacity = capacity
        s stop = 0

        s items = memory for s capacity number of 32 bit integers

        return s
```

- `void stack_free(Stack **sp)` : This function takes in a Stack double pointer `sp`. It does not return anything. Its purpose is to free all space used by the stack pointed to by `sp` and to set the pointer to NULL. This function was given in the assignment pdf. [1]

```
void stack_free(Stack **sp)
    if sp not NULL and sp pointer not NULL then
        if sp pointer items not NULL then
            free(sp pointer items)
            sp pointer items = NULL

        free(sp pointer)

    if sp not NULL
        sp pointer = NULL
```

- `bool stack_push(Stack *s, uint32_t val)` : This function takes in a pointer to a stack `s` and a 32 bit integer `val`. It returns either true or false. Its purpose is to add `val` to the top of the stack and increment the counter. If this operation is successful, it returns true. If not, it returns false. This function was given in the assignment pdf. [1]

```
bool stack_push(Stack *s, uint32_t val)
    if stack is full then
        return false

    s items[s top] = val
    s top = s top + 1

    return true
```

- `bool stack_pop(Stack *s, uint32_t *val)` : This function takes in a pointer to a stack `s` and a pointer to a 32 bit integer `val`. It returns either true or false. Its purpose is to set the integer pointed to by `val` to the item on the top of the stack and remove that item from the stack. If this happens successfully, it returns true. If not, it returns false.

```
bool stack_pop(Stack *s, uint32_t *val)
    if stack is empty then
        return false

    *val = s items[top - 1]
    s top = s top - 1

    return true
```

- `bool stack_peek(const Stack *s, uint32_t *val)` : This function takes in a pointer to a constant stack `s` and a pointer to a 32 bit integer `val`. It returns either true or false. Its purpose is to set the integer pointed to by `val` to the value at the top of the stack. If this operation succeeds, true is returned. Otherwise, its returns false.

```
bool stack_peek(const Stack *s, uint32_t *val)
   if stack is empty then
      return false

   *val = s items[s top - 1]
   return true
```

- `bool stack_empty(const Stack *s)` : This function takes in a pointer to a constant stack `s`. It returns either true or false. Its purpose is to check whether the stack `s` is empty. If it is it returns true; it returns false otherwise.

```
bool stack_empty(const Stack *s)
   if s top = 0 then
      return true
   return false
```

- `bool stack_full(const Stat *s)` : This function takes in a pointer to a constant stack `s`. It returns either true or false. Its purpose is to check whether the stack `s` is full. If it is it returns true; it returns false otherwise.

```
bool stack_full(const Stack *s)
   if s top >= s capacity then
      return true
   return false
```

- `uint32_t stack_size(const Stack *s)` : This function takes in a pointer to a constant stack `s`. It returns a 32 bit integer. Its purpose is to return the number of elements in the stack `s`.

```
uint32_t stack_size(const Stack *s)
   return s top
```

- `void stack_copy(Stack *dst, const Stack *src)` : This function takes in a stack pointers, `dst` and a constant stack pointer, `src`. It does not return anything. Its purpose is to copy the stack `src` to the stack `dst`.

```
void stack_copy(Stack *dst, const Stack *src)
   assert dst capacity <= src capacity
   dst top = s top
   memcopy(to dst items, from src items, for dst capacity * sizeof(uint32_t) bits)
```

- `void stack_print(const Stack *s, FILE *outfile, char *cities[])` : This function takes in three parameters: a pointer to a constant stack `s`, a pointer to a file `outfile`, and a pointer to a character string `cities[]`. It does not return anything. Its purpose is to print out the stack as a list of elements, given a list of vertex names, starting with the bottom of the stack. This function was given in the assignment pdf. [1]

```
void stack_print(const Stack *s, FILE *outfile, char *cities[])
   for i, 0 to i < s top
      print cities[s items[i]]\newline to outfile
```

Paths

- `Path *path_create(uint32_t capacity)` : This function takes in a 32 bit integer `capacity`. It returns a pointer to a Path data structure containing a stack and a weight of zero. Its purpose is to create the Path.

```
Path *path_create(uint32_t capacity)
   path pointer p = memory of size(sizeof(Path)) cast to type Path
   p vertices = create stack(capacity)
   p total_weight = 0
   return p
```

- `void path_free(Path **pp)` : This function takes in a double pointer to a path `pp`. It does not return anything. Its purpose is to free the path and all its allocated memory.

```
void path_free(Path **pp)
   if pp not NULL and pp pointer not NULL then
      if pp vertices not NULL then
         stack_free(pp vertices)
      free(pp pointer)

   if pp not NULL then
      pp pointer = NULL
```

- `uint32_t path_vertices(const Path *p)` : This function takes in a pointer to a constant Path `p`. It returns a 32 bit integer. Its purpose is to find the number of vertices in the path and return that value.

```
uint32_t path_vertices(const Path *p)
   v = stack_size(p vertices)
   return v
```

- `uint32_t path_distance(const Path *p)` : This function takes in a pointer to a constant Path `p`. It returns a 32 bit integer. Its purpose is to find the distance covered by the path and return that value.

```
uint32_t path_distance(const Path *p)
   d = p total_weight
   return d
```

- `void path_add(Path *p, uint32_t val, const Graph *g)` : This function takes in three parameters: a pointer to a constant Path `p`, a 32 bit integer `val`, and a pointer to a constant Graph `g`. It does not return anything. Its purpose is to add vertex `val` from graph `g` to the path. It also updates the distance and length of the path.

```
void path_add(Path *p, uint32_t val, const Graph *g)
   if stack_size(p vertices) = 0 then
      stack_push(p vertices, val)
      return
   start = stack_peek(p vertices)
   end = val
   stack_push(p vertices, val)
   p total_weight += graph_get_weight(g, start, end)
```

- `uint32_t path_remove(Path *p, const Graph *g)` : This function takes in two parameters: a pointer to a constant path `p` and a pointer to a constant graph `g`. It returns a 32 bit integer. Its purpose is to remove the most recently added vertex from the path. It also updates the distance and length of the path.

```
uint32_t path_remove(Path *p, const Graph *g)
    uint32_t end
    stack_pop(p vertices, &end)
    if stack_size(p vertices) <= 1 then
        p total_weight = 0
        return end
    start = stack_peek
    total_weight -= graph_get_weight(g, start, end)
    return end
```

- `void path_copy(Path *dst, const Path *src)` : The function takes in a pointer to a path `dst` and a pointer to a constant path `src`. It does not return anything. Its purpose is to copy the path `src` to the path `dst`.

```
void path_copy(Path *dst, const Path *src)
    dst total_weight = src total_weight
    stack_copy(dst vertices, src vertices)
```

- `void path_print(const *p, FILE *outfile, const Graph *g)` : This function takes in three parameters: a pointer to a constant `p`, a pointer to a file `outfile`, and a pointer to a constant graph `g`. It does not return anything. Its purpose is to print the path stored, using the vertex names from `g`.

```
void path_print(const *p, FILE *outfile, const Graph *g)
    **names = graph_get_names
    uint32_t v
    while stack_size(p vertices) > 0
        stack_pop(p vertices, &v)
        print(to outfile, names[v])
```

Psuedocode

```
main
    directed = false
    dash_i = false
    dash_o = false
    while (opt = getopt) != -1
        switch opt
            case 'h':
                print usageg message to stdio
                return 0
                break
            case 'd':
                directed = true
                break
            case 'i':
                infile_name = optarg
                dash_i = true
                break
            case 'o':
                outfile_name = optarg
                dash_o = true
                break
            default:
                print usage message to stderr
                return 1
```

```
FILE *infile = stdin;
if (dash_i) then
    infile = fopen(infile_name, read)
    if infile == NULL then
        print error and usage menu to stderr
        return 1

int num_vertices
if (fscanf(infile, "%d", &num_vertices) != 1) then
    print error message
    return 1

Graph *g = create_graph(num_vertices, directed)

for i, 0 to i < num_vertices
    char *name
    if (fscanf(infile, "%s", &name) != 1) then
        print error message and terminate
    graph_add_vertex(g, name, i)

int start
int end
int weight
bool done
int input_check
while (!done)
    input_check = fscanf(infile, "%d %d %d", &start, &end, &weight)
    if (input_check == EOF) then
        done = true
        continue
    if(input_check != 3) then
        print error message
        return 1

    graph_add_edge(g, start, end, weight)
}

current_path = path_create(num_vertices)
shortest_distance = UINT_MAX
shortest_path = path_create(num_vertices)

dfs(g, current_path, shortest_path, shortest_distance)

FILE *outfile = stdout;
if (dash_o) then
    outfile = fopen(outfile_name, write)
            if outfile == NULL then
                print error and usage menu to stderr
                return 1
if path_distance = 0 then
    print "No path found! Alissa is lost"
    return 0

print "Alissa starts at:" to outfile
path_print(shortest_path, outfile, g)
print "Total Distance: "
print(path_distance(shortest_path))

return 0
```

Error Handling

- Invalid command line arguments : If an invalid command line argument is given, the usage message will be printed and the program will be terminated.
- Invalid input file: The program will print an error message and the usage message to stderr and terminate the program.
- Invalid input in input file : For any invalid input, a corresponding error message will be printed and the program will terminate. Some invalid inputs are as follows:
  - Number of edges not provided
  - Invalid number of vertices
  - Invalid input of cities
  - Invalid edge input
- Invalid out file : If the output file cannot be opened, an error message will be printed and the program will terminate.

Testing

To ensure no memory leaks, the program will be run with valgrind using the commands `make debug` and `valgrind ./tsp -i maps/some-graph.graph`. The program was also tested using `scan-build make`.

To confirm the output of the program is correct, the `diff` command is utilized. To do this, the binary given in the resources repository will be run and the output will be directed into a file. This is done using a command such as `./tsp_ref -i maps/some-graph.graph > expect1.txt` or `./tsp 2> expect_err1.txt`. The same commands are used with this programs executable. The files generated can then be compared using the `diff` command.

Results

References

[1] Dr. Keery Veenestra and TAs. Assignment 5: Surfin' u.s.a. `https://git.ucsc.edu/cse13s/fall-2023-section-01/resources`, Fall 2023.