Assignment 3 Design

Michael Coe

October 2021

Provided Macro from professor Long in vertices.h : `VERTICES 26`

graph.c

Goal: to create an ADT that contains a plottable graph.

- `struct Graph`
  This structure is given via code from Professor Long, and contains 4 variables:
  `uint32_t vertices;` - Number of vertices.
  `bool undirected;` - Undirected graph?
  `bool visited[VERTICES];` - Where have we gone?
  `uint32_t matrix[VERTICES ][ VERTICES ];` - Adjacency matrix.

- `Graph *graph_create(uint32_t vertices, bool undirected)`
  This function was given in assignment documentation by Professor Long. This pointer function creates and returns a graph pointer created with with given values for the struct variables.

- `Graph *graph_delete(**G)`
  This function was given in assignment documentation by Professor Long. This function frees the allocated Graph memory and sets the pointer pointer to null.

- `uint32_t graph_vertices(Graph *G)` This function returns the value of `G->vertices`

- `bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k)`
  This function adds an edge of weight k from vertex i to vertex j. If the graph is undirected, add an edge, also with weight k from j to i. To accomplish this: use an if statement to see if j and i are less than `VERTICES` and k is non-zero, if they are, set `G->matrix[i][j]` to value k and set `G->matrix[j][i]` to value k as well if `G->undirected == true`

- `bool graph_has_edge(Graph *G, uint32_t i, uint32_t j))`

  This function will return true if `G->matrix[i][j] != 0` and return false otherwise

  This function will return the value of `G->matrix[i][j]` `VERTICES`if both i and j are less than `VERTICES` and return 0 otherwise.

- `uint32_t graph_edge_weight(Graph *G, uint32_t i, uint32_t j)`

  This function will return the value of `G->matrix[i][j]` `VERTICES`if both i and j are less than `VERTICES` and return 0 otherwise.

- `bool graph_visited(Graph *G, uint32_t v)`

  This function will return the value of `G->vertices[v]` if `v < VERTICES` and return false otherwise.

- `void graph_mark_visited(Graph *G, uint32_t v)`

  This function will set the value of `G->vertices[v]` to true if `v < VERTICES`.

- `void graph_mark_unvisited(Graph *G, uint32_t v)`

  This function will set the value of `G->vertices[v]` to false if `v < VERTICES`.

- `void graph_print(Graph *G)`
  This function will print out a representation of the graph using a series of for loops. if the graph is directed print out "directed graph", otherwise print out "undirected graph". Then start a for loop iterating through the i indexes with a nested for loop iterating through the j index in which will print `G->matrix[i][j]`. After that Another for loop can iterate through vertices to print out which vertices are true and which are false.

stack.c

Goal: to create an ADT that contains a functional stack

- `struct Stack`

  *This structure is given via code from Professor Long, and contains 3 variables:
  `uint32_t top`; -Index of the next empty slot.
  `uint32_t capacity`; -Number of items that can be pushed.
  `uint32_t *items`; -Array of items , each with type uint32_t.

- `Stack *stack_create(uint32_t capacity)`
  This function was given in assignment documentation by Professor Long. This pointer function creates and returns a stack pointer created with with given values for the struct variables.

- `void stack_delete(Stack **s)`
  This function was given in assignment documentation by Professor Long. This function frees the allocated stack memory and sets the pointer pointer to null.

- `bool stack_empty(Stack *s)`
  This function will return true if `s->top == 0` and return false otherwise.

- `bool stack_full(Stack *s)`
  This function will return true if `s->top == capacity` and return false otherwise.

- `bool stack_size(Stack *s)`
  This function will return `s->top`.

- `bool stack_push(Stack *s, uint32_t x)`
  This function will return false if `stack_full()`, otherwise set `s->items[s->top] = x`, increment `s->top` and return true.

- `bool stack_pop(Stack *s, uint32_t *x)`
  This function will return false if `stack_empty()`, otherwise set `*x = s->items[s->top - 1]`, decrement `s->top` and return true.

- `bool stack_peek(Stack *s, uint32_t *x)`
  This function will return false if `stack_empty()`, otherwise set `*x = s->items[s->top - 1]` and return true.

- `void stack_copy(Stack *dst, Stack *src)`
  This function will start with a while loop that will break when `dst->top == src->top`. Within that loop, create an integer x that will pass the pointer value into `stack_peek(src, &x)` and then push that value into `dst` using `push(dst, &x)`.

- `void stack_print(Stack *s, FILE *outfile, char *cities[])`
  This function was given in assignment documentation by Professor Long. Described from the documentation: "Prints out the contents of the stack to outfile using `fprintf()`. Working through each vertex in the stack starting from the bottom, print out the name of the city each vertex corresponds to."

paths.c

Goal: to create an ADT that tracks the path along a graph

- `struct Path`

  *This structure is given via code from Professor Long, and contains 2 variables:
  `Stack *vertices`; // The vertices comprising the path.
  `uint32_t length`; // The total length of the path.

- `Path *path_create(void)`
  This function creates a path pointer that creates a new path structure. Then initialize the vertices pointer to equal `stack_create(VERTICES)` and the length variable to equal 0.

- `void path_delete(Path **p)`
  This function will start by calling `stack_delete(*p->vertices)` and then calling `free(*p)`. afterwards it will set **p to null.

- `bool path_push_vertex(Path *p, uint32_t v, Graph *G)`
  This function will first create value: `uint32_t start` and pass it to `stack_peek(p->vertices, &start)` to hold the starting vertex. Then if `graph_has_edge(G, start, v)`, then if `stack_push(p->vertices, v)`, add the value of `graph_edge_weight(G, start, v)` to `p->length`, call `graph_visited(G, v)`, and return true. Otherwise, return false.

- `bool path_pop_vertex(Path *p, uint32_t *v, Graph *G)`
  This function will return false if `stack_pop(p->vertices, v)`. Otherwise create value: `uint32_t start` and pass it to `stack_peek(p->vertices, &start)` to hold the starting vertex. Then subtract the value of `graph_edge_weight(G, start, v)` to `p->length`, call `graph_unvisited(G, v)`, and return true. Otherwise, return false.

- `uint32_t path_vertices(Path *p)`
  This function returns stack_size(p-¿vertices)

- `uint32_t path_length(Path *p)`
  This function returns p-¿length

- `void path_copy(Path *dst, Path *src)`
  This function starts by setting dst-¿legth = src-¿length, and then calling stack_copy(&dst-¿vertices, &src-¿vertices).

- `void path_print(Path *p, FILE *outfile, char *cities[])`
  This function calls stack_print(p-¿vertices, outfile, cities).

tsp.c

Goal: To test out the various sorting methods to check their efficiency.

- void main(void)

  Using `getopt`, take parameters from the command line to perform actions:

  - -h: Prints out a help message describing the purpose of the graph and the command-line options it accepts, exiting the program afterwards.
    .
  - -v: Enables verbose printing. If enabled, the program prints out all Hamiltonian paths found as well as the total number of recursive calls to dfs().
  - -u: Specifies the graph to be undirected.
  - -i infile: Specify the input file path containing the cities and edges of a graph. If not specified, the default input should be set as `stdin`
  - -o outfile: Specify the output file path to print to. If not specified, the default output should be set as `stdout`.

  Using `fgets()`, scan the following lines:

  - Save the first value to variable `uint32_t vertices`, if `vertices` is greater than `VERTICES`, then print an error statement.
  - Save the next vertices number of lines to character array `cities`, using `strdup()` from `<string.h>`.

  Next create `Graph G` with given vertices undirected as true if indicated. Then create 3 variables i,j and k, and use `fscanf()` in a while loop to scan the next line and call `graph_add_edge(G,i,j,k)` until it reaches `EOF`. If any of the lines are malformed then print an error statement.

  Next, create 2 Path structs: `curr, shortest`. Then call `dfs(G,vertices, curr, shortest, cities, outfile)`.

- `void dfs(Graph *G, uint32_t v, Path *curr , Path *shortest , char * cities[], FILE *outfile)`
  Start the function by using a for loop to iterate through all the vertices, and for each vertex, if it is possible to push it to the current patch, recursively call dfs replacing v with the new vertex. If the path is complete (meaning all vertices are visited) and the final vertex is equal to v, then print the path and copy it to the shortest path if it is shorter or the shortest path is empty. Otherwise continue the loop. After the loop is complete return 0.