Shweta Jones
CSE 13S - Spring 2021
Prof Darrell Long

Assignment 3 Design - Sorting: Putting Your Affairs in Order

Design:

This program focuses on creating an efficient route using a series of different ADTs: Graph, Path. and Stacks. These structures use each other to develop these paths.

Files:

graphs.c: Implements graph ADT
stack.c: Implements stack ADT
path.c: Implements path ADT
tsp.c: Main function

Pseudocode:

tsp.c:

<u>Help</u>

Prints out the help information is -h is typed

<u>Main</u>

Argparse for the different options
- -h: prints the help message explaining the graph and the command-line options
- -v: enables verbose printing, prints the Hamiltonian paths, prints number of recursive calls to dfs()
- -u: makes the graph undirected
- -i: reads in the input file
- -o: specifies the output file

Read file to determine vertice count
- Check if it's valid

Read file to determine the cities → place into list
Graph *g = graph_create(vertices, undirected)
Read file to determine the edge points and weights
- Check for malformed edges
- Add edge weight

Create current path
Create shortest path
Call dfs
Print shortest path
If verbose is true → path printed
Print length
Print recursive calls
Close infile and outfile

Return 0

Graph.c:

Struct Graph{
uint32_t vertciesl
bool indirect;
bool visited[VERTICES];
uint32_t matrix[VERTICES][VERTICES]
}

Graph *graph_create(uint32_t vertices, bool undirected)
Graph *G = (Graph *) malloc(sizeof(Graph))
G->vertices = vertices
G->undirected = undirected
if (G->undirected == true)
printf("is undirected\n")
else
printf("not undirected\n")
for (uint32_t i = 0; i < vertices; i++)
G->visited[i] = false
for (uint32_t j = 0; j < vertices; j++)
for (uint32_t k = 0; k < vertices; k++)
G->matrix[j][k] = 0
return G

void graph_delete(Graph **G)
free(*G)

uint32_t graph_vertices(Graph *G)
return G->vertices

bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k)
G->matrix[i][j] = k
if (G->undirected == true)
G->matrix[j][i] = k
return true

bool graph_has_edge(Graph *G, uint32_t i, uint32_t j)
if (G->matrix[i][j] > 0)
return true
else
return false

uint32_t graph_edge_weight(Graph *G, uint32_t, uint32_t j)

```
  if (G->matrix[i][j] > 0)
      uint32_t val = G->matrix[i][j]
      return val
  else
      return 0

bool graph_visited(Graph *G, uint32_t v)
  if (G->visited[v] == true)
      return true
  else
      return false

void graph_mark_visited(Graph *G, uint32_t v)
   G->visited[v] = true

void graph_mark_unvisited(Graph *G, uint32_t v)
   G->visited[v] = false

Void graph_print(Graph *G)
  for (uint32_t m = 0; m < G->vertices; m++)
      for (uint32_t n = 0; n < G->vertices; n++)
         printf("%d\t", G->matrix[m][n])
      printf("\n")
```

Path.c:

```
Struct Path{
   Stack *vertices;
   uint32_t length;
}

Path *path_create(void)
   Path &(*(p->vertices)) = stack_create(VERTICES)
   p->length = 0
   return p

void path_delete(Path **p)
  stack_delete(&(*p)->vertices)
   free(*p)
   *p = NULL

bool path_push_vertex(Path *p, uint32_t v, Graph *G)
  uint32_t prev
   if (stack_empty(p->vertices)) {
      stack_push(p->vertices, v)
```

```
        path->length += graph_edge_weight(G, START_VERTEX, v)
        return false
    } else {
        stack_peek(p->vertices, &prev)
        stack_push(p->vertices, v)
        p->length += graph_edge_weight(G, START_VERTEX, &prev)
    }
    return true
```

<u>bool path_pop_vertex(Path *p, uint32_t v, Graph *G)</u>

```
  uint32_t prev
    if (stack_empty(p->vertices)) {
        return false
    } else {
        uint32_t prev
        stack_peek(p->vertices, &prev)
        stack_pop(p->vertices, v)
        p->length -= graph_edge_weight(G, v, &prev)
        return true
    }
```

<u>uint32_t path_vertices(Path *p)</u>

```
    return stack_size(p->vertices)
```

<u>uint32_t path_length(Path *p)</u>

```
    return p->length
```

<u>void path_copy(Path *dst, Path *src)</u>

```
  uint32_t y = src->length
    uint32_t z = dst->length
    z += y
```

<u>void path_print(Path *p, FILE *outfile, char *cities[])</u>

```
    stack_print((p->vertices), outfile, cities)
```

Stack.c:

```
struct Stack {
    uint32_t top
    uint32_t capacity
    int64_t *items
}
```

<u>Stack Create</u>

Stack *stack_create(uint32_t capacity)

```
    Stack *s = (Stack *) malloc (sizeof(Stack))
    if (s)
        s->top = 0
        s->capacity = capacity
        s->items = (int64_t *) calloc (capacity, sizeof(int64_t))
        if (!s->items)
                free(s)
                s = NULL
    return s
```

<u>Stack Delete</u>

```
void stack_delete(Stack **s)
    if (*s && (*s)->items)
        free((*s)->items)
            free(*s)
            *s = NULL
    return
```

<u>Stack Empty</u>

```
bool stack_empty(Stack *s)
    return s->top == 0
```

<u>Stack Full</u>

```
bool stack_full(Stack *s)
    return s->top == 1
```

<u>Stack Size</u>

```
uint32_t stack_size(Stack *s)
    return s->top
```

<u>Stack Push</u>

```
bool stack_push(Stack *s, uint32_t x)
    if (s->top == s->capacity)
        s->capacity = 2 * s-> capacity
            s->items = (int64_t *) realloc (s->items, s->capacity * sizeof(int64_t))
            if (s->items == NULL)
                return false
    s->items[s->top] = x
    s->top += 1
    return true
```

<u>Stack Peek</u>

```
bool stack_peek(Stack *s, uint32_t *x)
    if (s->top == 0)
```

```
        return false
    else
        return true
```

Stack Pop

```
bool stack_pop(Stack *s, uint32_t *x)
    if (s->top == 0)
        return false
    s->top -= 1
    *x = s->items[s->top]
    return true
```

Stack Print

```
void stack_print(Stack *s, FILE *outfile, char *cities[])
    for (uint32_t p = 0; p<s->top; p++)
        printf("%ld ", s->items[s->top])
    printf("\n")
```