For all questions, refer to code lines and file names when discussing your code. Your answers to these questions are likely to be quite substantial and may be several paragraphs each. Take as much time and space as you need in order to fully and completely answer the questions.

1. How do you represent edges in your graph representation? Where and how do you store distance between cities for your graph representation? Include file names and line numbers.

To represent edges, in adjlist.h line 36-38, I use linked list data structure which is stored in adj_t struct to store edges from starting city to its all destination cities, which is edges in graph.

```
typedef struct adj {
        node_t *head;
} adj_t;
```

In line 65-74, the function createNode take in neighboring city and distances and creates a new `node_t` struct to store the neighboring cities of the current city.

To store distance between cities for your graph representation, in adjlist.h line 30-34,

```
typedef struct node {
        int nei;
        int dist;
        struct node *next;
} node_t;
```

where `dist` is the distance between two adjacent cities, and `nei` represents the index of the neighboring city in the ‘city’ array, which is defined in adjlist.h line 40-52.

2. What struct or structs did you create to store your data in, and why did you create those structs with those fields? Explain each field in each struct. How are the structs related to each other? Refer to code lines and file names when discussing your code. Draw and include pictures if it helps.

In ‘parse_cities.h’ line 29-34, struct ‘city_t’ is used to store the city data that I’ve parsed from `city.dat`. char start[CHAR_SIZE]; is a string that represents the starting city. char end[CHAR_SIZE]; is a string representing the ending city. `dist` is the distance between starting city and the ending city. numLines is an integer represents number of lines between starting and ending city.

I create ‘city_t’ struct because I need to store the information I read from city.dat, and I need starting, ending, distance which is 3 columns of data, I also stores number of lines because it is useful when I need to create adjacency list, because in adjlist.h line 103, I need to know how many edges, and iterate all edges to create adjacency list.

In ‘adjlist.h’ line 30-34, struct ‘node_t’ is used to store adjacent node’s data. `int nei` is the index of adjacent city, `int dist` is the distance to target city, `struct node* next` is to point to the next adjacent city node in the linked list.

I created ‘node_t’ because to create a linked list, I need the nodes to store the data of adjacent cities, and distance to adjacent city, and a pointer to the next city in adjacent linked list.

In `adjlist.h` line 36-38, I created struct ‘adj_t’ because I need to store the adjacent list for each starting city. ‘node_t* head’ is a pointer used to store the head node of the linkedlist, which is first adjacent ending city of current starting city.

‘city_t’ struct stores city information that is from `city.dat`.

‘node_t’ stores the information of each adjacent city, and it gets city information from `city_t` struct.

‘adj_t’ struct stores the adjacent list of all neighboring cities, and it is a linkedlist consists of ‘node_t’ structs.

3. How do you verify that an edge exists in the graph? What code in which file verifies that an edge exists? What is the Big O of this operation, and why does it have that Big O? (Count the operations or explain the count of the operations for this entire edge verification process. Refer to code files and line numbers throughout.)

To verify that an edge exists in the graph, I use codes in 'main.c' line 68-78. In line 68, I initialize a node_t struct pointer as cur to the head of the linked list. In line 69, I iterate through the linked list until the cur pointer hits NULL. In main.c line 70, I am comparing neighboring city with the ending city that user input to verify if edge exist, if its same, in line 71, I print the message showing such edge exists. in line 72 break the loop.

In line 74 of main.c I increment the current pointer to its next node. In line 75 if cur pointer hits NULL, meaning that we iterate through the whole adjacent list and cannot find the ending city that user input is existing in the linked list, we decide that there are no edges that exist in that graph.

The big O of this operation is O(E) because in the worst case we have to traverse the whole linked list in order to find the ending city that the user input.

In line 68, node_t *cur = adjList[start].head; is 1 operation,

in line 69, 74 while (cur), cur=cur->next; are E operations,

in line 70 comparison is 1 operation,

in line 71, 76, printf function is 1 operation

in line 72 break is 1 operation.

in line 75, if(!cur) is 1 operation

total operations is: 2*E + 6 operation, so its O(E) //

4. Describe your plan for how you will implement the shortest path Dijkstra's algorithm using your chosen structs and data structures. Be sure to describe answers to all of the following below. Include headings to your answer for each sub-question below. Your answer for this question will likely be a minimum of 1 page; aim for 1-3 pages. Your answer to these questions will be based on not just how thoroughly you answer the question, but whether or not your implementation would work with the big O that you describe. (20 points)

- how you determine that all edges from a city have been visited

In adjlist.h line 30-34, I will update my node struct

```
typedef struct node {
    int nei;
    int dist;
    int visited;
    struct node *next;
} node_t;
```

I will add a new field ‘int visited’, it will be init to 0, when an ending city is visited, I will change it from 0 to 1, so that it will keep track of the city visited.

To determine that all edges from a city have been visited, I will use a loop to iterate through all adjacent city node’s `visited` field, if all of them are 1, then indicates all edges have been visited.

- how you add edges to the set of edges to be visited

To add edges to the set of edges to be visited, I will create a heap structure called heap_t in heap.h. The heap_t struct will have a pointer to the array of distance, and the size of the heap, and its max capacity. For example:

```
typedef struct minheap {
   int *arr;
   int size;
   int capacity;
} minheap_t;
```

It will also have methods such as createHeap, heapify, heapPush, heapPop, freeHeap, etc. The heap_t struct will store distance from starting city to current city, in increasing order of their distance from the starting city to the ending city, which is the int dist field in adjlist.c line 32.

When a city is visited, it will call ‘heapPop’, and then its adjacent unvisited cities will call `heapPush` to add the heap_t. And I will update the distance of that city by its previous value + current node’s value.

- how you keep track of the lowest cost path from the origin city to the destination city

I will maintain an array called **‘dist’**, its size of `CITIES` constant in adjlist.h line 22. It is used to store the shortest path of starting city to each ending city. `dist` array will be initiated to value of INT_MAX, which is implemented in the library #include <limits.h>, while I will set the starting city of be initial value of 0. Since its the starting point of algorithm. When a node is visited, it will update the value of the `dist` array, with the value of the shortest path to the current node.

I am also creating a **‘parent’ array**, its size of `CITIES` constant in adjlist.h line 22. to store the previous node of the current city in the shortest path.

During Dijkstra BFS, when a shorter path than what is currently stored in ‘dist’ array is found, I will update shorter path by updating ‘dist’ array as the new shorter path distance, and ‘parent’ array as the new shorter path parent city of the target city. I will also update *arr in priority queue regarding the new shorter path.

At the end, ‘dist’ array will store the shortest path to each ending city from the starting city, and ‘parent’ array will store parent city of each ending city, which is used to store the shortest path from starting city to ending city.

- how you keep track of the cities visited on your lowest cost path

To keep track of the cities visited on the lowest cost path, I use the **parent array** that was created in the last question. Starting from the ending city, we can follow the parent information back to the starting city. It will create a sequence of cities visited on the lowest cost path from the starting city to the ending city.

I will store the path in a **stack** while following parent information. When hitting the starting city, the stack will contain the lowest cost path from ending city to starting city.

Then we can pop the stack to get reverse order, which is shortest path from starting city to ending city.

- What the big O of your implementation of Dijkstra's will be, and why

The2 time complexity of this implementation is O(V log V + E log V), where E is the number of edges and V is the number of vertices in my adjacent list.

O(logV) comes from operation of ‘minHeap’, heapify, inserting and getting min element is cost of O(logV)

Since each edges will be visited, which is O(E), and `heap_t` struct has heapify operation which is O(logV) time complexity, so Its O(E logV).

Also it will need to update minHeap value for each vertex in graph, there are V vertex and each updating operation will cost O(logV) in minHeap. So its O(V logV).

In summary time complexity is O(V log V + E log V).

The space complexity of our implementation is O(V), where V is the number of vertice, which is the space required to store the arrays for ‘parent’ array, and ‘dist’ array, and ‘minHeap’ data structure.