Route Planning - DS Final Project 2022 Spring

Yijie LUO, SJTU ID 519260910009

Abstract. In this project for final of SPEIT's data structure 2022 spring lesson, I'm going to use various data structure to represent a geographic map of China, and to plan a shortest path on this map. My tool includes `C++`, `Make` and `CMAKE`, installed with `Docker`. By using adjacency representation, heap structure and finally implementing the Dijkstra algorithm, I successfully finished the tasks listed in the handout here.

1. T1

In this task, we will design AM(adjacency matrix), AL(adjacency list) and printAM, printAL

1.1. Adjacency Matrix

Adjacency matrix (abbreviate to AM) is a matrix where the columns and rows are the vertices of the graph, and the element of the matrix represents the adjacency condition. If we take row i that represents city A and column j that represents city B as example, the AM is written as `adjMat`.

- The item adjMat$_{i,j}$ = - `INF`, if the city A and B is not adjacent. As there is no direct route between A and B, we set the distance to negative infinity to represent the inaccessibility.
- The item adjMat$_{i,j} = d \geq 0$, in this case the cities are adjacent and the distance of the direct route is d. (Notice that the distance is always not negative)

We also notice that if $i = j$, A, B is the same city, then $d = 0$. Also we have adjMat$_{i,j}$ = adjMat$_{j,i}$, as the route is always bidirectional. We conclude that the AM is always a symmetric matrix with non-negative items.

In this situation, we first define the basic types for ou data, including vertices and edges.

```
1  /* include/types.h */
2  using vertice_type = std::string;
3  using edge_type = std::pair<vertice_type, vertice_type>;
4  using weighted_edge_type = std::pair<edge_type, int>;
```

A. IMPLEMENT THE **AM** DATA STRUCTURE

And we introduce the public interface of the class `AdjacentMatrix`. The class maintain all the vertices in a `std::vector<vertice_type>` container. And a bidirectional map from the vertices'

Email: excespo@sjtu.edu.cn.

names to their indices in the rows and columns of the matrix, with type `vertice_to_idx_map_type = std::map<vertice_type, int>` and type `idx_to_vertice_map_type = std::map<int, vertice_type>`. They'are quite useful when we want to convert between city names and the AM structure.

```
/* include/adjacency.h */
AdjacencyMatrix(std::vector<vertice_type> vs,
                std::vector<weighted_edge_type> wes);
~AdjacencyMatrix(){};
std::vector<vertice_type> get_vertices() { return _vertices; }
vertice_to_idx_map_type get_v2idx() { return _v2idx; }
idx_to_vertice_map_type get_idx2v() { return _idx2v; }
adj_matrix_type get_adjMat() { return _adjMat; }
int get_v_size(void) { return _v_size; }
int get_e_size(void) { return _e_size; }
void print();
```

You can see that the only part to implement is the constructor of the class and the `print` function. We will now explain how to constrcut with explicit arguments of all vertices and all weighted edegs(that's to say with the distances between cities). We experience three phases:

- With all vertices, map them to their position in the vector, and use the same position in the construction phases of the matrix
- Initialization phase of the matrix, the size of matrix being the number of vertices, and pad the diagonal line to 0 (always 0 whatever the map looks like)
- Traverse all the weighted edges to fill the matrix. Vacant positions in the matrix we pad them with `-INF` as they represent the inaccessibility.

By doing so, we successfully maintain the private variables of a vector of vertices, two maps between vertices and indices, and the matrix itself, being a two-dimensional vector.

B. Implement the generic **PrintAM** function

As for printing the adjacent matrix, we just need to leave space for the rows and columns, and print the matrix by traversing the vector container in order, which is quite naive and simple. The result is shown below:

```
================ TEST FOR AM BEGINS ================
Our map has 13 vertices and 24 edges.
Print Adjacency Matrix:
       WL   LZ   LS   XA   CD   KM   BJ   WH   GZ   SY   TJ   SH   FZ
  WL    0  130  120    N    N    N    N    N    N    N    N    N    N
  LZ  130    0  111   84   85    N    N    N    N    N    N    N    N
  LS  120  111    0    N  110    N    N    N    N    N    N    N    N
  XA    N   84    N    0   87    N   90   94    N    N    N    N    N
  CD    N   85  110   87    0   86    N   99  102    N    N    N    N
  KM    N    N    N    N   86    0    N    N  101    N    N    N    N
  BJ    N    N    N   90    N    N    0  100    N   88    9    N    N
  WH    N    N    N   94   99    N  100    0   98    N   97   96   95
  GZ    N    N    N    N  102  101    N   98    0    N    N    N   91
  SY    N    N    N    N    N    N   88    N    N    0   89    N    N
  TJ    N    N    N    N    N    N    9   97    N   89    0   93    N
  SH    N    N    N    N    N    N    N   96    N    N   93    0   92
  FZ    N    N    N    N    N    N    N   95   91    N    N   92    0
================ TEST FOR AM ENDS ================
```

Where `N` is a symbol that just represents the value `-INF`.

1.2. Adjacency List

C. IMPLEMENT THE **AL** DATA STRUCTURE

When cities are uniformly distributed geographically, it is likely that the number of edges is not much large than that of vertices (Imagine the extreme case where all vertices are connected to any other verice, and if the number of vertices is n, then number of edges can reach $n * (n - 1)$). Therefore the AM will be sparse.

To solve this problem, we can use a bucket to store all the neighbors of a vertice, and we store the buckets in another dimension. This is the idea of Adjacency list (abbreviate to AL) We use a vector of type `vertice_type` to store one bucket, and the list node of type `std::pair<vertice_type, std::vector<vertice_type>>` to store both the head vertice and its neighbor bucket. A vector of type being such node type helps us store the whole AL.

```
AdjacencyMatrix(std::vector<vertice_type> vs,
                std::vector<weighted_edge_type> wes);
~AdjacencyMatrix(){};
std::vector<vertice_type> get_vertices() { return _vertices; }
```

```
    vertice_to_idx_map_type get_v2idx() { return _v2idx; }
    idx_to_vertice_map_type get_idx2v() { return _idx2v; }
    adj_matrix_type get_adjMat() { return _adjMat; }
    int get_v_size(void) { return _v_size; }
    int get_e_size(void) { return _e_size; }
    void print();
```

The constructor of the AL is quite similar to that of AL.

D. IMPLEMENT THE GENERIC **PRINTAL** FUNCTION

We print the AL in the form given in the handout and the result is:

```
================ TEST FOR AL BEGINS ================
Our map has 13 vertices and 24 edges.
Print Adjacency List:
WL -> LZ -> LS -> N
LZ -> WL -> LS -> XA -> CD -> N
LS -> WL -> LZ -> CD -> N
XA -> LZ -> CD -> BJ -> WH -> N
CD -> LZ -> LS -> XA -> KM -> WH -> GZ -> N
KM -> CD -> GZ -> N
BJ -> XA -> WH -> SY -> TJ -> N
WH -> XA -> CD -> BJ -> GZ -> TJ -> SH -> FZ -> N
GZ -> CD -> KM -> WH -> FZ -> N
SY -> BJ -> TJ -> N
TJ -> BJ -> WH -> SY -> SH -> N
SH -> WH -> TJ -> FZ -> N
FZ -> WH -> GZ -> SH -> N
================= TEST FOR AL ENDS =================
```

That's all for *task 1*.

2. T2

2.1. Data Preparation

Before analysing the given map, we will first give interfaces for the map.

GENERATION OF MAP IN **AM**

We prepare an interface `AM genAM();` to generate an `AM` data structure from the given map. As our `AM` constructor requires a vector of vertices and a vector of edges with distance(/weight), we simply construct the two vectors by pushing all vertices and weighted edges. Finally we can get the correct AM corresponding to the map given in the handout.

A. IMPLEMENT THE GENERIC **GETEDGESAM**

The function prototype should be like `std::vector<weighted_edge_type> getEdgesAM(AM am);` in our case. To extract the edges from AM, we just need to undergo the following phases:

- traverse the half matrix, and get the position where the item isn't `-INF`
- convert from position to city names
- construct the edge with the names and item value being the distance(/weight)
- push the edge to the array to be returned.

And to better demonstrate our edges structure, we overload the `operator<<` with a friend function

```
1  /* include/types.h */
2  inline std::ostream &operator<<(std::ostream &cout,
3                                  weighted_edge_type w_edge) {
4      weight_type w = w_edge.second;
5      vertice_type v1 = w_edge.first.first, v2 = w_edge.first.second;
6      std::cout << "[" << v1 << " <--" << w << "--> " << v2 << "]";
7      return std::cout;
8  }  // need inline to avoid multiple definition
```

Our `GetEdgeAM` function take the given map AM as argument and give the following result:

```
============= TEST FOR getEdgeAM BEGINS =============
Got Weighted Edges in this AM:
[WL <--130--> LZ]
[WL <--120--> LS]
[LZ <--111--> LS]
[LZ <--84--> XA]
[LZ <--85--> CD]
```

```
[LS <--110--> CD]
[XA <--87--> CD]
[XA <--90--> BJ]
[XA <--94--> WH]
[CD <--86--> KM]
[CD <--99--> WH]
[CD <--102--> GZ]
[KM <--101--> GZ]
[BJ <--100--> WH]
[BJ <--88--> SY]
[BJ <--9--> TJ]
[WH <--98--> GZ]
[WH <--97--> TJ]
[WH <--96--> SH]
[WH <--95--> FZ]
[GZ <--91--> FZ]
[SY <--89--> TJ]
[TJ <--93--> SH]
[SH <--92--> FZ]
============== TEST FOR getEdgeAM ENDS ==============
```

That's the correct edges in the given map.

2.2. Sort of edges

As the prerequisite of the algorithms in the later tasks, we need to sort the edges in the ascending order.

B. Implement the generic **MinHeapSort**

The min heap structure maintain a partial ordered sequence (that the parent element is always smaller than its children), with the first element in the structure being always the smallest element. To output an ascending sequence, we transform the original sequence to a min heap structure, and every time we pop the first element and re-order the heap, until the heap is empty.

First we implement the `MinHeap` class. We maintain the private variables: a vector of keys of elements, and a vector of values of element. In our case, key refer to the edge structure, and value refer to the distance of the edge.

```
1  /* include/minHeap.h */
2  template <typename keyT, typename valT>
```

```
3   class minHeap {
4     private:
5       std::vector<keyT> _keys;
6       std::vector<valT> _vals;
7       int _size;
8
9     protected:
10      int siftUp(int r);
11      int siftDown(int size, int r);
12      void heapify(int size);
13
14    public:
15      minHeap() { _size = 0; }
16      minHeap(std::vector<keyT> ks, std::vector<valT> vs);
17      ~minHeap() {}
18      int size() { return _size; }
19      bool empty() { return _size == 0; }
20      keyT get_min_key() { return _keys[0]; }
21      valT get_min_val() { return _vals[0]; }
22      keyT get_key_at(int r) { return _keys[r]; }
23      valT get_val_at(int r) { return _vals[r]; }
24      valT get_val_at(keyT k);
25      bool set_val_with(keyT k, valT v);
26      void push_back(keyT k, valT v);
27      void print_keys();
28      void print_vals();
29      void print_all();
30      void delMin();
31      void sort();
32  };
```

The most important methods of the class is the three protected methods: `siftUp`, `siftDown`, `heapify`:

- `siftUp`, check if the current node satisfy the heap's partial order between itself and its sibling and its parent. If not satified, modified their positions.
- `siftDown`, check if the current node satisfy the heap's partial order between itself and its children.

If not satified, modified their positions.

- `heapify`, from an arbitrary array (in fact a vector), do `siftDown` for all the elements except the lowest level ones. It will give a heapified array.

All the three methods modify both the key's array and the value's array.

To fit with the handout's requirement, we encapsulate the functions concerning `minHeap` to one function: `void minHeapSort(std::vector<weighted_edge_type> &w_edges);` As the detailed procedures is too long, here we just post an extract and the sorted sequence. More details please run the executable under the root working directory with command: `build/test_edges`, and see the test for `MinHeapSort`. Extract of the executing procedure and the sorted sequence is:

```
============ TEST FOR minHeapSort BEGINS ============
Got Sorted Weighted Edges
...
...
...
Current heap of size: 3
Print current heap's keys:
[LZ <--111--> LS] [WL <--120--> LS] [WL <--130--> LZ]
Current heap of size: 2
Print current heap's keys:
[WL <--120--> LS] [WL <--130--> LZ]
Current heap of size: 1
Print current heap's keys:
[WL <--130--> LZ]
Current heap of size: 0
Print current heap's keys:

[BJ <--9--> TJ]
[LZ <--84--> XA]
[LZ <--85--> CD]
[CD <--86--> KM]
[XA <--87--> CD]
[BJ <--88--> SY]
[SY <--89--> TJ]
[XA <--90--> BJ]
[GZ <--91--> FZ]
```

```
[SH <--92--> FZ]
[TJ <--93--> SH]
[XA <--94--> WH]
[WH <--95--> FZ]
[WH <--96--> SH]
[WH <--97--> TJ]
[WH <--98--> GZ]
[CD <--99--> WH]
[BJ <--100--> WH]
[KM <--101--> GZ]
[CD <--102--> GZ]
[LS <--110--> CD]
[LZ <--111--> LS]
[WL <--120--> LS]
[WL <--130--> LZ]
============= TEST FOR minHeapSort ENDS =============
```

C. IMPLEMENT THE GENERIC **MERGESORT**

A typical merge sort is made up of two phases:

- sort phase: The array is recurrently bisected until the array has only less than two elements (it's trivial then, and is ordered, so we get many small segments of ordered sequences).
- merge phase: We recurrently merge all segments to a larger one. For two segments to be merges, we pop the smaller one of the smallest elements of the two segments, until segments are all empty. The merge goes until all segments have been merged to the original size, and as a result of merge phase, the result is necessarily ordered.

Our design for merge sort include three functions:

- `void merge_then_sort(std::vector<weighted_edge_type> &w_edges, int lo, int hi);`
  This function is responsible for bisecting the interval $[\text{lo}, \text{hi})$ into two subintervals $[\text{lo}, \text{mid})$ and $[\text{mid}, \text{hi})$, recursively calling the function to continue bisecting the subintervals until the subinterval is trivial (in this case it directly return), and then merging to larger segements.
- `void merge(std::vector<weighted_edge_type> &w_edges, int lo, int hi);`
  This function deal with the merge phase with the idea mentioned above, merge the subintervals in $[\text{lo}, \text{hi})$ and make this interval ordered.

- `void MergeSort(std::vector<weighted_edge_type> &w_edges);` An encapsulated version of merge sort fitted in the handout's context.

As for results, to save space, more details please run the executable under the root working directory with command: `build/test_edges`, and see the test for `MergeSort`. The Extract of the executing procedure and the sorted sequence is:

```
============= TEST FOR MergeSort BEGINS =============
Got Sorted Weighted Edges
...
...
...
After merge:
[LZ <--84--> XA] [LZ <--85--> CD] [CD <--86--> KM] [XA <--87--> CD]
[XA <--90--> BJ] [XA <--94--> WH] [CD <--99--> WH] [CD <--102--> GZ]
[LS <--110--> CD] [LZ <--111--> LS] [WL <--120--> LS]
[WL <--130--> LZ] [BJ <--9--> TJ] [BJ <--88--> SY] [SY <--89--> TJ]
[GZ <--91--> FZ] [SH <--92--> FZ] [TJ <--93--> SH] [WH <--95--> FZ]
[WH <--96--> SH] [WH <--97--> TJ] [WH <--98--> GZ] [BJ <--100--> WH]
[KM <--101--> GZ]
After merge:
[BJ <--9--> TJ] [LZ <--84--> XA] [LZ <--85--> CD] [CD <--86--> KM]
[XA <--87--> CD] [BJ <--88--> SY] [SY <--89--> TJ] [XA <--90--> BJ]
[GZ <--91--> FZ] [SH <--92--> FZ] [TJ <--93--> SH] [XA <--94--> WH]
[WH <--95--> FZ] [WH <--96--> SH] [WH <--97--> TJ] [WH <--98--> GZ]
[CD <--99--> WH] [BJ <--100--> WH] [KM <--101--> GZ]
[CD <--102--> GZ] [LS <--110--> CD] [LZ <--111--> LS] [WL <--120--> LS]
[WL <--130--> LZ]
Results:
[BJ <--9--> TJ]
[LZ <--84--> XA]
[LZ <--85--> CD]
[CD <--86--> KM]
[XA <--87--> CD]
[BJ <--88--> SY]
[SY <--89--> TJ]
[XA <--90--> BJ]
[GZ <--91--> FZ]
```

```
[SH <--92--> FZ]
[TJ <--93--> SH]
[XA <--94--> WH]
[WH <--95--> FZ]
[WH <--96--> SH]
[WH <--97--> TJ]
[WH <--98--> GZ]
[CD <--99--> WH]
[BJ <--100--> WH]
[KM <--101--> GZ]
[CD <--102--> GZ]
[LS <--110--> CD]
[LZ <--111--> LS]
[WL <--120--> LS]
[WL <--130--> LZ]
============== TEST FOR MergeSort ENDS ==============
```

2.3. QAs

We will answer the questions concerning the two functions given above.

D. TIME COMPLEXITY

The time complexity of `MinHeapSort` and the `MergeSort` are both $O(nlogn)$

E. EXPLANATION OF TIME COMPLEXITY

Time complexity of MinHeapSort is O(nlogn) As for a heap structure, we construct the heap with `siftDown` from the bottom. Suppose there are N elements, as a heap is stored in array but is seen as a hierarchy structure (like tree), we can say the height of heap is $L = log(N)$. As for an element of height H, it will undergo a `siftDown` phase, it's up to H, and there are 2^{L-H-1} elements of height H. So the complexity is evaluated by $\sum_1^L H \cdot 2^{L-H-1} \approx 2^L = N$ (Construction begins from height 1, ends to height L). So the construction of heap is of time complexity $O(n)$, and it's done inplace.

To output the ordered sequence in a min heap, we use `delMin`, which is an element-wise swap and `siftDown`, it is therefore of complexity $O(logn)$

The complete `MinHeapSort` requires a head-construction and `delMin` for n steps, the complexity is $O(n) + n \times O(logn) = O(nlogn)$. And the sort require $O(n)$ extra space to store the temporal min heap structure.

Time complexity of MergeSort is O(nlogn) As for `MergeSort`, Suppose we the time complexity of

an array with size n is $T(n)$, then consider merge phase, we merge two already sorted array to one, this phase require time of $C(n) = n$, so we have the recurrent formula $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + C(n)$, we have $T(n) = nlogn = O(nlogn)$. The extra space is $O(n)$.

That's all for *task 2*.

3. T3

Establish the minimum cost spanning tree of the given map graph with Krustal's algorithm. The Krustal's algorithm holds the idea that to make the cost of spanning a tree in graph minimum, we take the smaller edges as much as possible.

3.1. Code Designs

A. IMPLEMENT THE GENERIC **GENMST**

In detail, the Krustal's algorithm can be written in the following executing order:

- Sort all the edges in ascending order to a candidate sequence, initiate the graph to empty.
- Select the current shortest edge (if there is multiple choices, it's arbitrary), check if adding this edge to the graph conflict with the tree properties (say, result in a cycle in the graph). If so, roll back (say, don't choose this edge). Whether the edge conflicts or not, erase it from the candidate sequence.
- Repeat the above step until the number of edges in the graph equals to the number of nodes in our given map minus 1 (property of a spanning tree). And we can ensure that the candidate sequence can provide enough candidate edges until we generate the spanning tree.

Above is the algorithm and also the backbone of our functions. Then introduce the interfaces.

```
void EdgeSort(std::vector<weighted_edge_type> &w_edges);
```

We use `EdgeSort` in `include\edges.h`, which is a quite version of `MinHeapSort` (the original version is verbose and will give the detailed intermediate steps which is not necessary here)

```
std::vector<connected_graph_type> connected_components(
                        std::vector<weighted_edge_type> w_edges);
```

This function in `include\MST.h` divise the current graph to connected components, and it will be easier to check if there is a cycle in a connected graph by simply calculating the number of nodes and that of edges.

```
std::vector<weighted_edge_type> GenMST(
                        std::vector<weighted_edge_type> w_edges);
```

This function in `include\MST.h` executes the whole algorithm described before

3.2. QAs

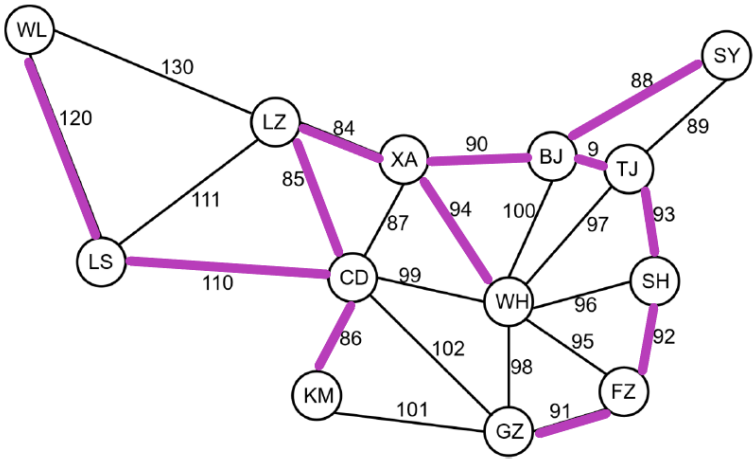
B AND C. RESULT OF KRUSTAL'S ALGORITHM

```
================ Test for GenMST ================
MST including edges:
[BJ <--9--> TJ] [LZ <--84--> XA] [LZ <--85--> CD] [CD <--86--> KM]
[BJ <--88--> SY] [XA <--90--> BJ] [GZ <--91--> FZ] [SH <--92--> FZ]
[TJ <--93--> SH] [XA <--94--> WH] [LS <--110--> CD] [WL <--120--> LS]
================ Test for GenMST ================
MST weights:
1042
```

There are in all 12 routes, and the minimum cost is 1042

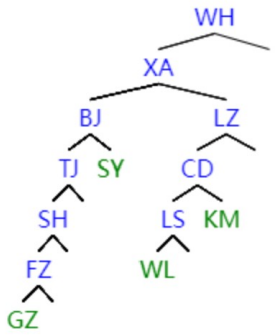
D. DRAW THE ESTABLISHED MST

The established MST is:



E. MST WITH BINARY TREE

When `WH` is chosen as the root node, the MST tree can be spanned in this order:



In this case the MST is a binary tree. We give the traverses of this tree:

- Preorder: WH - XA - BJ - TJ - SH - FZ - GZ - SY - LZ - CD - LS - WL - KM
- Inorder: GZ - FZ - SH - TJ - BJ - SY - XA - WL - LS - CD - KM - LZ - WH
- Postorder: GZ - FZ - SH - TJ - SY - BJ - WL - LS - KM - CD - LZ - XA - WH

That's all for *task 3*.

4. T4

Realize the minimum distance with Dijkstra's algorithm. The algorithm holds the idea that in a graph where there is no cycle of negative weights, we can recurrently determine the minimum distance from the already-determined nodes.

4.1. Code Designs

A AND B. IMPLEMENT THE GENERIC **DIJKSTRAAM**

The only function is `preceding_vertice_map_type dijkstraAM(AM am, vertice_type src, vertice_type dst);`
We want to in the meanwhile calculate the minimal distance, and be able to construct the route planned. So the return type of this function is:
`using preceding_vertice_map_type = std::map<vertice_type, vertice_type>;`
which is a map from the current node to its previous one. With this structure we can easily establish the chosen route.

4.2. QAs

C. COMPLETION OF PSEUDO CODE OF DIJKSTRA'S ALGORITHM

The required fill in the blank is boxed and emphasized in the pseudo code in next page.

D. DEMONSTRATE **DIJKSTRAAM**

To save space, more details please run the executable under the root working directory with command: `build/test_dijkstra`. The Extract of the executing procedure and the sorted sequence is:

```
Updating around: SY
turn: 1
Current heap of size: 14
Print current heap's keys:
SY BJ TJ SY GZ FZ SH WL XA LS WH LZ KM CD
```

Algorithm 1 Dijkstra's Algorithm

Push the start vertex S into Queue
while Queue is not empty and the destination vertex D has not been popped **do**
  Pop the vertex with **MINIMUM** distance from Queue
  for vertex in the non-traversed neighbors of the popped vertex **do**
    if vertex is not in Queue **then**
      push vertex (with its current distance to S) into Queue
    end if
    if vertex already exists in Queue **then**
      update the distance of vertex in Queue if necessary
    end if
  end for
end while

```
Print current heap's values:
0 88 89 2139062143 2139062143 2139062143 2139062143 2139062143 2139062143
2139062143 2139062143 2139062143 2139062143 2139062143
Poping: SY : 0
Updating around: BJ
turn: 2
...
...
...
Updating around: LS
turn: 12
Current heap of size: 3
Print current heap's keys:
LS WL SY
Print current heap's values:
373 392 2139062143
Poping: LS : 373
Minimum distance from src: SY, dst: LS >> 373
BJ : SY
CD : XA
FZ : WH
GZ : WH
KM : CD
```

```
LS : LZ
LZ : XA
SH : TJ
TJ : SY
WH : TJ
WL : LZ
XA : BJ
LS <- LZ <- XA <- BJ <- SY
```

The minimal distance from `ST` to `LS` is $\boxed{373}$

That's all for *task 4*.

A. How to *reproduce* all the results in this report

Please refer to the `README.md` in the root directory of the codes. You will see the usage of `make`, which helps you how to establish the same working environment and then to execute and also there are other functions.