PROGRAMMING ASSIGNMENT 4
Finding Time-Dependent Shortest Path

Subject: Directed graphs and shortest path
TAs: Selim Yilmaz, Levent Karacan, Merve Ozdes

Due Date: 17.05.2018 23:59:59

1 Introduction

The main objective of this assignment is to gain experience on handling directed graphs, or *digraph*s, and on implementing the shortest path algorithm (i.e., *Dijkstra*) within a scenario comprising a number of design constraints.

The following two subsections introduce the basic concepts of the directed graphs (Section 1.1) and Dijkstra's shortest path algorithm, one of the well-known algorithm (see Section 1.2).

1.1 Directed graph

A directed graph is a set of *vertices* and a collection of *directed edges* that each connects an ordered pair of vertices. We say that a directed edge points from the first vertex in the pair and points to the second vertex in the pair. The vertices are also called *nodes* or *points*, while directed edges are also referred to as *links* or *lines*. When working with real-world examples of graphs, we sometimes refer to them as *networks*.

Adjacency-lists representation is one of the convenient ways for a graph representation. In this representation, we often maintain a vertex-indexed array of lists of the vertices connected by an edge to each vertex. Figure 1 demonstrates adjacency-list representation of a digraph containing 5 vertices and a couple of edges connecting some of them.

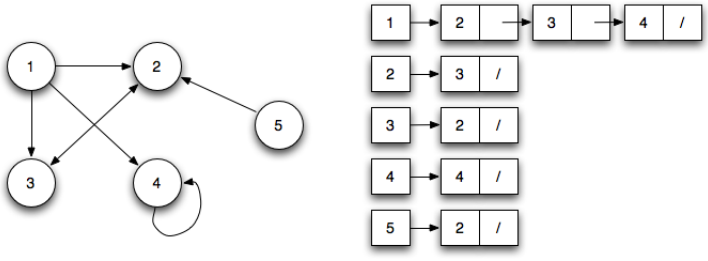


Figure 1: An example of a digraph and a corresponding adjacency-list representation.

1.2 Dijkstra's shortest path algorithm

An edge-weighted digraph is a digraph where we associate weights or costs with each edge. The shortest path from vertex s to vertex t is a directed path from s to t with the property

that no other such path has a lower weight.

Dijkstra's algorithm, conceived by a computer scientist Edsger W. Dijkstra in 1956, is an algorithm for finding the shortest paths between the vertices in a digraph, which may represent, for example, road networks.

A step-by-step introduction of Dijkstra's algorithm[1] is given below.

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for the starting node and to infinity for all other nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the currently assigned value and assign the smaller one.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. Move to the next unvisited node with the smallest tentative distances and repeat the above steps which check neighbors and mark visited.
6. If the destination node has been marked visited or if the smallest tentative distance among the nodes in the unvisited set is infinity then stop. The algorithm has finished.
7. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Section 2 defines the problem and the scenario designed within the context of this assignment. Section 3 provides a set of instructions that you need to consider to accomplish this assignment. Section 4 gives a complete example. Finally, important remarks regarding submission are outlined in Section 5.

2 Problem Definition

Railway network in an imaginary country, say *Neverland*, consists of a set of intersections (I) and a set of rails (R) connecting some of them. There is a switch in every intersection pointing to one of the rails going out of the intersection. Therefore, when a railway driver enters one of the intersections, (s)he can leave only in a direction where the switch is pointing. If the railway driver wants to go some other direction, (s)he needs to manually change the switch, which leads to an increase in arrival time.

The problem that you are expected to address in this assignment is to help a railway driver

[1] A simulation that applies Dijkstra's shortest path algorithm can be found here.

for finding his/her optimal route from source to the destination by implementing Dijkstra's algorithm over a dynamic digraph where the topology changes occasionally.

The idea behind using Dijkstra's algorithm is to ensure time-dependent shortest path for the railway driver. To elaborate it further, Figure 2 demonstrates an imaginary railway network. In this network, there are 6 intersections (from A to F) and 9 rails (edges). The number next to each rail is the distance (in km.) between the intersections that the rail connects. The black rails indicate the rails where the switch in the source intersection is pointing.

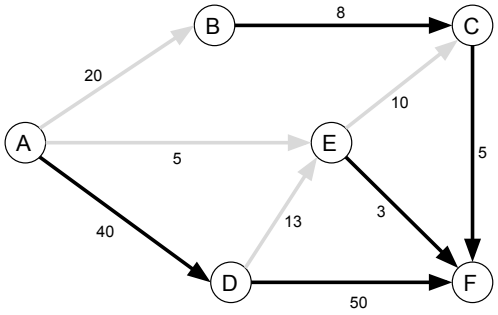


Figure 2: A example of a railway network

Considering this railway network, suppose a railway driver expects one route that gives least arrival time from A to C. From the network, we can see that there exist three routes (A-B-C, A-E-C, and A-D-E-C). It requires one switch change for the selection of the first route (A-B-C) while it takes two switch changes if (s)he selects the second (A-E-C) or the third route (A-D-E-C). As mentioned, every switch change leads to an additional time; and thus negatively affects the total arrival time to the destination. So, there is a trade-off between selecting longer rail or changing the switch's direction at every intersection for the railway driver.

The arrival time (T_A) of a rail vehicle, such as tram, metro, and train, from source to destination is the sum of all the times taken for it to pass all the rails within the path P (i.e., $T_A = \sum T_{r_{i,j}} \,|\, r_{i,j} \in P; P \subseteq R \,|\, i,j \in I \,|\, R, I \neq \emptyset$). It is quite straightforward to calculate the time to pass across a rail connecting X and Y ($T_{r_{X,Y}}$) and it is equal to the distance (in km) from X to Y ($dist_{X,Y}$) divided by the rail vehicle's velocity (vel):

$$T_{r_{X,Y}} \begin{cases} dist_{X,Y}/vel & already\,points\,Y \\ C + (dist_{X,Y}/vel) & otherwise \end{cases} \tag{1}$$

where C is a time (in min) that takes railway driver to change switch's direction, which should be used in the case where the switch in X points any direction other than Y.

3 Assignment Task

This assignment expects you to implement Dijkstra's shortest path algorithm and to apply it on a digraph with dynamic topology.

The first thing you need to do is to construct a railway network. As it is the most case in the real railway networks where there exist two rails pointing opposite directions between any pair of intersections, you should also construct a directed graph to represent the railway network instead of an undirected graph. Clearly saying, if there exists a rail from X to Y then a rail in opposite direction (from Y to X) should also exist in the network.

You are provided three files and two of which are for the construction of railway network (whereas the third file consists of various commands to ensure a dynamic digraph). Every line in the first file begins with a label of an intersection followed by a set of intersections as neighbors (each of which is separated by commas) and ends with a label of one of the neighbors its switch is currently pointing.

```
A:B,C>C
B:A>A
C:A>A
```

From the file content above, it can be seen that intersection A has 2 neighbors, in other words, there are 2 rails going from A, and the switch in A points to the intersection C. The second file you are given contains the distances between every intersection in the railway network. As the distance from X to Y is equal to that from Y to X, this file does not necessarily state all the distances.

```
A-B 10
A-C 4
```

Every intersection in the railway network has *i)* a label, *ii)* a state variable indicating whether it is under maintenance or not, and *iii)* a variable showing the total number of trains that pass over itself. Every rail, however, has *i)* source and destination intersections, *ii)* a state variable indicating whether the switch in the source intersection points it or not, *iii)* a state variable that indicates whether it is broken or not, and *iv)* a variable storing the distance between source and destination. Figure 3 gives an abstract representation of the corresponding railway network that is constructed according to the topology which is stated in the files above. Figure 4, however, gives an adjacency-list representation of this railway network.

After you are managed to construct the railway network, you can now move one step ahead and it is time for you to make this network dynamic through a couple of commands that are provided in the third file.

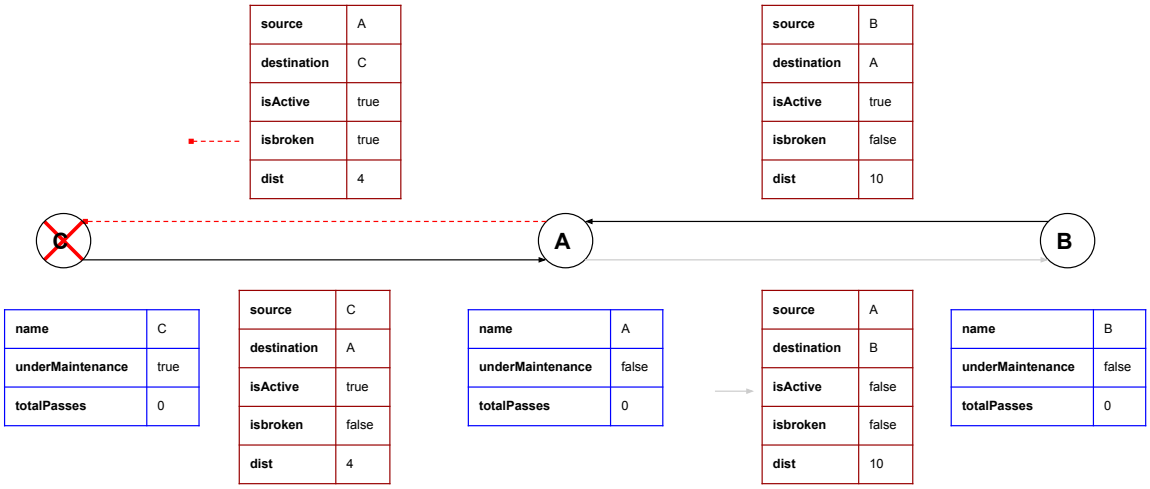


Figure 3: An abstract representation of a railway network

Commands

Rather than simply applying Dijkstra's algorithm on a static graph, you are expected to handle a dynamic graph where its topology changes occasionally as in the real-world. The topology change occurs through a couple of commands provided in the third file as mentioned earlier. Description of all the commands is provided below:

- `MAINTAIN {INTERSECTION}`: It sets a specified intersection as under maintenance. If one intersection is maintained, it should be treated as if it was removed from the railway network. Therefore, one should ignore this intersection and thus all the rails going to this intersection while finding optimal route until it is put into service again.

  ```
  Input:
  MAINTAIN A

  Output:
  COMMAND IN PROCESS >> MAINTAIN A
      Command "MAINTAIN A"  has been executed successfully!
  ```

- `SERVICE {INTERSECTION}`: It puts specified intersection that is currently under maintenance into service again.

  ```
  Input:
  SERVICE A

  Output:
  COMMAND IN PROCESS >> SERVICE A
      Command "SERVICE A"  has been executed successfully!
  ```

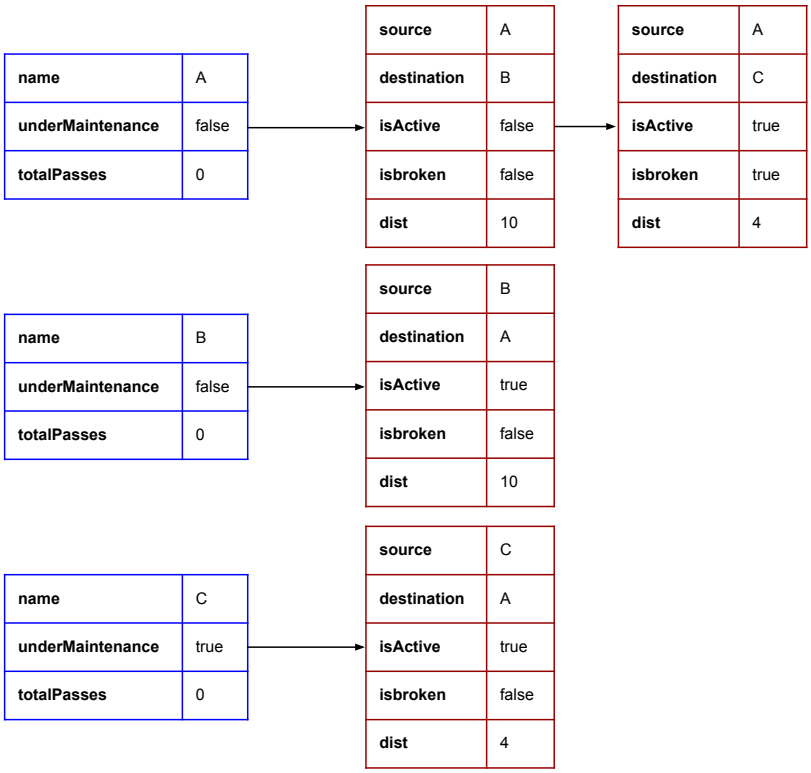


Figure 4: A graph representation of a railway network

- `BREAK` {SOURCE}>{DESTINATION}: It interrupts an existing connection/rail/link between `SOURCE` and `DESTINATION`. When a rail is broken, one should ignore it during the process of finding optimal route until it is repaired again.

```
Input:
BREAK A>B

Output:
COMMAND IN PROCESS >> BREAK A>B
    Command "BREAK A>B"  has been executed successfully!
```

- `REPAIR` {SOURCE}>{DESTINATION}: It repairs an existing broken rail and puts it into service.

```
Input:
REPAIR A>B

Output:
COMMAND IN PROCESS >> REPAIR A>B
    Command "REPAIR A>B"  has been executed successfully!
```

- `ADD {INTERSECTION}`: It creates a new intersection into the railway network.

```
Input:
ADD D

Output:
COMMAND IN PROCESS >> ADD D
    Command "ADD D"  has been executed successfully!
```

- `LINK {SOURCE}:{NEIGHBOR-DIST}+>{POINTING RAIL}`: It connects newly added `SOURCE` intersection with other existing ones. This command also adjusts the direction of the switch in `SOURCE` intersection. Please note that you should also construct the rails with opposite direction of what is provided as neighbors.

```
Input:
LINK D:A-12,B-7,C-4>C

Output:
COMMAND IN PROCESS >> LINK D:A-12,B-7,C-4>C
    Command "LINK D:A-12,B-7,C-4>C"  has been executed successfully!
```

- `ROUTE {SOURCE}>{DESTINATION}` {*vel*}: It initiates a process that is finding optimal route between `SOURCE` and `DESTINATION` for a rail vehicle moving with a certain velocity (*vel*) (km/h). Remember that you need to consider maintenance and repair status of intersections and rails before you initiate, respectively. After the process is completed, it may or may not have found a route. An appropriate message should be displayed.

```
Input:
ROUTE A>D 3

Output:
COMMAND IN PROCESS >> ROUTE A>D 3
    No route from A to D found currently!
    Command "ROUTE A>D 3"  has been executed successfully!

or

Output:
COMMAND IN PROCESS >> ROUTE A>D 3
    Time (in min): 160,000
    Total # of switch changes: 1
    Route from A to D: A C D
    Command "ROUTE A>D 3"  has been executed successfully!
```

  As explicitly stated above, you should print a warning message in the case of no route found. Otherwise, you should print *i)* time (in min) taken for a rail vehicle to arrive `DESTINATION` from `SOURCE`. Please refer to Section 2 for the calculation of total arrival time, *ii)* total number of switch change that is necessary to ensure the least arrival time, *iii)* labels of the intersections in the visiting order.

The further commands are intended for testing your implementation, which reveals every change that you are expected to perform in the railway network topology.

- `LISTROUTESFROM {INTERSECTION}`: It lists all the rails (including those currently being repaired) going from `INTERSECTION` in an alphabetical order.

```
Input:
LISTROUTESFROM C

Output:
COMMAND IN PROCESS >> LISTROUTESFROM C
    Routes from C: A D
    Command "LISTROUTESFROM C"  has been executed successfully!
```

- `LISTMAINTAINS`: It lists all the intersections that are currently under maintenance in an alphabetical order.

```
Input:
LISTMAINTAINS

Output:
COMMAND IN PROCESS >> LISTMAINTAINS
    Intersections under maintenance: A B
    Command "LISTMAINTAINS"  has been executed successfully!
```

- `LISTACTIVERAILS`: It lists all the rails that are currently being pointed by all switches in the intersections in an alphabetical order with respect to the source intersection.

```
Input:
LISTACTIVERAILS

Output:
COMMAND IN PROCESS >> LISTACTIVERAILS
    Active Rails: A>C B>A C>A D>C
    Command "LISTACTIVERAILS"  has been executed successfully!
```

- `LISTBROKENRAILS`: It lists all the rails that are currently broken with `BREAK` command in an alphabetical order.

```
Input:
LISTBROKENRAILS

Output:
COMMAND IN PROCESS >> LISTBROKENRAILS
    Broken rails: A>B
    Command "LISTBROKENRAILS"  has been executed successfully!
```

- `LISTCROSSTIMES`: It lists every intersection with the total number of trains that pass over that intersection thus far in alphabetical order. Note that, you should ignore an intersection over which no rail vehicle passes.

```
Input:
LISTCROSSTIMES

Output:
COMMAND IN PROCESS >> LISTCROSSTIMES
    # of cross times: A:1 C:1 D:1
    Command "LISTCROSSTIMES"  has been executed successfully!
```

- `TOTALNUMBEROFJUNCTIONS`: It prints out total number of intersections (including those that are currently under maintenance) in the railway network.

```
Input:
TOTALNUMBEROFJUNCTIONS

Output:
COMMAND IN PROCESS >> TOTALNUMBEROFJUNCTIONS
    Total # of junctions: 4
    Command "TOTALNUMBEROFJUNCTIONS"  has been executed successfully!
```

- `TOTALNUMBEROFRAILS`: It prints out total number of rails (including those that currently being repaired) in the railway network.

```
Input:
TOTALNUMBEROFRAILS

Output:
COMMAND IN PROCESS >> TOTALNUMBEROFRAILS
    Total # of rails: 10
    Command "TOTALNUMBEROFRAILS"  has been executed successfully!
```

- *In the case of any command other than what are described above, the following message should arise.*

```
Input:
DUMMYCOMMAND

Output:
COMMAND IN PROCESS >> DUMMYCOMMAND
    Unrecognized command "DUMMYCOMMAND"!
```

4 A Complete Example

The implementation you are expected to design will take 4 parameters to be run. The order of these parameters should be:

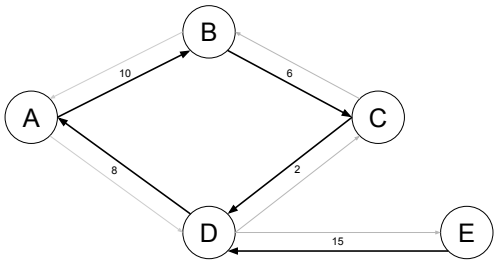
1. a complete path (including the file name and extension) of the file that describes railway network topology
2. a complete path (including the file name and extension) of the file that states distances of intersections
3. a complete path (including the file name and extension) of the command file
4. time for a switch change in an intersection (C in Eq. 1) in the network.

Suppose after your implementation is run with the following command as well as with the file given thereafter, you get a network demonstrated in a figure given in the initial step.

```
java network.in distances.in commands.in 2
```

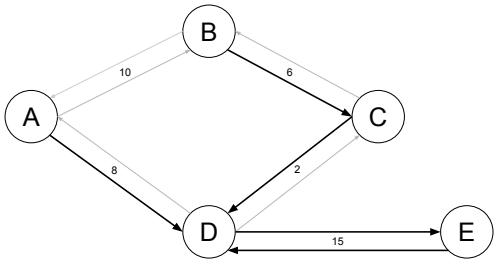
network.in	distances.in	commands.in
A:B,D>B	A-B 10	ROUTE A>E 3
B:A,C>C	A-D 8	BREAK A>D
C:B,D>D	B-C 6	ROUTE A>E 10
D:A,C,E>A	C-D 2	MAINTAIN C
E:D>D	D-E 15	ROUTE A>E 1
		ADD G
		LINK G:B-19,E-41>B
		ROUTE A>E 6
		LISTCROSSTIMES

- Initial step:



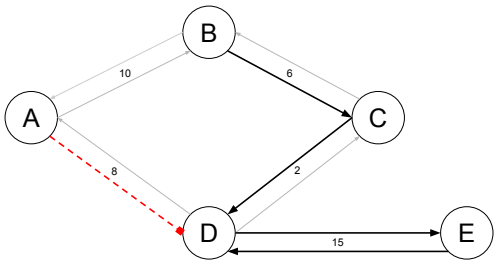
- ROUTE A>E 3

```
COMMAND IN PROCESS >> ROUTE A>E 3
    Time (in min): 464,000
    Total # of switch changes: 2
    Route from A to E: A D E
    Command "ROUTE A>E 3"  has been executed successfully!
```



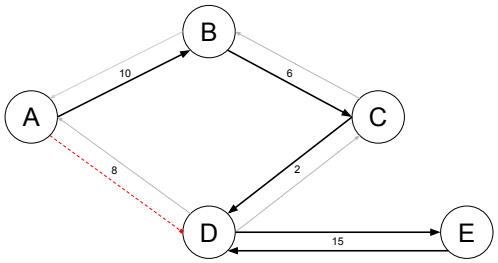
- BREAK A>D

```
COMMAND IN PROCESS >> BREAK A>D
    Command "BREAK A>D"  has been executed successfully!
```



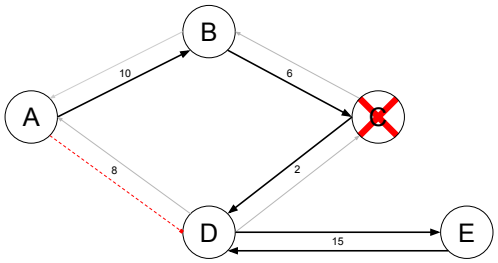
- ROUTE A>E 3

```
COMMAND IN PROCESS >> ROUTE A>E 10
    Time (in min): 200,000
    Total # of switch changes: 1
    Route from A to E: A B C D E
    Command "ROUTE A>E 10"  has been executed successfully!
```



- MAINTAIN C

```
COMMAND IN PROCESS >> MAINTAIN C
    Command "MAINTAIN C"  has been executed successfully!
```

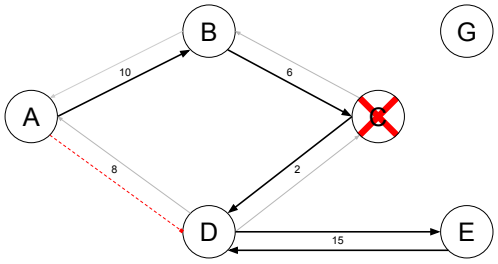


- ROUTE A>E 1

```
COMMAND IN PROCESS >> ROUTE A>E 1
    No route from A to E found currently!
    Command "ROUTE A>E 1"  has been executed successfully!
```

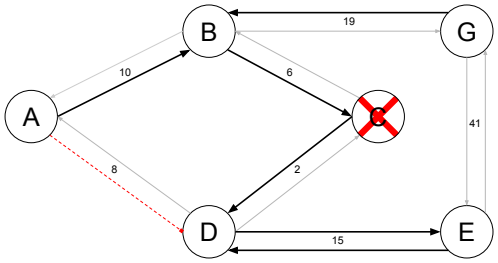
- ADD G

```
COMMAND IN PROCESS >> ADD G
    Command "ADD G"  has been executed successfully!
```



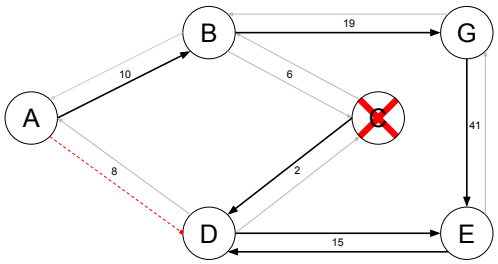
- LINK G:B-19,E-41>B

```
COMMAND IN PROCESS >> LINK G:B-19,E-41>B
    Command "LINK G:B-19,E-41>B"  has been executed successfully!
```



- ROUTE A>E 6

```
COMMAND IN PROCESS >> ROUTE A>E 6
    Time (in min): 704,000
    Total # of switch changes: 2
    Route from A to E: A B G E
    Command "ROUTE A>E 6"  has been executed successfully!
```



- LISTCROSSTIMES

```
COMMAND IN PROCESS >> LISTCROSSTIMES
    # of cross times: A:3 B:2 C:1 D:2 E:3 G:1
    Command "LISTCROSSTIMES"  has been executed successfully!
```

5 Submission Notes

- Do not miss the deadline.
- Save all your work until the assignment is graded.
- The assignment must be original, individual work. All the duplicate or Internet works (even if a citation is provided) are both going to be considered as cheating.
- You can ask your questions via Piazza and you are supposed to be aware of everything discussed in there.
- You will submit your work from https://submit.cs.hacettepe.edu.tr/index.php with the file hierarchy as below:

→ <student id.zip>
→ src.zip <DIR>

- The class name in which main method belongs should be **Assignment4.java**. All classes should be placed in **src** directory in **src.zip**. Feel free to create subdirectories, corresponding the package(s), but each should be in **src** directory.
- This file hierarchy must be zipped before submitted (Not .rar, only .zip files are supported by the system)