Mandatory Assignment 2, IN3130, 2021

Deadline: Monday October 25

Comments concerning this mandatory assignment:

- This assignment consists of three exercises, and each student must turn in their own, independent solution to both.
- Exercises 1 and 3 require code. Your programs must be written in Java. Variable and function names should be self-explanatory; comment as you see fit. Please make sure your code compiles at the department computers. Some datasets for testing your programs are on the course site.
- Exercise 2 requires a textual answer, including some discussion. It is equally important to give good answers to this exercise as it is to program well!
- In the event of minor errors or ambiguities in the assignment text, you are generally expected to state your reasonable assumptions in the PDF or commented in the source code as to the intended meaning in your answer.
- We might give further comments or corrections to this assignment. If so, they will appear as messages on the course page. Check regularly.
- For 3-day extension, submit your request in the nettskjema at the course page.

Exercise 1 (A-search)*

The exercise is to write a program for solving the 8-puzzle and 15-puzzle; in general, the ($N*N-1$)-puzzle (on an $N \times N$-board). Your program should be made for an $N \times N$-board, but you can assume that $N \leq 5$. A state for the game with $N = 3$ can be shown like this:

```
1 2 3
0 4 5
7 8 6
```

Here 0 indicates the empty square. The program should find a way of moving the tiles (a sequence of legal *moves*) that ends up in the final state:

```
1 2 3
4 5 6
7 8 0
```

(and similar for $N \times N$-boards). A legal move is to let the empty square "switch place" with one of the (maximally 4) neighbouring tiles. Such a move can be described *by how the empty square is "moved"*, with either L, R, U, D (left, right, up, down).
Note that the solution you give as an answer must be optimal, in the sense that no solution with fewer moves should exist. Thus, the solution to the problem above would be RRD. Your solution should be written to an output file, on the format described below.

Your program should use the A*-algorithm. It is OK to use a straightforward Manhattan-heuristic, but you are welcome to try other monotone heuristics if you want to.

NB: Your program *must* be able to solve *all* instances of the 8-puzzle (3×3 board) within reasonable time. It is also interesting to see if your program can solve at least a few simple 15-puzzle instances (4×4 board), but it is not required. To check this, you should use the test files on the website.

Input

The program is to read its input from a designated file. The filename is given to the program, together with the name of an output file, as command line arguments. The first line of the input file is the number *N*, then follows the start state over *N* lines, see below.

Output

The output should have the following elements, each starting on a new line:

1. The start-situation (not including the value *N*)
2. The solution, with the number of steps, and a sequence of the letters L, R, U, D (see example below)
3. The number of times a new state is encountered (= the number of inserts in the priority queue of a state that is not already there). This should not include the start state.

The first value of the solution line will depend only on the instance. It should be followed by the solution represented by a sequence of L, R, U, D (but note that multiple optimal solutions are possible for some instances). The number in the last line may vary depending on your implementation, choice of heuristic etc.

Example input	**Example output**

`3`
`1 2 3`
`0 4 5`
`7 8 6`
	`1 2 3`
	`0 4 5`
	`7 8 6`
	`Solution: 3, RRD`
	`States seen: 7`

Implementation tips

The states must be represented in some way. The easiest is to use objects of a class State with a two-dimensional byte array (and maybe in addition two integers saying where the empty square is, which might speed up the algorithm).

A*-search requires several quite advanced date structures: You will at least need some kind of priority queue, and some efficient way of looking up already visited nodes (e.g., a hashmap, splaytree or similar).

Exercise 2 (NP-completeness)

Let **HC** (Hamiltonian-cycle) = {<G> | The directed graph G contains a Hamiltonian cycle}.

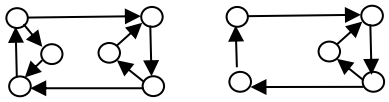
We define it in such a way that graphs with only one vertex, and graphs with two vertices connected to each other, are YES-instances of **HC**.

The problem **HC** is known to be **NP**-complete.

Prove that **2CC (**2-Cycle-cover**),** defined below, is **NP**-complete.
Describe the reduction function *f*, and prove that it is correct. Argue that the reduction is polynomial.

2CC
Input: A directed graph *G*.
Question: Are there two directed, simple, non-overlapping cycles in the graph, each with at least three vertices, that together include all the vertices of *G*? A cycle is simple if it never visits a vertex more than once.

Example
The following graphs show a YES-instance and a NO-instance of **2CC**, respectively:



Exercise 3 (Network flow)

The exercise is to implement the FordFulkerson-algorithm, using the shortest augmentation path in each step (Edmonds and Karp's version). Given a graph with capacities your program shall output the value of an optimal flow, the flow over each edge, and a cut (the one given by the algorithm) proving that the flow is optimal.

The graph is a directed graph, i.e. the capacity from vertex u to vertex v can be different from the capacity from v back to u. All capacities are integer and non-negative. (We use the term *vertex* in this exercise; it is sometimes also called a *node*.)

Input

The program shall read its input from a designated file. The filename is given to the program, together with the name of an output file, as command line arguments. The input file contains:

- First a line with the number of vertices m.
- Then m lines with m numbers each (a matrix) defining the capacities between each pair of vertices. The number in line i and column j is the capacity from vertex i to vertex j, in other words the capacity of the edge (i, j) in the graph.

The vertices are numbered 0 through m-1, with 0 as the source and m-1 as the sink. Note that there may be a positive capacity both from a vertex v to a vertex u, and from u back to v. On the diagonal all capacities are 0. There are no edges going into the source or out of the sink (the leftmost column and the last line (row) contain only zeros).

Output

Make sure your program produces the given format, including the texts given in the examples below!

- First a line with value of the optimal flow.
- Then a line with the vertices (given by their number and sorted by these) on the *source* side of a cut with a capacity equal to the flow given in the first line. The source vertex should be included.
- Then a line giving the number of steps (increments) that are used to obtain the optimal flow value. This number may vary slightly, depending on choices.
- Finally, m lines with m numbers each (a matrix) where the flow is written, in the same format as the capacities of the input.

Example input	**Example output**
5 0 5 1 0 0 0 0 1 4 0 0 2 0 0 6 0 0 1 0 1 0 0 0 0 0	Max Flow: 4 Cut: 0 1 3 Steps: 4 0 3 1 0 0 0 0 1 2 0 0 0 0 0 3 0 0 1 0 1 0 0 0 0 0

Figure 1 below shows the network of the example input, with capacities on the edges. (Vertex 0 is source, and 4 the sink.)

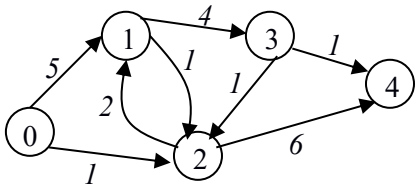


Figure 1 Network with capacities

Implementation tips

There is multiple ways of solving this problem. You may represents the nodes as objects and treat it as a graph or you may use multiple 2d arrays, or another data structure. Thus, you could use three or four $m \times m$-arrays. For instance, one for the original problem N, one for the flow in this step f, and one for the possible flow changes $N(f)$. It might also be convenient with a Boolean array that says whether a vertex has been seen in the current search. To obtain the Edmonds and Karp's version (with as few edges as possible) you can use breath-first search.

Handing it all in

Hand in source code plus *one* PDF-document. Comments, specifications, assumptions etc. can be included at the start of the PDF.

- Put the source code and test files with Exercise 1 and 3 in separate folders and a document called oblig2-<username>.pdf in a folder called: `oblig2_<your username>`.
- Compress the folder into a zip file with the same name.
- Hand in the compressed file at: `https://devilry.ifi.uio.no`

[end]