Exercise sheet 8.

Advanced Algorithms

WiSe 2020/21

Instructor: László Kozma

Due 18:00, January 17th, 2021

Exercise 1 Single-source shortest paths *7 pts*

(a) (3 pts) Show that in an *acyclic* directed graph, SSSP can be solved in $O(m+n)$ time, even in the presence of negative weights.
Hint: find a good order of edge-relaxation.

(b) (4 pts) Suppose you are given the pairwise exchange rates between n different currencies (not necessarily symmetric). Describe an algorithm to detect whether there is an opportunity for arbitrage. (An arbitrage is a sequence of exchanges guaranteed to yield a profit.)
Hint: model the problem as a directed graph with suitable edge-weights.

Exercise 2 Widest paths *4 pts*

Given is a directed graph with non-negative edge-weights. The *width* of a path is the *minimum* weight of an edge on the path. We would like to find *maximum* width paths from a source vertex s to every other vertex in a graph.

Show that the problem can be solved in time $O(m + n \log n)$ by adapting Dijkstra's algorithm appropriately.

Exercise 3 Colorful paths *6 pts*

Given is a directed graph $G = (V, E)$ with edges colored either red or blue and two special vertices $s, t \in V$. Describe efficient algorithms for each of the following tasks. What are the running times of your solutions?
Hint: you can use or adapt the SSSP algorithms discussed in class.

(a) (2 pts) Find a path from s to t with more red edges than blue edges, or report that there is no such path.

(b) (2 pts) Find the path from s to t with the smallest number of blue edges. (In case of ties, pick one arbitrarily.)

(c) (2 pts) Find a path from s to t with at least 3 red edges, or report that there is no such path.

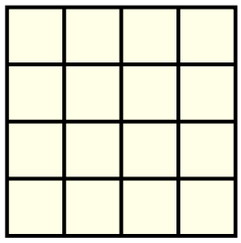
(d) (bonus 3 pts) Find a path from s to t with exactly 3 red edges, or report that there is no such path.

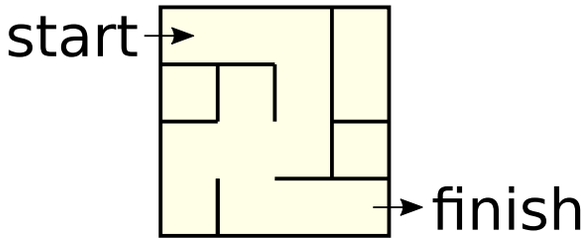
Exercise 4 Building a maze *8 pts*

This is a programming exercise. You may submit the source code (if it is compact and readable) or a high level description of your approach, otherwise. You should also submit the output (for some reasonable size, say $n = 60$). The goal of the exercise is to think about efficient data structures for the task. (You can of course use the data structures provided by the programming language/library you use, but think of how they may be implemented at a lower level and how that affects the efficiency of your solution.)

Start with a complete n-by-n grid, i.e. the collection of horizontal segments between $(i, j), (i + 1, j)$, for $i = 0, \ldots, n - 1$, $j = 0, \ldots, n$, and vertical segments between $(i, j), (i, j+1)$, for $i = 0, \ldots, n$, $j = 0, \ldots, n-1$. Repeatedly remove a horizontal or vertical segment from the collection picked randomly (but excluding the segments on the boundary), until a path opens up between the upper left and the lower right grid cells. (see figure for example with $n = 4$).

For extra credit, come up with some heuristics to make the maze more difficult/interesting to solve.





Total: 25 points. Have fun with the solutions!

From: Yumeng Li and Thore Brehmer

Exercise 1 Single-source shortest paths *7 pts*

(a) (3 pts) Show that in an *acyclic* directed graph, SSSP can be solved in $O(m+n)$ time, even in the presence of negative weights.
Hint: find a good order of edge-relaxation.

(b) (4 pts) Suppose you are given the pairwise exchange rates between n different currencies (not necessarily symmetric). Describe an algorithm to detect whether there is an opportunity for arbitrage. (An arbitrage is a sequence of exchanges guaranteed to yield a profit.)
Hint: model the problem as a directed graph with suitable edge-weights.

a) no cycles and directed graph

Idea: to find d(s,e) topological sort the graph ✓
and update dist acording to the edge weights

1. init all dist values with ∞ and the start vertice dist with 0
2. use topo sort (remove vertices with no incoming edges)
   (after the start vertice has been discovered:)
   - for current vertice = c :
     - test for all outgoing edges if :
       dist(c.next) > dist(c) + edge(c, c.next)
       if so: update dist = dist(c) + edge(c, c.next) ✓

The algorithm has O(m+n) complexity, because we use mainly topo sort which has O(m+n) complexity

3P

b) 1p

- interpret the pairwise exchanges as edges:=E and the rates as edge-weights and the n different currencies as vertices:=V

→ Model a Graph:=G from (V,E) with the edge-weights

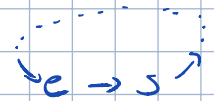
this is unfortunately too inefficient; the number of cycles is superexponential.
eg. # cycles of length n is (n-1)!

1. find in Graph G all cycles:=C (as only in cycles there is an opportunity for arbitrage)

1.1. order C like: number of cycles in c_i < number of cycles in c_{i+1}. $C=[c_1,\dots c_i,c_{i+1},..]$

2. for each cycle:=c in C:

run Bellman-Ford with any vertice:=s as start and s.previous:=e as end vertice



2.1 stop the Bellman-Ford alg after it changed dist(s)
(we have to stop it, as it may contain a negativ cycle)

if dist(s) has a negative value:

What are exactly the edge-weights here?
What is meaning of a neg. distance?

⇒ there is an opportunity for arbitrage in that cycle:= c'

⇒ there is an opportunity for arbitrage for the currencies in that cycle

2.2 remove any cycles:= c^* from C which contain c'
as c^* will also have an opportunity for arbitrage, as it contains c'

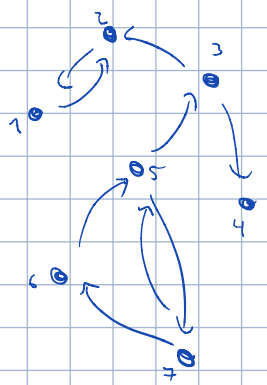
The alg will always stop because the cycles we are testing with Bellman-Ford will max contain 1 negative cycle in them.

Why negative cycles?

Because of the steps 1.1 and 2.2.
Also it will work for 1 negative cycle through 2.1

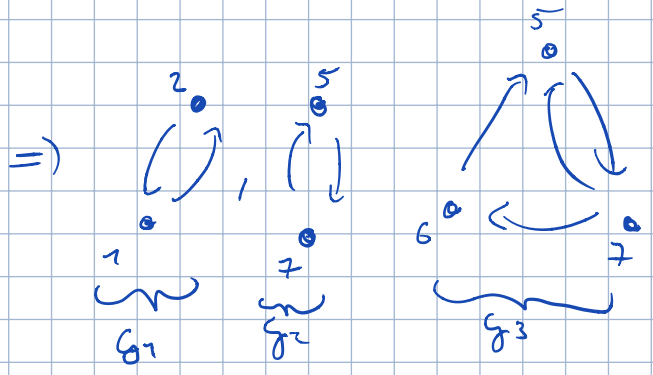
You could simply run B-F with proper weights, no need to sort cycles.

e.g.



Again, this is too much, kind of a complete graph

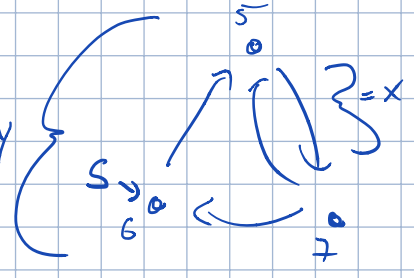
1. Find all cycles and sort them from amount of cycles in the cycle



2. Run Bellman-Ford and stop it after dist(s) (start) changed

if dist(s) has negative value ⇒ opportunity for arbitrage

e.g. for G_3.



suppose x is negative

⇒ Bellman-Ford would not stop because it never returns to dist(s)

For that case we have 2.1

any inner cycle was tested beforhand because of 1.

2.1 if any cycle:=x has an opportunity for arbitrage: test if that cycle x is an inner cycle of any other cycle:=y. That cycle y also has an opportunity for arbitrage. Remove that cycle y

⇒ if our alg tests y with Bellman-Ford ⇒ x can not be negative

Exercise 2 Widest paths *4 pts*

Given is a directed graph with non-negative edge-weights. The *width* of a path is the *minimum* weight of an edge on the path. We would like to find *maximum* width paths from a source vertex s to every other vertex in a graph.

Show that the problem can be solved in time $O(m + n \log n)$ by adapting Dijkstra's algorithm appropriately.

Adapt Dijkstra like the following:

1. save, just like dist, to every vertice a variable max(n) = 0
2. whenever dist gets compared: (dist(c) > dist(n) + edge-weight(n,c))
   2.1 also compare if max(c) < edge_weight(n,c)
   - if so store max = edge_weight(n,c)
   2.2 also compare if max(c) < max(n)
   - if so store max(c) = max(n)

Computing distances is no longer needed, weights should not be added now.

width is the min weight of a path

Works because for each path: we test if the last added edge (2.1) is bigger than every edge we encountered before hand (2.2)

Still in $O(m + n \log n)$ as we only added 2 edges

This doesn't quite work, bec. you only consider paths that are visited by Dijkstra according to distance, but here we need to replace distance by width, to guide the search.

Exercise 3 Colorful paths *6 pts*

Given is a directed graph $G = (V, E)$ with edges colored either red or blue and two special vertices $s, t \in V$. Describe efficient algorithms for each of the following tasks. What are the running times of your solutions?
Hint: you can use or adapt the SSSP algorithms discussed in class.

(a) (2 pts) Find a path from s to t with more red edges than blue edges, or report that there is no such path.

(b) (2 pts) Find the path from s to t with the smallest number of blue edges. (In case of ties, pick one arbitrarily.)

(c) (2 pts) Find a path from s to t with at least 3 red edges, or report that there is no such path.

(d) (bonus 3 pts) Find a path from s to t with exactly 3 red edges, or report that there is no such path.

a)

Runtime like Bellman

1. go through all edges
   - if edge = red:
     edge-weight = -1
   - elif edge = blue:
     edge-weight = 1

1P ✓

2. run dijkstra from s to t with (V, E) and the new edge-weights
   adapt dijkstra like following:

   Dijkstra w. neg.-weights?

   save, just like dist, to every vertice a variable previous

   whenever dist gets updatet: (dist(c) > dist(n) + edge-weight(n,c))
   update previous aswell: previous(c) = n

   2.1 if $|d(s,t)| \geq 0$:
   ⇒ there are more or equal blue edges
   ⇒ return False

   Why Dijkstra? just run B-F w/ given weights.

   2.2 else: ($d(s,t) < 0$)
   ⇒ there are more red edges
   ⇒ return the path

The path can be read out from the table with the previous variable (just follow t.previous until previous = s)

3. if Bellman does not stop after n (|E|) iterations

⇒ there is a negative cycle

3.1 make the cycle non-negative (red_edge_weight = 0)

3.2 and test if there is a path from the cycle to t

if so ⇒ there is a path with "∞" red edges

else run the algorithm once more (with non-negative cycle)

(negative cycle ⇒ ∞ red edges path, just needs to be connected to the end)

The alg will always try to pick red edges over blue edges

Becaus red-edge-weight < blue-edge-weight.

Runtime like Bellman

6)

Just like in a with these changes:

1. go through all edges
   - if edge = red:
     edge-weight = 0
   elif edge = blue:
     edge-weight = 1 ✓ 2p

The alg will pick the path with least amount of blue edges
Becaus red-edge-weight = 0 and blue-edge-weight = 1
So only blue-edges have an negative impact on the path and the dist

c)

Runtime like Bellman

Just like in a) with these changes

1. go through all edges
   - if edge = red:
     edge-weight = -1
   - elif edge = blue:
     edge-weight = 0
2. run Bellman from s to t with (V, E) and the new edge-weights

   if $|d(s,t)| \geq 3$:
   return true

   elif $|d(s,t)| < 3$:
   return false

2p

The alg will always try to pick as many red edges as possible. Because red-edge-weight < blue-edge-weight and red-edges-weight<0 with $|d(s,t)|$ we can test how many red edges we used, because the blue-edge-weight is 0, hence it wont contribute to $d(s,t)$ / only red-edge-weight will change $d(s,t)$

Exercise 4 Building a maze *8 pts*

This is a programming exercise. You may submit the source code (if it is compact and readable) or a high level description of your approach, otherwise. You should also submit the output (for some reasonable size, say $n = 60$). The goal of the exercise is to think about efficient data structures for the task. (You can of course use the data structures provided by the programming language/library you use, but think of how they may be implemented at a lower level and how that affects the efficiency of your solution.)

Start with a complete n-by-n grid, i.e. the collection of horizontal segments between $(i, j), (i + 1, j)$, for $i = 0, \ldots, n - 1$, $j = 0, \ldots, n$, and vertical segments between $(i, j), (i, j+1)$, for $i = 0, \ldots, n$, $j = 0, \ldots, n-1$. Repeatedly remove a horizontal or vertical segment from the collection picked randomly (but excluding the segments on the boundary), until a path opens up between the upper left and the lower right grid cells. (see figure for example with $n = 4$).

For extra credit, come up with some heuristics to make the maze more difficult/interesting to solve.

Our algorithm works the following:

0. create the nxn maze

1. Select the top left cell
   (each has 4 walls top, left, down and right)

2. test if the selected cell has walls

   2.1 if so. pick one of those walls at random

   - remove that wall

   - push the current cell to the stack

   - select the cell which was connected to the just removed wall

   2.2 if it has no walls

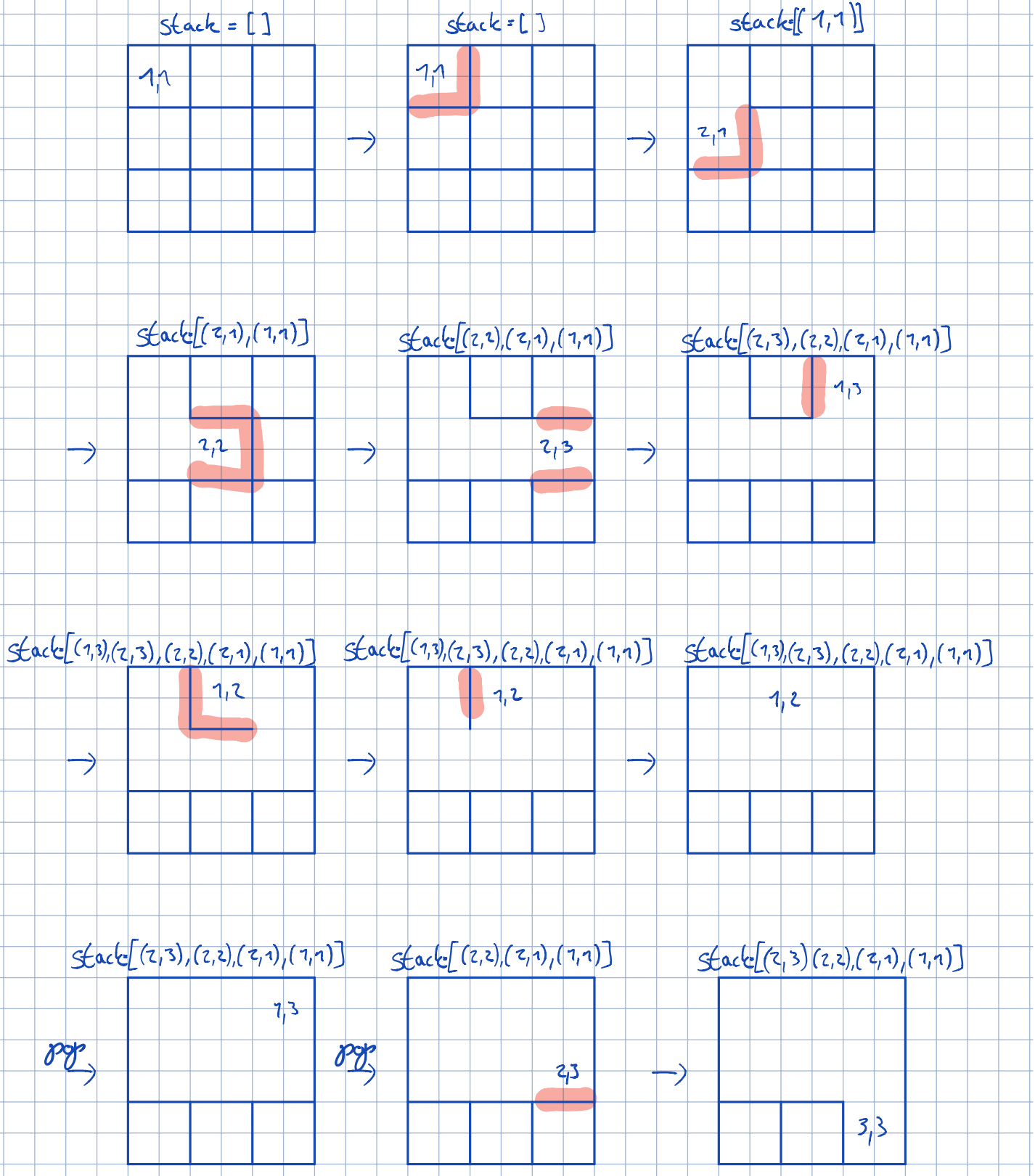
   - pop the last cell from the stack and select it

while the selected cell is ≠ bottom right cell

Works like Depth first search

$O(n^2)$ as in the worst case each cell gets visited in the while loop / the bottom right cell is the last unvisited cell remaining

e.g.



this works, but it "clears up" a large part of the maze.

We didn't use any special data structer as the described algorithm (depth first search) doesent profit from that.

We also used arrays for storing the maze. But because we only used the arrays for storing data and not for searching or sorting, there is also no need for another datastructure

Ok, this is nice and simple, just that the maze is a bit "empty" as many walls are removed. You could have removed some wall globally at random, to keep it more dense.

For n=60 and size=10 (drawing size of rooms)

Let me solve it