CS2040S

Recitation 8
AY22/23S2

SSSP Review

See here for a quick review of SSSP.

Recitation goals

- Advanced graph modelling and transformations
- Modifying shortest paths algorithms
- Explore shortest paths applications


StonksX Trader
stonks

Problem description

- You trade in a Forex exchange for global currencies
- There are currently n currencies being traded
- You have a matrix R containing all exchange rates where $\mathrm{R}[i, j]$ is the exchange rate from currency i to j
- Note that exchange rates are not symmetric: $\mathrm{R}[i, j] \neq \mathrm{R}[j, i]$
- No arbitrage possible: if you start with one currency and then convert it to another, and so on, and then back to the first currency, you will end up with *no more* money then what you started with

Problem Illustration (Matrix and the problem)

Currency	SGD	INR	RM	Yuan	GBP
SGD	1	62	4	6	1/2
INR	1/62	1	1/19	1/12	1/100
RM	1/4	19	1	2	1/6
Yuan	1/6	12	1/2	1	1/9
GBP	2	100	6	9	1

Find the sequence of conversions that will maximise the amount of GBP.

1 SGD -> GBP = ½ GBP

1 SGD -> INR -> GBP = ?

Problem Illustration (Matrix and the problem)

Currency	SGD	INR	RM	Yuan	GBP
SGD	1	62	4	6	1/2
INR	1/62	1	1/19	1/12	1/100
RM	1/4	19	1	2	1/6
Yuan	1/6	12	1/2	1	1/9
GBP	2	100	6	9	1

Find the sequence of conversions that will maximise the amount of GBP.

1 SGD -> GBP = ½ GBP

1 SGD -> INR -> GBP = 62/100 GBP

Test yourself!

What graph DS is matrix $\mathbb{R}$?

Currency	SGD	INR	RM	Yuan	GBP
SGD	1	62	4	6	1/2
INR	1/62	1	1/19	1/12	1/100
RM	1/4	19	1	2	1/6
Yuan	1/6	12	1/2	1	1/9
GBP	2	100	6	9	1

Test yourself!

What graph DS is matrix R?

Answer: Adjacency Matrix.

Problem 1.a.

How do you model this problem as a graph?

What are its vertices and edges?

Is it weighted or non-weighted?

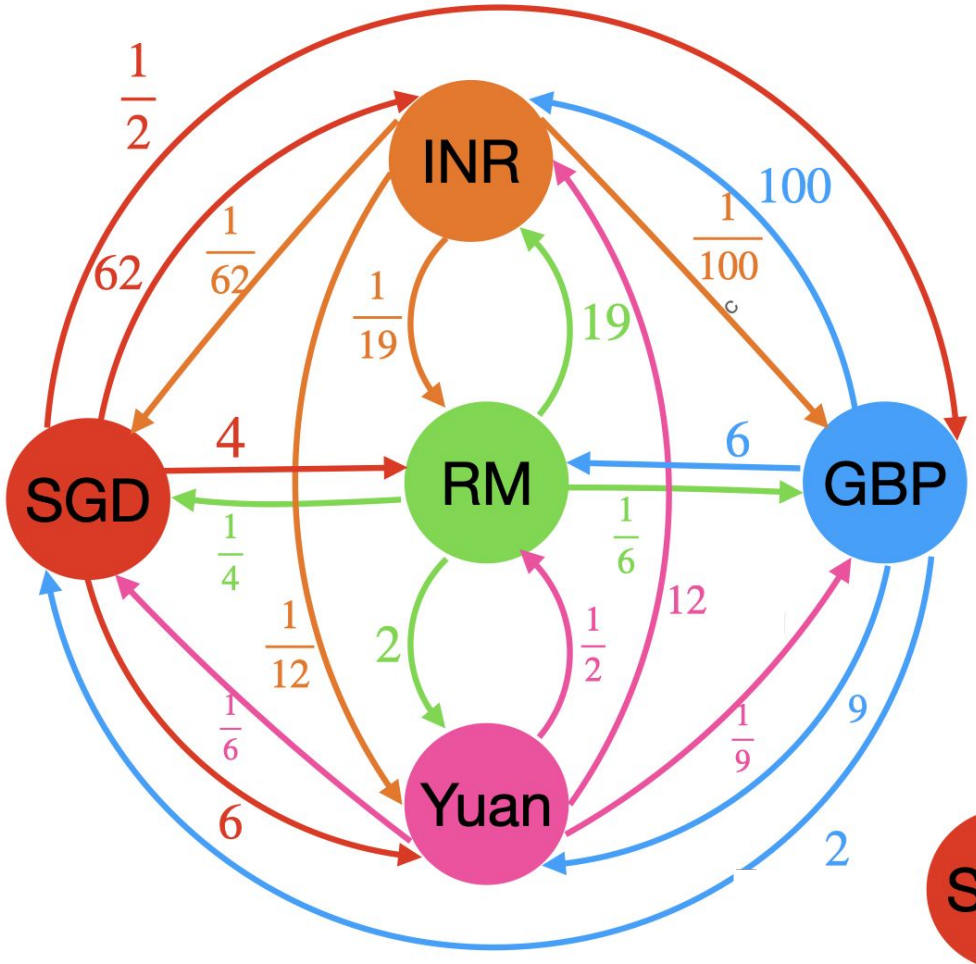
Are the edges directed or non-directed?

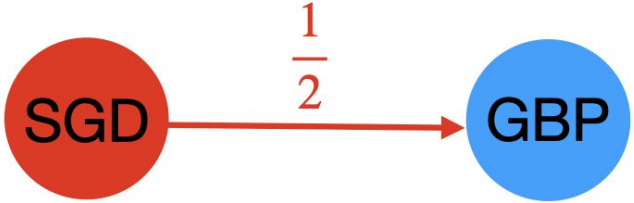
Graph modelling

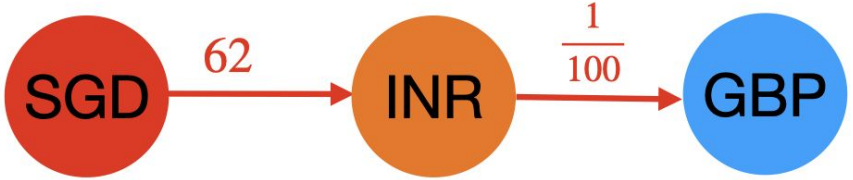
Vertex: Currency

Edge (i, j):

- Direction: Can convert from currency i to j
- Weight: Exchange rate


INR
SGD
RM
GBP
Yuan
1/2
62
1/62
1/19
19
1/100
100
4
6
1/4
1/6
1/12
2
1/2
12
1/6
1/9
9
6
2


SGD
1/2
GBP


SGD
62
INR
1/100
GBP

Problem 1.b.

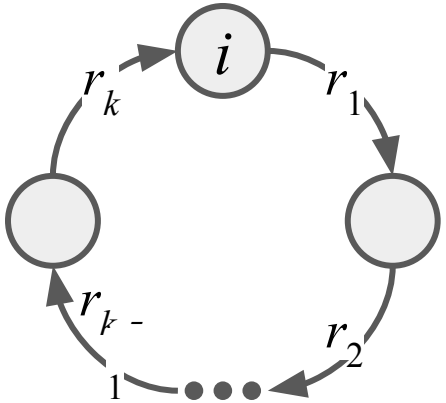
In your graphical model, which of its property reflects the “no arbitrage” rule?

What would happen if such a rule is not enforced by the exchange?

No arbitrage

$$i \times r_1 \times r_2 \times \ldots \times r_k \leqslant i$$

$$r_1 \times r_2 \times \ldots \times r_k \leqslant 1$$

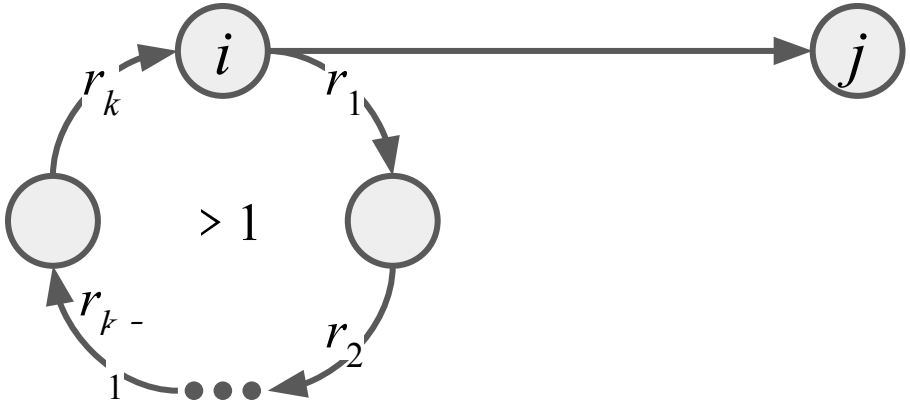


Thus graphically, this means that we won't have a cycle where the product of all its edge weights exceed 1.

No arbitrage

What if there's no such rule enforced by the exchange? Well that means we can get infinitely rich!

Suppose we're converting from currency i to j. If there exists a cycle from i back to itself such that the total product of edge weights within that cycle is more than 1, then we can endlessly get more money than we started with. We can do this *infinitely many* times before converting to currency j!



Problem 1.c.

What graph algorithm will you use to solve the problem?

Which modifications are necessary?

Single Source *Longest Product* Path?

Suppose we wish to convert from currency i to j, we want to find a path such that it *maximizes* the path *length* where *length* is measured as the *product* of all the edge weights in the path.

Realize that we can either modify a SSSP algorithm or transform our graph.

Test yourself!

Which part(s) of a SSSP algorithm can we modify to solve this problem?

Test yourself!

Which part(s) of a SSSP algorithm can we modify to solve this problem?

Answer: We need to modify the **relax** operation to cater for 2 changes.

Assuming we are relaxing edge (u, v, w)

1. "Product instead of sum": Compare $D[u] \times w$ instead of $D[u] + w$
2. "Maximization instead of minimization": Update when new estimate is *greater* than previous instead of lower

Solution 1

$relax(u, v, w)$

if $D[v] < D[u]*w$ **then**
    $D[v] \leftarrow D[u]*w$
end

We need make these modifications (highlighted in green) to the relax operation.

Test yourself!

Given our modifications to the relax operation, what is the triangle inequality for our SSSP algorithm now?

Test yourself!

Given our modifications to the relax operation, what is the triangle inequality for our SSSP algorithm now?

Answer: $\delta(s,v) \geq \delta(s,u) \times w$

Test yourself!

What distance estimates should we initialize on the nodes for our SSSP algorithm?

Test yourself!

What distance estimates should we initialize on the nodes for our SSSP algorithm?

Answer:

- Source node initialized as 1 due to the *multiplicative identity property* of 1
- All other nodes initialized as 0 due to the *zero-product property*

Test yourself!

Which SSSP algorithm is *not suitable* for use with our modified relax operation and our graph? Why?

Test yourself!

Which SSSP algorithm is *not suitable* for use with our modified relax operation and our graph? Why?

Answer:

Dijkstra's, because there can be edge weights > 1.

What's the problem here?

Discussion

In the standard SSSP problem on positive weighted graphs, Dijkstra's is a *greedy algorithm* exploiting the property that traversing an edge can only **increase** a path length.

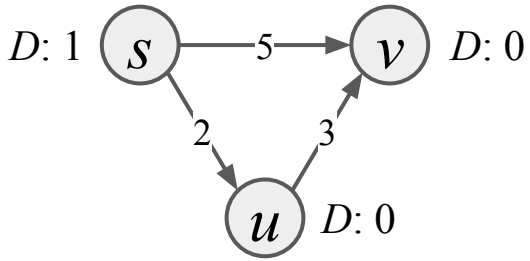
After each relaxation, Dijkstra's **finalizes** the shortest distance to the node concerned, because

- Even if we revisit the node in the future then the new path must have used more hops than before (since Dijkstra's uses BFS propagation)
- More hops means greater distance so that cannot be more minimal than the minimum distance we found earlier with lesser hops

Discussion

What about our “modified Dijkstra’s” (with the modified relax operation and a max-PQ) on the currency exchange graph? We just need to check if its “modified” greedy strategy still works.

Consider this graph:



When we run “modified Dijkstra’s” on source node s, it will greedily relax $s \rightarrow v$ and finalize the $\delta(v)$ as 5, although $s \rightarrow u \rightarrow v$ provides a better path because $2 \times 3 = 6 > 5$. So our greedy strategy fails here!

Discussion

This is a great example to show that it can be tricky to modify graph algorithms while still ensuring correctness. We have to be very careful!

This is why it is often recommended to transform the graph into an equivalent form in which we can apply standard graph algorithms directly.

Solution 2

In order to use SSSP algorithms without modifying them, we have to *cast* / *reduce* the problem to a SSSP problem.

Specifically, we need to:

1. Redefine length: Change product to summation
2. Update objective: Change maximization to minimization

We have to transform our graph to achieve these.

Solution 2: Redefine length

maximize length $= \text{maximize } r_1 \times r_2 \times \ldots \times r_k$

maximize log length $= \text{maximize } \log(r_1 \times r_2 \times \ldots \times r_k)$

$= \text{maximize } \log r_1 + \log r_2 + \ldots + \log r_k$

Our new definition for length!

So now the graph becomes:

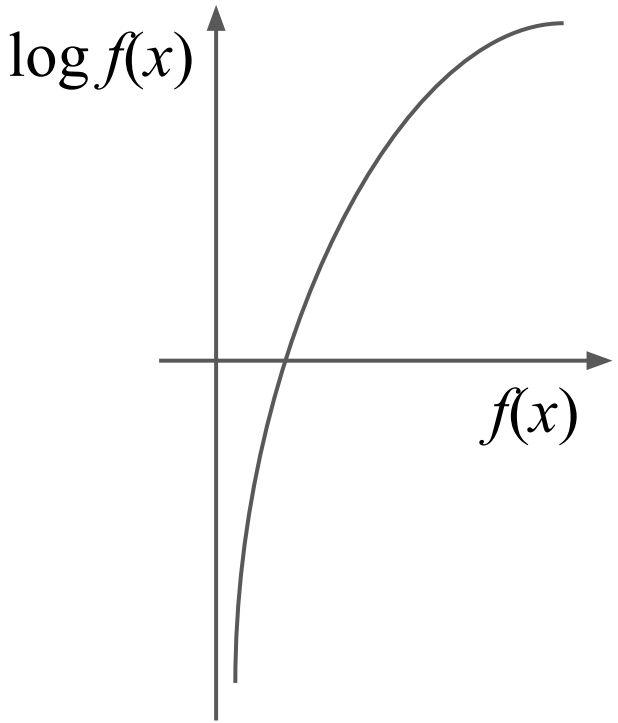
Solution 2: Log trick

Why is maximizing $f(x)$ the same as maximizing $\log f(x)$?

This is because the log function is a *monotonically increasing* function:

$\log f(x)$ increases *if and only if* $f(x)$ increases!

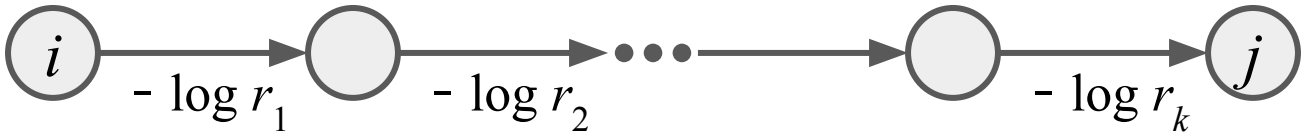
It is convenient here because it breaks up product into summation. This is a classic trick often used in statistical maximum likelihood estimation.



Solution 2: Update objective

Lastly we need to change the objective from a maximization into a minimization one. Thankfully this is trivial because maximizing $f(x)$ is the same as *minimizing* $-f(x)$.

So we simply negate all edge weights:



Now you can just run your favourite SSSP algorithm that can handle negative edges!

Test yourself!

Will we need to be concerned about negative weighted cycles after this conversion?

Test yourself!

Will we need to be concerned about negative weighted cycles after this conversion?

Answer: No, this naturally follows from the “no arbitrage” rule because there’s no way we would be caught in a cycle to get infinitely rich. To prove, recall that we have established that for any cycle,

$$r_1 \times r_2 \times \ldots \times r_k \leqslant 1$$

Hence taking negative log on both sides of the inequality,

$$-\log(r_1 \times r_2 \times \ldots \times r_k) \geq -\log 1$$

$$-\log r_1 - \log r_2 - \ldots - \log r_k \geq 0$$

Test yourself!

What is the tradeoff in this solution?

Test yourself!

What is the tradeoff in this solution?

Answer: Shortest distance $\delta = -\log r_1 - \log r_2 - \ldots - \log r_k$ now no longer represents the max currency conversion product $L = r_1 \times r_2 \times \ldots \times r_k$.

How do we *recover* L? Here's how:

$$
\begin{aligned}
\delta &= -\log_a r_1 - \log_a r_2 - \cdots - \log_a r_k \\
-\delta &= \log_a r_1 + \log_a r_2 + \cdots + \log_a r_k \\
a^{-\delta} &= a^{\log_a r_1 + \log_a r_2 + \cdots + \log_a r_k} \\
&= a^{\log_a r_1} a^{\log_a r_2} \ldots a^{\log_a r_k} \\
&= r_1 \times r_2 \times \cdots \times r_k \\
&= L
\end{aligned}
$$

Solution 3

This graph transformation is essentially

1. First taking reciprocal of edge weights (turning a maximization into a minimization)
2. Then taking $\log$ (turning product chain to a summation)

Why does taking reciprocal work here but not in problem 3?

Solution 3: Triangle inequality

Our “raw” triangle inequality in this question:

$$\delta(v) \geq \delta(u) \times w$$

Where $\delta(v)$ is result of $w_i \times \ldots \times w_k$ from current best estimated path to v.

By taking reciprocal on both sides, we get:

$$1/\delta(v) \leq 1/\delta(u) \times 1/w$$

$$\delta'(v) \leq \delta'(u) \times w'$$

So in this question taking reciprocal works due to the commutativity of multiplication.


Problem 2
Single Player Shortest Victory

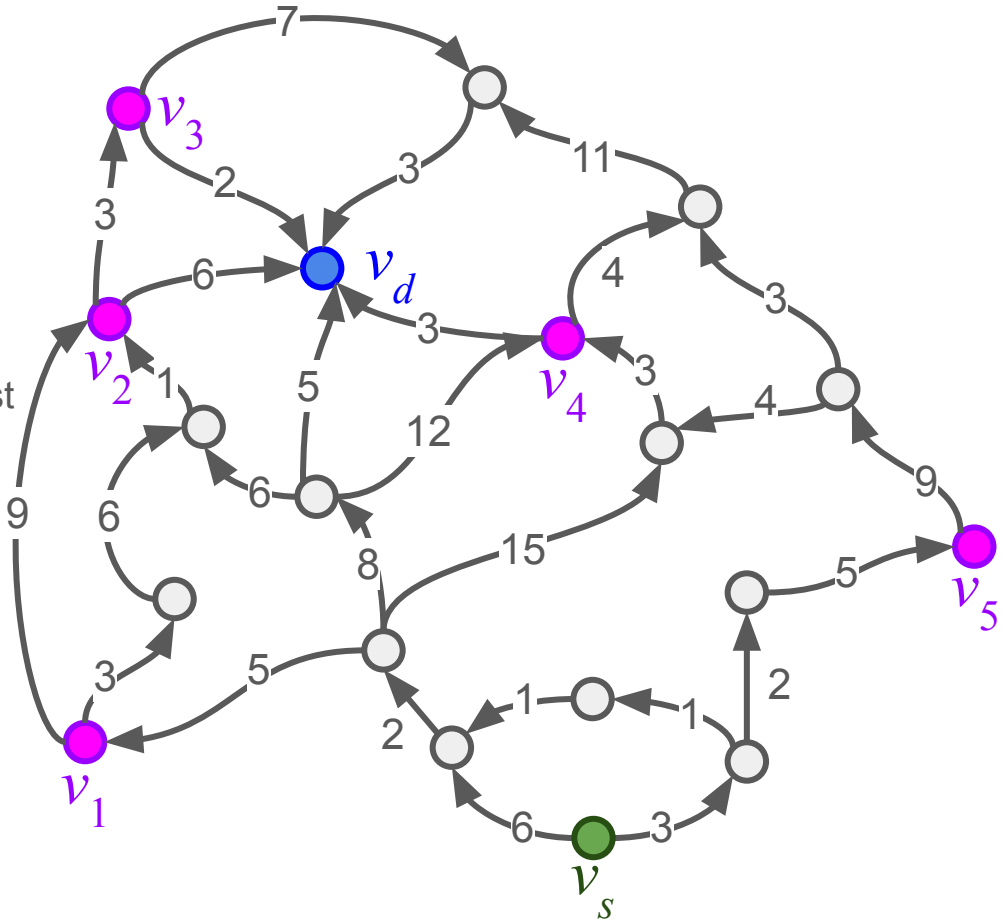
PUBG Scenario

- You are at **green** point
- Play area will be converging on **blue** point
- Supply drops are indicated by **magenta**
- *Travel times* are indicated by edge weight
- You need to visit *at least one* supply drop before taking the fight to the final play area
- You want to take the *fastest* route

Erangel Map from: https://www.foxsportsasia.com/

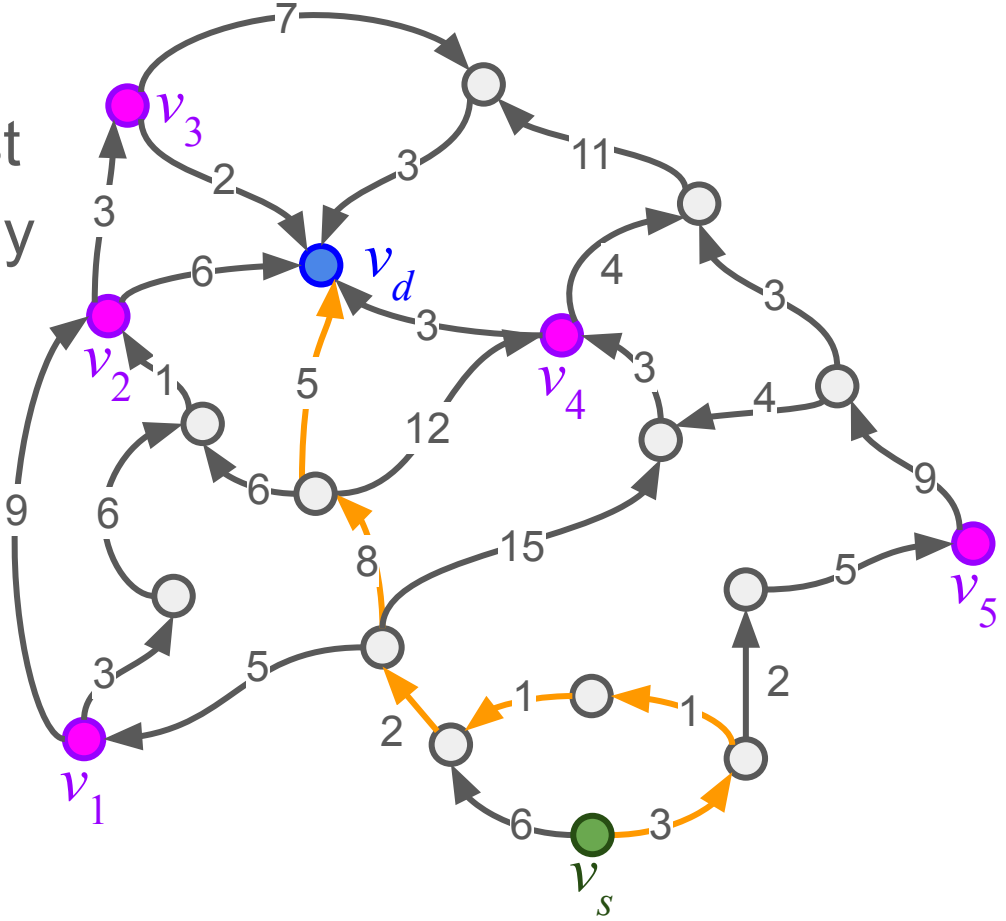
The graph

- Graph $G=(V, E)$
- Source vertex: v_s
- Destination vertex: v_d
- Mandatory vertices: $v_1, \ldots, v_k$
  - Mandatory here means we must visit at least one of them

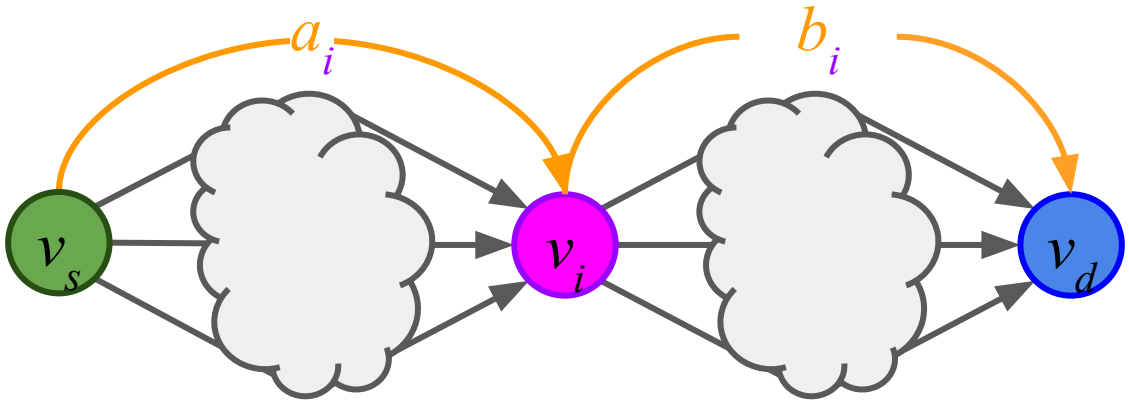


The graph

Unfortunately the shortest path does not pass by any supply drops :’(



Solution 1: Exhaustive SSSP

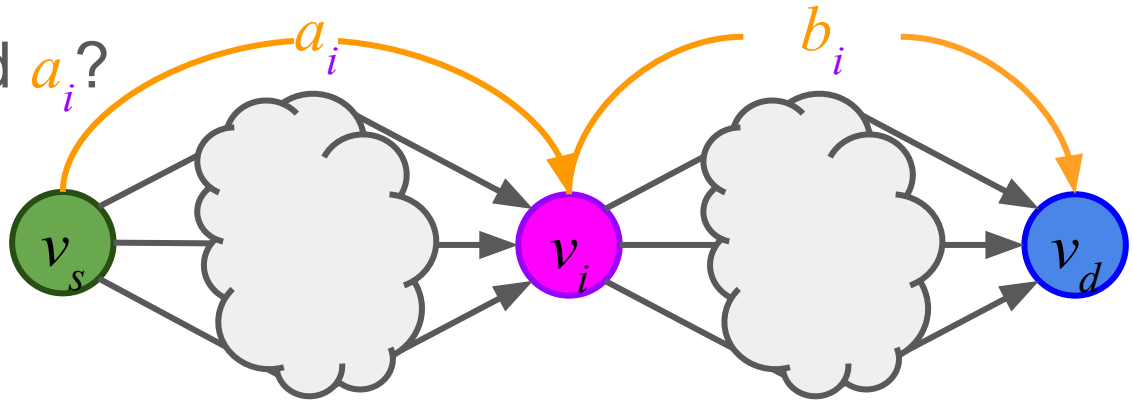


a_i: Shortest distance from source vertex v_s to mandatory vertex v_i
b_i: Shortest distance from mandatory vertex v_i to destination vertex v_d

Objective: We want to find $a_i + b_i$ that is the *minimum* across all mandatory vertices $i \in [1, k]$.

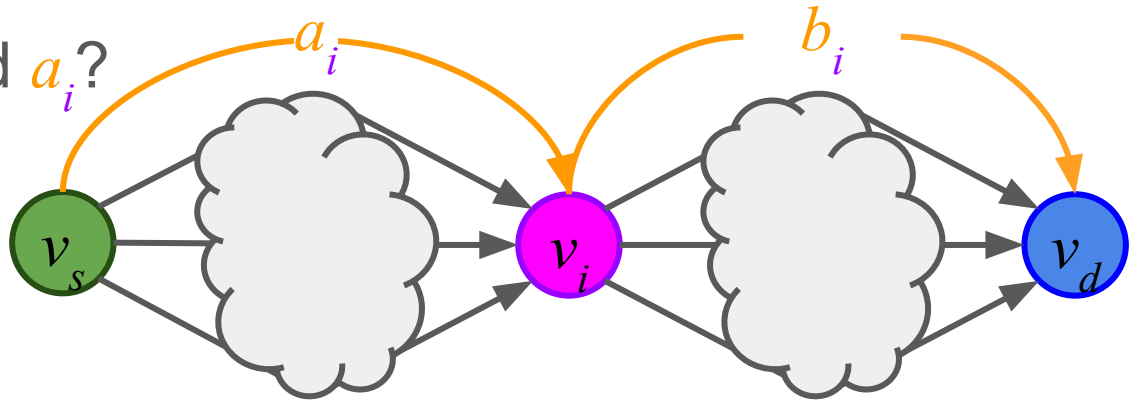
Test yourself!

How to find a_i?



Test yourself!

How to find a_i?



Answer:

Simply run SSSP on v_s to obtain table D_s, then a_i is just $D_s[i]$.

What about b_i? How do we find the b_i for each mandatory vertex i?

Realize that

SSSP *pertains only to the chosen source vertex* and not any destination vertex in particular since it computes shortest paths from source to *all other vertices* in the graph (i.e. all vertices in the graph are destinations).

Solution 1: Exhaustive SSSP

Naive strategy:

Since the shortest distances output by a SSSP algorithm are only valid for the chosen source vertex, we can exhaustively let each mandatory vertex v_i take turns being the source in the SSSP problem in order for us to find b_i, the shortest distance from every v_i to the destination v_d.

Solution 1: Exhaustive SSSP

Algorithm:

1. Run SSSP from v_s to obtain $D_s[1], \ldots, D_s[k]$
2. Run SSSP from each mandatory vertex $v_1, \ldots, v_k$ to obtain distances tables $D_1, \ldots, D_k$ (i.e. one shortest distance table for *each* mandatory vertex)
3. Solve the minimum of $D_s[i] + D_i[d]$

Test yourself!

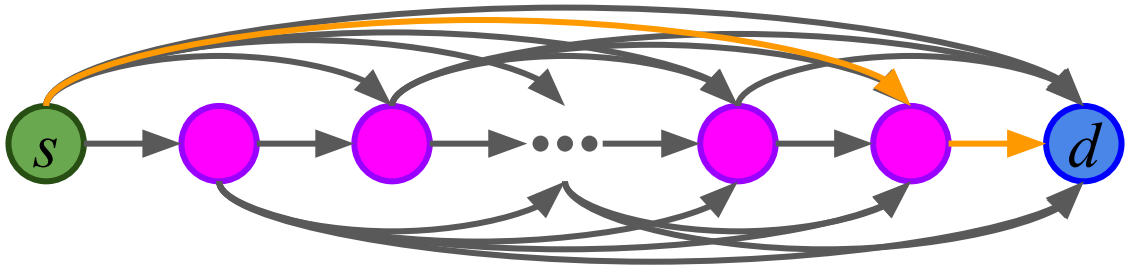
What is the time complexity of this solution if you used Bellman-Ford for the SSSP algorithm?

Test yourself!

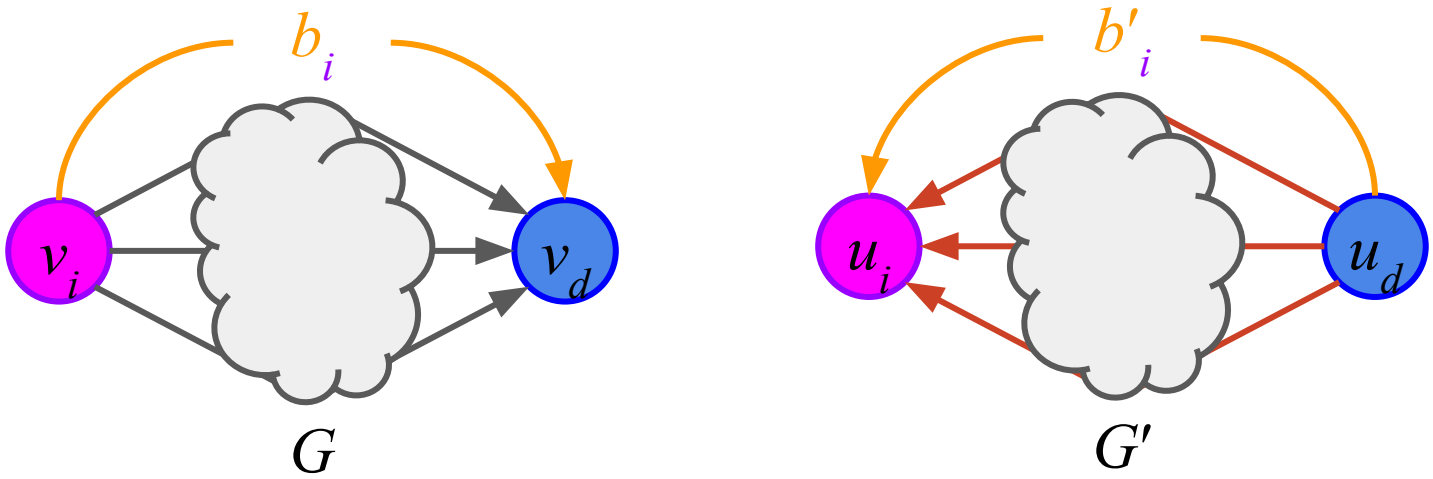
What is the time complexity of this solution if you used Bellman-Ford for the SSSP algorithm?

Answer: $O(kVE)$

If in the worst case all vertices are mandatory, then this algorithm will run in $O(V^2E)$, which is bad...



Solution 2: Key observation



- Suppose we reverse all edges in G and create a new graph G'.
- Then by construction, the shortest distance from v_i to v_d in graph G must be the same as the shortest distance from u_d to u_i in graph G'
- Thus $b_i = b'_i$

Solution 2: Key idea

If we can turn mandatory vertices from source vertices into *destination* vertices, then there's no need to compute each b_i by running SSSP on each mandatory vertex v_i as the source.

To achieve that, we can just *reverse all edges* on the graph, then run SSSP from the destination vertex v_d to obtain distance table D_d. That way, our shortest path b_i from each mandatory vertex v_i to destination vertex v_d is simply $D_d[i]$!

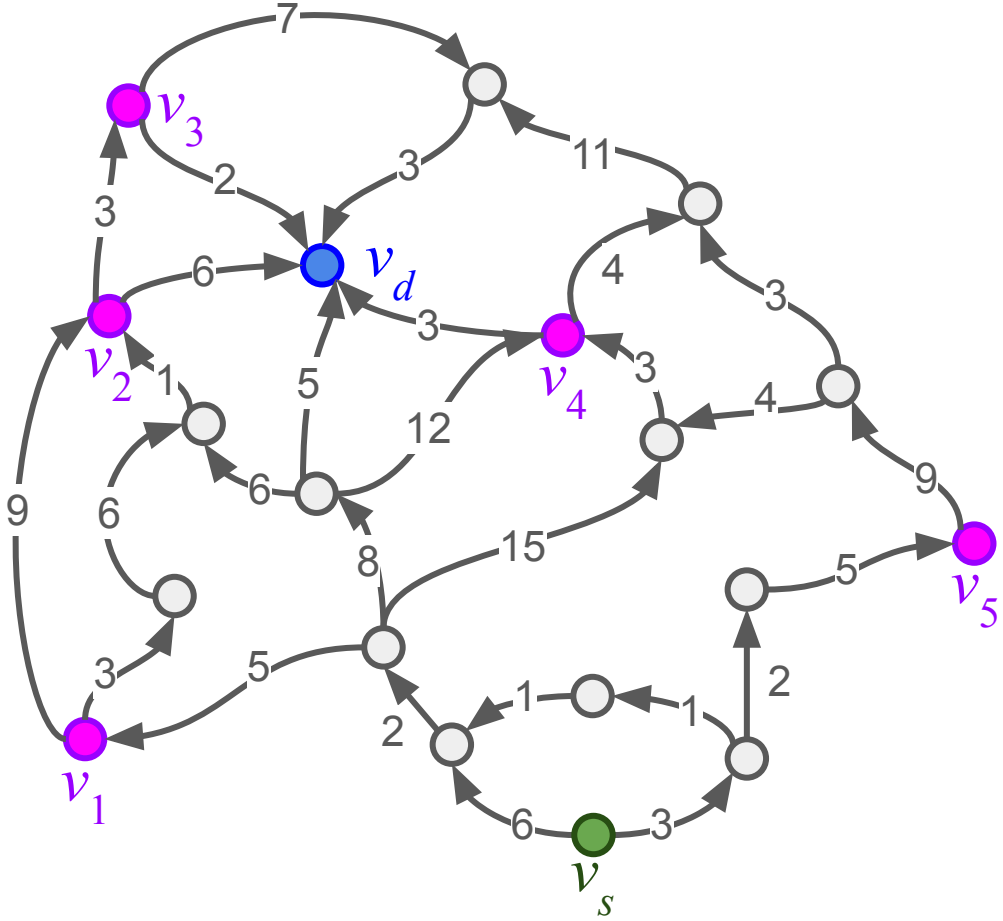
Solution 2: Graph reversal

Algorithm:

1. Run SSSP from v_s to obtain $D_s[1]$, …, $D_s[k]$
2. Reverse all edges in graph G
3. Run SSSP from destination vertex v_d to obtain shortest distance table D_d
4. Solve the minimum of $D_s[i] + D_d[i]$

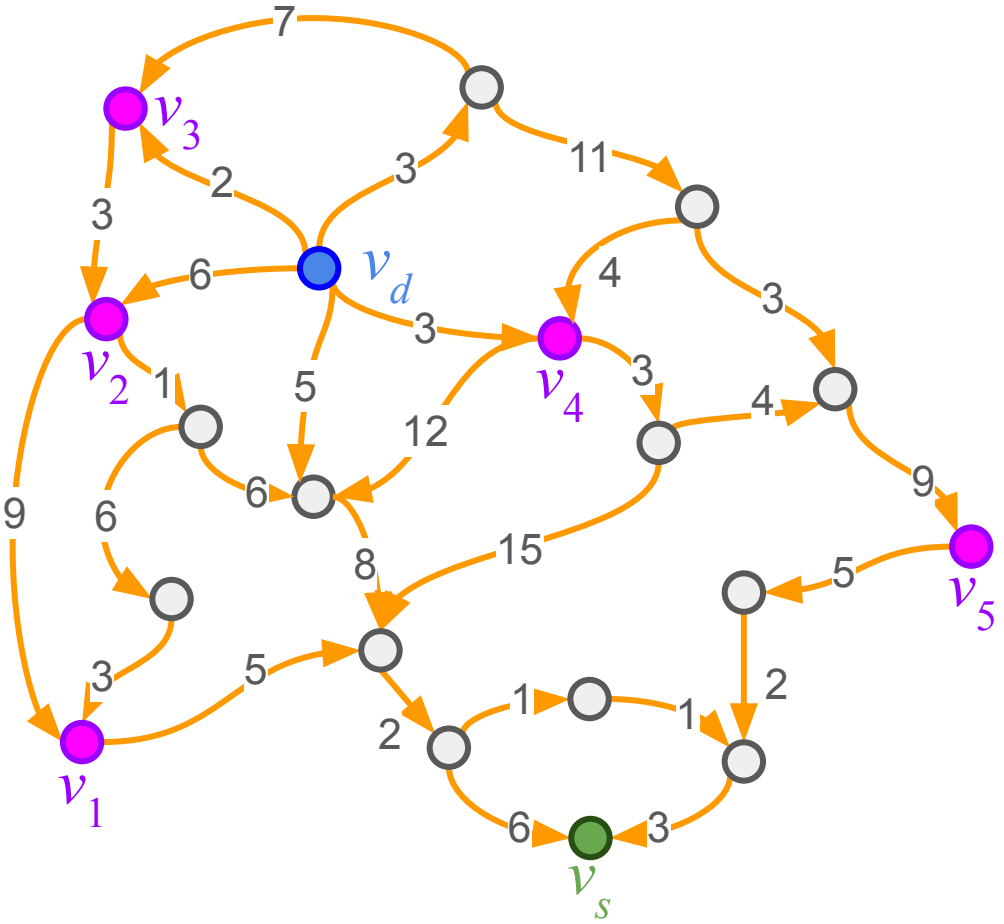
The trick

From this..



The trick

To this!



Test yourself!

What is the time complexity for this solution if we run Bellman-Ford or Dijkstra?

Test yourself!

What is the time complexity for this solution if we run Bellman-Ford or Dijkstra?

Answer:

Bellman-Ford: $O(VE)$

Dijkstra: $O(E + V \log V)$ (Using Fibonacci heap)

Since we are just running them twice.

Test yourself!

What is the space complexity for this solution if adjacency list is used to encode the graph?

Test yourself!

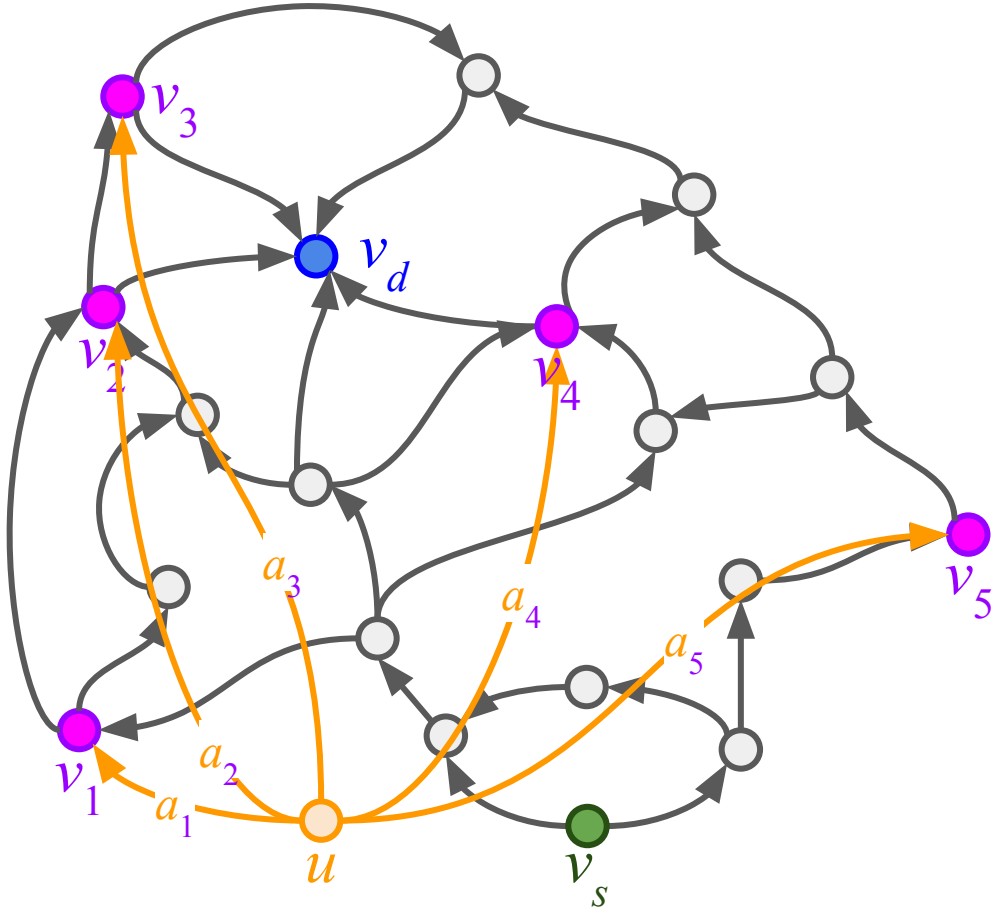
What is the space complexity for this solution if adjacency list is used to encode the graph?

Answer: $O(V+E)$ from storing the reversed graph.

Solution 3

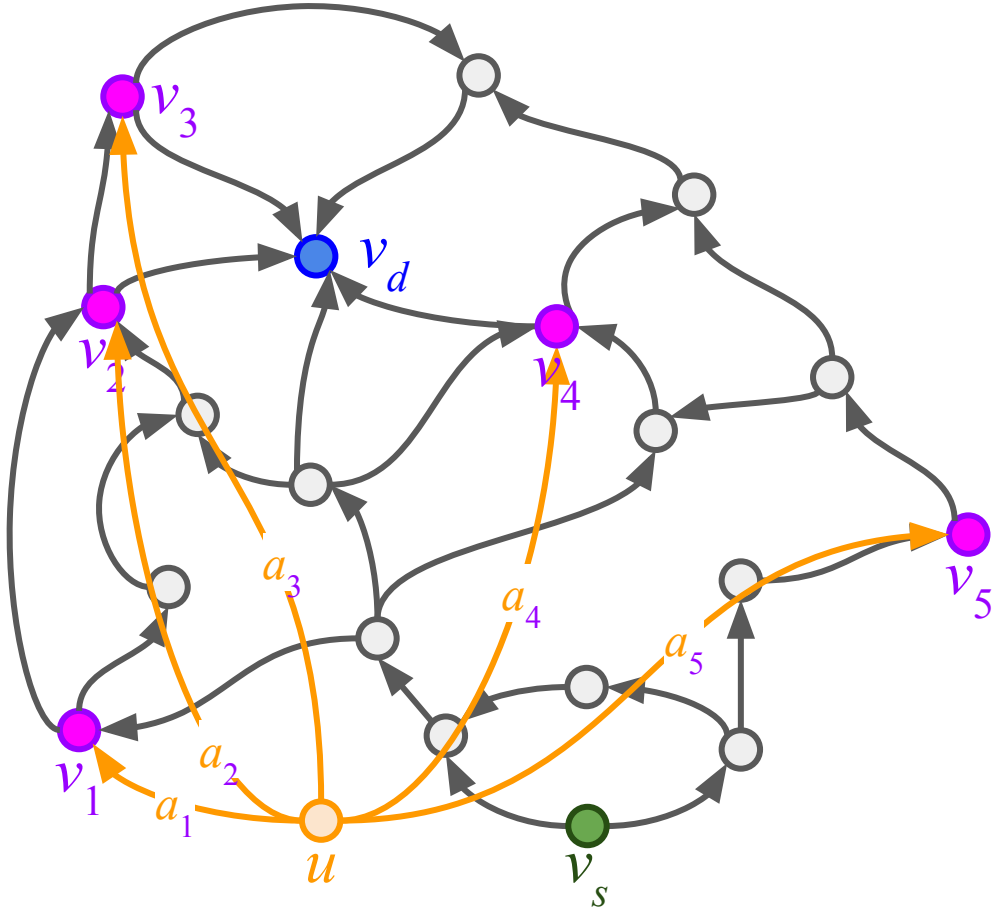
Create a dummy vertex u and connect it to all mandatory vertices via outgoing edges.

In addition, we assign weight a_i to edge $u \rightarrow v_i$.



Solution 3

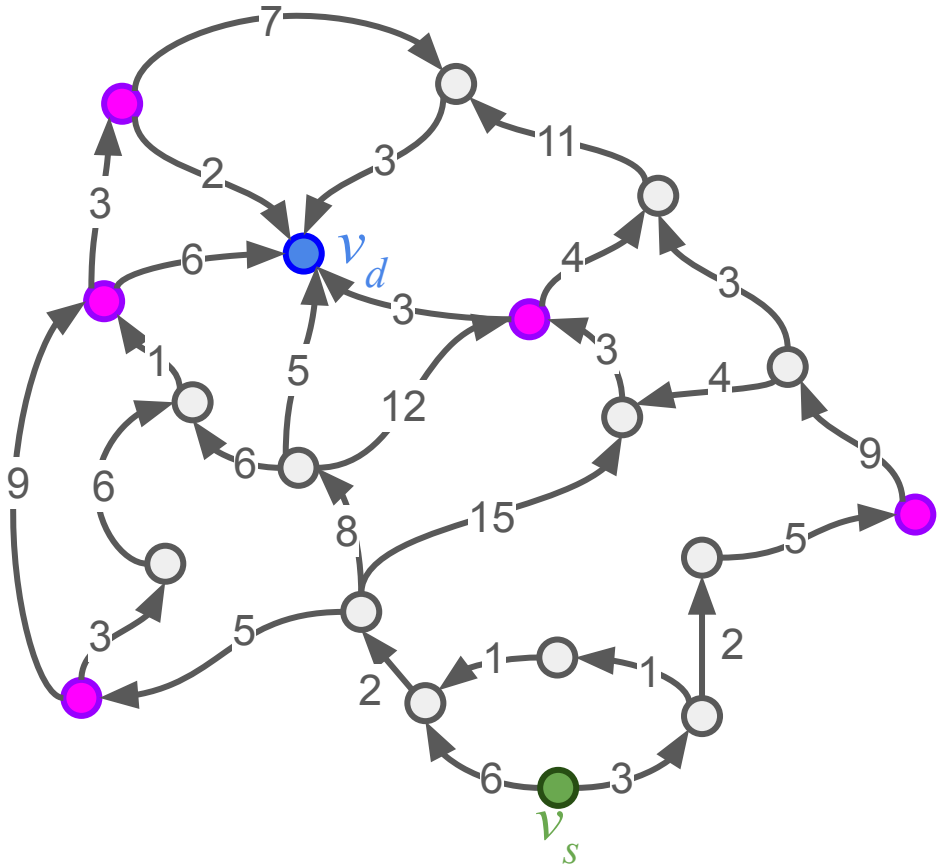
Now simply run SSSP from u and we'll directly obtain the shortest path that passes through a mandatory vertex! Of course, we'll have to merge with the SSSP solution from the first part (from computing a_i) to recover the full path.

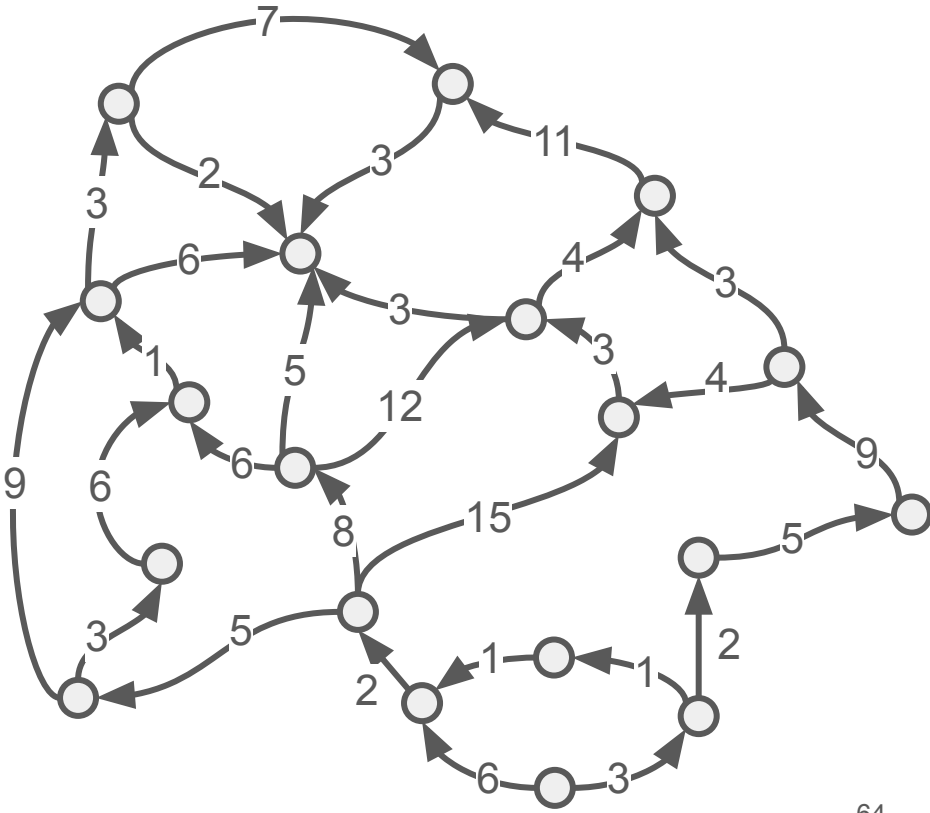


Solution 4

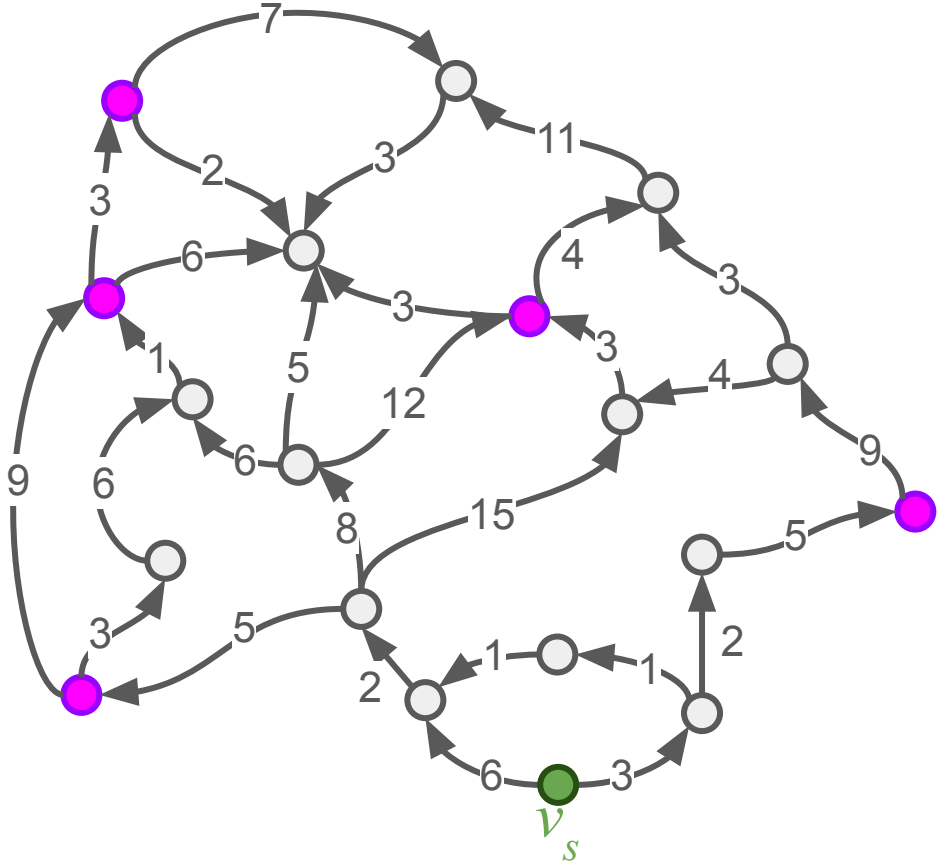
There exists yet another method to find the shortest path from v_s to v_d by *forcing* the exploration to pass through at least one mandatory vertex along the way.

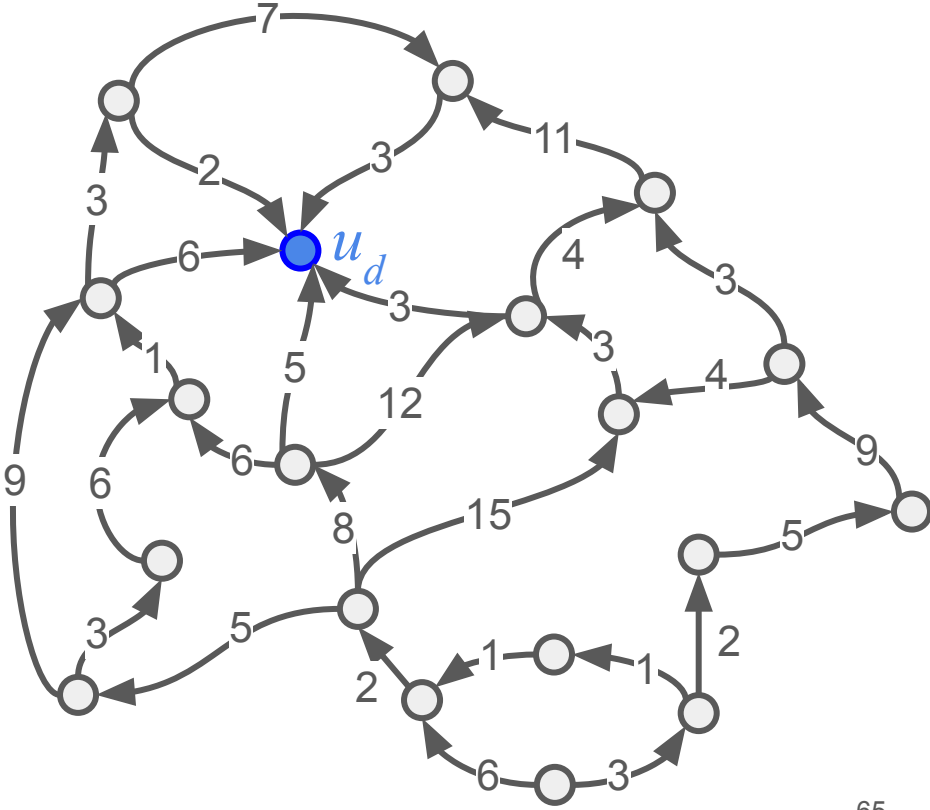
Step 1: Make a duplicate graph G'



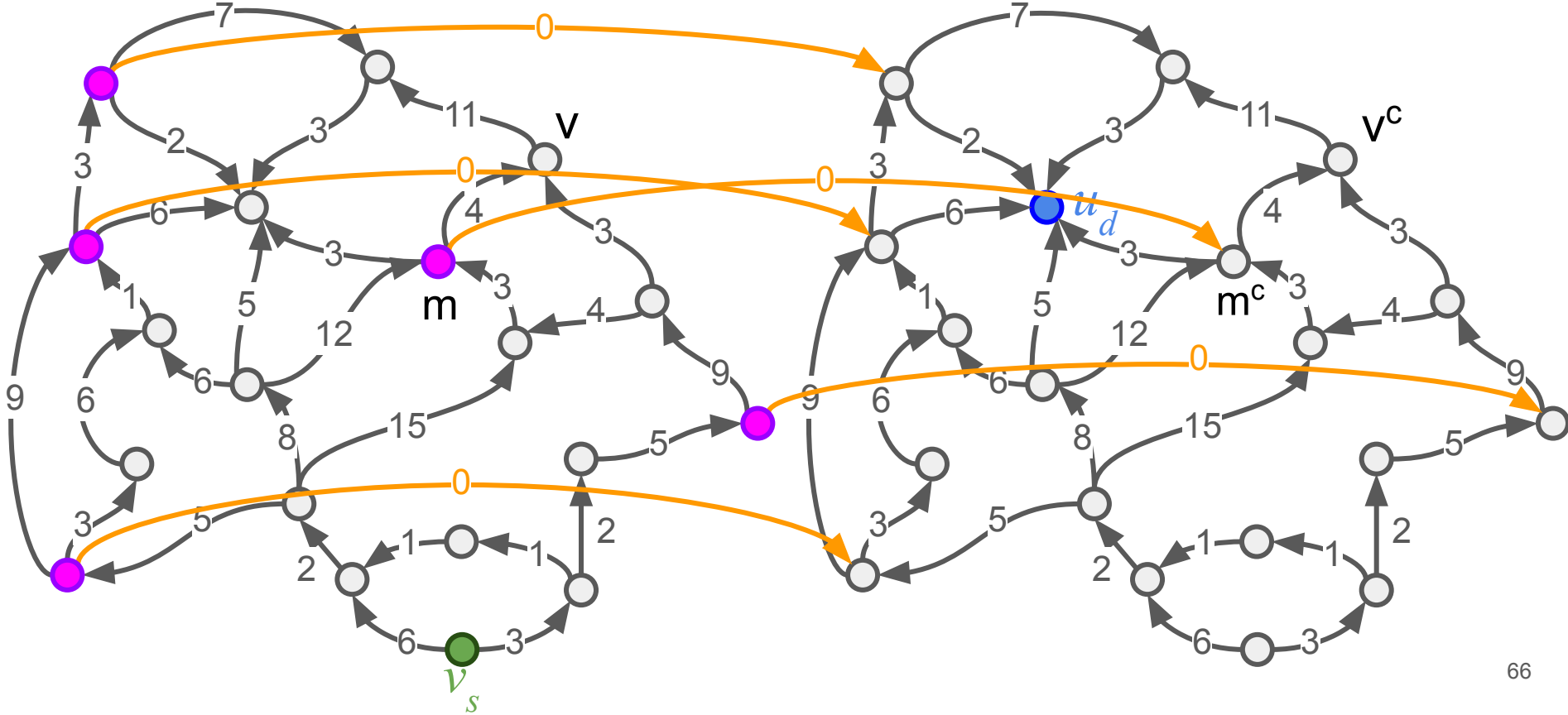


Step 2: Set vertex in G' as our new destination





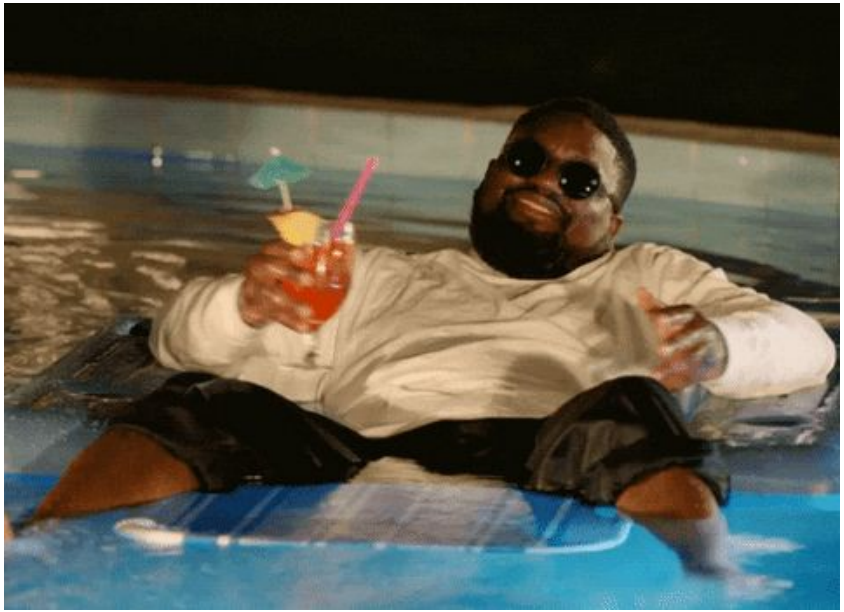
Step 3: Connect outgoing edges to second graph



Step 4

Just run your favourite SSSP algorithm as per normal!

The SSSP exploration will now be forced to visit the mandatory vertices because they are now the only *bridges* to our destination vertex in graph G'.



Test yourself

Do we have to make a literal copy of the graph to implement this solution?

Test yourself

Do we have to make a literal copy of the graph to implement this solution?

Answer:

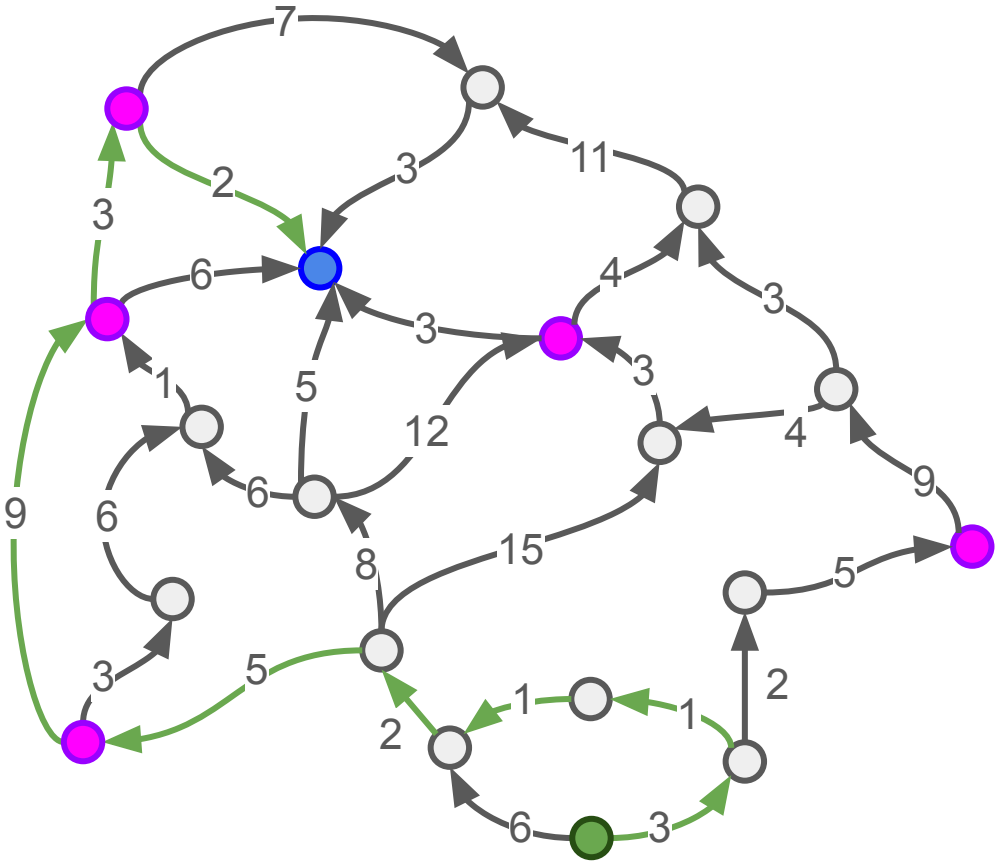
No we don't. We can traverse G' implicitly by using a separate distance table D' to indicate distance estimates of vertices in the copied graph. When we relax a vertex u during an exploration which visited a mandatory vertex before, we update estimate for u in D' instead of D.

Space complexity is $\mathrm{O}(V)$.

Answer

Total time: 26

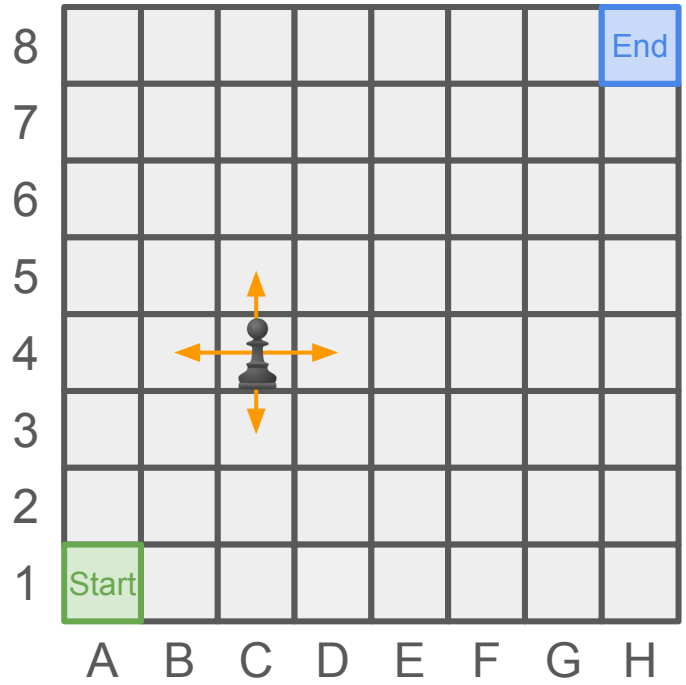
We passed by 3 supply drops!



Winning the Games

Problem 3: Description

- You have a $n \times n$ chessboard
- Your piece start at A1 and the endpoint is H8
- Your piece may only move one step at a time in ←↑↓→ if the move is valid



Problem 3: Description

- Each piece is associated with reward points
- A valid move is one in which the piece is moving into a square of < points than the current square
- Goal: find the sequence of valid moves that give the most points

	A	B	C	D	E	F	G	H
8	32	44	92	32	14	10	8	5
7	20	54	61	71	18	16	12	32
6	46	80	88	90	9	18	90	7
5	61	83	30	29	22	20	25	10
4	97	50	51	57	30	80	78	15
3	92	89	60	54	88	47	82	33
2	99	83	68	50	23	92	9	68
1	100	76	73	99	55	74	73	22

Example

- A valid sequence of moves amounting to 586 points
- This is not the best path however

	A	B	C	D	E	F	G	H
8	32	44	92	32	14	10	8	5
7	20	54	61	71	18	16	12	32
6	46	80	88	90	9	18	90	7
5	61	83	30	29	22	20	25	10
4	97	50	51	57	30	80	78	15
3	92	89	60	54	88	47	82	33
2	99	83	68	50	23	92	9	68
1	100	76	73	99	55	74	73	22

Problem 3.a.

Explain how the problem can be modelled as a directed graph.

How many nodes does your graph have?

State one important property of your graph.

Draw a small example (e.g., for $n=2$ or $n=3$) to illustrate.

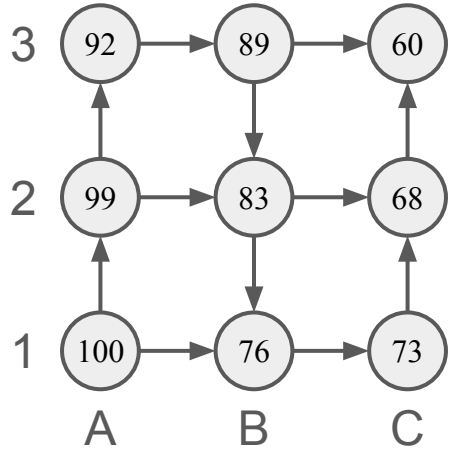
Discuss

How should we model this?

3	92	89	60
2	99	83	68
1	100	76	73
	A	B	C

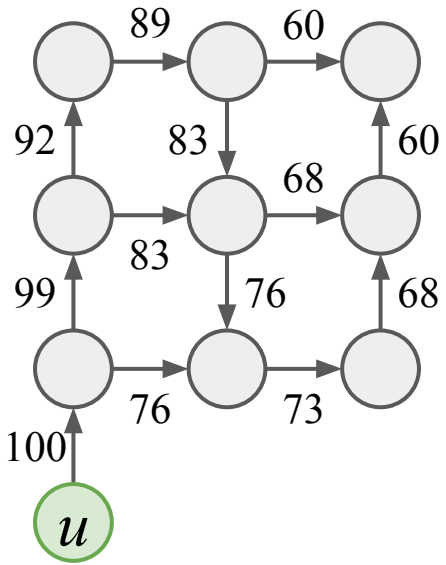
Modelling choice 1: Vertex weights

	A	B	C
3	92	89	60
2	99	83	68
1	100	76	73



Modelling choice 2: Edge weights

	A	B	C
3	92	89	60
2	99	83	68
1	100	76	73



Problem 3.b.

Give an efficient algorithm for finding the sequence of moves which attains the maximum possible reward.

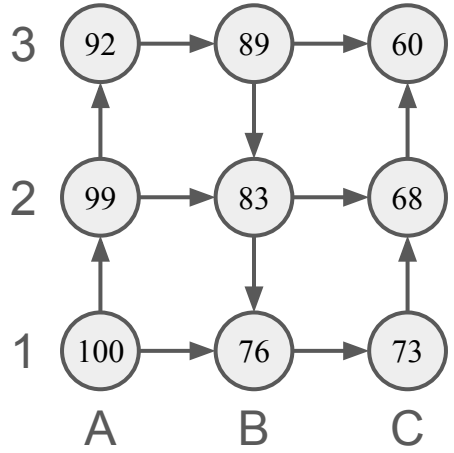
You may use any algorithm from class in unmodified form without reproducing it.

What is the (asymptotic) performance of your algorithm?

Solution 1

Modelling reward points as *vertex weights*.

	A	B	C
3	92	89	60
2	99	83	68
1	100	76	73



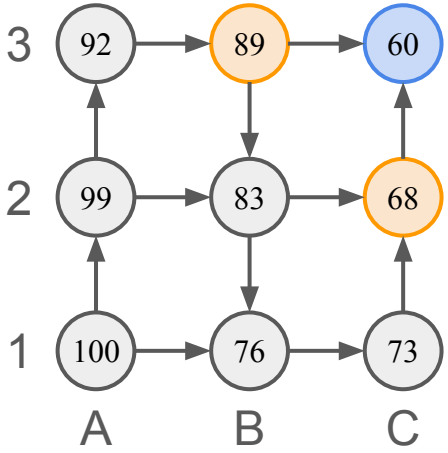
Observation

Let us denote $D[v]$ as the most rewarding path to vertex v. Then,

$$D[\text{C3}] = \max(D[\text{B3}], D[\text{C2}]) + 60$$

In other words, the most rewarding path to C3 is the maximum of:

- Most rewarding path of B3 + 60
- Most rewarding path of C2 + 60



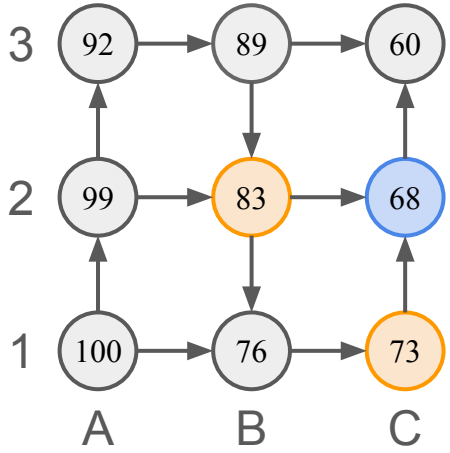
Observation

Likewise,

$$D[C2] = \max(D[B2], D[C1]) + 68$$

The most rewarding path to C2 is the maximum of:

- Most rewarding path of B2 + 68
- Most rewarding path of C1 + 68

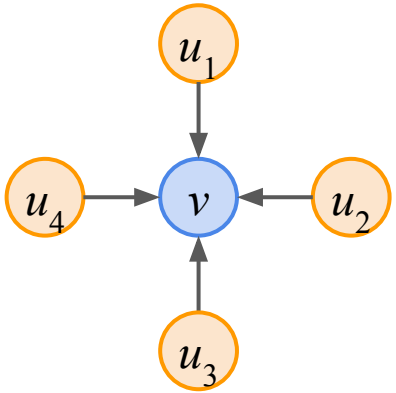


Observation: Subproblem

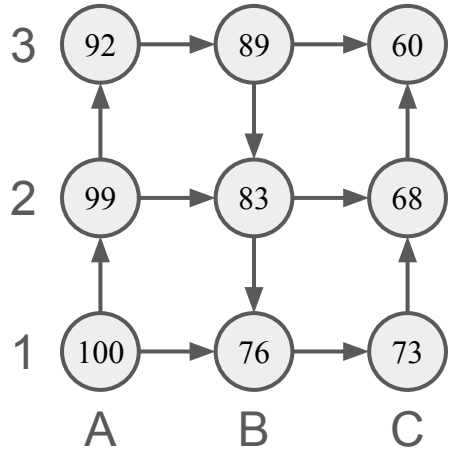
The most rewarding path to a vertex v is the *maximum* of all the most rewarding paths to u where $u \rightarrow v$, plus v's weight.

This means if we can solve all the subproblems *in sequence*, ending with the destination vertex, we would be able to incrementally compute the most rewarding path leading up to it.

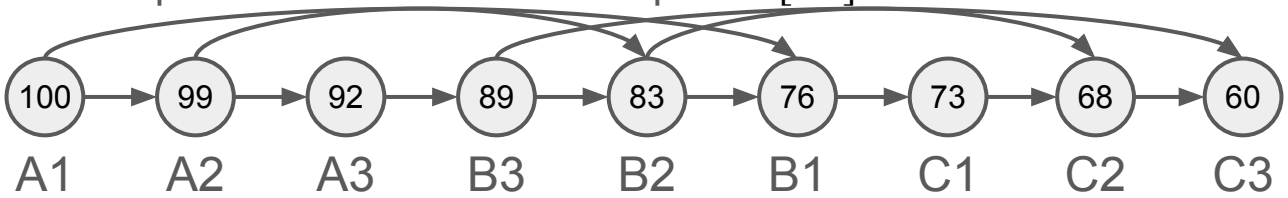
What sequence are we referring to?



Solution 1



We can simply compute $D[u]$ for every vertex from left to right in the sequence below until we compute $D[\mathrm{C3}]$.



Test yourself!

How did we obtain the sequence for which we solve the subproblems leading to the final solution?

Test yourself!

How did we obtain the sequence for which we solve the subproblems leading to the final solution?

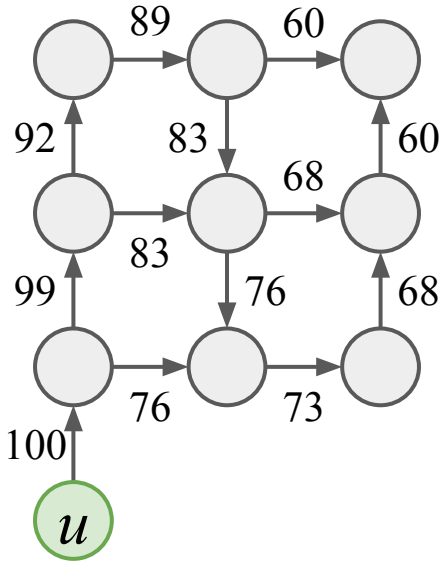
Answer: It is just the topological ordering!

Recall that topological order for a graph means that for any edge $u \rightarrow v$, u will appear before v in the topological sort. Hence by solving this problem in topological ordering, by the time we get to v, we are guaranteed to have obtained the best paths for all the incoming vertices to v.

Solution 2

Modelling reward points as *weighted edges*.

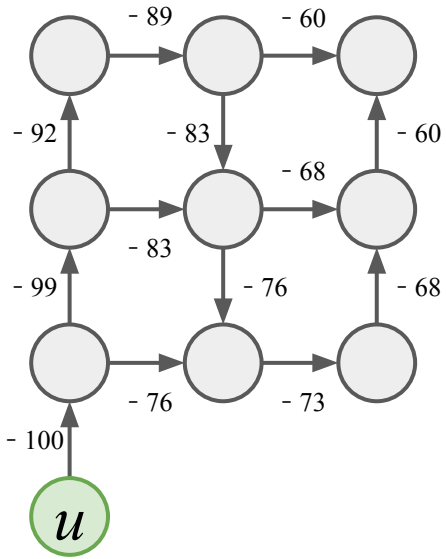
3	92	89	60
2	99	83	68
1	100	76	73
	A	B	C



Solution 2

Graph transformation: negate all the edges

Algorithm: SSSP with vertex u as the source



Test yourself!

Which SSSP algorithm should we use and how can it be more optimal for this question?

Test yourself!

Which SSSP algorithm should we use and how can it be more optimal for this question?

Answer: We should use Bellman-Ford since there are negative weighted edges. However we don't have to spend $O(VE)$ time.

We can simply obtain the topological ordering in $O(V+E)$ time and then do a single $O(E)$ pass of relaxations with that order.

Test yourself!

What’s the danger of running SSSP algorithms on a graph with negative edges? Do we risk the same here?

Test yourself!

What's the danger of running SSSP algorithms on a graph with negative edges? Do we risk the same here?

Answer:

The risk is that we run into negative weighted cycles.

Fortunately we won't run into them in this problem because the graph is *acyclic*!

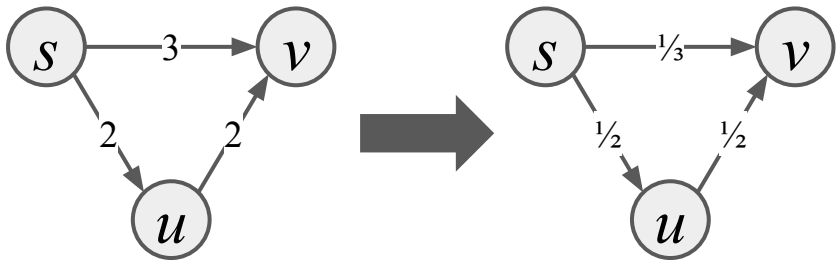
Test yourself!

If we take the reciprocal of each edge weight instead of negating it (so that we avoid negative edges), will that also work?

Test yourself!

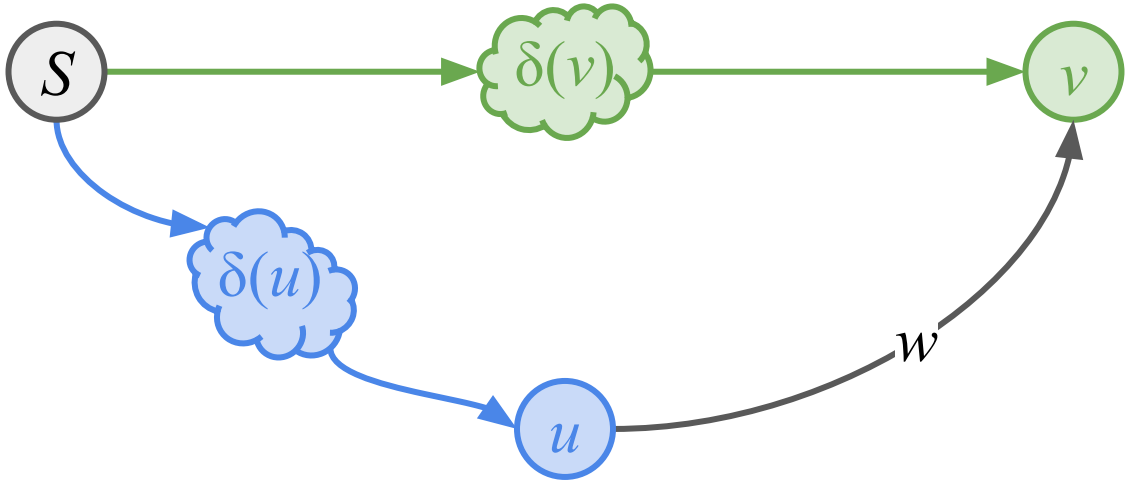
If we take the reciprocal of each edge weight instead of negating it (so that we avoid negative edges), will that also work?

Answer: No. Consider the example on the right. We should pick $s \to u \to v$, but after taking reciprocal of edge weights, the SSSP algorithm will pick $s \to v$.



So which type of modifications can we make and not make?

Triangle inequality



SSSP's triangle inequality:

$$\delta(v) \leq \delta(u) + w$$

Triangle inequality

The “raw” triangle inequality in this problem we aim to achieve:

$$\delta(v) \geq \delta(u) + w$$

Where $\delta(v)$ is result of $w_i + \ldots + w_k$ in the best path to v.

Therefore we can negate edges to reduce it to a “SSSP-friendly” form:

$$-\delta(v) \leq -\delta(u) + -w$$

$$\delta'(v) \leq \delta'(u) + w'$$

Triangle inequality

However if we take reciprocal, we are telling SSSP to solve using the following inequality:

$$\delta'(v) \leq \delta'(u) + 1/w$$

This cannot be reduced back to our raw inequality due to the non-commutativity of addition:

$$\delta'(v) = 1/w_i + \ldots + 1/w_k \neq 1/\delta(v)$$

Therefore it is no longer the same problem!

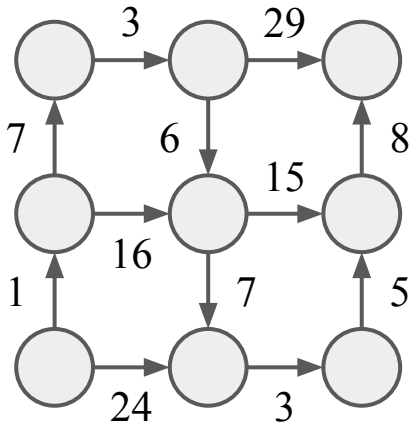
The counterexample for this was illustrated earlier.

Wrong solution

Some students have suggested modelling edge weights as the difference in points across cells, then solve the graph as a SSSP problem to minimize the “points lost”.

This might appear reasonable until you realize all paths sum up to the same amount! Why?

	A	B	C
3	92	89	60
2	99	83	68
1	100	76	73



Non-graph solution

Did you know: there is a direct correspondence between Dynamic Programming (DP) problems and DAGs.

Revisit this problem again after you have learnt about DP :)