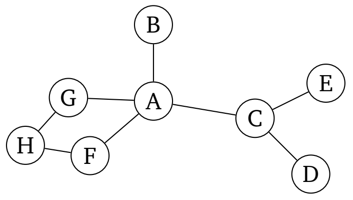
Section 07: Solutions

1. Graph traversal

(a) Consider the following graph. Suppose we want to traverse it, starting at node A.



If we traverse this using *breadth-first search*, what are *two* possible orderings of the nodes we visit? What if we use *depth-first search*?

Solution:

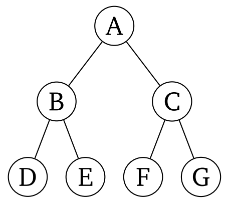
Here are two possible orderings for BFS:

- A, G, F, B, C, H, D, E
- A, C, B, F, G, D, H, E

Here are two possible orderings for DFS:

- A, G, H, F, C, D, E, B
- A, B, C, E, D, F, H, G

(b) Same question, but on this graph:



Solution:

Here are two possible orderings for BFS:

- A, B, C, D, E, F, G
- A, C, B, F, G, D, E

Here are two possible orderings for DFS:

- A, B, D, E, C, F, G
- A, C, G, F, B, E, D

2. Implementing graph searches

(a) Come up with pseudocode to implement *breadth-first search* on a graph, given a starting node and an adjacency list representation of the graph. Is your method recursive or not? What data structures do you use?

Solution:

See the pseudocode given in lecture.

(b) Come up with pseudocode to implement *depth-first search* on a graph, given a starting node and an adjacency list representation of the graph. Is your method recursive or not? What data structures do you use?

Solution:

See the pseudocode given in lecture.

3. Design Problem: Pathfinding in mazes

Suppose we are trying to design a maze within a 2d top-down video-game. The world is represented as a grid, where each tile is either an impassable wall, an open space a player can pass through, or a *wormhole*. On each turn, the player may move one space on the grid to any adjacent open tile. If the player is standing on a wormhole, they can instead use their turn to teleport themselves to the other end of the wormhole, which is located somewhere else on the map.

Now, suppose the there are several coins scattered throughout the map. Your goal is to design an algorithm that finds a path between the player and some coin in the fewest number of turns possible.

Describe how you would represent this scenario as a graph (what are the vertices and edges? Is this a weighted or unwighted graph? Directed or undirected?). Then, describe how you would implement an algorithm to complete this task.

Solution:

We can represent this as an undirected, unweighted graph where each tile is a vertex. Edges connect tiles we can travel between. When we have a wormhole, we add an extra edge connecting that wormhole tile to the corresponding end of the wormhole.

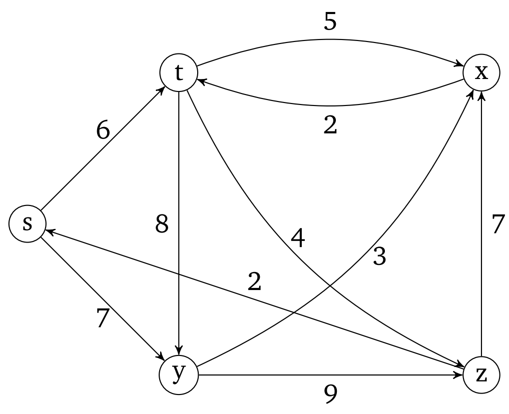
Because it takes only one turn to travel to each adjacent tile, there is actually no need to store edge weights: it costs an equal amount to move to the next vertex.

All paths are bidirectional, so we can also use an undirected graph. (If there are paths or wormholes that are one-way, we can switch to using a directed graph).

To find the shortest path, we can run BFS starting with the player and stop the moment we hit a coin.

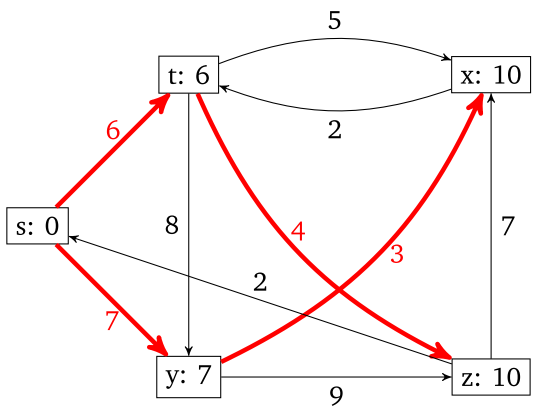
4. Simulating Dijkstra's

(a) Consider the following graph:



Suppose we run Dijkstra's algorithm on this graph starting with vertex s. What are the final costs of each vertex and the shortest paths from s to each vertex?

Solution:

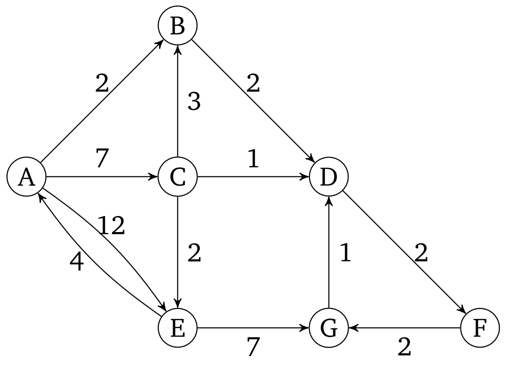


The table below shows the steps of running Dijkstra's algorithm.

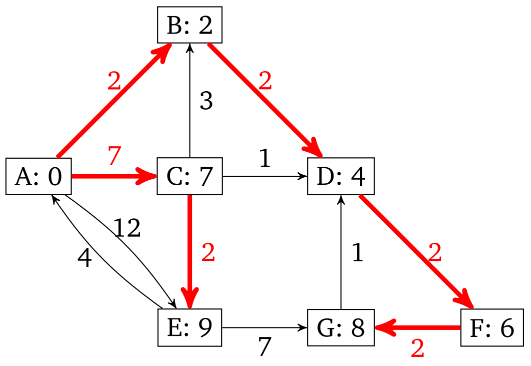
Vertex	Distance	Predecessor	Processed
s	0	–	YES
t	6	s	YES
y	7	s	YES
z	10	t	YES
x	10	y	YES

Note that the order of the fourth and fifth vertices are interchangable, since their costs are the same.

(b) Here is another graph. What are the final costs and shortest paths if we run Dijkstra's starting on node A?

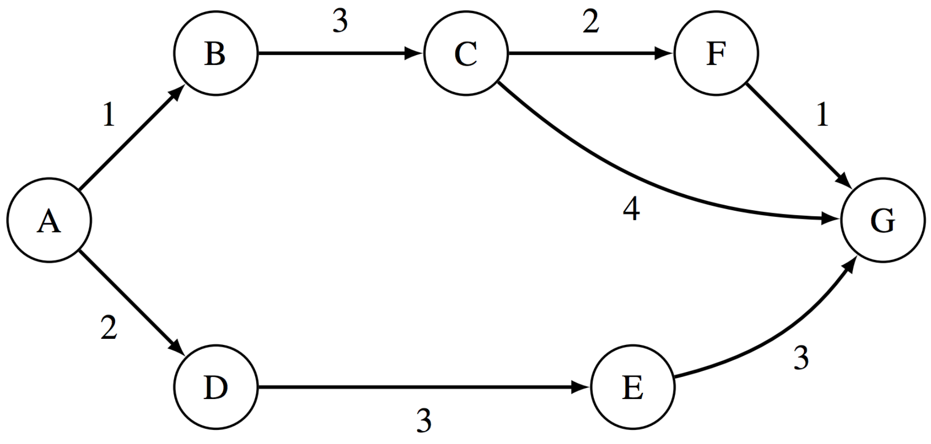


Solution:



5. A* Search

For the graph below, let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`.



Heuristics
$h(A, G) = 8$
$h(B, G) = 6$
$h(C, G) = 5$
$h(F, G) = 1$
$h(D, G) = 6$
$h(E, G) = 3$

(a) Given the weights and heuristic values for the graph above, what path would A* search return, starting from A and with G as a goal?

Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, h(A))
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
    poppedNode, poppedPriority = PQ.pop()
    if (poppedNode == goal): terminate

    for child in poppedNode.children:
        if PQ.contains(child):
            potentialDist = distTo[poppedNode] + edgeWeight(poppedNode, child)

            if potentialDist < distTo[child]:
               distTo.put(child, potentialDist)
               PQ.changePriority(child, potentialDist + h(child))
```

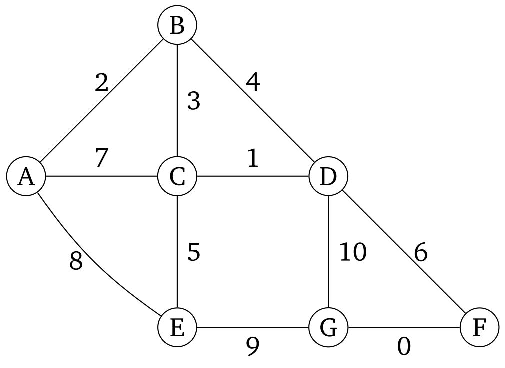
Solution:

A* would return $A - D - E - G$. The cost here is $2 + 3 + 3 = 8$.

Explanation: A* runs in a very similar fashion to Dijkstra's. The only difference is the priority in the priority queue. For A*, whenever computing the priority (for the purposes of the priority queue) of a particular node n, always add $h(n)$ to whatever you would use with Dijkstra's.

6. Minimum spanning trees

Consider the following graph:



(a) What happens if we run Prim's algorithm starting on node A? What are the final costs and edges selected? Give the set of edges in the resulting MST.

Solution:

Step	Components	Edge
1	{A} {B} {C} {D} {E} {F} {G}	(A,B)
2	{A,B} {C} {D} {E} {F} {G}	(B,C)
3	{A,B,C} {D} {E} {F} {G}	(C,D)
4	{A,B,C,D} {E} {F} {G}	(C,E)
5	{A,B,C,D,E} {F} {G}	(D,F)
6	{A,B,C,D,E,F} {G}	(F,G)

(b) What happens if we run Prim's algorithm starting on node E? What are the final cost and edges selected? Give the set of edges in the resulting MST.

Solution:

Step	Components	Edge
1	{A} {B} {C} {D} {E} {F} {G}	(E,C)
2	{C,E} {A} {B} {D} {F} {G}	(C,D)
3	{C,D,E} {A} {B} {F} {G}	(C,B)
4	{B,C,D,E} {A} {F} {G}	(B,A)
5	{A,B,C,D,E} {F} {G}	(D,F)
6	{A,B,C,D,E,F} {G}	(F,G)

(c) What happens if we run Prim's algorithm starting on *any* node? What are the final costs and edges selected? Give the set of edges in the resulting MST.

Solution:

The answer would be the same as the one we get above, since for each node, we always choose the smallest-weight edge that links to it.

(d) What happens if we run Kruskal's algorithm? Give the set of edges in the resulting MST.

Solution:

We'll use this table to keep track of components and edges we processed. The edges are listed in an order sorted by weight.

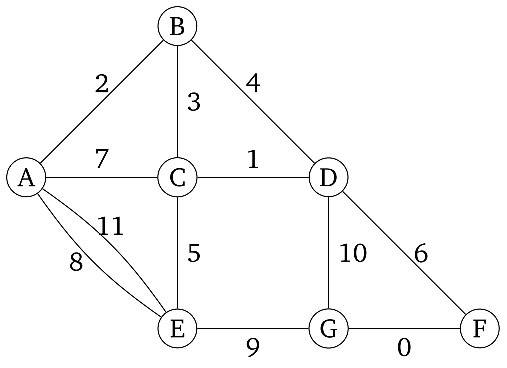
Step	Components	Edge	Include?
1		(F,G)	
2		(C,D)	
3		(A,B)	
4		(B,C)	
5		(B,D)	
6		(C,E)	
7		(D,F)	
8		(A,C)	
9		(A,E)	
10		(E,G)	
11		(D,G)	

After executing Kruskal's algorithm on the above graph, we get

Step	Components	Edge	Include?
1	{A} {B} {C} {D} {E} {F} {G}	(F,G)	Yes
2	{A} {B} {C} {D} {E} {F,G}	(C,D)	Yes
3	{A} {B} {C,D} {E} {F,G}	(A,B)	Yes
4	{A,B} {C,D} {E} {F,G}	(B,C)	Yes
5	{A,B,C,D} {E} {F,G}	(B,D)	No
6	{A,B,C,D} {E} {F,G}	(C,E)	Yes
7	{A,B,C,D,E} {F,G}	(D,F)	Yes
8	{A,B,C,D,E,F,G}	(A,C)	No
9	{A,B,C,D,E,F,G}	(A,E)	No
10	{A,B,C,D,E,F,G}	(E,G)	No
11	{A,B,C,D,E,F,G}	(D,G)	No

The resulting MST is a set of all edges marked as *Include* in the above table.

(e) Suppose we modify the graph above and add a heavier parallel edge between A and E, which would result in the graph shown below. Would your answers for above subparts (a, b, c, and d) be the same for this following graph as well?



Solution:

The steps are exactly the same, since we don't consider the heavier edge when there are parallel edges. The reason is that the heavier edge would be considered as the best edge when there is a lighter one that can be added to the graph.

7. More MSTs

Answer each of these true/false questions about minimum spanning trees.

(a) A MST contains a cycle.

Solution:

False. Trees (including minimum spanning trees) never contain cycles.

(b) If we remove an edge from a MST, the resulting subgraph is still a MST.

Solution:

False, the set of edges we chose will no longer connect everything to everything else.

(c) If we add an edge to a MST, the resulting subgraph is still a MST.

Solution:

False, an MST on a graph with n vertices always has $n-1$ edges.

(d) If there are V vertices in a given graph, a MST of that graph contains $|V|-1$ edges.

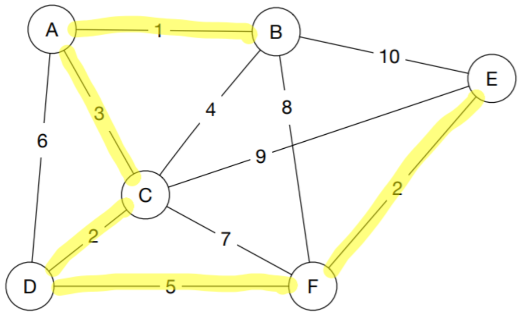
Solution:

This is true (assuming the initial graph is connected).

Answer these questions about Kruskal's algorithm.

(a) Execute Kruskal's algorithm on the following graph. Fill the table.

Solution:



Step	Components	Edge	Include?
1	{A} {B} {C} {D} {E} {F}	A, B	Yes
2	{A, B} {C} {D} {E} {F}	D,B, C	Yes
3	{A,B} {C,D} {E} {F}	E, F	Yes
4	{A,B} {C, D} {E, F}	A, C	Yes
5	{A, B, C, D} {E, F}	B, C	No
6	—"—	D, F	Yes
7	{A, B, C, D, E, F}	A, D	No
8	—"—	C, F	No
9	—"—	B, F	No
10	—"—	C, E	No
11	—"—	E, B	NO

(b) In this graph there are 6 vertices and 11 edges, and the for loop in the code for Kruskal's runs 11 times, a few more times after the MST is found. How would you optimize the pseudocode so the for loop terminates early, as soon as a valid MST is found.

Solution:

Use a counter to keep track of the number of edges added. When the number of edges reaches $|V|-1$, exit the loop.

8. Graph Modeling 1: DJ Kistra

You've just landed your first big disk jockeying job as "DJ Kistra."

During your show you're playing "Shake It Off," and decide you want to slow things down with "Wildest Dreams." But you know that if you play two songs whose tempos differ by more than 10 beats per minute or if you play only a portion of a song, that the crowd will be very disappointed. Instead you'll need to find a list of songs to play to gradually get you to "Wildest Dreams." Your goal is to transition to "Wildest Dreams" as quickly as possible (in terms of seconds).

You have a list of all the songs you can play, their speeds in beats per minute, and the length of the songs in seconds.

(a) Describe a graph you could construct to help you solve the problem. At the very least you'll want to mention what the vertices and edges are, and whether the edges are weighted or unweighted and directed or undirected.

Solution:

Have a vertex for each song. Draw a directed edge from song A to song B if (and only if) song B is slower than A, but the difference between their speeds is at most 10 beats per minute. Add a weight equal to the length of song B to each such edge.

(b) Write pseudocode to construct your graph from the previous part. You may assume your songs are stored in whatever data structure makes this part easiest. Assume you have access to a method `makeEdge(v1, v2, w)` which creates an edge from `v1` to `v2` of weight `w`.

Solution:

```
foreach(Song s1){
    foreach(Song s2){
        if( s2.bpm < s1.bpm && |s1.bpm - s2.bpm| <= 10)
            insert(s1, s2, s2.songLength)
    }
}
```

As long as our data structure as an efficient iterator this algorithm will run in $O(|V|^2)$ time. If your song is stored in a data structure that can be sorted by bpm, you can increase the speed to $O(S \log S + E)$ where S is the number of songs and E is the number of edges in the resulting graph by adding an early exit to the loop.

(c) Describe an algorithm you could run on the graph you just constructed to find the list of songs you can play to get to "Wildest Dreams" the fastest without disappointing the crowd.

Solution:

Run Dijkstra's from "Shake It Off." When the algorithm finishes, use back pointers from "Wildest Dreams" (and reverse the order) to find the songs to play.

(d) What is the running time of your plan to find the list of songs? You should include the time it would take to construct your graph and to find the list of songs. Give a simplified big-O running time in terms of whatever variables you need.

Solution:

The answer will depend on what you chose in the previous parts. The sorted list approach gives a running time of $O(S \log S + E \log S)$

9. Graph Modeling 2: Snow Day

After 4 snow days last year, UW has decided to improve its snow response plan. Instead of doing "late start" days, they want an "extended passing period" plan. The goal is to clear enough sidewalks that everyone can get from every classroom to every other **eventually** but not necessarily very quickly.

Unfortunately, UW has access to only one snowplow. Your goal is to determine which sidewalks to plow and whether it can be done in time for the first 8:30 AM lectures.

You have a map of campus, with each sidewalk labeled with the time it will take to plow to clear it.

(a) Describe a graph that would help you solve this problem. You will probably want to mention at least what the vertices and edges are, whether the edges are weighted or unweighted, and directed or undirected.

Solution:

Have a vertex for each building and an edge for each section of sidewalk. The edges should be undirected, and weighted by the time it will take the snowplow to clear it.

(b) What algorithm would you run on the graph to figure out which sidewalks to plow? Explain why the output of your algorithm will be able to produce a "extended passing period" plowing plan.

Solution:

Run an MST algorithm (either Kruskal's or Prim's). Whatever edges are chosen are the sidewalks the plow should clear. Why is this valid for the extended passing period plan? For example, why can students get from every classroom to every other?

(c) How can you tell whether the plow can actually clear all the sidewalks in time?

Solution:

Look at the weight of the MST. That's how long it will take to plow. If the plow can start in time to finish by 8:30, then we can start on time!

10. Design Problem: Negative edge weights

If you enjoy reading Pokémon jokes, you can read the flavor text to understand where the graph problem comes from. Otherwise, you can skip those parts and just read the formal statements.

Flavor Text You and your trusty Pikachu have made it halfway through Viridian Forest, but things have taken a turn for the worst. That last Weedle poisoned your Pikachu, and you're all out of antidotes.

In the Pokémon world, the poison doesn't do any damage as long as you stay *perfectly still*. But every time you take a step, the poison does a little bit of damage to your poor friend Pikachu.

Thanks to Bulbapedia[1], you know the exact map of Viridian Forest. Knowing that each step will cost your Pikachu exactly one of its precious hit points, you will need to find an efficient path through the forest.[2]

Formal Statement In a video game you are playing, each step you take costs a character (Pikachu) one unit of health. You have a map of the level (Viridian Forest) – your goal is to reach the end of the level (marked on your map) while losing as little health as possible.

[1]Like Wikipedia, but for Pokémon!

[2]Don't worry about running into wild Pokémon. For some reason you have a huge number of repels. Next time, maybe invest in full heals or potions instead.

(a) Describe a graph and an algorithm run on that graph to find the path through the forest to save as many of Pikachu's hit points as possible (i.e. the path with the fewest number of steps).

Solution:

Have a vertex for each possible location in Viridian Forest, and an edge between every two vertices we can move between in one step. Since Pikachu loses the same amount of hit points per step, we can just leave the graph unweighted.

Since the graph is unweighted, we can just run BFS, starting from our current location, with a target of the end of Viridian Forest.

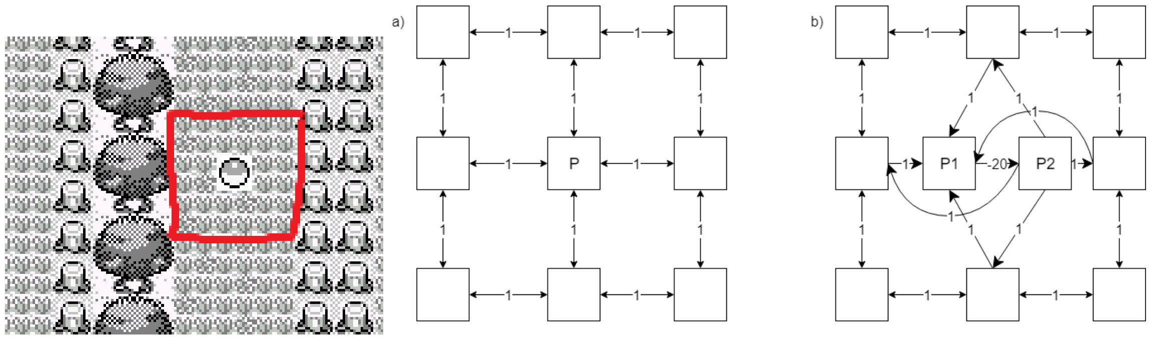
You could use either a directed graph or an undirected graph for this part.

(b) **Flavor Text** You run your algorithm and come to a devastating realization – the edge of Viridian Forest is at least 25 steps away, and Pikachu has only 20 hit points left. If you just walk to the end of the forest, Pikachu will faint before reaching the next Pokémon Center. So you come up with a backup plan. Returning to Bulbapedia, you see there is a potion just a little bit out of the way of the fastest path.

Brock tells you he knows how to update your graph to find the best path now. He says he'll add a dummy vertex to the graph where the potion is and connect up the new vertex with a (directed) edge of length -20, to represent undoing the loss of 20 hit points.

Formal Statement You realize your character doesn't have enough health to make it to the edge of the forest. But you know there is a healing item (a "potion") somewhere in the forest, that will give you back 20 units of health.

A friend (Brock) suggests the following update: add a dummy vertex to the graph where the healing item is and connect up the new vertex with a (directed) edge of length -20, to represent undoing the loss of 20 hit points.



9 spots in Viridian Forest, the corresponding vertices before Brock's transformation and the same vertices after the transformation.

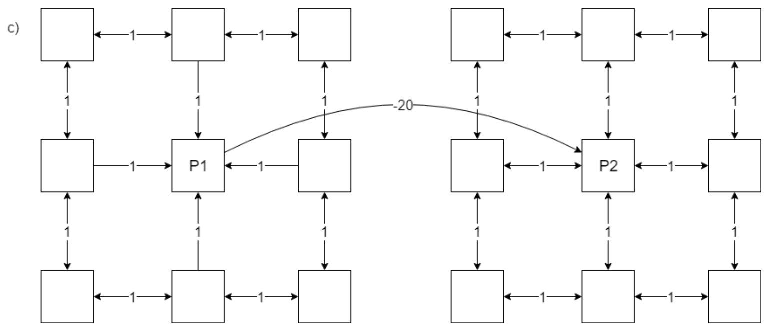
Tell Brock why his representation isn't quite going to work (hint: you can only use the potion once. What happens if the potion edge is part of a cycle?).

Solution:

The potion edge is part of a cycle.

What happens if you go around the cycle repeatedly? Each time the distance you've gone gets "shorter!" So no matter how many times you've gone around the cycle you should go around once more, and you'll be able to find an even shorter path from your current location to the edge of the forest. With Brock's representation, the shortest path isn't even defined!

(c) You convince Brock to change the graph representation. You'll now have two copies of the original Viridian Forest graph, in copy 1 the potion is still unused. In copy 2, the potion is no longer there. You add an edge of weight -20 from copy 1 to copy 2 at the location of the potion (crossing that edge represents using that potion). His new graph looks something like this.



Brock says he'll start running Dijkstra's. Should you trust the output?

Solution:

No, Dijkstra's isn't guaranteed to work when there are negative edges. Poor Brock. He knows so much about rock Pokemon but so little about algorithms.

Luckily your Pokédex gets good data service in Viridian Forest, and you look up the Bellman-Ford algorithm for finding shortest paths with negative edge weights and find the new best path.

(d) **Challenge Problem** Misty says she knows about a second potion somewhere else in the forest. Describe how to modify the graph to handle both of the potions.

Solution:

We now want 4 copies of the graph. One for each of (no potions used, only potion 1 used, only potion 2 used, both potions used). Make edges of weight -20 to connect these in the same way as you did in the last part.

To make it easier to choose a final destination, add a dummy destination vertex. Then add a weight 0 edge from each copy of the edge of the forest to the dummy destination.