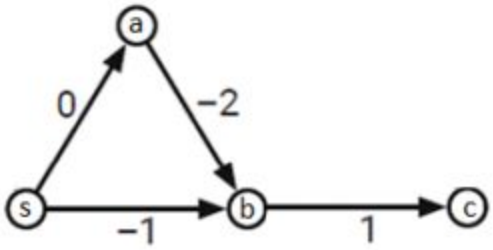
Carson Sklare

2.a

Graph shown:



Code used:

```
from dijkstar import Graph, find_path
g = Graph()
#s = 1, a = 2, b = 3, c = 4
g.add_edge(1,2,0)
g.add_edge(1,3,-1)
g.add_edge(2,3,-2)
g.add_edge(3,4,1)
print(find_path(g,1,4))
```

Result:

```
PathInfo(nodes=[1, 3, 4], edges=[-1, 1], costs=[-1, 1], total_cost=0)
```

Because it goes from a,b,c without taking into account s,a,b,c, this is using the nonnegativeDijkstra algorithm. This answer is shown because it only relaxes an edge once without checking because it would not need to check twice for nonnegative numbers. Because the value can go down due to a negative number the answer for this is wrong.

2.b

Code:

```
from networkx import *
#s = 1, a = 2, b = 3, c = 4
g2 = nx.DiGraph()
g2.add_node(1)
g2.add_node(2)
g2.add_node(3)
g2.add_node(4)
g2.add_edge(1,2,weight = 0)
g2.add_edge(1,3,weight = -1)
g2.add_edge(2,3,weight = -2)
g2.add_edge(3,4,weight = 1)
print(nx.dijkstra_path(g2,1,4))
```

Part of output, it will not compute the path because it detected negative weights:

```
ValueError: ('Contradictory paths found:', 'negative weights?')
```

2.c

By removing the node 4 and connecting edge from 3 to 4, I was able to get an answer of [1,3] meaning s directly connected to b without checking going from s to a to b.

Code:

```
from networkx import *
#s = 1, a = 2, b = 3, c = 4
g2 = nx.DiGraph()
g2.add_node(1)
g2.add_node(2)
g2.add_node(3)
g2.add_node(4)
g2.add_edge(1,2,weight = 0)
g2.add_edge(1,3,weight = -1)
g2.add_edge(2,3,weight = -2)
g2.add_edge(3,4,weight = 1)
print(nx.dijkstra_path(g2,1,3))
```

Result:

```
[1, 3]
```

2.d

For Bellman Ford, it got both answers correct as it went from 1,2,3,4 for going from s to c and 1,2,3 from going to s,b

Code:

```
from networkx import *
#s = 1, a = 2, b = 3, c = 4
g2 = nx.DiGraph()
g2.add_node(1)
g2.add_node(2)
g2.add_node(3)
g2.add_node(4)
g2.add_edge(1,2,weight = 0)
g2.add_edge(1,3,weight = -1)
g2.add_edge(2,3,weight = -2)
g2.add_edge(3,4,weight = 1)
print(nx.bellman_ford_path(g2,1,4))
print(nx.bellman_ford_path(g2,1,3))
```

Result:

```
[1, 2, 3, 4]
[1, 2, 3]
```

3. Use either BFS, DAGSSSP, Dijkstra or Bellman Ford:
   a. You've come up with the following board-game: you have a chessboard (8x8, or larger), and a king in the lower left-hand corner (A1) of the board. The goal is to move the king to the upper right-hand corner (H8). Your king can move one square in each step, and only towards the goal, so the king can only move up, right, or upper-right. Moving between squares can bestow treasure (say A3->A4 gives $100, and B7->B8 gives $50), but some other crossings between squares cost money (say A2->A3 costs $40). You want to find a path for the king that leaves him as rich as possible. Which algorithm do you use?
      i. **If there was no costs nor gains to the board it would be an undirected graph, probably would use BFS. There also are only three ways to move with only going towards the goal. I think because of this factor it would be best to make this into a DAG. I would think of using the DAGSSSP algorithm as it can relax nodes until the goal, making a best path along the way.**
   b. Same board-game as in a), but the king is now allowed to move in all directions (for one square), including up, upper left, left, lower-left, down, and lower-right. Which algorithm would you use to find a path on which the king makes most money?
      i. **Because the limitation of going in specific directions is gone, I think there will be a lot more possibilities to make more money to the goal. I think the best one to do for this chess problem would be BFS because I can tell from each position on the board what would be the best path because I can see all the neighboring spaces around the king.**
   c. You are given n cities in the plane, including Chicago. Distances between cities are the travel distances from one city to another (car, plane, for example). You want to find the

shortest routes from Chicago to each of the other n-1 cities. Which algorithm do you use?

   i. **If we're traversing to different cities from one place to the other without negative weight or looking back, I would think the Dijkstra algorithm would be the most efficient since I could find the shortests paths to each city from one another.**

d. Remember the glass problem from homework 1? You have a bunch of glasses (could be more than four), and there are certain rules on which glasses can be flipped in a single move (in homework 1 we were allowed to flip neighboring glasses); the rules are such that you can always go back and undo a move. You are given an initial and a target constellation for the puzzle, and are asked to find a quickest solution to the puzzle, that is, the shortest number of moves that get you from the initial to the target constellation. Which algorithm do you use?

   i. **Because you would have to look back at prior moves or do different types of moves to get to the same place more efficiently, the Bellman Ford algorithm would be useful. The algorithm could relax each node, or I think in this case each combination, so that I would be able to make a path from a starting position to the ending position only using the least amount of moves.**

e. [Extra Credit, +2EC] In the moving the king board game from a), some crossings are not allowed at all, e.g. there may be a wall between C4 and C5 which cannot be crossed. How can this be modelled in the shortest path problem?

   i. **I would think if we're keeping the restrictions from a, then we would still be using a DAG. I think in terms of that type of graph you would disconnect the edge of say C4 to C5. Because it is a DAG there will be another way to get to the end goal without having to go through certain edges.**

4.

You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where exactly should you meet?

You are given a weighted graph G = (V, E), where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight w(e) equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location. Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

Because they are traveling from different cities, in terms of a graph there will be no negative weight and since this seems similar to problem 3c, I would want to use the Dijkstra algorithm. Say we have a graph with p and q on different ends and we want to meet at t somewhere in the graph. I would want to run a Dijkstra algorithm from p to t to find my shortest distance to t and my friend would run the Dijkstra algorithm from q to t

to find their best walk to t. That way we both have our best path laid out for us to both meet at t.