Part 3:

- **What would you conclude the complexity of your two algorithms to be for dense graphs?**
  We know from the question that in dense graphs, Dijkstra has complexity O (E + V log V), or O (V^2) if the graph is dense and Bellman-Ford has complexity O (VE), or O (V^3) if the graph is dense. At the same time, we want to do the all-source shortest circuit, so we must perform Dijkstra, or Bellman-Ford, once for each vertex. so, the total complexity is O ($V \times V^2$) = O(V^3) (for Dijkstra) or O ($V \times V^3$) = O (V^4) (for Bellman Ford). Therefore, if we use Dijkstra to do all-source shortest circuit in the case of dense graphs, the complexity should be O (V^3), if we use Bellman Ford to do all-source shortest circuit in the case of dense graphs, the complexity should be O (V^4).

Part 4:

- **What issues with Dijkstra's algorithm is A* trying to address?**
  In Dijkstra's algorithm, it expands and updates the nodes only based on their distance from the source. This will lead to inefficiency since it didn't consider the weight and path after that node. A common possible situation is that it has less cost so far but from the node you in right now to the destination has more cost than other trace. Thus, we need to update more. In contrast, A* uses a heuristic function that estimates the cost to reach the destination from the current nodes, effectively "guiding" the search in the right direction. A* tends to focus the search around the most promising paths and reduces unnecessary computation.
- **How would you empirically test Dijkstra's vs A*? Describe the experiment in detail.**
  To compare and design empirically test for Dijkstra's Algorithm and A*, I will design the following detail:
  - Input Graph:
    1. For the input graph, we can either randomly generate like the lab we did before or use the London Subway system data provided.
    2. Try to choose the data base has multiple source-destination pairs to ensure a variety of path lengths and complexities.
  - Implementation:
    1. For Dijkstra's Algorithm and A*, make sure they will handle the same data structures. There are many ways to implement directed or undirected graphs, make sure the algorithms will work with the same data and use a common code base for consistency.
    2. Ensure that both implementations use a similar underlying data structure, such as a priority queue.
  - Generate Plot:
    1. Measure the run time for each algorithm which has the same input.

2. Put them in the plot and compare them to see the performance
   - Experimental Variations:
     1. By the trait of heuristic function, we could have several variations of how good the heuristic function is. What I mean by "how good" is how close is the estimate cost compared to the actual cost. We could have $h(n) = 0$, $h(n) < actual\ cost$, $h(n) = actual\ cost$, and $h(n) > actual\ cost$.
     2. Also, we can change graph density and size to see how each algorithm scales under different conditions, but we can ignore this if we use the London Subway system database.

- **If you generated an arbitrary heuristic function (like randomly generating weights), how would Dijkstra's algorithm compare to A*?**
  If heuristic function has a random generated weight, it will not provide any useful information for guiding which loses its mean purpose. It can misguide A*'s search, causing it to explore paths in a manner like Dijkstra's algorithm. Since the function is randomly generated, the algorithm might not guarantee an optimal solution for the shortest path. On the other hand, Dijkstra's algorithm will always guarantee an optimal solution if the graph does not have a negative weight. After A* loses its main ability to prioritize nodes that are more promising for reaching the destination, the performance of both algorithms would be comparable.
- **What applications would you use A* instead of Dijkstra's?**
  - GPS Systems: By the satellite, the system can estimate the distance from a location to anothor, however the actual distance won't just be the straight-line distance. Then, the straight-line distance could be a good heuristic. The actual distance will follow the road and trace that record in the system.
  - Network Routing: In cases where routes have spatial significance, A* can efficiently find the optimal path between nodes when the heuristic reflects geographical distances.

Part 5:

Under the database of the London subway system, A* always outperforms Dijkstra. In theory, A* and Dijkstra will be comparable when the estimate cost calculated by the heuristic function is equal to zero. However, we calculate the estimated by the latitude and longitude. Our heuristic function uses latitude and longitude to calculate the Euclidean distance and use it for estimating cost. It is a very relative guidance and provides a directional bias toward the destination, which helps A* prune some unnecessary nodes. Since the heuristic is meaningful and good, it helps A* more efficiently, which basically makes A* always outperforms Dijkstra. In our experiment, we considered the factor of the number of transfers in the shortest path as the following figure:

```
Source: 175, Destination: 106 | Transfers: 2 (multiple_transfers) | Dijkstra: 0.000119s, A*: 0.000117s
Source: 175, Destination: 69 | Transfers: 2 (multiple_transfers) | Dijkstra: 0.000129s, A*: 0.000136s
Source: 175, Destination: 86 | Transfers: 2 (multiple_transfers) | Dijkstra: 0.000205s, A*: 0.000140s
Source: 175, Destination: 152 | Transfers: 2 (multiple_transfers) | Dijkstra: 0.000201s, A*: 0.000146s
Source: 175, Destination: 174 | Transfers: 0 (same_line) | Dijkstra: 0.000063s, A*: 0.000092s

Average Running Times by Transfer Category:
Same_line: Dijkstra = 0.000149s, A* = 0.000125s
One_transfer: Dijkstra = 0.000226s, A* = 0.000156s
Multiple_transfers: Dijkstra = 0.000306s, A* = 0.000223s
```

We basically find the shortest path of each combination of stations and then calculate the number of transfers we must do for each of them. As we can observe, A* always outperforms Dijkstra. Also, as the number of transfers increases, A* seems to be more efficient. The more transfer the trip needs, implicitly implies the potential longer path. It will make the route more complex and makes the heuristic less accurate. However, even if the heuristic is less "tight" for complex routes, A* still benefits from using goal information. While a less accurate heuristic might lead to increased node expansion compared to an ideal scenario, in practice, A* can still outperform Dijkstra's algorithm because it leverages even imperfect guidance toward the destination and longer trip will makes A* outperforms more with its efficiency.

Part 6:

- **Discuss what design principles and patterns are being used in the diagram.**

  The UML diagram demonstrates several core design principles. It uses the **Strategy Pattern**, allowing different pathfinding algorithms (Dijkstra, Bellman_Ford, A_Star) to be used interchangeably through the SPAlgorithm interface. This promotes flexibility and separation of concerns. It also applies **abstraction and encapsulation** by hiding the internal workings of the Graph and exposing only essential methods. The design follows the **Open/Closed Principle**, making it easy to add new algorithms without changing existing code, and adheres to the **Liskov Substitution Principle**, as all algorithm classes can be used wherever SPAlgorithm is expected. Together, these principles create a modular, extensible, and maintainable system.

- **The UML is limited in the sense that graph nodes are represented by the integers. How would you alter the UML diagram to accommodate various needs such as nodes being represented as Strings or carrying more information than their names.? Explain how you would change the design in Figure 2 to be robust to these potential changes.**

  The current UML diagram assumes that graph nodes are represented by integers, which limits flexibility and scalability in real-world applications where nodes may need to be represented by strings, unique identifiers, or objects carrying additional information. To make the design more robust, the graph-related classes should be refactored to use **generics**, allowing them to handle any node type by defining the Graph class as Graph<T> and updating all method signatures to use T instead of int. Additionally, a

dedicated **Node class** could be introduced to encapsulate node-related data such as names, labels, or metadata. This class would replace simple integers and allow more expressive graph modeling. The SPAlgorithm interface and its implementing classes would also need to be updated to accept the generic node type or Node objects. These changes would improve the design's flexibility, support a wider range of applications, and maintain type safety while making the system adaptable to future requirements.

Here is an example how we change figure 2 to be robust to our changes