Assignment Peking Express

Group: №6
Aleksandar Georgiev, Ivaylo Ivanov

Approach

The initial concept for the algorithm was to use a modified version of one of the common pathfinding algorithms and adapt it to our use case. The first thing that came to mind was Dijkstra's algorithm, especially considering the fact that we also had to keep a budget in mind. However, in our case we must not consider the least expensive route, but the shortest that fits in the given budget. In that case BFS would make sense, since it will give us the shortest path. The problem that we run into now is taking budget into consideration - if the shortest path does not fit in the given budget, we need to find the second best option. That is not easily achievable with the above mentioned algorithms (we stumbled upon a nice approach online, but doing this for a third shortest, fourth shortest etc. would overcomplicate things a lot https://stackoverflow.com/a/4972027).

So the next best thing to do was generate all possible paths, sort them by length, then find the first one that fits within our budget. This way we will always find the best possible option, even if it is the 10th shortest path.

At first, critical locations seemed like a non-issue, since the assignment document states that the final node 88 should be reached in as few moves as possible, not as few turns as possible. This makes waiting around a non-move, making waiting around the best approach. Later it turned out that it should actually be done in the fewest turns. Our implementation still waits around if the next location in the path is critical and occupied. Another approach would have been to search for the next best route in that situation, however there are a lot of unknowns. The next best route may contain more critical locations than the current one, it may actually be better to wait around than to take it etc. This is why we stuck to the original idea of waiting around, we can not be sure that changing the path will yield a better result.

Another minor issue our implementation does not take into account is the number of critical locations along a possible path. However, similarly to the previous issue described, it may not always be best to take the path with least critical locations in a distributed game.

Code

https://pastebin.com/viRxAWnK
The code contains two classes - one for the graph (map) of the game, and one for the game itself. The graph class stores the following important fields:

1. vertices
2. paths - stores as a dictionary where the key is the node and the values are all connections to that node
3. paths - a list where all possible paths will be put
4. prices - a list where the price for each vertex is put
5. pathPrices - a list with prices where pathPrices[i] is the price of paths[i]

Additionally, this class has the following methods:

1. addEdge - adds a directional edge, is called twice for an undirected graph
2. calculatePathPrices - for all paths in self.paths, put the according price in pathPrices
3. pathsUtil - a utility function to generate all possible paths from a source to a destination node recursively
4. generateAllPaths - generates all paths from source to destination. It has the limitation that it can handle graphs up to 88 nodes for a game of peking express. No index of a node shall exceed 88.

The PekingGame class contains an instance of Graph, as well as a few other important components. As a constructor argument, it takes a path to a testfile. It extracts all the important information from it - parses the json graph, the start location, the budget, and the occupied locations for each turn.

It is important to know that since the game is not meant to be played in a distributed manner anymore, we have chosen to execute everything with the construction of the game object. This means that the game is ‘played’ when it is created. To get the final path our ‘player’ will take, it is only needed to print ‘finalPath’.

The class itself has the following methods:

1. initializeMap - parses the json input
2. setStartLocation
3. setBudget
4. updateTurnCount - called then a turn is played
5. updateOccupiedLocations - currentOccupiedLocations is the list where we can’t go on the current turn, this method makes sure to keep it updated
6. generateGraph - generates a graph from the given json, generates all paths in the graph, sorts them by length and calculated their prices
7. pickPath - finds the shortest path within the given budget
8. nextMove - returns the nextMove (it is required that a graph is already generated and a path is already chosen). It essentially only checks whether the next move in the chosen path is legal.

9. generateFinalPath - calls nextMove until the final node is reached. The path taken is found in finalPath.

Testing

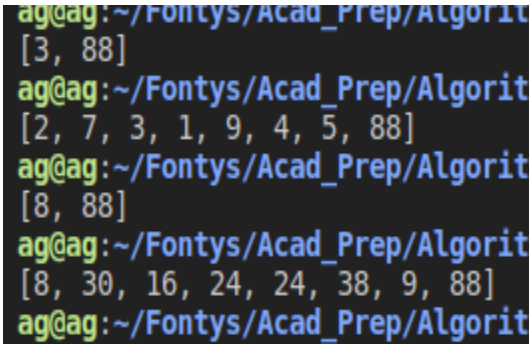
We started our testing with the test files that were provided in the FHICT portal (sharepoint), After a bit of confusion about the format of the data in the file we managed to get the correct results from the tests.

Testfile_1:

testFile_2:

Testfile_3:

Testfile_4:



After we were sure that those tests passed correctly. We constructed our own tests so we can thoroughly test our implementation in more specific situations - such as, when the shortest path is above our budget etc.

REST API

Before the change of the assignment to a non distributed manner we had implemented a REST API using flask(Code: https://pastebin.com/EusutsZ5). The rest api has four endpoints:

- initializeMap(POST): It receives a json object with the corresponding data inside and it sends that object to the peking express class for the initialization. We have an argument checking for every endpoint so we can be sure that they receive the data they need. Thus, if the endpoint receives a request without the json object or a json object with missing data inside, it will respond with status code *400 BAD REQUEST* and a help message.
- location (POST): It expects a json object with integer inside. When it receives the object it sends it to the peking express class for initialization
- nextMove (GET): It calls the peking express class and sends the result.

- UpdateCompretitorLocation (POST): it waits for a json object with two arguments inside. Then it again sends it to the peking express class so it locations can be updated.