Dynamic Shortest Path - Essentials

General search algorithm

A **search** problem consists usually of a **set** of nodes (problem **states**) where an algorithm is used to search through the set, to find some information. The information can be a targeted node, a set of nodes or related data to the topology of the problem.

The relation between the nodes form a topology called **Graph** in data structure.

Tools needed

Open List (Frontier)

An open list is needed to push in and pop out nodes.

Data structure: Represented as a priority queue or any container that can be sorted.

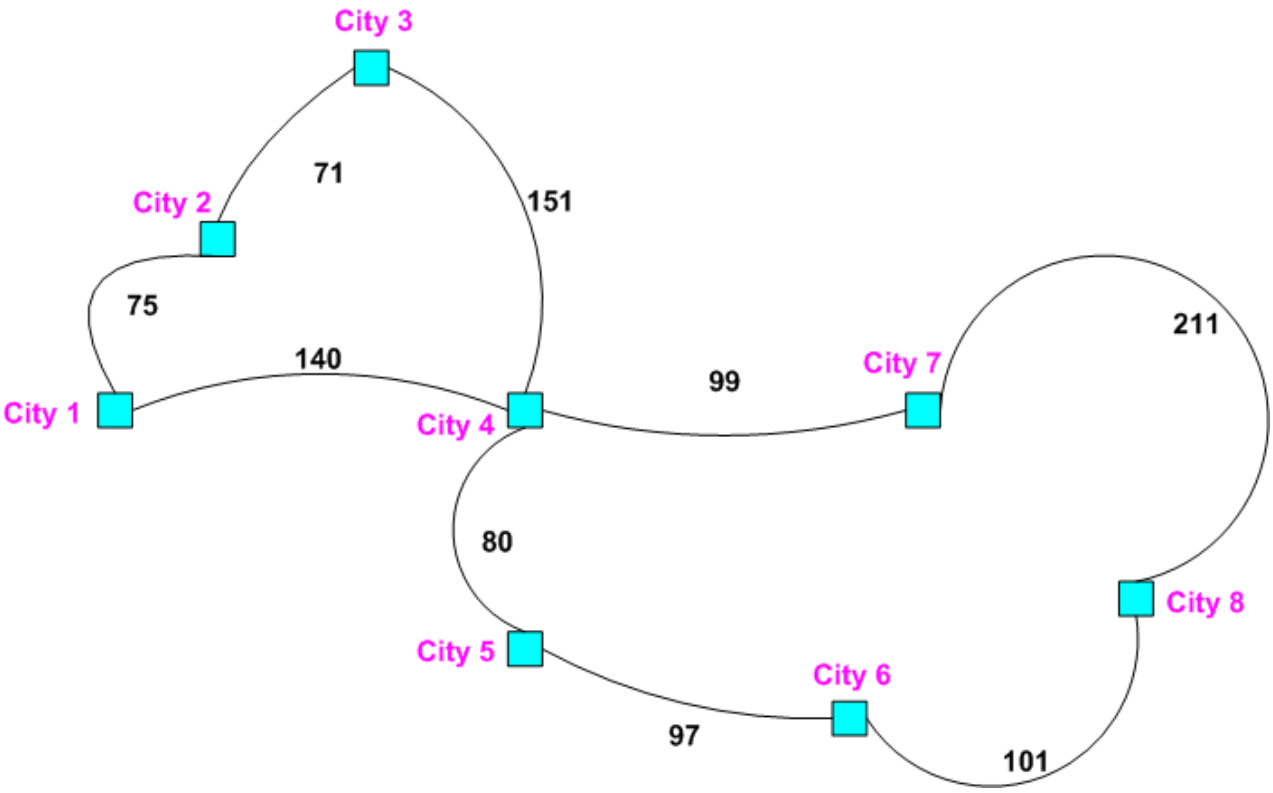
Closed List (Visited)

A closed list is needed to push in nodes.

Data structure: Any container that its items can be searched and compared

Problem

Given the following map of 8 cities, we want to find a (possible shortest) path that takes us from **Citi 1** to **Citi 8**.



The numbers on the edges of the map represent the actual distance cost between two cities.

The actual shortest search method (Uniform-Cost Search)

The algorithm relies on the historical previous search data computed since the start node.

```
struct Node
{
        void * data;
        Node * parent;
        uint actualCost;
};
```

startNode = Citi 1 (actualCost = 0)

targetNode = Citi 8

Add startNode to the open list.

```
ActualCost()
{
        While(open list not empty)
        {
                currentNode = Pop the cheapest node from open list
                if(currentNode == targetNode)
                        return PATH_FOUND

                Add currentNode to closed list

                //Expand currentNode
                for(all neighborNodes to currentNode)
                {
                        if(neighborNode not in open list and not in closed list)
                                Push neighborNode into open list //see Note2
                        else if(neighborNode is already in open list with higher cost)
                                Replace the node in the open list with the new one
                }
        }
        return NO_PATH
}
```

Note1: Removing the **blue** pseudo-code, turns the algorithm to **Breadth-First Search** algorithm, where the open list changes to a **queue** (FIFO) data structure.

Note2: Every time we add a neighborNode to the open list, we set its actual cost to:
actualCost = currentNode->actualCost + Cost(from currentNode to neighborNode)

Statistics:

Solving the problem, the following nodes will be expanded in sequence until the goal is found:

1,2,4,3,5,7,6

As a result, we can see that this method is very slow but it was able to find the optimal solution:

1 -> 4 -> 5 -> 6 -> 8

The greedy search method (Best First Search)

The algorithm relies on future best estimations to target node.

Best estimated distance cost to City 8	
City 1	366
City 2	374
City 3	380
City 4	244
City 5	213
City 6	100
City 7	176

```
struct Node
{
	void * data;
	Node * parent;
	uint estimatedCost;
};
```

startNode = Citi 1

targetNode = Citi 8 (estimatedCost = 0)

Add startNode to the open list.

```
GreedySearch()
{
	While( open list not empty)
	{
		currentNode = Pop the cheapest node from open list
		if(currentNode == targetNode)
			return PATH_FOUND

		Add currentNode to closed list

		//Expand currentNode
		for(all neighborNodes to currentNode)
		{
			if(neighborNode not in open list and not in closed list)
				Push neighborNode into open list //see Note1
		}
```

```
        }
        return NO_PATH
}
```

Note1: Every time we add a neighborNode to the open list, we set its estimated cost to:
estimatedCost = Read from the table

Note2: The best estimated distances calculated in the table above, are, in this particular problem, the shortest distances between the cities. E.g. **374** is the distance calculated between the actual **City2** and **City8** positions on the map, which is a straight line between both cities. A straight line is the best direct estimation we can do between, in distance, to go from **City2** to **City8**.

Statistics:

Solving the problem, the following nodes will be expanded in sequence until the goal is found:

1,4,7

As a result, we can see that this method is very fast but it wasn't able to find the optimal solution:

1 -> 4 -> 7 -> 8

A* method - The combination of the greedy and the actual

The algorithm is the combination of the historical search data from the startNode and the best estimation possible to the targetNode. At every node the evaluated cost is the addition of both the actual and the estimate.

```
struct Node
{
        void * data;
        Node * parent;
        uint actualCost;
        uint totalCost;  // = actualCost + estimatedCost
};
```

startNode = Citi 1

targetNode = Citi 8 (estimatedCost = 0)

Add startNode to the open list.

```
AStar()
{
        While( open list not empty)
        {
                currentNode = Pop the cheapest node "totalCost" from open list
                if(currentNode == targetNode)
                        return PATH_FOUND

                Add currentNode to closed list

                //Expand currentNode
                for(all neighborNodes to currentNode)
                {
                        if(neighborNode not in open list and not in closed list)
                                Push neighborNode into open list //see Note1
                        else if(neighborNode is already in open list with higher cost)
                                Replace the node in the open list with the new one
                }
        }
        return NO_PATH
}
```

Note1: Every time we add a neighborNode to the open list, we set its actual and total cost to:
actualCost = currentNode->actualCost + Cost(from currentNode to neighbor)
totalCost = actualCost + estimatedCost(read from the table)

Statistics:

Solving the problem, the following nodes will be expanded in sequence until the goal is found:

1,4,7,5,6,8

As a result, we can see that this method is faster than the first one and it was able to find the optimal solution.

1 -> 4 -> 5 -> 6 -> 8

Final Thoughts

The previous algorithms are not only used for navigation systems and searching for path-findings. These algorithms can be used in any search finding for any problem with a set of states in a state space.

In this particular problem, that we saw, the state space was the map of cities, and the states were the cities themselves represented as geographical positions.

Reference

Book: Artificial Intelligence a Modern Approach, Third edition – S. Russel, P. Norvig – Chapter03.