Continuation of minimum cost walk algorithms

1. **Bellman-Ford algorithm** (also known as Ford or Bellman-Kalaba)

The idea of this algorithm is to check all the edges multiple times and for each edge to update the distance and previous of its terminal vertex, if needed. As UCS (Uniform Cost Search), this algorithm works with negative cost edges but not negative cost cycles.

```
Algorithm name: Bellman-Ford (or Ford or Bellman-Kalaba)
Input:
	g: a weighted graph (must be directed if there are negative cost edges)
	s: a starting vertex in g
Output:
	distance: a map containing the distance from s to any vertex
	previous: a map containing the previous vertex in the minimum cost walk between
		  s and any vertex
Algorithm:
	Map distance
	Map previous
	for v in graph.vertices(): #same initialization as UCS and Dijkstra
		distance[v] = infinity
		previous[v] = -1
	end_for
	distance[s] = 0
	changed = true #a flag needed to stop the algorithm if there are no more updates
	while changed do
		changed = false
		for (v1, v2) in g.get_edges():
			dist_with_v1 = distance[v1] + g.cost(v1,v2)
			if distance [v2] > dist_with_v1:
				distance[v2] = dist_with_v1
				previous[v2] = v1
				changed = true
			end_if
		end_for
	 end_while

	return distance, previous
```

The time complexity of this algorithm depends on the complexity of get_edges and on how many iterations are needed to finish the algorithm. The way this algorithm works, after iteration *k* we have the minimum cost walks of length at most *k*. So, after the first iteration we have the minimum cost walk of length 1, after the second the walks of length 2 and so on. Now, if we want the minimum cost walk, it doesn't make sense to repeat edges or vertices (unless we have negative cost cycles, in which case this algorithm does not work) so the minimum cost walk is also a path. In any graph n-1 is the highest length possible for a path (it includes all n vertices, and there are n-1 edges between n vertices). So, we know that at each iteration we get the minimum cost walk of length *k* but the longest possible minimum cost walk is n-1. Therefore the algorithms runs at most n-1 iterations. How much time does each iteration take? Well, if implemented on a list of neighbors, the get_edges function should take

Theta(m) (and actually iterating over m edges should take m steps). So, in total, a while iteration takes Theta(m) and there are n-1 while iterations at most so the algorithm runs in O(m*n).

What happens if there are negative cost edges? The algorithm gets stuck in an infinite loop. But we know that there should be at most n-1 iterations. So we can set a counter and if there are more than n-1 iterations then we detected a negative cost cycle.

As with UCS, if there are no negative cost edges, the algorithm could be stopped early once the distance for t is less than infinity, if we do not care about the walk to other vertices.

2. Floyd-Warshall algorithm

This algorithm is made to find the minimum cost walks between ALL pairs of vertices. That is, not just between the start vertex and all other vertices (as Dijkstra, UCS or Bellman-Ford), but between ALL pairs (i.e. if we consider all vertices start positions).

What this algorithm returns is something similar to the distance and previous map of Dijkstra/UCS/Bellman-Ford, but for all vertices – i.e. a matrix. It returns the Distance matrix, where at point i,j is the distance vrom vertex i to vertex j. And returns the Next matrix, where at point i,j is the next vertex in the walk from i to j (this is the reverse of how the previous map worked, it returns the next in the walk from i to j, not the previous in the walk from s to i).

```
Algorithm name: Floyd-Warshall
Input:
      g: a weighted graph (must be directed if there are negative cost edges)
Output:
      Distance: a matrix, where at position i,j the distance from i to j
      Next: a matrix, where at position i,j the next vertex in the walk from i to j
Algorithm:
      Matrix[n by n] Distance
      Matrix[n by n] Previous
      #First initialize all distances with infinity except where there are edges,
       there we initialize with the cost of the edge
      for i=0 to n-1:
          for j=0 to n-1:
              if i==j:
                  Distance[i,j] = 0
              else if g.is_edge(i,j):
                  Distance [i,j] = g.cost(i,j)
                   Next[i,j] = j
             else
                  Distance[i,j] = infinity
              end_if
          end_for
      end_for

      #For each vertex we check if that vertex would be better in the walk between all
       other pair of vertices
      for k=0 to n-1:
          for i=0 to n-1:
```

```
            for j=0 to n-1:
                 distanke_with_k = Distance[i,k] + Distance [k,j]
                if Distance[i,j] > distanke_with_k:
                    Distance [i,j] = distanke_with_k
                   Next[i,j] = Next[i,k]
                end_if
            end_for
        end_for
    end_for
```

Time complexity of hte algorithm is straightforward: there are 3 nested for loops each running for n iterations => Time complexity of Theta(n^3)

3. Best First Search (BeFS)

Best First Search (BeFS to make it different from BFS – Breadth First Search) is a *class* of searching algorithms. That means that it is a generic algorithm that is further specialized when implemented concretely, and, when implemented, it becomes several others specific algorithms. Or, viewed in another way, it is a template to be used to implement some type of searching algorithms.

Best First Search can be generalized for use in different problems, but we will only consider it for minimum cost walk search. As said in the previous lecture, UCS is a Best First Search algorithm. The BeFS algorithm for minimum cost walks is given in the following:

```
Algorithm name: Best First Search (BeFS)
Input:
      g: a weighted graph (must be directed if there are negative cost edges)
      s: a starting vertex in g
Output:
      distance: a map containing the distance from s to any vertex
      previous: a map containing the previous vertex in the minimum cost walk between
s and any vertex
Algorithm:
      Map distance
      Map previous
      PriorityQueue pq
      for v in graph.vertices(): #same initialization
            distance[v] = infinity
            previous[v] = -1
      end_for
      distance[s] = 0
      pq.push(s)
      while pq is not empty:
            current = pq.pop()
            for n in g.get_neighbors(current):
                  dist_with_current = distance[current] + g.cost(current, n)
                  if distance[n] < dist_with_current:
                        distance[n] = dist_with_current
                        previous[n] = current
                        pq.push(n)
                  end_if
            end_for
```

```
    end_while

    return distance, previous
```

As you can see this algorithm is identical to UCS. So, what gives? Well, Best First Search is a *class* of algorithms in the sense that the above is the algorithm but what *best* means is up to you. In other words different BeFS algorithms use different priorities. UCS uses the current distance of a vertex as the priority.

It is important to note that this is not written in stone. In graph problems (and many algorithms in general) the algorithms used tend to have different implementations, additions or things left out depending on what problem needs to be solved and how to best optimize it. The simple example is the early exit if we want only the s-t walk, and we know that there are only positive cost edges. Depending on what we want to do, what priority we use and so on things can change on how the algorithm is implemented and used.

4. Heuristics, A* and Greedy search

In computer science, and especially in the Artificial Intelligence domain, heuristics are heavily used to improve algorithms.

What are heuristics? It is a term used in many domains, from psychology to economics to computer science. In general an heuristic is a kind of function or rule that gives a good estimation of something that would be impractical to find exactly, or would take much longer to compute exactly. For example, any "rule of thumb" is a kind of heuristic.

In computer science, heuristics usually refer to rules based on *domain knowledge*, that allow for a much more efficient implementation of an algorithm (possibly subject to some limitations or constraints or loss of guarantees). It is important that heuristics are based on *domain knowledge.* Domain knowledge is information about the domain of the problem you are trying to solve, for example if you know that the problem is solving something in electrical engineering or construction or planning etc., you might know that in those cases some special rules/constraints/hijinks might apply. This information might let you implement a more efficient algorithm either by disregarding some cases that would make the algorithm harder otherwise, or by being able to introduce some shortcuts because you have extra information.

The easiest example for heuristics is in the domain of pathfinding. In pathfinding you know that you are trying to find a path between some points in some space. So, you can estimate the length of a path between 2 points with the Euclidian distance (or some other distance metric). This is how heuristics can help algorithms. If in our weighted graph we know that the vertices represent locations in some space, we could compute the Euclidean distance between vertices and that would give us a hint on what vertices are actually close together and what vertices are far away. This kind of information is heavily used in path finding algorithm (such as google maps searching for the best route to follow.

Going back to Best First Search, other types of BeFS algorithms (other than UCS) are based on heuristics. Two main ones are A* and Gredy Search.

A*

A* is the most efficient pathfinding algorithm. While it is heavily used in pathfinding it is a general algorithm. It is a BeFS where the priority is **distance + heuristic.** In pathfinding, the heuristic is usually the Euclidian distance or Manhattan distance. More exactly, when computing the priority for the A* algorithm for vertex v when searching a walk from s to t:

$$p(v) = d(s,v) + h(v,t)$$

where p(v) is the priority of the vertex v, d(s,v) is the distance from the start vertex s to vertex v and h(v, t) is the heuristic value that estimates the distance between vertex v and the terminal vertex t.

It is important to keep in mind that not all heuristics work for A*. A* is guaranteed to give the optimal solution only if, for all vertices *v*, we have $h(v) \leq d(v, t)$ (in other words, the estimation is always an underestimation).

Greedy search

Greedy search is a BeFS where the priority is the heuristic value. Practically p(v) = h(v,t) – from the previous equation, just without the d(s,v) part.

Greedy search is an algorithm that does **not** give the optimal solution. It is intended to give a very fast solution (basically it is Theta(length_of_walk*m/n)) but it might not give best solution, maybe not a good solution. The usefulness of Greedy search is entirely dependent on how good the heuristic is for the problem.

5. Introduction to subgraph search

Many problems in graph theory revolve around finding a subgraph with specific properties (a practical example can be finding communities in a graph representing social connections). Most often we want an induced subgraph, and finding such a subgraph is equivalent to finding a subset of vertices. In the following are presented 2 types of subsets of vertices that are often searched for in graphs.

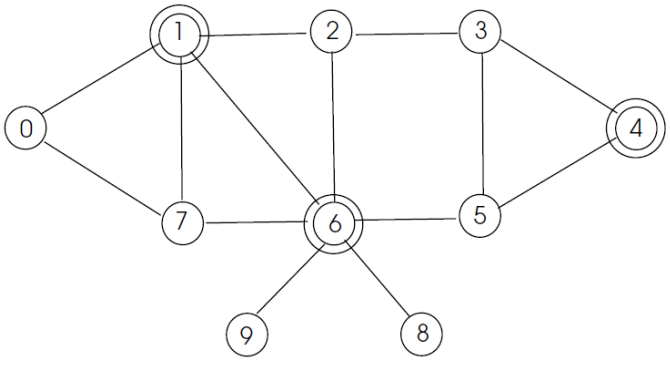

Figure 1

Independent sets

An **independent set** is a set of vertices in a graph, where no 2 vertices are adjacent.
Let $G = (V, E)$ be a graph. $M \subseteq V$ is an independent set of G if $\forall v_1, v_2 \in M, (v_1, v_2) \notin E$.

In graph theory the problem is to find a maximum independent set – an independent set with the greatest possible size. **Independence number** is the size of the largest possible independent set in the graph.

A trivial independent set is a set with only one vertex.

Considering the graph in Figure 1, a maximum independent set is $\{1, 6, 4\}$. The independence number of the set is 5. There is not only one maximum independent set, another example might be $\{9,8,5,7,2\}$.

Dominating set

A **dominating set** of a graph G is a subset of its vertices such that any vertex of G is either in this set or adjacent to a vertex in this set.
Let $G = (V, E)$ be a graph. $M \subseteq V$ is a dominating set of G if $\forall v_1 \in V$, either $v_1 \in M$, or $\exists v_2 \in M$ such that v_1 and v_2 are adjacent.
In graph theory the problem is to find a minimum dominating set – a dominating set with the smallest possible size. **Domination number** is the size of the smallest possible dominating set.

A trivial dominating set is the set containing all vertices.

Considering the graph in Figure 1, a minimum dominating set is $\{0,4,2,8,9\}$. The dominance number of the set is 3. There is not only one minimum dominating set, another example might be $\{3,6,0\}$.