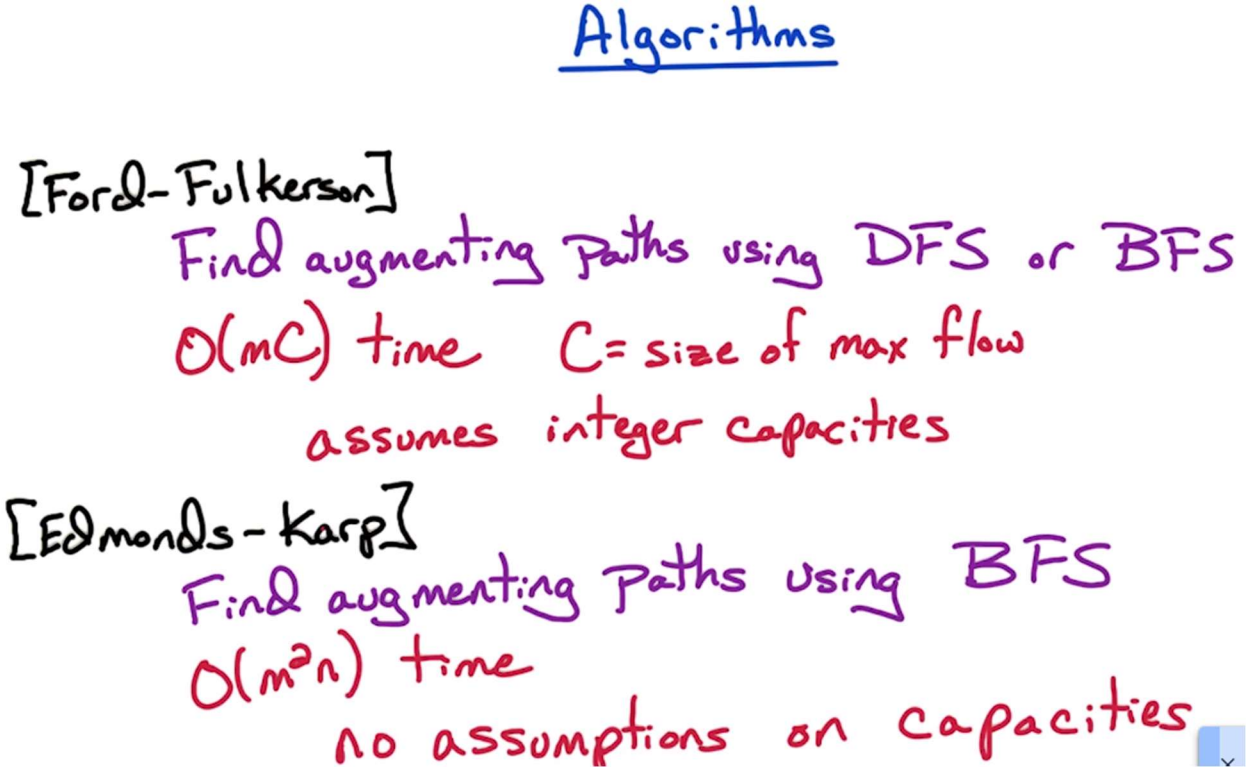
LESSON: MF4: Edmonds-Karp Algorithm

***

(Slide 1) Max-Flow Min-Cut Algorithms



We've seen the Ford-Fulkerson algorithm for solving the max-flow problem. And in this lecture, we'll look at the Edmonds-Karp algorithm for solving the max-flow problem. Before we dive into the Edmonds-Karp algorithm, let's do a quick contrast of the two algorithms.

Now, in the Ford-Fulkerson algorithm, the main step is to find augmenting paths which are paths from s to t in the residual graph and to do that, we either use DFS or BFS. Now the running time of Ford-Fulkerson algorithm is oO(mC), and recall, C is the size of the max-flow. And recall that this running time analysis assumes that the capacities are integer value.

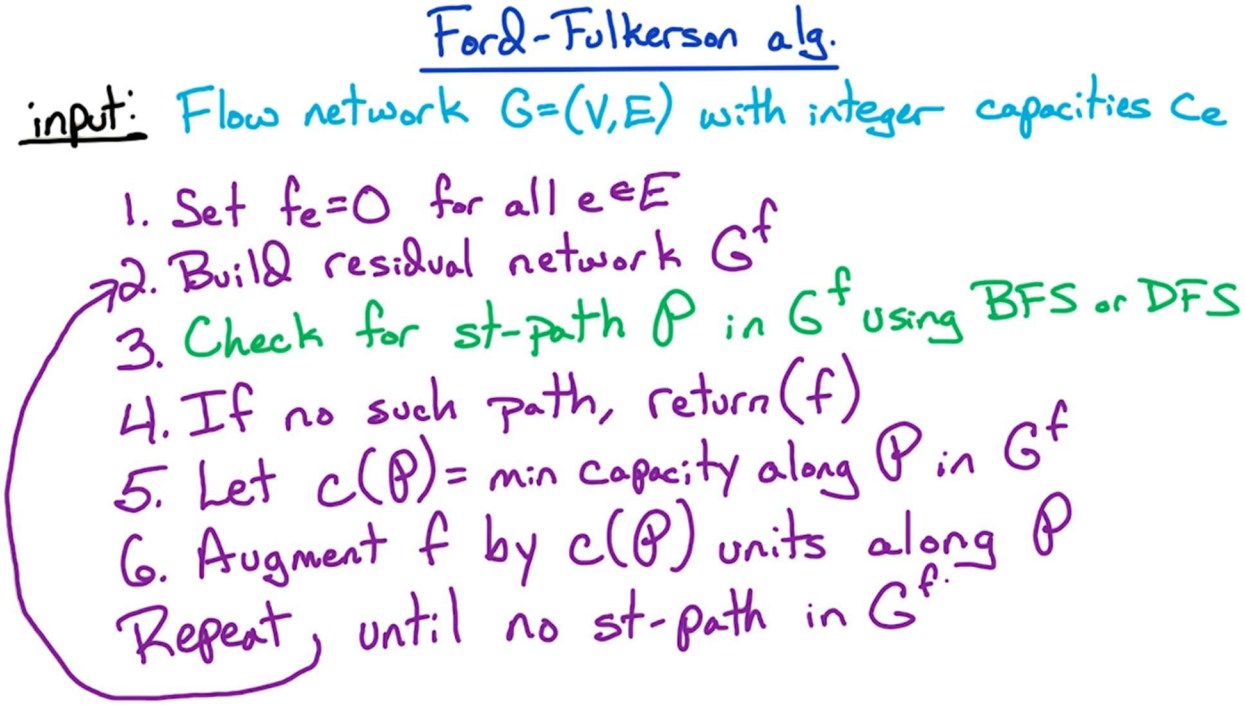
Now. in contrast, Edmonds-Karp finds augmenting paths using BFS. So, Edmonds-Karp algorithm is an example of a Ford-Fulkerson algorithm. The running time analysis for Ford-Fulkerson algorithms still applies to Edmonds-Karp algorithm where we assume integer capacities? But, in fact, we can often make a stronger guarantee on the running time - we'll prove that the running time is $O(m^2 n)$ – and, this doesn't require that the capacities are integer values. We have no assumptions on the capacities other than, of course, that they're positive valued.

Now, the algorithms are quite similar, so we'll give a quick recap with a Ford-Fulkerson

algorithm and then we'll point out the differences in the Edmonds-Karp algorithm. And then afterwards, we'll dive into the running time analysis for the Edmonds-Karp algorithm.

***

(Slide 2) Ford-Fulkerson Algorithm



Let's start with the Ford-Fulkerson algorithm. Recall the input to the Ford-Fulkerson algorithm is a flow network which is a directed graph with capacities along the edges specified by c_e. And, for the Ford-Fulkerson algorithm, we assume that these capacities are integer values.

Ford-Fulkerson alg:
Input: Flow network G={V,E} with integer capacties c_e.

- We start off by setting the flow to 0 along every edge of the graph.
- Next, we build the residual network for the current flow f. We denote this residual network by G^f.
- Now, we look for an augmenting path in the residual network. More precisely, we check for a path from s -> t in the residual network. And we do this using either BFS or DFS.
- If such a path exists, that's denoted by P. If no such path from s -> t in the residual network exists, then we return the current flow.

  Note: We'll prove later that if there is no augmenting path, no path from s -> t in the residual network, then the current flow is guaranteed to be a maximum size

flow.

- Now if there is such a path, then we want to augment along this path as much as possible. Therefore, we let c(P) be the capacity of this path - more specifically, it's the minimum, over the edges of this path, of the capacity of these edges in the residual network - this is the maximum amount that we can augment this path.

  Finally, we augment the current flow by c(P) units along this path,

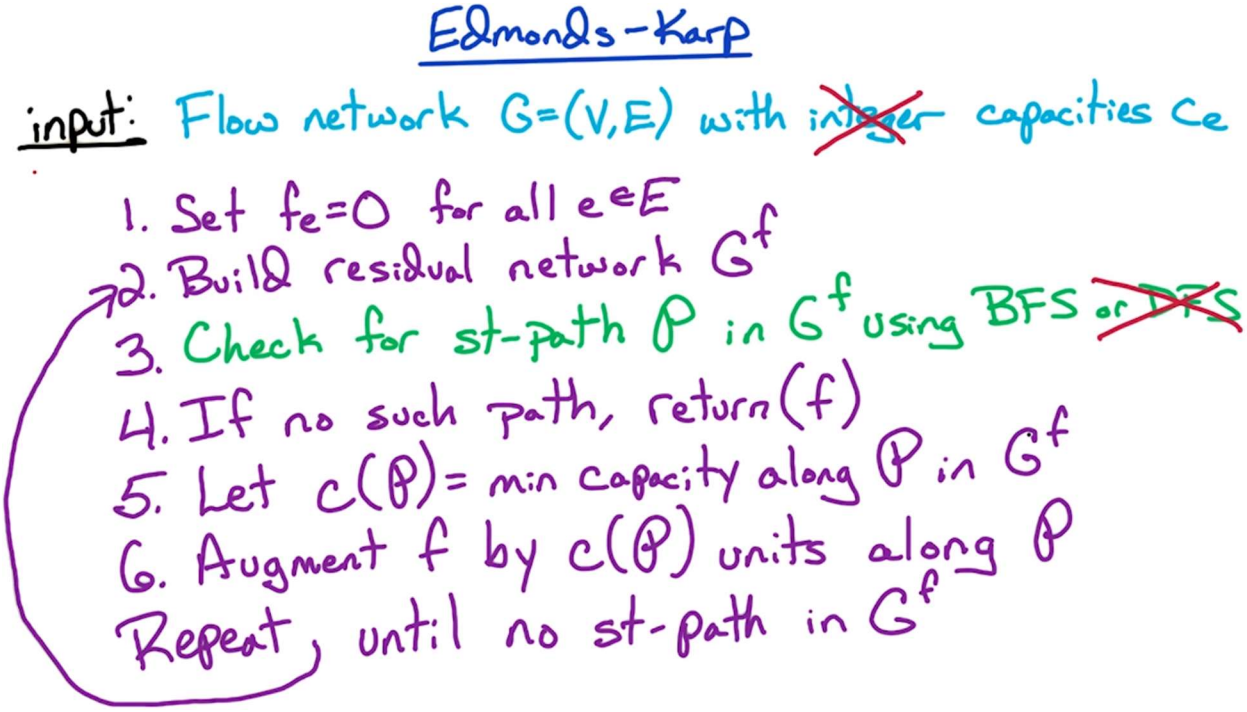
- and then we repeat the algorithm. We use the current flow and we build a new residual network, and we check for a path and continue.

  … And, we stop when there's no st-path in the residual network.

This completes the description of the Ford-Fulkerson algorithm. Let's look now at the Edmonds-Karp algorithm.

***

(Slide 3) Edmonds-Karp Algorithm



In the Edmonds-Karp algorithm, there's only one distinction. When we check for an st-path in the residual network, we have to use BFS. We cannot use DFS. The running time analysis assumes that we're using BFS to find this path. That's the only difference between Edmonds-Karp and Ford-Fulkerson algorithm.

Now, our running time analysis will not assume that the capacities are integer values, so we can remove this assumption. All that's required is that the capacities are positive.

Before we dive into the running time analysis, let's look at one basic property of the Edmonds-Karp algorithm and also the Ford-Fulkerson algorithm: Notice that the residual network has to change by at least one edge in every round. In particular, at least one edge reaches full capacity. We augment along P until we reach the full capacity of at least one edge on that path.

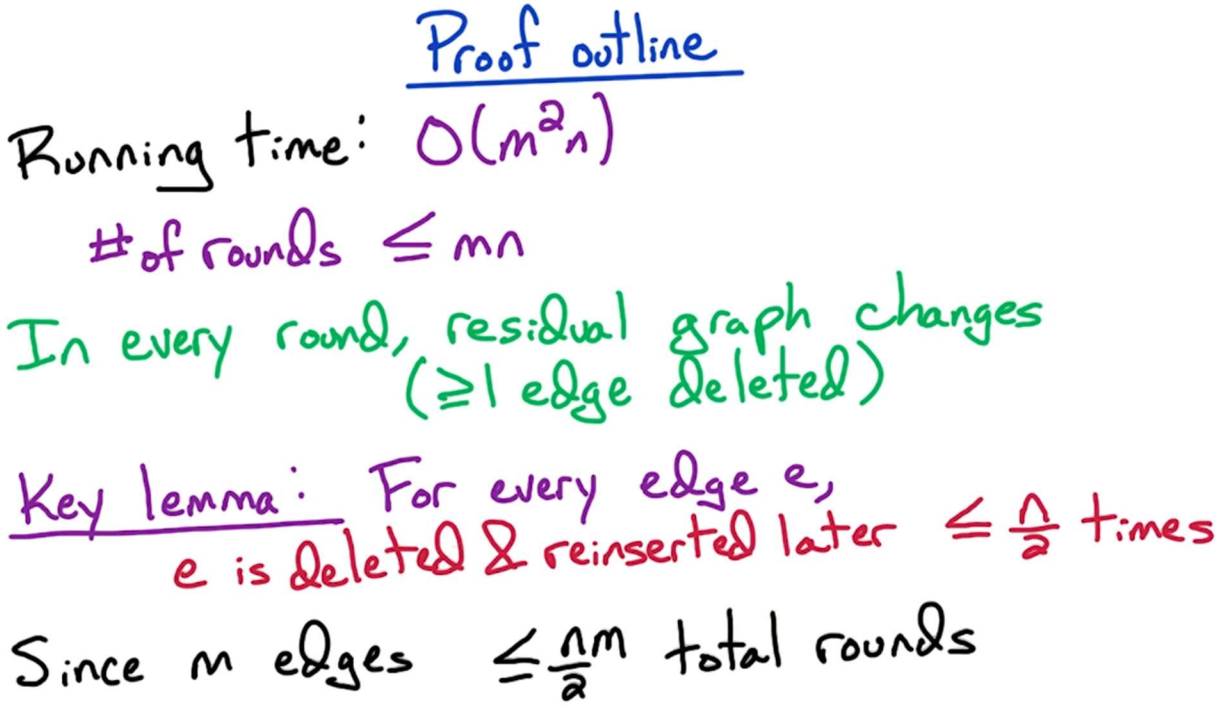
Now, that edge, which reaches its full capacity in the residual graph, will be removed from the residual graph in the next stage. That edge might be a forward edge or a backward edge, but, regardless, it will be removed from the residual graph in the next stage.

There may be additional edges which are removed and there might be other edges which are added back into the residual network, and we're going to need some understanding of which

edges are added in or removed from the residual network. But one key property is that at least one edge is removed in every stage.

***

(Slide 4) Proof Outline



Now, we're going to prove that the running time of Edmonds-Karp algorithm is $O(m^2 n)$. To prove that, we're going to prove that the number of rounds of the algorithm - the number of times we augment the path - is at most m x n.

Now, in each round, we run BFS. BFS takes linear time. So assuming the number of edges is at least the number of vertices, that's O(m) time. Therefore, it's O(m) time per round and O(mn) rounds. So. the total run time will be $O(m^2 n)$. So, our main task is to prove that the number of rounds is at most mn.

As we just discussed, in every round, the residual graph changes from the previous round. In particular, at least one edge is deleted from the residual graph in every round. This edge corresponds to the one with minimum capacity along the augmenting path.

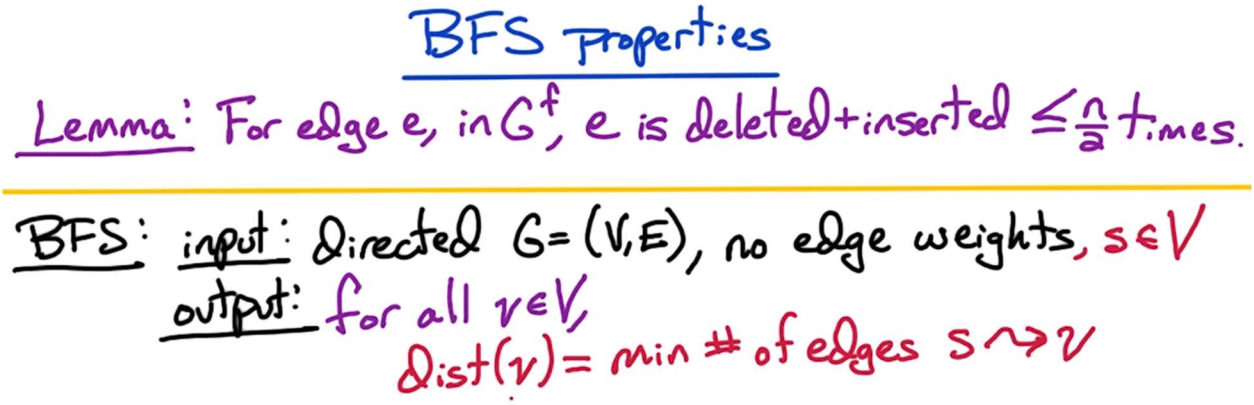
Now, edges make it add it back in in the residual graph. And in particular, this edge might be deleted in this round and in later rounds might be added back in. Now what we're going to prove is that for every edge of the graph, we can bound the number of times that the edge is deleted in the residual graph and then reinsert it back later. In particular, we're going to show that the number of times this edge e is deleted from the residual graph, and then later reinserted into the residual graph, is at most n/2 times.

Now. since there are m edges in the graph (and this holds for every edge of the graph), we know that every edge is deleted at most n/2 times - at least one edge is deleted in every round. Therefore, there are at most n/2 x m total rounds. This gives our desired bound on the number of rounds of the algorithm which proves the desired running time.

So let's dive into the proof of this key lemma.

***

(Slide 5) BFS Properties



Now we want to prove the key lemma, from the previous slide, which states that, for every edge of the graph in the residual graph, this edge is deleted and then reinserted later at most n/2 times.

So this deletion / insertion process can occur at most n/2_times. Now, of course, to prove this lemma, we're going to have to use properties of the BFS. Let's recall the setting for BFS, breadth-first search:

- The input is a directed or undirected graph G (we're going to be working with directed graphs).

- Our directed graph, the residual network, we'll have edge weights. But, in fact, BFS doesn't pay attention to edge weights.

- BFS also specifies as a start vertex s.

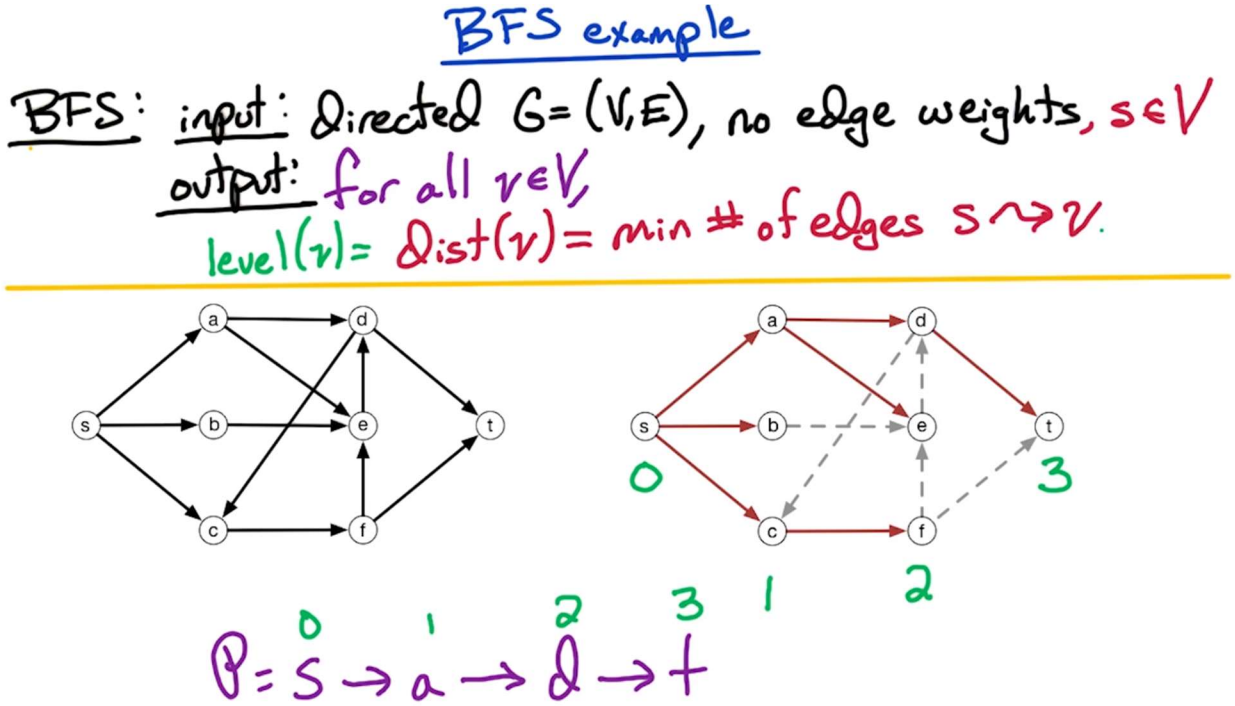
- Now what is the output of the breadth-first search algorithm? Well, for every vertex of the graph, so it outputs an array of size n, and dist(v) is defined as the minimum number of edges to go from s to v.

  Notice, dist just counts the number of edges, it pays no attention to the edge weights. If we want to pay attention to the edge weights, then we have to use an algorithm such as Dijkstra's algorithm.

- Now in addition to computing this distance for every vertex, it also finds such a path from s to v which uses the minimum number of edges.

***

(Slide 6) BFS Example



Let's take a look at BFS on one of our earlier examples.

Here's one of our examples of a flow network. Let's assume that we're at the initial flow. So the flow is zero along every edge of the graph. Therefore, the capacities along these edges don't really matter. Now, since the flow is 0 along every edge, there's no back edges, and the residual network at this stage will be exactly the same flow network.

Now, let's run BFS on this residual network:

- We start off from vertex s, and we explore the edges out of s, and we see a, b and c.

- Next, we explore the edges out of a, b and c.
- From vertex a, we see d and e.
- From vertex b, we see e, which is already explored. So this is an unexplored edge.
- Then from vertex c, we see vertex f.

  So the solid red edges are the tree edges, the BFS tree edges, and the dash gray edges are the non-explored edges.

- Now we explore the edges out of d, e and f.
- From vertex d we see vertex t, our end vertex, and then we have a bunch of unexplored edges.

Now at this stage, BFS is done.

Now, let's look at some properties of BFS:

- Notice this is exploring the graph in layers.

  Now, in the first layer or level is only vertex s itself. So this is at level 0. Now then, the neighbors of s are at level 1. So a, b and c are level 1. Then we have that d, e, f are at level 2. And finally, vertex t is at level 3.

  Now the level of vertex is the same as the minimum number of edges to go from s to that vertex. So formerly, the level of vertex is the same as the distance.

- Now, what is the path P that BFS finds from s => t? - well, it's this path, s -> a -> d -> t.

  Notice, on this path from s to t, look at the levels. s is at level 0, a is at level 1, d is at level 2, and t is at level 3. So, the level goes up by +1 at every edge.

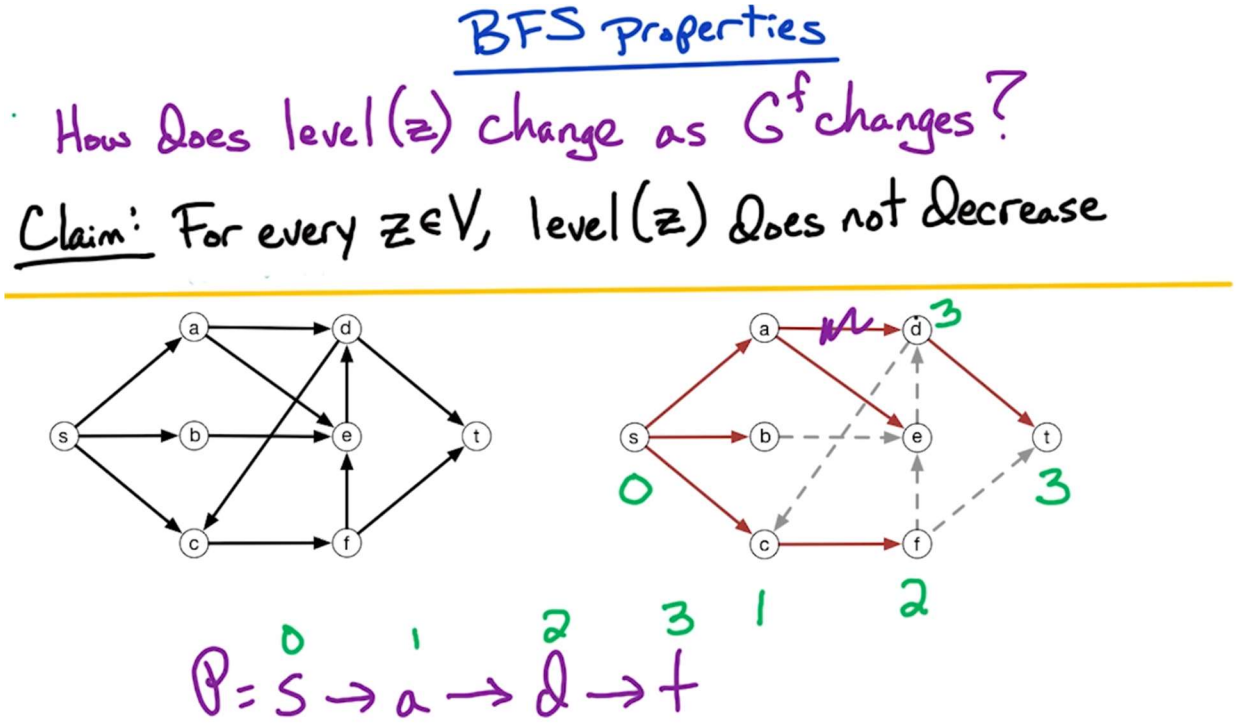
  Now, notice that the level cannot go up by more than one in one edge, because of the definition of the level - It's the minimum number of edges from s to that vertex.

  Similarly, the level cannot stay the same or go down along an edge - otherwise, there's a shorter path to the vertex.

  So the path that BFS finds from s to t, is going to be a path where the level goes up by plus one at every edge. That's the key property that we need from BFS, that the path obtained from s to t, the levels of the vertices go up by plus 1 along every edge of their path.

***

(Slide 7) BFS Properties – Part 2



Now, to prove our main lemma about the number of rounds of the Edmonds-Karp algorithm, we're going to analyze the level of a vertex during the course of the Edmonds-Karp algorithm. In particular, we want to look at a vertex z, and we want to look at how its level changes as the residual network changes.

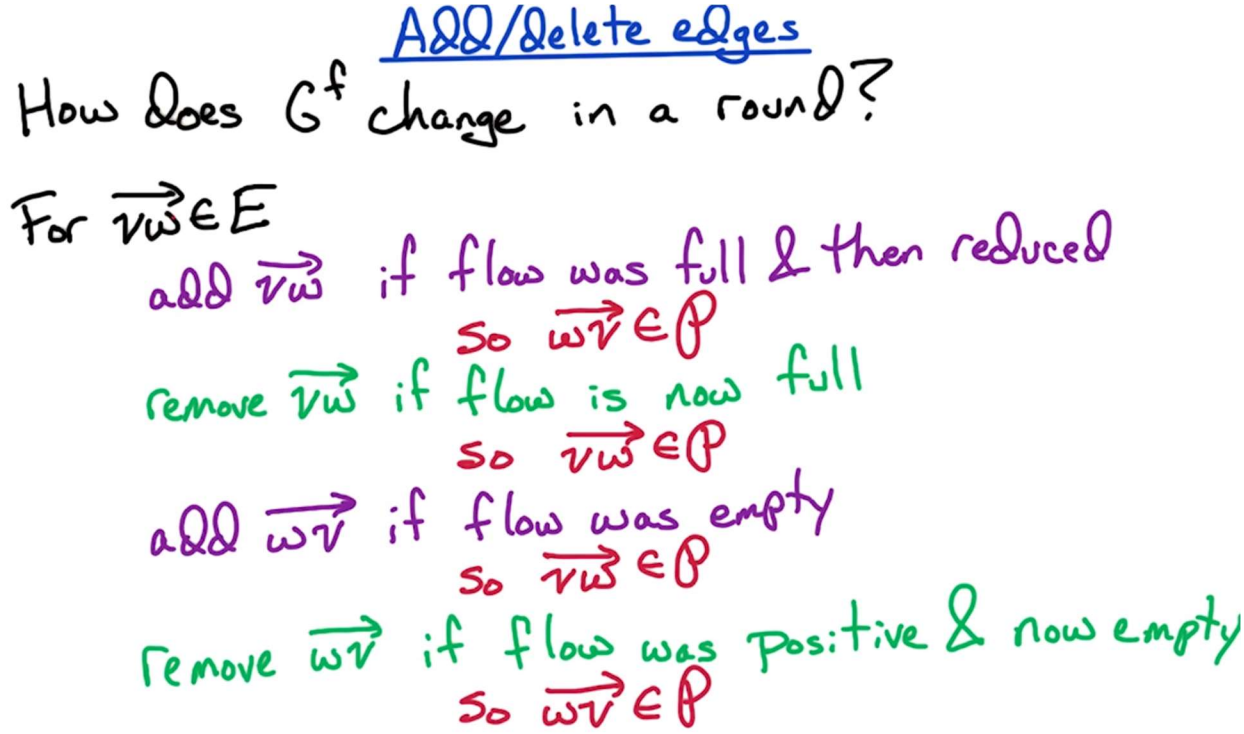
Now, what can happen in the residual network? - well, edges can get deleted. For instance, suppose we delete this edge from a -> d, then what happens to the level of d? - well, the minimum number of edges to get from s => to d is 1, 2, 3. So, d will be at level 3. So, the level of d will increase when we delete this edge from a -> D.

Now, we're also going to add edges into the residual network. For instance, what if we add an edge from b to d? Or maybe we'll even add this edge back from a -> D, then will the level of D decrease back to level 2? = well, we're going to prove that this does not occur. Even though we're adding edges into the residual network (sometimes), the level of vertex is never going to decrease. We're going to prove, that for every vertex Z, it's level over the course of the algorithm does not decrease. It can stay the same or can increase in some levels. But, if for instance, this vertex d – if its level increases to 3, then it can never go back down to 2. It can only stay the same or increase further.

Now, to prove this claim, we have to understand which edges are added into the residual network, and which edges are removed from the residual network. So, let's look at that in more detail.

***

(Slide 8) Add/Delete Edges



Let's take a look in detail how the residual graph changes potentially in a round.

Now, consider an edge of the original flow network. This edge itself might be in the residual network or its back edge from w -> v might be in the residual network. Let's look at when the forward or backward edge might be added or removed from the residual network.

- When do we add this forward edge to the residual network? Well, in order to not be there before, the flow had to be full. So if the flow was full, it means the flow equaled the capacity along that edge, and then the flow was reduced. Then this edge would have leftover capacity. So it would be added into the residual network.

  Now, how can the flow be reduced along this edge? That means we send flow along the back edge. To send flow along the back edge, that means the back edge was in the augmenting path P. So, if this forward edge was added into the residual network, then the back edge must have been on the augmenting path. Now, let's look at the other case where we remove this forward edge from the residual network.

- Now, we will remove it if there was some leftover capacity, but, now, there's no leftover capacity. So, the flow equals the capacity along this edge - that means we augmented the flow, increased the flow along this edge. So, this forward edge itself is on the

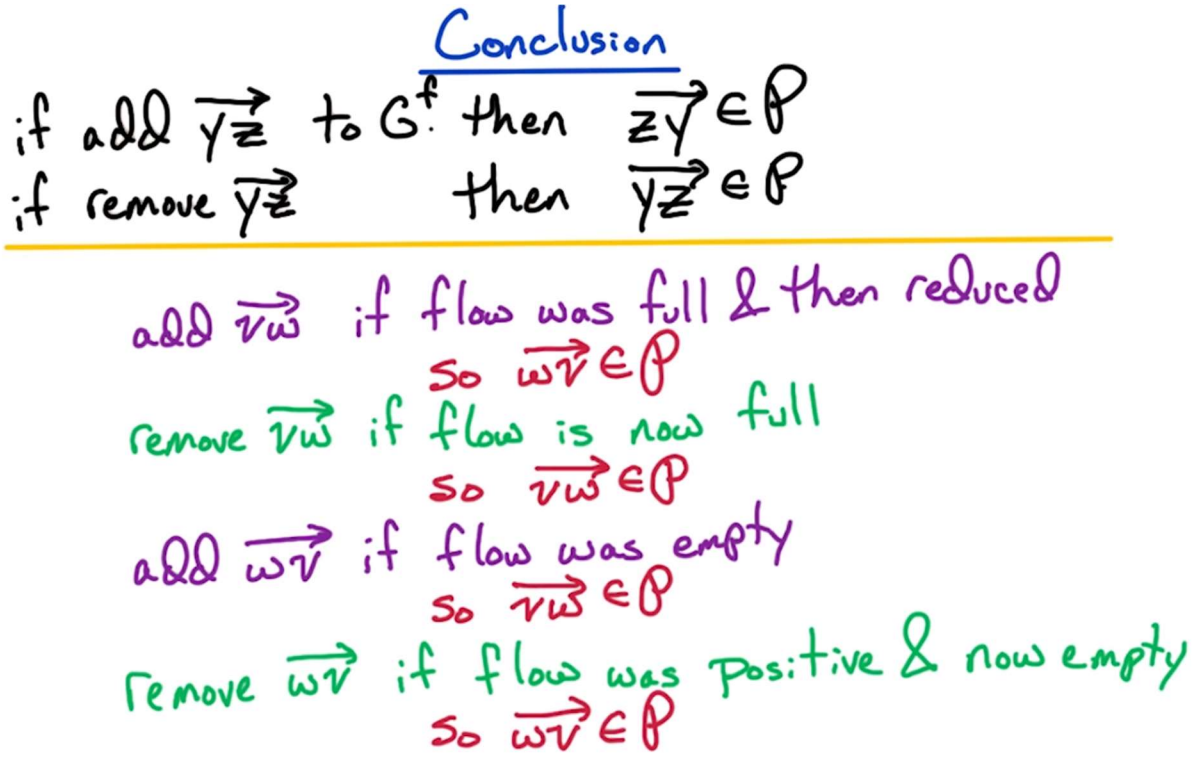
augmenting path. In order to remove this forward edge from v -> w from the residual graph, this forward edge from v -> w must be on the augmenting path because we increased the flow along this edge.

- Now, the other type of edge that we can add into the residual network is the back edge from w -> v. To add the back edge in, the flow must have been empty and then we increase the flow along the edge from v -> w. To increase the flow from v -> w, this edge from v -> w must be on the augmenting path.

- The final case is when we will remove the back edge from the residual network. Now, in order to remove this back edge from the residual network, there must have been some flow along the forward edge, and then the flow is now empty. So we must have decreased the flow along this forward edge. To decrease the flow along the forward edge, that means we send flow along the back edge. So the back edge from w -> v must be on the augmenting path P.

***

(Slide 9) Conclusion



Here are the key conclusions that we need from this analysis.

- If we add an edge from y -> z to the residual network, then what can we conclude? - well that's this case, and this case. Notice that in both of these cases, the opposite direction edge is always on the augmenting path. So if we add this edge from y -> z to the residual network, then we know that the edge from z -> y is on the augmenting path.

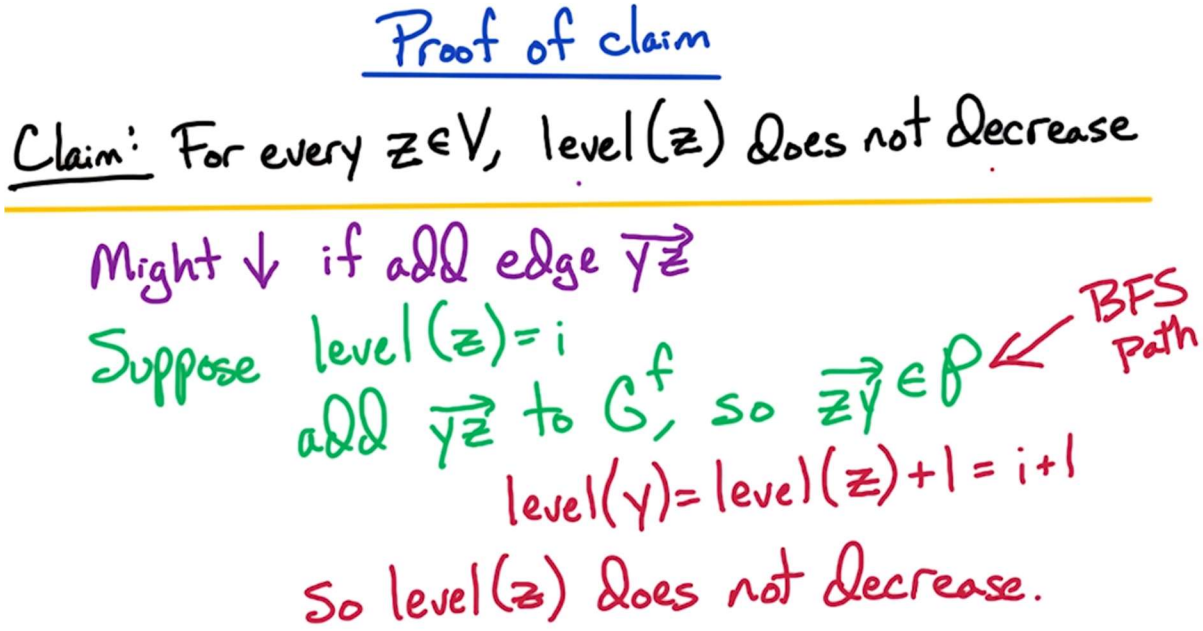
- Similarly, if we remove the edge from y -> z from the residual graph, then what can we conclude in this case? Well, when we remove, then we notice that the edge itself is on the augmenting path. So in this case, we know that the edge from y -> z is on the augmenting path. So if we add an edge y -> z to the residual graph, then the opposite direction edge is on the augmenting path. If we remove an edge, then the edge itself is on the augmenting path.

  This should make intuitive sense, because to remove an edge from the residual graph, that means we remove the leftover capacity. In order to remove the leftover capacity, that means we have to increase the flow along this edge. To increase the flow along the edge, the edge has to be on the augmenting path in order to augment along it). To add an edge to the residual network, we have to increase the spare capacity. That means we

have to decrease the flow along this edge. To decrease the flow along an edge, then the reverse direction edge must be augmented. So it must be on the augmenting path.

***

(Slide 10) Proof of Claim (Edmonds-Karp Algorithm)



Now, let's prove the main claim that we made earlier. For every vertex Z, the level of that vertex does not decrease during the course of the Edmonds-Karp algorithm.

Now, the residual graph may change. We may add edges or delete edges from the residual graph, but the level of the vertex never decreases. How potentially can the level of a vertex decrease? - while the level of a vertex might decrease, if we add an edge from y -> z and y gives a shorter path to z - now, suddenly removing edges can decrease the level of a vertex so we can ignore removing edges. We simply have to look at adding edges and their potential effect.

So suppose the level of vertex z is currently I (level(z) = i) and then suppose that we add this edge from y to z into the residual network. Now, if we add this edge from y -> z to the residual network, what do we know? - well, as we just saw, then the reverse direction edge must be on the augmenting path.

Now, we have to utilize the properties of the Edmonds-Karp algorithm. Edmonds-Karp algorithm says this path, this augmenting path is a BFS path. So, we obtained this path by running BFS.

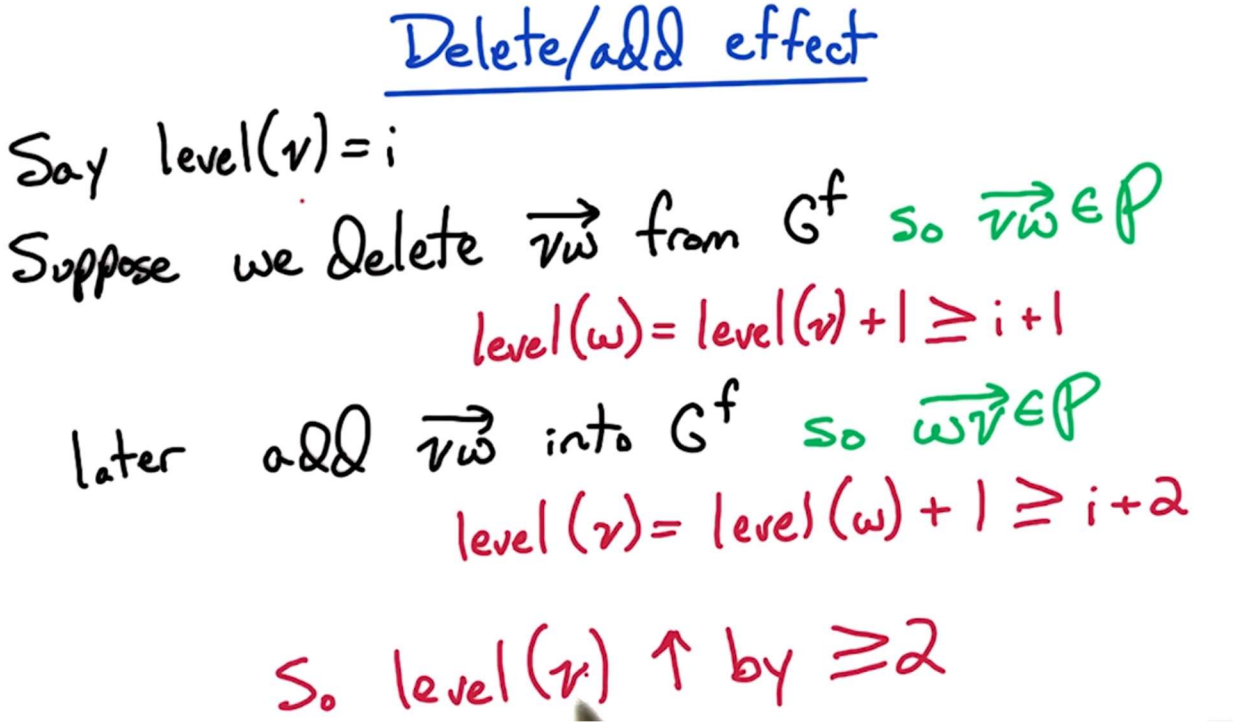
Now what do we know about paths from BFS? We know that on this path, the level of the vertices increases by +1 along every edge. Therefore, the level(y) = level(z) + 1. We said the level of Z is i so therefore the level(y) = i + 1. So, we added this edge from y -> z but this edge

goes from a higher level to a lower level. So it certainly does not decrease the level of vertex z - It does not give a shorter path to z. So the level of the vertex z does not decrease when we add this edge in. Now, this completes the proof of this claim.

So we proved that the level of a vertex never decreases during the course of the Edmonds-Karp algorithm. This completes the proof of the claim; but, in order to bound the number of rounds of the algorithm, we're going to have to prove something stronger. We're going to have to prove that, occasionally, the level of a vertex strictly increases and then this will give us a bound on the number of rounds of the algorithm.

***

(Slide 11) Delete/Add Effect



Now, our main lemma bounds the number of times that we can delete an edge and then add it back in. Let's look at the effect of deleting an edge and adding it back in on the level of the vertices.

Let's take a particular vertex v and let's say the current level(v) = i. Now suppose at some later round we delete this edge from v -> w from the residual network. What can we conclude about the level(w) when we delete this edge from v -> w? - now, if we were deleting this edge from v -> w from the residual network G^f, what do we know about the augmenting path? - we know this edge itself from v -> w is on the augmenting path P. That means the level(w) = level(v) + 1.

Now, at the time that we deleted this edge, level(v) might have changed from the earlier time, so it might not be level i anymore. But, what do we know about level(v)? We just proved that the level(v) never decreases. So, this level(v), at this current time, must be at least i. So level(v) + 1 >= i+1. So level(w) >= i+1.

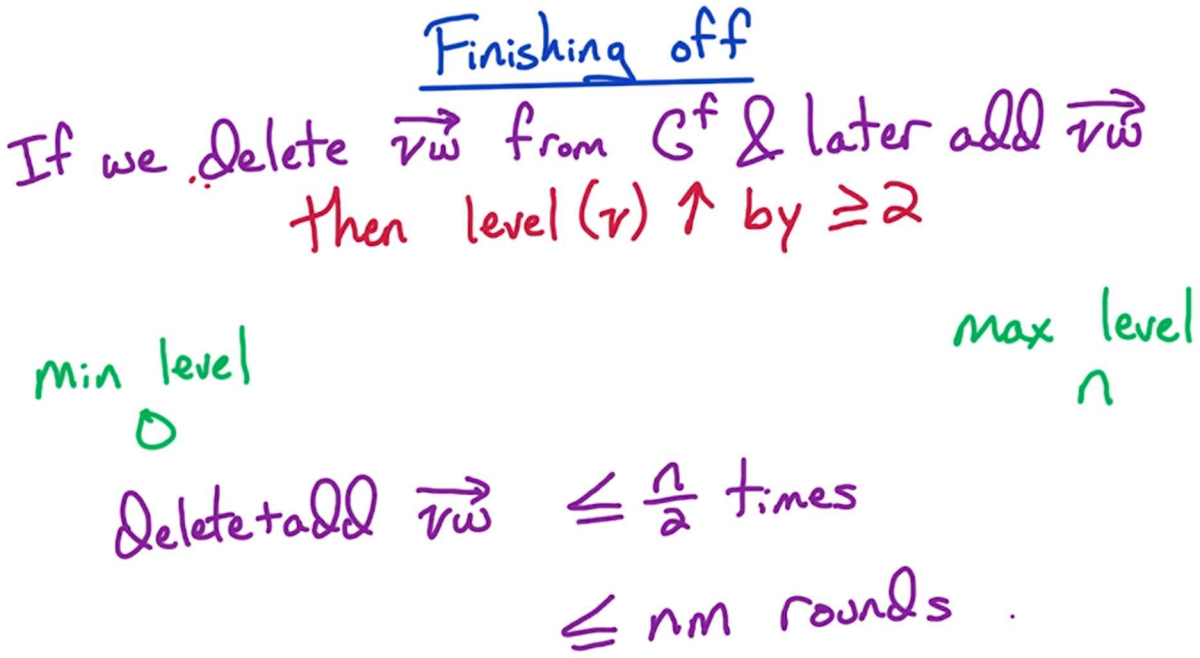
Now suppose later we add this edge from v -> w back into the residual network G^f. Now if we're adding this edge from v -> w into the residual network, then we know that the reverse direction edge, the edge from w -> v must be on the augmenting path. Therefore, at this current time, the level(v) = level(w) + 1. By our claim, we know that the level(w) did not decrease from earlier times and we earlier showed that the level(w) was at level at least i + 1. So this is that

level at least i+2: level(v) = level(w) + 1 >= i + 2.

Notice what we just proved. Suppose the level(v) is i, now we take an edge going out from vertex v, and we delete that edge and later we added it back in. After we added this edge back in, we know that the level of the vertex is at least two greater than what it previously was. So the level of this vertex increased by at least two.

***

(Slide 12) Finishing Off



We just showed that if we delete an edge v -> w from the residual network G^f and later we add this edge back into the residual network, then the level(v) will increase >= 2.

If we look at the level of vertex v before this deletion compared to after the insertion, the level of the vertex increase by at least two. Now, what do we know about the level of vertices? The minimum level is 0, that's for vertex s itself. The maximum level is n. Therefore, what's the maximum number of times that we can delete and then add this edge back into the residual network? We can delete it and then reinsert it back in at most n/2 times.

Since there m edges in the graph, this proves our main result that there is are most nm rounds in the Edmonds-Karp algorithm. And that completes the analysis of the Edmonds-Karp algorithm.