Graph Algorithms: Network Flows

This visualization is incomplete; it shows just Dinitz algorithm, and even this part is not fully finished.

See Appendix to learn how to manipulate with certain controls in the control bar. Read also the definition in the section "Excess Passing".

Introduction

The initial scene is just an illustrution, proceed to the next one.

Greedy Method

The section not yet implemented.

Excess Passing

The section not yet implemented. However, read the following text:

Usually, augmenting path algorithms modify the flow by increasing or decreasing the amount of flow in particular edges of the network:

if there is a sequence of nodes source $= v_0, v_1, \ldots, v_k =$ sink such that for each $i = 1, \ldots, k$

- either $e_i = (v_{i-1}, v_i)$ is an edge such that $r_i =$ capacity of $e_i -$ flow through e_i is greater than 0;
- or $e_i = (v_i, v_{i-1})$ is an edge such that $r_i =$ flow through e_i is greater than 0;

then for each $i = 1, \ldots, k$

- if $e_i = (v_{i-1}, v_i)$ is an edge, then increase the flow through e_i by Δ;
- else if $e_i = (v_i, v_{i-1})$ is an edge, then decrease the flow through e_i by Δ;

where Δ is the minimum of all $r_1, \ldots, r_k$.

We are using an equivalent, but simplified way of increasing the flow. Define as an *excess* of a node v the sum of flows through the edges that terminate in v minus the sum of flows through the edges originating in v. (Note that the sum of excesses of all nodes of a network must be equal to 0.)

For a flow, the eccess of all internal nodes (i.e., nodes other than the source and the sink) must be 0. The excess of the sink is usually positive and its value is the size of the flow. Of course, the excess of the source is minus the excess of the sink, and hence it is usually negative. We will allow that, during the computation, the excess of some internal nodes can temporarily become greater than 0 (but drom down to 0 at least at the end of the computation).

We asssume that a network is an oriented graph, and if $e = (u, v)$ is an edge of the network, than (v, u) is an edge as well. If this is not satisfied, add formally (v, u) as an edge with capacity 0: our assumption will be satisfied, but the flows are unchanged, because the flow through a new edge must always be 0. If $e = (u, v)$ is an edge, then (v, u) will be called an opposite to e and is denoted by e^{opp}.

It is obvious that we can suppose that either the flow through e or the flow through e^{opp} is 0 for any edge e: it doesn't make sense to send some positive flow from u to v and return it immediately back to u. If both $f(e)$ and $f(e^{opp})$ are decreased by $\min(f(e), f(e^{opp}))$, we get a flow of the same size that verifies our assumption.

A residual capacity or *reserve* of an edge e is the number $r(e) = c(e) - f(e) + f(e^{opp})$. What happens if the flow through e is increased by Δ:
- the reserve of e decreases by Δ;
- the reserve of e^{opp} increases by Δ;
- the excess of u decreases by Δ;
- the excess of v increases by Δ.

What happens if the flow through e^{opp} is decreased by Δ:
- the reserve of e decreases by Δ;
- the reserve of e^{opp} increases by Δ;
- the excess of u decreases by Δ;
- the excess of v increases by Δ.

We can see that both operations along an augmenting path have the same effect. Thus, we will see a modification of the flow along a path as source $= v_0, v_1, \ldots, v_k =$ sink as sending Δ units of excess from the source to v_1, then from v_1 to v_2, etc., until the Δ units of excess eventually arrive to the sink.

How much excess can be sent along an edge e? Exactly $r(e)$:
First, if $f(e^{opp}) > 0$, we can send up to $f(e^{opp})$ units of excess by decreasing the flow in e^{opp}.
If $f(e^{opp}) = 0$ or decreasing the flow in e^{opp} is not sufficient, we can increase the flow through e by up to $c(e) - f(e)$ units of excess.

Hence we can compute first reserves of edges and then operate with reserves of edges and sending excess. Would it be possible to recover the information about the actual value of flows through particular edges? Yes, and in a unique way, iw we assume that at most one of the values $f(e)$ and $f(e^{opp})$ is greater than 0 for any edge e:
- if $r(e) < c(e)$, then $f(e) = c(e) - r(e)$ (and $f(e^{opp}) = 0$));
- if $r(e) = c(e)$, then $f(e) = 0$ (and $f(e^{opp}) = 0$));
- if $r(e) > c(e)$, then $f(e) = 0$ (and $f(e^{opp}) = r(e) - c(e)$)).

To prove it,
- if $r(e) < c(e)$, then (since $c(e) - f(e) \leq r(e)$) $c(e) - f(e) < c(e)$, i.e., $f(e) > 0$, $f(e^{opp}) = 0$, and hence $r(e) = c(e) - f(e)$;
- if $f(e) > 0$, then $f(e^{opp}) = 0$ and $r(e) < c(e)$; therefore $r(e) = c(e)$ implies $f(e) = 0$, which in turn gives $f(e^{opp}) = 0$;
- $r(e) > c(e)$ can not occur for $f(e^{opp}) = 0$; hence $r(e) > c(e)$ implies $f(e) = 0$ and $r(e) = c(e) + f(e^{opp})$.

Ford-Fulkerson Algorithm

The section not yet implemented.

Dinitz Algorithm

In the case of Dinitz algorithm, it is possible to choose how edges of the network are labeled. "Automatic Labels" means that any edge is labeled by its reserve, but only if this value is important in a particular moment of the computation, otherwise the label is not shown. The other labeling possibilities are obvious. It is recommended to use (at least at the beginning) the automatic labeling.

The Dinitz algoritm is an implementation of the Ford-Fulkerson algorithm. Since an augmenting path is composed od edges that have positive reserve, network edges wit reserve equal to 0 are *not shown* throughout the whole Dinitz section.

Note that at the beginning, when the flow through any edge is 0, reserves of edges are equal to their capacities. The edges of zero capacity that were formally added to have the opposite edge for any edge of the network, are therefore not shown at the beginning, on the other hand all original edges are visible (assuming that the original network did not involved any zero capacity edges).

Dinitz Algorithm

Essentially, the Dinitz algorithm is the Ford-Fulkerson algorithm with the rule that if we have a choice among more augmenting paths, we must choose the one that has the smallest number of edges (resolving ties arbitrarily). This rule would guarantee that the algorithm is quite fast even in the worst case. This idea had been independently found by Edmonds and Karp, but Dinitz observed that there are usually many augmenting paths of the minimum length, and introduces a preprocessing that allows the algorithm to search for paths faster.

The pseudocode shows the maing loop of the algorithm that consists of the following actions:

Stratify network

Since most of the following actions do not need the information about reserves of edges (except whether the edge reserve is 0), labels are not shown (and any edge is shown only if its reserve is positive).

In order to search for shortest augmenting paths faster, the algorithm first divides nodes of the network into layers: the source is the only node in the layer 0, a node v is in the k-th layer if it is not in the $(k-1)$-th layer, but there is a node u of the $(k-1)$-th layer such that (u, v) is an edge that has the reserve greater than 0.

Stratifying is done by a simple breadth-first search starting in the source. Since the breadth-first search is a topic of another Algovision tour (not yet implemented in Javascript), no detailed explanation of the operation is given.

The index of the layer that contains the sink gives the length of the shortest augmenting path in the given moment of the computation. If the bread

No path from the source to the sink exists

If no path from the source to the sink exists that is composed from edges that are displayed, i.e., have positive reserve, then the actual flow has the miximum size, and the algorithm exits the main loop. One execution of the main loop of the algorithm is called a *phase*.

Hide slow edges

The Dinitz algorithm must always use an augmenting path of the shortest length. Since the algorithm has not exited the loop at least one such path exists, and its length (the number of edges) is given by the index of the layer containing the sink node. It is clear that any path of the shortest length must be composed of "forward" edges - such edges that their terminus is one layer right of their origin.

The computation could be faster if the algorithm does not consider "slow" edges - such edges that their terminus is in the same layer as their origin or even left of it. It is quite easy to detect such edges. The color of the slow edges is changed to yellow. The control in the control bar gives a user option to hide yellow edges (with one exception mentionned later).

Clean

Preprocessing of nodes according to Dinitz continues. There might be nodes different from the sink such that no edge leaving the node is a blue "forward" edge. One should avoid such nodes, because no shortest augmenting path passes through them (e.g., the node 16 in Example 0). This is why all edges arriving to the node are re-colored to dark yellow. This could generate another node with no blue out-edges (e.g., the node 14 in Example 0). All in-edges of such a node are recolored to dark yellow, etc.

Even though it is not really necessary (we would neve enter such nodes), nodes with no incoming blue edges are cleaned as well using analogous procedure.

The procedure is explained in more details in a later scene, the present scene just shows the outcome of the "cleaning".

Find saturated flow

After stratification, hiding the slow edges, and cleaning, *any* path starting in the source would eventually finish in the sink.

Now, we keep repeating the following actions until all blue edges disappear (i.e., until there is no augmenting path of the present length).

First, an augmenting path from the source to the sink is found. While Edmonds-Karp implementation used backtrack to find such a path (or to find that no such path exists), in the present clean stratified network the path finding is a very straightforward one, there is no danger of entering a dead-end edge or to loose one or more steps by using a slow yellow edge. The action takes time proportional to the length of the path that is bounded by the number of all edges, while the Edmonds-Karp backtrack generally needed the time proportional to the number of edges of the network.

While constructing the path, the algorithm keeps track of the smallest reserve of an edge of the path. This is why the "Automatic Label" mode shows labels (reserves) of the path edges to determine Δ, the minimum of the reserves of the path. The path edges also become green.

When the path is known and Δ determined, the excess of the size Δ is propagated through the path. The size of excess is visualized using a rectangle thet extend up from the center of the node. The only exception is the source (negative excess), where the rectangle goes down.

What happens with the edges of the network: the reserve of the edges of the path (green, later again blue) decreases by Δ. For at least one edge of the path its reserve drops to 0 and such an edge becomes invisible and not good for the future augmenting paths (during the present phase of the computation).

On the other hand, opposites of the path edges increase their reserve by Δ. However, during the present phase, all such edges are *yellow* (slow) and hence not good for augmenting paths in the present phase. This is the key moment which makes the algorithm fast: during one phase, blue edges only disappear and each path processing kills at least one blue edge.

During a path processing, the opposites of the path edges are shown (with yellow color) as well as their labels, even if Hide Yellow is checked.

A path processing kills at least one blue edge and hence "dead-end" nodes could be created. In order to facilitate the next path search, the procedure of cleaning, described above and in one succeeding scene, is applied again to kill dead-end nodes.

End of loop

Why Short Paths

This scene repeats the computation of the Dinitz algorithm and shows why the number of executed steps is limited by a function of the number of nodes and edges independently of the capacity values.

At an appropriate moment you will see two numerical values in the north-east corner of the window: the length of the shortest path from the source to the sink, composed only of "blue" edges, and the number of the blue edges. In the lexicographical order, the couple of values only decreases. This means that any change of values is such that either the path length decreases (no matter what the number of blue edges does), or the path length remains the same and the number of blue edges decreases.

As already shown in the previous scene, an augmenting path according Dinitz algorithm can be composed only of blue edges. The reserves of edges change only during a path update - for at least one blue edge its reserve decreases to 0; on the other hand, there are edges the reserves of which are increased, but all of them are opposites to the path edges and hence they are *slow*. Thus, during one phase the number of blue edges decreases.

The path update could not create a new augmenting path of the length shorter or equal to the shortest augmenting path length. Consequently, if all blue edges are "killed" and a phase of the computation finishes, there is no more augmenting path of the length that has been considered during the phase, and the "Path Length" value increases.

In the new phase, any edge that has been classified as "slow" in the preceeding phase could become "blue". However, the initial number of the blue edges in the new phase could not be, of course, greater than the number of all edges of the network. Similarly, the length of any augmenting path could not be greated than the number of nodes of the network.

Now, we are able to give the upper bound to the number of changes of the displayed couple of values: if the network has N nodes and M edges (we can assume that the network is connected and hence $N \leq M + 1$), then there are not more than N changes of the path length, and during each phase, i.e., when the path length remains the same, at most M changes of the number of blue edges - not more than NM changes during the whole computation.

Finally, let us estimate, how many steps the algorithm are associated with any particular change. We can associate the path length change with stratifying the network at the beginning of the corresponding phase a determination of slow edges. Since the stratifying is implemented as a breadth-first search, this action takes $O(M)$ time during one phase, and $O(NM)$ for all at most N phases.

One possible way a blue edge is killed during a particular phase is that becomes saturated (the reserve drops to 0) during a path update. Since the path construction and update work in time proportional to the path length (because the subnetwork of the blue edges has been cleaned and contains no dead-ends), this time is $O(N)$.

Another way to kill a blue edge is to find that it leads to a dead-end node. It is easy to see that cleaning works in the time $O(1)$ per eliminated node (the scene "Cleaning" will explain this fact in detail).

Thus, eliminating one blue edge within a particular phase takes not more than $O(N)$ time. If follows that the total time during one phase is $O(NM)$ and $O(N^2M)$ during all at most N phases of the computation.

Combining the results given above, we can see that the computing time of the Dinitz algorithm is $O(N^2M)$,

Why Not Long Paths

This scene shows once more why it is important for the computing time to use shortest augmenting paths. The scene is explained with respect to Example 0, other choices might not work well due to the absence of a long augmenting path that is used to show that non-Dinitz implementation might loop in the way that shown in the Ford-Fulkerson scenes.

The crucial point of the analysis of the Dinitz algorithm in the previous scene was that a path update could not create new augmenting path of the length *smaller or equal* to the actual shortes aubmenting path length. This scene shows that this might not be true if a longer aubmenting path is processed.

The scene begins in the same way as the previous two scenes: the network is stratified, slow edges are identified and the network is cleaned. Now, start to create the firs augmenting path, leaving the checkbox Hide Yellow unchecked. At the beginning, the algorithm follows the Dinitz rule and nodes are 0, 1, 6, 11, 13. However, the next edge added to the path is not $(13, 18)$ (which would and must be selected by the Dinitz algorithm), but $(13, 8)$, which leads us far back in the system of stripes. The remaining part of the path construction is standard, and we finally get the path 0, 1, 6, 11, 13, 8, 12, 15, 18.

When processing the path, sending 25 units of excess along the edge $(13, 8)$ increases the reserve of the opposite edge $(8, 13)$ to 25, which creates a *new augmenting path* 0,2,8,13,18, which is *textbfshorter* than 5, the length of the actual shortest augmenting path.

The scene stops here, our goal has been illustrated and we are not interested in the following computation.

Cleaning

Algorithm of 3 Indiens

The section not yet implemented.

Goldberg Algorithm

The section not yet implemented.