Algorithmic Combinatorics (Discrete Mathematics II) WS 2018 / 2019
Tibor Szabó

Connected components, connectivity, and spanning trees.

These are some rough informal notes—to augment your own notes.

Algorithms for connectedness. How to find out whether a graph G is connected? The most natural idea is to take a vertex v, explore its neighbors, then its neighbors' neighbors, and so on. This way we explore the connected component of v. If at the end this is $V(G)$, then G is connected, otherwise not. How to formalize this a bit more?

Algorithm `Comp`

Input: graph G, vertex $v \in V(G)$

Idea: maintain a set Q of vertices that were reached (on a path from v), but not yet explored for their neighbors, and a set W of vertices whose neighbors were explored. W will be disjoint from Q.

Initialization: $Q = \{v\}$, $W = \emptyset$

Iteration: For $i \geq 1$ Step i:

- Define v_i to be an "arbitrary" vertex in $Q \setminus W$ and move it over to W. That is, update $Q := Q \setminus \{v_i\}$ and $W := W \cup \{v_i\}$
- Update $Q := Q \cup (N(v_i) \setminus W)$ (some of the added vertices might already be in Q)
- IF $Q = \emptyset$ then STOP and return $W =: W^*$ (as the component of v in G)
- ELSE iterate ($\rightarrow$ Step $i + 1$)

Theorem 1. *The algorithm terminates in at most n steps. The connected component $C_G(v)$ of v in G is equal to the set W^* that* `Comp` *outputs*

Proof. In each step the algorithm either terminates or moves a vertex from Q to W, where it stays forever. So after at most n steps there are no more vertices not in W, hence Q is empty and `Comp` stops. (One uses here that $Q \cap W = \emptyset$, which can be proved by induction on the number of steps in the iteration.)

It can also be proved by induction on the number of steps in the iteration that every vertex u that was ever in Q is connected to v. (A vertex u becomes part of Q in Step i only if it is the neighbor of vertex v_i which was in Q in Step $(i-1)$, so by induction was connected to v.)

Then it follows that $W^* \subseteq C_G(v)$, since W^* is just the final state of W and a vertex u is moved to W only if in the previous step it was part of Q, and hence connected to v.

In the other direction, suppose (for a contradiction) that there is a vertex $u \in C_G(v) \setminus W^*$. Let P be an arbitrary vu-path (which exists since u is in the connected component of v).

Let $w \in V(P)$ be the first vertex on P that is not in W^* (there is such a vertex since u is one for example; w might even be equal to u). Let w^- be the predecessor of w on P. Then w^- must be in W^* (by the definition of w). But in the step when w^- was moved to W, all its neighbors NOT in W (w^- among them) was moved to Q. Hence w^- became part of Q during the algorithm and the only way to leave Q is to be moved to W. Since Q is empty upon termination, w^- had to be moved to W at some point, where it would have stayed and ended up in W^*, a contradiction. □

Note that in the description of the algorithm, we left it open[1] that in what order we choose the next vertex in Q which is to be moved to W. In the next section we elaborate on this: there are a couple of very useful, special ways to select.

Spanning Trees, DFS-trees, BFS-trees. Our goal here is to use Algorithm `Comp` to not only identify the component of v, but also build a spanning tree of it. To this end whenever a vertex v_i is moved to W from Q, we also identify an edge going from v_i to W and add it to a set F of edges we maintain, which forms a spanning tree on W. (Note that there is always at least one between v_i: if v_i was added to Q in Step $j < i$, then $v_i \in N(v_j) \setminus W$ and in the same step the neighbor v_j of v_i was added to W). This way, together with the set W, we recursively also build a spanning tree on W: in each step we add a leaf anda pendant edge

To facilitate the building of this tree, when we put a vertex x into Q in Step i, we do so by recording the pair (x, v_i), which is a potential edge from x to W: an edge we might choose into our spanning tree F when we later add x to W.

Sometimes we just want any spanning tree, but sometimes we prefer one tailored to our specific problem at hand.

What kind of tree we will build depends heavily on our rule how we choose the next "arbitrary" vertex to be moved from Q to W. For this we will maintain Q as a list, in each step adding the vertices of $N(v_i) \setminus W$ somewhere to this list (not paying attention whether we add the same vertex multiple times), and always choosing v_{i+1} to be the first on this list.

To where exactly we add $N(v_i) \setminus W$ to the list produces very different trees.

If we imagine Q like a "queue", where the new neighbors must stand to the *end of the line*, the process is called `BreadthFirstSearch` and the tree produced is called a BFS-tree. `BreadthFirstSearch` prefers to first explore all (still unexplored) neighbors of the vertex that was added to W the earliest. A BFS-tree will contain paths as short as possible.

If, instead, we imagine Q like a "stack" of items, where we always pack the new neighbors *on the top*, the process is called `DepthFirstSearch` and the tree produced is called a DFS-tree. `DepthFirstSearch` prefers to first explore a neighbor of the vertex that was most recently added to Q. A DFS-tree tends to be enlonagted, containing long paths of G.

Finding a minimum weight spanning tree. Given a connected graph G with edge weights $w : E(G) \to \mathbb{R}$, the task is to find a spanning tree $T \subseteq G$ whose weight $w(T) : \sum_e \in$

[1] we said "arbitrary"

$E(T)w(e)$ is minimum among the weights of all spanning trees of G. (In this case we call T a MST (minimum-weight spanning tree).)

A naive algorithm would take a look at all spanning trees, and calculate their weight. This potentially could involve checking all n^{n-2} spanning trees on n labeled vertices (DMI, Cayley's Theorem) and performing $n-2$ additions for each; all together $\Theta(n^{n-1})$ number of steps (which is huge ...)

In DMI that the greedy type algorithm of Kruskal, which maintains a spanning forest F of G and iteratively adds the cheapest possible edge that does not create any cycle, does find an MST.

Here we give a tiny bit more concrete description of this algorithm, so we can bound its running time (and prove that it is polynomial).

`Kruskal`$[G, w]$

Input: graph G and weight function $w : E(G) \to \mathbb{R}$

Idea: Maintain spanning forest F and iteratively add the cheapest edge that does not create a cycle. For each vertex $v \in V(F)$, maintain a label c_v (that serving to identify which vertices are in the same component of F)

Initialization: $V(F) = V(G)$, $E(F) = \emptyset$. Let $c_v := v$.

Step 0 order edges according to weight, say $w(e_1) \leq \cdots \leq w(e_m)$, where $E(G) = \{e_1, \ldots, e_m\}$.

Step i Iteratively, for every $i \geq 1$, check whether the next edge $e_i := uw$ would close a cycle when added to F. That is,

IF $c_u = c_w$ then iterate ($\to (i+1)$).

ELSE update $F := F + uw$ and for every $z \in V$ with $c_z = c_w$, update $c_z := c_u$. Then iterate ($\to (i+1)$).

Step LAST Output $F^* := F$ (claiming it be a MST).

We have seen in DMI, that this algorithm terminates and its correct, that is, F^* is min weight spanning tree of G. (Note that the labeling maintains the properties that at before each step of the algorithm, (1) the labels within each component of F are the same, and (2) the labels of two vertices in different components are different. This implies then F is always a forest, since we add and edge $e = uw$ to F if and only $c_u \neq c_w$, which happens if and only if u and w are in different components. Hence we (1) throw away all edges that are going within the same component (and hence would create a cycle) and we (2) add any edge that connects two distinct components (and then adjust the labels so the crucial properties of the labels are maintained).

This way Kruskal Algorithm terminates with a spanning tree after adding $n-1$ edges to F.

What is the running time for a graph with n vertices and m edges? The sorting at the beginning takes $O(m \log m)$ comparisons. In each iteration we check whether they are in the same component of F, by checking whether $c_u = c_w$, which takes just $O(1)$ step. Then, depending on the result, we either do nothing else in this step, or perform the updating of the

index of the components of one of the vertices, which takes at most $O(n)$ steps. There are m iterations, so the cumulative running time is $O(m \log m) + O(m) + O(n^2) = O(m \log m + n^2)$. This is $O(m \log m)$ if G is dense (i.e., has a lot of edges; concretely if $m = \Omega\left(\frac{n^2}{\log n}\right)$). Otherwise the running time is at most $O(n^2)$ (but with an appropriate data structure the second step also only needs $m \log m$ steps).