50 Representation of graphs

In this chapter we will peek into the area of data structures for representation of graphs. Our ultimate goal is to design a data structure that represents a forest with weighted vertices and allows efficient path queries (e.g. what is the cheapest vertex on the path between u and v) along with cost and structural updates.

Let us define the problem more formally. We wish to represent a forest $F = (V, E)$, where each vertex v has cost $c(v) \in \mathbb{R}$.[1] We would like to support following operations:

- *path query* — find the vertex with minimum cost on a path $u \to v$;[2]
- *point update* — set $c(v) \leftarrow d \in \mathbb{R}$;
- *path update* — increase cost of each vertex on the path $u \to v$ by $\delta \in \mathbb{R}$;
- *structural update* — connect two trees via edge (u, v) or delete edge (u, v).

50.1 Static path

As a warm-up we build a data structure for F being a static path, without structural updates. This will also be an important building block for the more general case.

Let us denote the vertices $v_1, \ldots, v_n$ according to the position on the path and let us denote $c_i = c(v_i)$. We build an range tree T over the costs $c_1, \ldots, c_n$. That is, T is a complete binary tree with $c_1, \ldots c_n$ in its leaves (in this order) and inner nodes contain the minimum of their children. Note that each node represents a subpath of F with leaves being the single vertices.

Theorem: Static path representation via range tree can perform *path query*, *point update* and *path update* in $\mathcal{O}(\log n)$ time.

Proof: Any $v_i \to v_j$ subpath of F forms an interval of leaves of T and each such interval can be exactly covered by $\mathcal{O}(\log n)$ subtrees and they can be easily found by traversing T top to bottom. The answer to the path query can then be easily calculated from the values in the roots of these subtrees.

The point update of c_i is simply a matter of recalculating values in the nodes on path from root of T to the leaf c_i, so it takes $\mathcal{O}(\log n)$ time.

[1] We could also had weighted edges instead.

[2] Generally, we can use any associative operation instead of minimum.

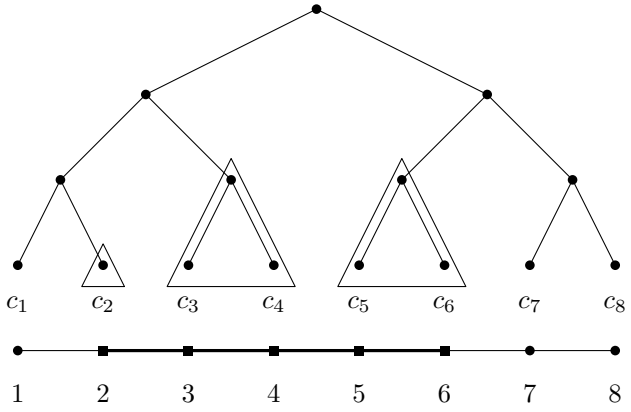

Figure 50.1: An example of a range tree for path on eight vertices. Marked subtrees cover the subpath $2 \to 6$.

The path updates are more tricky. As in the path query, we can decompose the update to $\mathcal{O}(\log n)$ subtrees. But we cannot afford to recalculate the values in these subtrees directly as they can contain $\Theta(n)$ nodes. But we can be lazy and let others do the work for us.

Instead of recalculating the whole subtree, we put a *mark* into the root along with the value of δ. The mark indicates "everything below should be increased by δ". Whenever an operation touches node during a top-down traversal, it checks for the mark. If the node is marked, we update value in the node according to the mark and move the mark down to both children. If the children are already marked, we simply add the new mark to the existing one.

This way, other operations can work as if there were no marks and path updates can be performed in $\mathcal{O}(\log n)$ time. Note that this lazy approach requires other operations to always traverse the tree top-down in order to see correct values in the nodes.

□

50.2 Heavy-light decomposition

Now we are ready build data structure for static trees using *heavy-light decomposition*. We assume our tree F is rooted and we orient all edges up, towards the root.

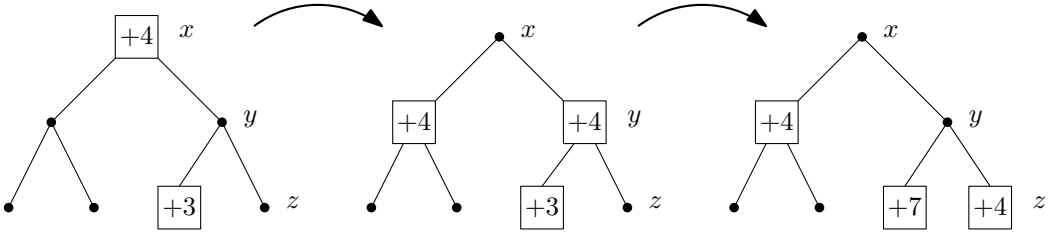

Figure 50.2: Example of range tree traversal with marks. We wish to travel from x to z. The node x is marked, with $\delta = +4$, so we need to increase value stored in x by 4 and transfer mark to both children of x. Then we can visit x and move along to y. Node y is also marked now, so we update y and transfer mark to both children. Left child of y was already marked by $+3$, so we have change the mark to $+7$.

Definition: Let F be a rooted tree. For any vertex v we define $s(v)$ to be the size of subtree rooted at v (including v). Let u be a child of v, we say the edge (u, v) is *heavy* iff $s(u) \geq s(v)/2$, otherwise we say (u, v) is *light*. Finally, a *heavy path* is simply a path containing only heavy edges.

Observation: For any vertex v, there is at most one heavy edge from v to its children. Therefore, each vertex v lies on exactly one heavy path (the path can consist of only v).

Observation: Any root-to-leaf path in F contains at most $\log n$ light edges.

This gives us the decomposition of the tree into heavy paths that are connected via light edges. The decomposition can be easily found using depth-first search in linear time.

Lowest common ancestor

A simple application of heavy-light decomposition is a data structure to answer lowest common ancestor (LCA) queries. We will also need to calculate LCA in order to evaluate path queries and updates.

For each vertex v we store an identifier of the heavy path it lies on and we also store the position of v on that path. For each heavy path H we store the light edge that leads from the top of the path and connects H to the rest of the tree. These information can be precalculated in $\mathcal{O}(n)$ time.

To answer $\mathrm{LCA}(x, y)$ we start at both x and y and we jump along heavy paths up, towards the root. Once we discover lowest common heavy path, we compare position of ''entry-points'' to decide which one of them is LCA. We have to traverse $\mathcal{O}(\log n)$ light

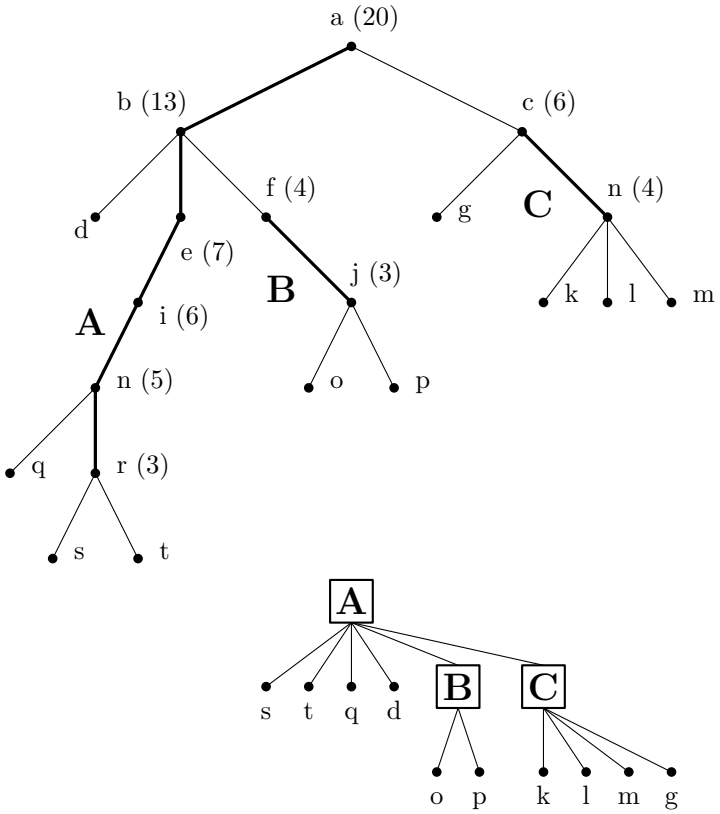

Figure 50.3: Example of heavy-light decomposition. Top part shows a tree with heavy paths marked by thick lines. Numbers in parenthesis show the value of $s(v)$ (ones are omitted). Bottom part shows the tree after compression of non-trivial heavy paths.

edges and we can jump over a heavy path in constant time, thus we spend $\mathcal{O}(\log n)$ time in total.

Path queries and updates

Let us return to the original problem of path queries and updates. The idea is straightforward: Heavy-light decomposition turns the tree into a system of paths and we already know a data structure for a static path. The following lemma gives a recipe on how to evaluate path queries and updates:

Lemma: Every path $x \to y$ in F can be partitioned into $\mathcal{O}(\log n)$ light edges and $\mathcal{O}(\log n)$ subpaths of heavy paths.

Proof: If the path is top-down we have $O(\log n)$ light edges and these edges split the path into $O(\log n)$ heavy subpaths. Otherwise, path $x \to y$ can be divided into two top-down paths at the lowest common ancestor of x and y. □

We represent each heavy path using the range tree structure for static path from the previous chapter. The root of each range tree will also store the light edge that leads up from the top of the path and connects it to the rest of the tree. We also need to store the extra information used in the LCA algorithm.

Both queries and updates are then evaluated per partes. We partition the query (update) into $\mathcal{O}(\log n)$ queries on heavy paths plus $\mathcal{O}(\log n)$ light edges. To do so, we need to calculate LCA of query path's endpoints which takes $\mathcal{O}(\log n)$ time. Each subquery can be evaluated in $\mathcal{O}(\log n)$ time and so we get $\mathcal{O}(\log^2 n)$ in total.

Corollary: A data structure based on heavy-light decomposition can perform *path queries*, *point updates* and *path updates* in time $\mathcal{O}(\log^2 n)$, it can be built in $\mathcal{O}(n)$ time and requires $\mathcal{O}(n)$ space.

Static costs

Let us analyze the partitioning of a path in a bit more detail:

Observation: When we partition a path into $\mathcal{O}(\log n)$ heavy subpaths, all of the subpaths, with one exception, are a prefix or a suffix of heavy path.

We can use this observation to make path queries faster but at the cost of keeping the costs static and forgoing the path updates. For each heavy path we calculate and store prefix and suffix minimums. This allows us to answer almost all subqueries in constant time and the one remaining subquery can be answered in $\mathcal{O}(\log n)$.

Corollary: On a static tree with static costs, the path queries can be answered in $\mathcal{O}(\log n)$ time.

50.3 Link-cut trees

Link-cut trees are dynamic version of the heavy-light decomposition. They allow us to change structure of the represented forest by either linking two trees or by cutting an edge inside a tree. Link-cut trees were introduced in a paper by Sleator and Tarjan in 1982. However, we will show later version from 1985, also by Sleator and Tarjan, that

uses splay trees instead of original biased binary trees. Although it achieves the time complexity only in amortized case, it is significantly easier to analyze.

Link-cut tree represents a forest F of *rooted* trees; each edge is oriented towards the respective root. It supports following operations:

- Structural queries:
  - PARENT(v);
  - ROOT(v) — return root of v's tree
- Structural updates:
  - CUT(v) — remove an edge between v and PARENT(v);
  - LINK(u, v) — create edge (v, u) such that v becomes a child of u (v needs to be a root before the call of LINK and both vertices cannot lie within the same tree);
  - EVERT(v) — reroot the tree and make v the root
- Cost queries:
  - COST(v);
  - PATHMIN(v) — return minimum cost vertex on the path $v \to$ ROOT(v)
- Cost updates:
  - SETCOST(v, x);
  - PATHUPDATE(V,δ) — increase all costs on the path $v \to$ ROOT(v) by δ.

Link-cut trees use a similar approach as the heavy-light decomposition and each tree is decomposed into a system of paths. Instead of heavy and light edges we have *fat* and *thin* edges. However, whether an edge is fat or thin is not given by the structure of the tree but by the history of the data structure. The only requirement is that every vertex has at most one incoming fat edge. This requirement assures that the tree can be decomposed into a system of fat paths interconnected by thin edges. Unlike heavy-light decomposition, we don't have bound on the number of thin edges on the path $v \to$ ROOT(v). In fact, it is possible that there are only thin edges in the tree! Nevertheless, we will show that everything works out in the amortized case.

The key ingredient of our data structure is the EXPOSE operation. EXPOSE(v) changes the fat-thin decomposition in such a way that v and ROOT(v) are the endpoints of a single fat path. Internally, all operations on trees are implemented using EXPOSE and a constant number of operations on a fat path:

- ROOT(v) = EXPOSE(v), then return the top of the fat path;

– Cut(v) = Expose(Parent(u)), then remove thin edge;

– Link(u, v) = Expose(u), then join two fat paths;

– Evert(v) = Expose(v), then reverse fat path;

– PathMin(v) = Expose(v), then return path minimum for the fat path;

– PathUpdate(v) = Expose(v), then perform path update on the fat path.

Conceptually, Expose(v) is straightforward. At the beginning, we turn a fat edge below v into thin edge (if there was one) and make v the endpoint of the fat path. Now, assume v lies on a fat path A. We start at v and jump along A to its top t. Unless t is the root of the tree (which means we are done), t is connected to a fat path B via thin edge (t, p), see Figure 50.4. We cut B by turning fat edge below p into a thin edge. Then we join top half of B with A by making edge (t, p) fat. This is one step of the Expose. Now we jump to the top of the newly created fat path and repeat the whole process.

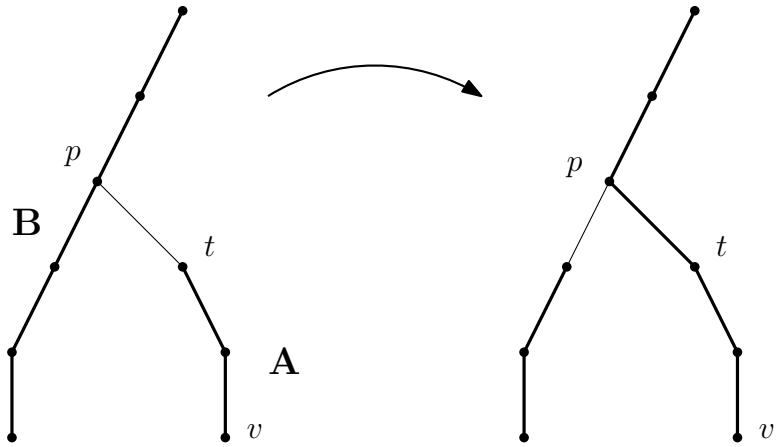

Figure 50.4: One step of Expose in the thin-fat decomposition.

Theorem (Sleator, Tarjan'82): Expose operation performs $\mathcal{O}(\log n)$ steps amortized.

By using a balanced binary tree to represent fat paths, we obtain $\mathcal{O}(\log^2 n)$ amortized time complexity for Expose and all other operations. But we can do better! In the original paper, Sleator and Tarjan use biased binary trees to improve the bound to $\mathcal{O}(\log n)$ amortized and even show that the construction can be deamortized to obtain this complexity in the worst case. However, this construction is quite technical and complicated. Instead,

we use splay trees to represent fat paths. This yields $\mathcal{O}(\log n)$ amortized complexity of EXPOSE, but with significantly easier analysis.

Intermezzo: Splay trees

Let us start by a brief recapitulation of splay trees. For more thorough description and analysis, we refer the reader to the Chapter **??**.

Splay tree is a self-adjusting binary search tree that uses SPLAY operation to rebalance itself. SPLAY(v) brings node v to the root of the tree using double rotations and at most one single rotation. All operations on the tree (including FIND) are either preceded or followed by a SPLAY operation in order to ensure $\mathcal{O}(\log n)$ amortized complexity. General rule of thumb is that whenever we travel the tree, we need to splay the deepest node we touch.

To analyze amortized complexity of a splay tree we use a potential method. Each node v is assigned an arbitrary *weight* $w(v) > 0$ and we define a *size* of v to be $s(v) \sum_{u \in T_v} w(u)$, where T_v are the nodes in the subtree rooted at v (including v). Note that the weights are just for the analysis and the data structure is not aware of their existence in any way. Base on the size we define the *rank* of the node as $r(v) = \log s(v)$. Finally, the potential Φ of the splay tree is defined as the sum of the ranks of all its nodes.

The key claim about the complexity of the SPLAY operation is the access lemma:

Theorem (Access lemma): Let t be the root of splay tree. Then SPLAY(v) costs at most $3(r(t) - r(v)) + 1$ rotations.

By setting all weights to one, the size of the node is exactly the size of the respective subtree and we obtain the $\mathcal{O}(\log n)$ amortized complexity of the SPLAY. But we can use different weights to obtain many curious properties of splay trees. And one of them is the application in link-cut trees.

Representation of fat paths via splay trees

We describe a fat path by a splay tree whose nodes have one-to-one correspondence with the vertices of the fat path. Nodes of the tree have no keys, the ordering is given by the order of respective vertices on the fat path. That is, left to right inorder traversal of the tree returns the vertices exactly in the order in which they lie on the fat path.

We deal with the costs in the same way as in our data structure for a static path. Each node contains the cost of the respective vertex and the path minimum for the subpath the whole subtree corresponds to. The only catch is that we need to recalculate minima during rotation. But this can be easily done in a constant time.

With this representation, we can easily answer ordering queries like previous vertex on the path, first and last vertex on the path etc. in an amortized $\mathcal{O}(\log n)$ time. We just have to make sure to splay the deepest node we touch during the operation.

Path minimum queries can also be easily answered in amortized $\mathcal{O}(\log n)$. Recall that we only need to answer path queries only for a prefix of the fat path, since PathMin(v) queries minimum on the path $v \to$ Root(v) and Root(v) is going to be the first vertex on the path after performing Expose(v). Thus, we simply splay v to the root and return the precalculated path minimum in the root's left son.

Path update can be performed in a similar way. Like in the case of a static path, we evaluate updates lazily and we propagate and clean marks during rotations.

Finally, to implement Evert we need to reverse the path. We can implement reverse lazily using the same trick as in the path update. Each node of the tree contains a single bit to indicate that the whole subtree below has switched orientation. As with the path updates, we propagate and clean the bits during rotations. Note that this switch is relative to the orientation of the parent node. Thus, in order to know the absolute orientation, we need to always travel in the direction from root to the leaves.

Representation of trees

Now that we know how the fat paths are represented, we need to add thin edges and connect fat paths into trees. And this is exactly what we do.

Apart from left and right son, we allow each node v in a splay tree to have multiple *middle sons*. Middle sons represent the thin edges leading into a vertex v. More precisely, if there is a thin edge leading from a fat path A into the vertex v, then the root of the splay tree representing A is a middle son of v. To distinguish between middles sons and left and right son, we will call left and right son *proper sons*.

However, a node does not know about its middle sons. We do not store a pointer from a node to its middle son. We only store a pointer from the middle son to its parent. Note that we need to update this pointer when the root of the subordinate splay tree changes, as we require the middle son to be the root of the splay tree. On the other hand, rotations in parent's splay tree have no influence on the middle sons. Also note that lazy updates (from path updates and path reversals) are *not* propagated to middle sons.

This way, each tree is represented by one *virtual tree*, where virtual tree contains two types of edges. One type represent the edges in splay trees and the other represents thin edges and middle sons.

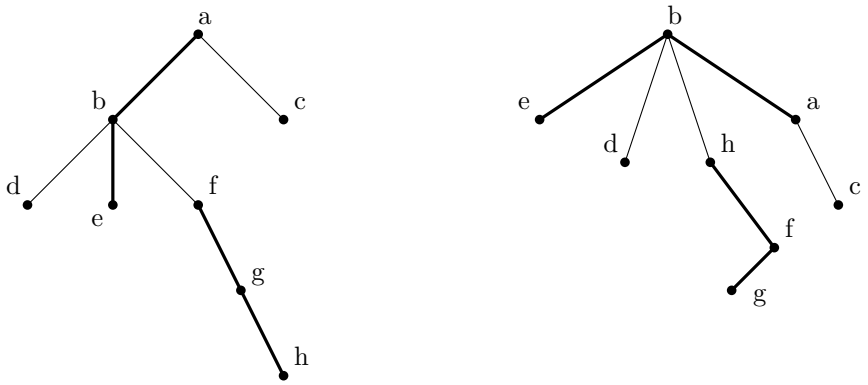

Figure 50.5: Example of a tree decomposed into a system of fat paths and a possible corresponding virtual tree.

Implementation of EXPOSE

At the start of EXPOSE(v) we need to turn a fat edge below vertex v into a thin edge in order to make v an endpoint of fat path. Let v lies on a fat path A' which is represented by a splay tree $T_{A'}$. We splay node v to the root of $T_{A'}$. Since v is the root now, its left subtree contains exactly the vertices that are bellow v on the path A'. Thus, we just turn left son of v into a middle son.

Now we show a single step of EXPOSE(v). Vertex v is the bottom node of a fat path A which is connected via a thin edge to the vertex p on a fat path B. Both A and B are represented by a splay trees T_A and T_B respectively. We assume v is the root of T_A, since we splayed it during the initial phase.

Similarly to the initial phase of EXPOSE(v), we need to cut the path B below the vertex p. So we splay p and once p is the root of T_B, we turn its left son into a middle son. Now we can join A and the remnant of B by making node v the left son of p. Then we move to the next step, where vertex p takes the role of vertex v. In the end, when v and ROOT(v), are within the same splay tree, we splay v.

Analysis of EXPOSE

For the sake of analysis we split EXPOSE(v) into three phases. Let $(r_1, p_1), \ldots, (r_k, p_k)$ be the thin edges on the path from v to the root of the virtual tree, see Figure 50.7. Let us also denote $p_0 = v$. In the first phase, we splay all of $p_0, \ldots, p_k$ into the roots of their respective splay trees. After first phase, p_k is the root of the virtual tree and $p_0, \ldots, p_k$ form a path made of thin edges. In the second phase, we turn edges $(p_0, p_1), (p_1, p_2), \ldots, (p_{k-1}, p_k)$

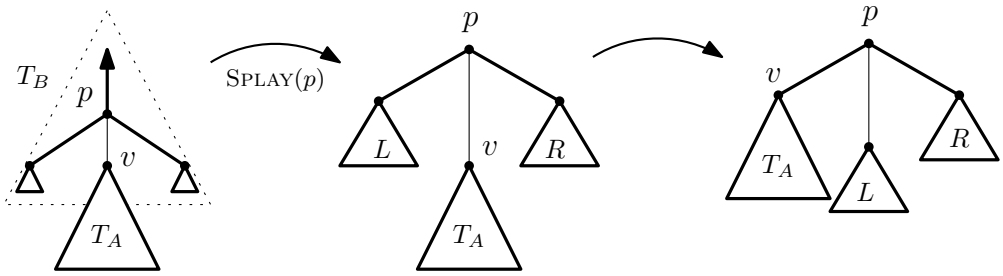

Figure 50.6: One step of EXPOSE(v) in a virtual tree.

into proper splay tree edges and left sons of $p_1, \ldots, p_k$ into middle sons, just as described in the previous section. Now that $p_0, \ldots, p_k$ form a path inside one splay tree, we splay $p_0 = v$ into the root of that tree – this is the third phase.

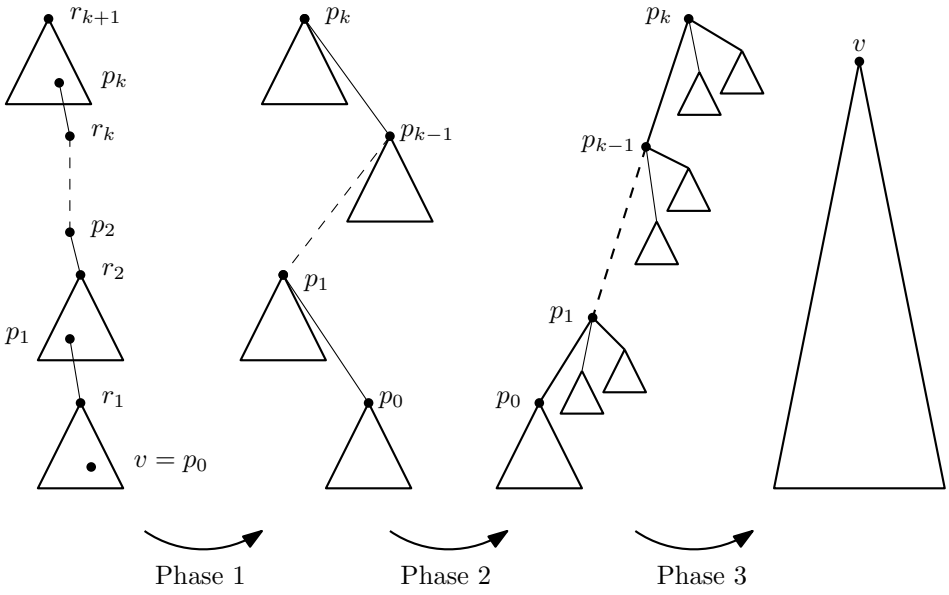

Figure 50.7: Three phases of EXPOSE(v).

The key phase we need to deal with in our analysis the first phase. Third phase is a simple splay in a splay tree, which should have have $\mathcal{O}(\log n)$ amortized complexity,

unless we break potential in previous phases, of course. Real cost of the second phase can be bounded by the real cost of the third phase, so the third phase can also pay for the second phase. However, we somehow have to make sure that swapping thin edges and splay edges does not break the potential stored in the respective trees.

The first phase, on the other hand, performs a series of splay operation and each of them possible costs up to $\Omega(\log n)$ amortized, if we use the basic bound on the complexity of splay. So, how are we going to achieve the complexity we promised?

The trick is to set weights of nodes in splay trees in a way that the whole virtual tree basically acts as one large splay tree. Recall that in the analysis of the splay tree, we assign an arbitrary non-negative weight to each node of the tree. In our analysis, we set the weight of a node to be

$$w(v) = 1 + \sum_{u \in M(v)} s(u),$$

where $M(v)$ is the set of middle sons of v. That is, size of v, $s(v)$, is the number of nodes in the subtree below v *including all nodes reachable through middle sons.*

We define a potential Φ of the virtual tree T to be the sum of the potential of each splay tree. This also means that $\Phi = \sum_{v \in T} r(v)$, where $r(v) = \log(s(v))$ is the rank of v.

Let us star with the third phase. We have only a single splay in a splay tree, so according to Access lemma we get amortized complexity $\mathcal{O}(r(p_k) - r(v) + 1)$, where the ranks are just before the third phase. Since size of any node is bound by the total number of vertices, we get amortized complexity of third phase to be $\mathcal{O}(\log n)$.

Observation: Turning a proper son into a middle son and vice versa does not change the potential.

This observation ensures that second phase does not change the potential. Thus, third phase can completely pay for the second phase.

Finally, the dreaded first phase. We perform k splays in k different splay trees. According to the Access lemma, the total amortized cost is at most

$$\sum_{i=1}^{k+1} 3(r(r_i) - r(p_{i-1})) + 1,$$

where r_{k+1} is the root of the virtual tree before the first phase. Observe that this sum telescopes! Since r_i is a middle son of p_i, we have $s(r_i) < s(p_i)$ and by monotonicity the same holds for the ranks. Using this we get that the amortized complexity is $\mathcal{O}(r(r_{k+1}) -$

$r(v) + k) = \mathcal{O}(\log n + k)$. This is still not the desired logarithmic complexity as k can be possibly quite large, perhaps $\Theta(n)$. But do not despair, the third phase can save us. Notice that $\mathcal{O}(k)$ can be upper-bounded by the real cost of the third phase. Thus, third phase will also pay for the $+k$ part of the first phase.

There is one last subtle problem we have to deal with. The problem is that structural updates can change the potential of the virtual tree outside of the Expose. The simple case is Cut. Since we remove a subtree from the virtual tree, we only decrease the potential in the original virtual tree and the potential of the new tree is paid by the potential of the removed subtree.

Link(u, v) is slightly more complicated, since adding a subtree means increasing size of some nodes. However, notice that after Expose(u) the node u has no right son as it is the last vertex on the fat path and also the root of the virtual tree. Thus, by linking u and v we only increase the size of u and we increase it by at most n, so only need to pay $\mathcal{O}(\log n)$ into the potential.

50.4 Application: Faster Dinic's algorithm

To show a non-trivial application of link-cut trees we describe how they can be used to make faster Dinic's algorithm. Recall that Dinic's algorithm is an algorithm to find a maximum flow from source vertex s to a targer vertex t in network G. We won't describe the algorithm in full detail here and we focus only on the parts important for our application of link-cut trees.

The key part of Dinic's algorithm is to find a *blocking flow* in the *level graph*. A *blocking flow* is a flow satisfying the property that every (directed) path from source to target contains a saturated edge, i.e. edge where the flow equals the capacity. The *level graph* contains exactly the vertices and edges which lie on some shortest path from s to t in the residual network⟨3⟩. The important property of level graph is that it is acyclic and it can be decomposed into levels such that there are no edges between vertices in each level, see Figure 50.8 level graph.

Dinic's algorithm starts with a zero flow and in each iteration it finds a blocking flow in the level graph and augments the flow with the blocking flow. It can be shown that we need n iterations to find the maximum flow in G.

Traditionally, the search for the blocking flow is done greedily in $\mathcal{O}(nm)$ time, which results in $\mathcal{O}(n^2m)$ complexity of the whole algorithm (construction of the leveled graph

⟨3⟩ Residual network is a network containing the edges with non-zero residual capacity, that is, difference between capacity and a flow. Capacity of each edge in residual network is exactly the residual capacity.

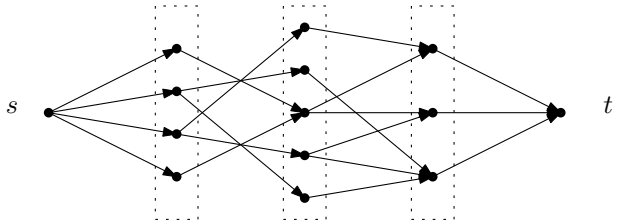

Figure 50.8: Example of a level network.

etc can be easily done in $\mathcal{O}(m)$ time per iteration). With link-cut tree, however, we can achieve $\mathcal{O}(m \log n)$ time per iteration.

We use link-cut trees with weighted edges – the cost of an edge is its residual capacity. At the beginning of iteration each vertex in the level graph arbitrarily selects a single outgoing edge. These edges form a forest F oriented towards t and we build a link-cut tree over F. We also mark the edges of F as processed.

We maintain multiple invariants throughout the algorithm. First, t is always a root of a tree in F while s is a root if and only if there are no outgoing edges from s in F. Second, each vertex has at most one outgoing edge in F. Finally, F contains only marked edges and if an edge is removed from F, it is never added back to F again.

The search for blocking flow consist of two steps – *augment step* and *expand step*. If $\textsc{Root}(s) = t$ there is a path path from s to t in F and we can perform augment step. We find the minimal edge e on the path $s \to t$ and perform a path update that decreases costs on $s \to t$ by $\textsc{Cost}(e)$. This corresponds to increasing the flow on $s \to t$ by $\textsc{Cost}(e)$. If $\textsc{Cost}(e) = 0$ we simply cut e and remove it from F.

Procedure AUGMENT STEP

1. If $\textsc{Root}(s) = t$, then:
2. $\quad e \leftarrow \textsc{PathMin}(s)$
3. $\quad$ If $\textsc{Cost}(e) = 0$:
4. $\quad\quad \textsc{Cut}(e)$
5. $\quad$ else:
6. $\quad\quad \textsc{PathUpdate}(s, -\textsc{Cost}(e))$

The other step is expand step which is performed when $\textsc{Root}(s) = r \neq t$. If r has an unmarked outgoing edge e we simply add e to F using $\textsc{Link}(e)$ and we mark e. However, if there is no such edge, all $s \to t$ paths via r are already blocked and we may remove r

from F for good. That is, we remove all edges that lead to r. Marked edges still present in F are cut and deleted from F, unmarked edges are marked but not added to F.

Procedure EXPAND STEP

1. $r \leftarrow \text{ROOT}(s)$
2. If $r \neq t$:
3. If exists unmarked edge e going from r:
4. MARK(e)
5. LINK(e)
6. else:
7. For each edge $e = (u, r)$:
8. MARK(e)
9. CUT(e) ◁ *Cut does nothing if e is not in the tree*

We repeat these steps until $\text{ROOT}(s) = s$ and all edges going out from s are marked. That is, we stop when s would be removed from F during expand step. Using simple induction, we can easily show that in such case all paths from s to t have been blocked and we indeed found a blocking flow.

We can see that the number of steps we execute is bounded by the number of edges in the level graph. Augment step either removes an edge from F or it decrease a cost of an edge to zero so it will be removed in the next step. Expand step either adds an edge to F or it removes at least one edge. Since we never add an to F twice we have at most $\mathcal{O}(m)$ steps. Each step takes amortized $\mathcal{O}(\log n)$ time which gives $\mathcal{O}(m \log n)$ time per iteration of the algorithm and $\mathcal{O}(nm \log n)$ time to find the desired maximum flow.