

Caching for Graph Algorithms

Lecturer: Seth Gilbert

October 2018

1 Breadth First Search

Consider the problem of performing a breadth first search on a graph given as an adjacency list. That is, assume we are given an array of nodes V where each node u contains a pointer to an array E_u that lists the edges adjacent to edge u . Typically, given a graph $G = (V, E)$, a BFS would take time $O(|V| + |E|)$, enumerating all the nodes in the graph in BFS order. With caching costs, though, we have to be more careful in order to avoid a cache miss with every edge traversal. For simplicity, we are going to assume that the graph is connected.

We are going to think of the BFS in terms of level sets. That is, we will construct the BFS level-by-level. To begin with, define the first level $L_0 = \{s\}$, where s is the source node for the search. We will proceed to construct level L_{i+1} from level L_i by enumerating all the neighbors of L_i that have not yet been visited. (A standard BFS works in much the same way, except using a queue to store the nodes from the previous level set, while the next level set is being constructed.)

The construction will proceed in several steps. At each step, we will maintain the lists in sorted order. (E.g., we can assume as an inductive invariant that L_i and L_{i-1} are sorted when constructing the sorted version of L_{i+1} .) As notation, define $N(L_i)$ to be the set of edges adjacent to nodes in L_i .

1. Enumeration. First, we iterate through all the nodes in L_i . For each node $u \in L_i$, we iterate through the adjacency list E_u and copy the nodes in that list to L_{i+1} . Note that this may include duplicates, as well as many other nodes that we do not want in L_{i+1} , e.g., because they have already been included in an earlier level. The cost of this enumeration is $2|L_i| + |N(L_i)|/B$. Notice that we have to pay one cache miss for each node that we visit, as there is no guarantee that the nodes in L_i will be stored nearby in memory. On the other hand, scanning the edges adjacent to some edge u will have cost at most $|E_u|/B + 1$. (The additional cost of one is due to the final block which may contain only one edge. All of these “plus one” costs add up to at most $|L_i|$.)

2. Duplicate removal. Next, we sort the list L_{i+1} . From this point onwards, we will keep L_{i+1} sorted. We can now remove duplicates by simply scanning through the list and marking for deletion any copies of nodes that appears more than once in the array. The cost of this is $\text{sort}(N(L_i))$ for sorting, and $N(L_i)/B$ for the scanning and marking deleted duplicates.

3. Removing already visited nodes. Notice that some of the nodes added to L_{i+1} may already have been visited in some earlier level. An important fact is that if $u \in N(L_i)$, then it is impossible for $u \in L_j$ for $j < i - 1$. That is, it is possible that $u \in L_{i-1}$ or $u \in L_i$, but it is impossible for u to be in any level earlier than $i - 1$. Imagine that this were not the case, e.g., that a node w in level L_i had a neighbor z in level L_{i-2} (or earlier); then when we enumerated level L_{i-2} we would have discovered node w and added it to level L_{i-1} (or earlier), i.e., it would not have been in level L_i . Hence if we have constructed all the levels from L_0 to L_i correctly, then all the neighbors of nodes in L_i will be in levels L_{i-1} or later.

Thus we can scan L_{i+1} and L_i to find nodes that appear in both. Since both lists are sorted, we can scan the two arrays together. Imagine we have one pointer in each of the two arrays, both initially pointing to the first element in their respective arrays. At every step:

- If the two pointers are indicating identical nodes, then we mark the node for deletion in L_{i+1} .
- Otherwise, if the pointer in L_{i+1} points to a smaller element than the pointer in L_i , then we advance the pointer in L_{i+1} .
- Otherwise, we advance the pointer in L_i .

In this way, we advance through the two arrays, finding all the common elements. We then repeat and do the same thing with L_{i-1} and L_{i+1} , scanning and marking common elements for deletion. When we are done, we copy L_{i+1} to a new array, removing all the deleted elements. The total cost of this step is $O(1)$ scan operations, i.e., total cost $O(|N(L_i)|/B + |L_i|/B + |L_{i-1}|/B)$.

When we are done, we now have the next level set L_{i+1} . We can continue iterating through this process until every node has been visited.

Total cost. The cost of constructing level L_{i+1} from level L_i is:

$$O(|L_i| + |N(L_i)|/B + \text{sort}(N(L_i)) + |L_{i-1}|/B + |L_{i-2}|/B)$$

An important fact is that each node appears in at most one L_j , and each edge appears in at most two $N(L_j)$. So when we sum over all the levels, we discover that $\sum_i |L_i| = |V|$, and $\sum_i (|N(L_i)|/B) = |E|/B$. Similarly, $\sum_i \text{sort}(N(L_i)) \leq \text{sort}(|E|)$. Thus, summing all the costs together yields the following:

Theorem 1 *Cache-efficient BFS has cost $O(|V| + \text{sort}(|E|))$, where $\text{sort}(E) = O((E/B) \log_{M/B}(E/B))$.*

Notes. The algorithm presented here is one developed by Munagala and Ranade. For dense graphs, where $\text{sort}(E) > V$, this algorithm is efficient. For sparse graphs, e.g., where $E = O(V)$, this algorithm is not very efficient. Assume that all the adjacency lists are stored consecutively in memory. A better solution in that case was given by Mehlhorn and Meyer, who gave an algorithm that runs in time $O\left(\sqrt{\frac{VE}{B}} + \text{sort}(E)\right)$ for a connected graph. For example, in this case, if $E = O(V)$, then we get a running time of $O(V/\sqrt{B} + \text{sort}(V))$. Finding an optimal solution remains an open question.

2 Maximal Independent Set

In the maximal independent set, you are given a graph $G = (V, E)$ and the goal is to find a subset of the nodes $S \subseteq V$ such that:

- *Independent:* no two nodes in S are neighbors in E .
- *Maximal:* every node in $V \setminus S$ has a neighbor in S .

Note that maximal independent sets are far from maximum independent sets. For example, if you take a star, the center of the star is a maximal independent set of size 1, while the leaves are a maximum independent set of size $|V| - 1$.

There is a simple greedy algorithm for finding a maximal independent set:

- Begin with $S = \emptyset$.
- Repeat until V is empty:
 - Let w be a node in V .

- Add w to S .
- Delete w and all of its neighbors from V and E .

Implemented carefully, this runs in $O(V + E)$ time. However, it is not going to be very efficient from a caching perspective.

We begin by describing Luby's Algorithm, an alternative randomized algorithm for finding an MIS. Luby's Algorithm has many advantages: it is much more efficient when running in parallel (or in a distributed system), and it also yields a more cache-efficient solution. After analyzing Luby's Algorithm, we show how to make it cache-efficient.

2.1 Luby's Algorithm

We begin by describing Luby's Algorithm for finding an MIS, and then show how we can make it efficient in the external memory model. As before, we begin with an empty set S , and repeat the following iteration until V is empty:

- For each node u , mark u with probability $1/2d(u)$, where $d(u)$ is the degree of u .
- For every edge (u, v) , if u and v are both marked, then:
 - If $d(u) < d(v)$ then unmark u .
 - If $d(v) < d(u)$, then unmark v .
 - If $d(u) = d(v)$: if $u < v$, then unmark u ; otherwise, unmark v .
- Add every marked node to S .
- Delete every marked node from V and E .

We repeat this procedure until the graph is empty, and then return S .

First, it should be clear that S is a maximal independent set. It is independent by construction: no two marked nodes are neighbors (after the unmarking step), and whenever we add a node to S , we delete all of its neighbors from V so that they can never be added to S later. And it is maximal in that every node deleted from V is either in S or a neighbor of a node in S . The key challenge is showing that it terminates quickly.

2.2 Analyzing Luby's Algorithm

To analyze the algorithm, for each node we look at the set of neighbors that have larger and smaller degree. For a given node u , define $H(u)$ to be the neighbors of u with higher degree, that is, the set of neighbors w such that $d(w) > d(u)$, or $(d(w) = d(u) \text{ and } w > u)$. Notice that if both u and a neighbor w are marked, if $w \in H(u)$, then u is unmarked. Conversely, define $L(u)$ to be all the neighbors of u with smaller degree, that is, the set of neighbors w such that $d(w) < d(u)$, or $(d(w) = d(u) \text{ and } w < u)$.

We say that a node u is *good* if at least $1/3$ of its neighbors have smaller degree, i.e., if $L(u)$ contains at least $1/3$ of its neighbors. Otherwise, the node is *bad*, which means that at least $2/3$ of its neighbors have larger degree, i.e., at least $2/3$ of its neighbors are in $H(u)$.

A good edge (u, v) is one where either u or v is good. If both u and v are bad, then (u, v) is bad. We first observe that lots of edges are good:

Claim 2 *For every graph $G = (V, E)$, at least half of all edges are good.*

Proof For every edge (u, v) , orient it toward the larger degree node. That is, if $d(u) < d(v)$, then orient it toward v ; if $d(u) > d(v)$, then orient it toward u ; if $d(u) = d(v)$, then orient it toward the larger id. Notice that if $w \in H(u)$, then an edge (u, w) is oriented toward w , i.e., it is an outgoing edge for u ; if $w \in L(u)$, then an edge (u, w) is oriented toward u , i.e., it is an incoming edge for u .

Look at some bad node v that has at least one adjacent bad edge. We know that at least $2/3$ of v 's neighbors have larger degree than v —that is why v is bad. That is, at least $(2/3)d(v)$ edges adjacent to v are outgoing edges, and at most $(1/3)d(v)$ edges adjacent to v are incoming edges. Since we have twice as many outgoing edges as incoming edges, we can map each incoming bad edge to two unique outgoing edges.

Since each bad edge has both endpoints bad, we know that each bad edge is an incoming edge for some bad node. Thus, we map each bad edge to two unique edges. Notice that the assignment is distinct: no two bad edges are assigned to the same edge. Assume for the sake of contradiction that there are more than $|E|/2$ bad edges. Then they are collectively mapped to more than $(|E|/2) \cdot 2 = |E|$ edges, which is impossible. Thus we conclude that at least $|E|/2$ edges are good. \square

Next, we argue that each good node in the graph is likely to be deleted during an iteration with some constant probability. Specifically, we show two facts:

Claim 3 Consider an iteration in which $u \in V$ and u is good. Then during that iteration, with probability at least 2α where $\alpha = (1/2)(1 - 1/e^{1/6})$, at least one neighbor of u is marked.

Proof First, notice that the probability that at least one neighbor of u is marked is at least as large as the probability that a neighbor in $L(u)$ is marked. (If a neighbor in $L(u)$ is marked, then at least one neighbor is marked; if no neighbor in $L(u)$ is marked, there is still the possibility that another neighbor of u is marked.) Thus we focus on bounding the probability that at least one neighbor in $L(u)$ is marked.

Now we calculate the probability that no neighbor of $L(u)$ is marked. Since each neighbor $w \in L(u)$ is marked independently with probability $1/2d(w)$, the probability that no neighbor is marked is at least:

$$\begin{aligned} \prod_{w \in L(u)} \left(1 - \frac{1}{2d(w)}\right) &\leq \prod_{w \in L(u)} \left(1 - \frac{1}{2d(u)}\right) \\ &\leq \left(1 - \frac{1}{2d(u)}\right)^{|L(u)|} \\ &\leq \left(1 - \frac{1}{2d(u)}\right)^{d(u)/3} \\ &\leq \left(\left(1 - \frac{1}{2d(u)}\right)^{2d(u)}\right)^{1/6} \\ &\leq e^{-1/6} \end{aligned}$$

Notice this follows because for every node $w \in L(u)$, we know that $d(w) \leq d(u)$, and since u is good we know that $|L(u)| \geq d(u)/3$. Finally, the last approximation follows from the fact that $(1 - 1/k)^k \leq e^{-1}$. Thus we conclude that with probability at least $1 - e^{-1/6}$, at least one neighbor of u (in $L(u)$) is marked. \square

We can also show that once a node is marked, it is unmarked with probability $\leq 1/2$:

Claim 4 Consider an iteration in which $u \in V$ and u is marked. Then with probability at least $1/2$, node u is not unmarked.

Proof Node u is unmarked only if another node in $H(u)$ was marked. Each node w in $H(u)$ was marked with probability $1/2d(w) \leq 1/2d(u)$. Thus, by a union bound we see that the probability that u is unmarked (i.e., the probability that *any* neighbor in $H(u)$ is marked) is at most:

$$\begin{aligned} \sum_{w \in H(u)} \frac{1}{2d(w)} &\leq \sum_{w \in H(u)} \frac{1}{2d(u)} \\ &\leq \frac{d(u)}{2d(u)} \\ &\leq \frac{1}{2} \end{aligned}$$

Thus with probability at least $1/2$, u remains marked. □

Putting the pieces together, we can now show that a good node u has a constant probability of being deleted in an iteration:

Claim 5 *Consider an iteration in which $u \in V$ and u is a good node. Then u is deleted in that iteration with probability at least $\alpha = (1/2)(1 - 1/e^{1/6})$.*

Proof Node u is deleted if any of its neighbors are marked and not unmarked. We know that with probability at least 2α , at least one neighbor of u is marked. Let M be the set of marked neighbors of u . Conditioned on the fact that $|M| > 0$, we know that the probability that at least one neighbor of u is marked and not unmarked is at least $1/2$ —since we simply selected any one node in M and discover that it remains marked with probability at least $1/2$. Thus:

$$\begin{aligned} &\Pr[\text{at least one neighbor is marked and remains marked}] \\ &\geq \Pr[|M| > 0] \cdot \Pr[\text{at least one neighbor is marked and remains marked} \mid |M| > 0] \\ &\geq 2\alpha \cdot \frac{1}{2} \\ &\geq \alpha \end{aligned}$$

(Recall that for events X and Y , we know by the rules of conditional probability that $\Pr[X] = \Pr[X \mid Y] + \Pr[X \mid \text{not } Y]$.) □

Corollary 6 *Consider an iteration in which (u, v) is a good edge. Then edge (u, v) is deleted with probability at least $\alpha = (1/2)(1 - 1/e^{1/6})$.*

Proof Since (u, v) is good, we know that either u or v is good, and whichever endpoint is good is deleted with probability at least α , leading the edge to being deleted. □

Let E_j be the set of edges at the beginning of iteration j . That is, define $E_0 = E$, and E_1 to be the number of edges remaining after the first iteration, etc. We now want to calculate the expected number of edges after j iterations, i.e., $E[|E_j|]$. We have to be a little careful because we cannot just multiply expectations.

Claim 7 *For iteration j , $E[|E_j|] = |E|(1 - \alpha/2)^j$.*

Proof First, we observe that conditioned on the number of edges in E_{j-1} , we can easily compute the expected number of edges in E_j . Specifically, from the above claims, we see that:

$$\mathbb{E}[|E_j| | |E_{j-1}| = k] \leq k - \alpha(k/2) = k(1 - \alpha/2).$$

This follows because we know that at least $k/2$ edges are good, and each good edge is deleted with probability α . The calculation then follows by linearity of expectation.

We can now compute the expectation for E_j by summing over all the k possible values of E_{j-1} :

$$\begin{aligned} \mathbb{E}[|E_j|] &= \sum_{k=1}^{n^2} \mathbb{E}[|E_j| | |E_{j-1}| = k] \Pr[|E_{j-1}| = k] \\ &= \sum_{k=1}^{n^2} k \cdot (1 - \alpha/2) \cdot \Pr[|E_{j-1}| = k] \\ &= (1 - \alpha/2) \cdot \sum_{k=1}^{n^2} k \cdot \Pr[|E_{j-1}| = k] \\ &= (1 - \alpha/2) \mathbb{E}[|E_{j-1}|] \end{aligned}$$

Proceeding by induction, with the base case that $\mathbb{E}[|E_0|] = |E|$, we conclude that $\mathbb{E}[|E_j|] = |E|(1 - \alpha/2)^j$. \square

We can now use this calculation, along with Markov's Inequality, to bound the expected number of iterations we will need:

Claim 8 *The expected number of iterations until the algorithm completes is $O(\log(|E|))$.*

Proof For simplicity, we are going to say that iteration t is the final iteration if there is only one edge (or fewer) remaining at the end of iteration t . (In reality, it might take one more iteration to finish that last iteration.)

First, we define $F(t)$ to be the probability that iteration $t + 1$ or later is the final, i.e., at the end of iteration t there is more than one edge remaining. The expected number of iterations to complete is $1 + \sum_{t=0}^{\infty} F(t)$. (We are adding one here to compensate for the one extra iteration needed to remove the last edge.)

Next, we know by Markov's Inequality that $F(t) = \Pr[|E_{t+1}| \geq 1] \leq \mathbb{E}[|E_{t+1}|] \leq |E|(1 - \alpha/2)^{t+1}$. We also know that $F(t) \leq 1$, since it is a probability distribution.

Therefore, define $T = (2/\alpha) \ln |E|$. Notice that $F(T) \leq |E|(1 - \alpha/2)^{(2/\alpha) \ln |E|} \leq 1$. Now define $F'(t)$ as follows:

- If $t < T$, define $F'(t) = 1$.
- If $t \geq T$, define $F'(t) = |E|(1 - \alpha/2)^{t+1} \leq |E|(1 - \alpha/2)^{T+t-T} \leq (1 - \alpha/2)^{t-T}$.

In both cases, $F(t) \leq F'(t)$. We can now bound the expected completion time as:

$$\begin{aligned}
1 + \sum_{t=0}^{\infty} F(t) &\leq 1 + \sum_{t=0}^{\infty} F'(t) \\
&\leq 1 + T + \sum_{t=T}^{\infty} (1 - \alpha/2)^{t-T} \\
&\leq 1 + T + \sum_{t=1}^{\infty} (1 - \alpha/2)^t \\
&\leq 1 + T + \frac{1 - \alpha/2}{\alpha/2} \\
&\leq 1 + T + \frac{2}{\alpha} - 1
\end{aligned}$$

Thus, the total running time is $O(T)$, in expectation. □

2.3 Cache-efficient Luby's Algorithm

To develop a cache-efficient version of Luby's Algorithm, we have to think about how to implement each of the iterations. In order to implement a parallel Luby's implementation, we will store the graph as an array of edges with the following properties:

- Each edge (u, v) appears in the array twice: once as the forward edge (u, v) and once as the reverse as (v, u) . We will think of (u, v) as the part of the edge owned by node u and (v, u) as the part of the edge owned by v . Notice that when we sort the array, if we scan the array it is equivalent to reading the adjacency list of each node.
- Each edge (u, v) in the array has attached to it $d(u)$ and $d(v)$, i.e., the degree of each node. (We leave it as an exercise to calculate the degrees.)
- Each edge (u, v) also has some additional space for indicating if the edge is marked, unmarked, deleted, etc.

We will refer to this array of edges as E .

One common operation will be to sort the edges E . Sometimes we will sort the array by the first component of each edge, and sometimes by the second component of each edge. For example, if we have the edge array $\langle (a, g), (d, a), (c, f) \rangle$, then if we sort it by the first component, we will get $\langle (a, g), (c, f), (d, a) \rangle$; if we sort it by the second component, we will get $\langle (d, a), (c, f), (a, g) \rangle$.

We now go through the steps of a Luby's iteration and see how to implement them in the context of the edge array E , transforming it into a new edge array that stores the results of the iteration.

As a running example, we will consider the following adjacency list:

$$(a, c), (a, d), (b, c), (c, a), (c, b), (c, f), (d, a), (d, e), (d, f), (e, d), (f, c), (f, d)$$

Marking nodes. The first step of Luby's Algorithm is to mark edge node u with probability $1/2d(u)$. To do this, we sort the array E by the first component and scan it from beginning to end. Recall that the array can be (conceptually) divided into chunks associated with the edges adjacent to each node, i.e., first the edges adjacent to a , then the edges adjacent to b , etc. Whenever the scan first encounters a new chunk, it makes a random choice to decide whether to

mark that node. More precisely, the first time we see an edge (u, \cdot) , we flip a coin with probability $1/2d(u)$ and decide whether to mark node u . If we decide to mark node u , then as we continue to scan the array, we mark each node (u, \cdot) that we encounter. Since these nodes are all arranged consecutively in the array (after sorting), this is easily feasible. Notice that just because we have marked (u, v) does not mean that we will have marked (v, u) —the latter edge we are associating with the adjacency list of node v .

In our example above, let us assume that as we go, we decide to mark nodes a , b , and c . Then the resulting edge list will look as follows, where M indicates that an edge is marked:

$$M(a, c), M(a, d), M(b, c), M(c, a), M(c, b), M(c, f), (d, a), (d, e), (d, f), (e, d), (f, c), (f, d)$$

Unmarking nodes. We next want to unmark nodes that have a marked neighbor of higher degree. To do this, we first make a copy of the array and sort the copy by the second component. In our running example, we get the following:

$$\begin{aligned} E &= M(a, c), M(a, d), M(b, c), M(c, a), M(c, b), M(c, f), (d, a), (d, e), (d, f), (e, d), (f, c), (f, d) \\ E' &= M(c, a), (d, a), M(c, b), M(a, c), M(b, c), (f, c), M(a, d), (e, d), (f, d), (d, e), M(c, f), (d, f) \end{aligned}$$

Notice that when you sort the edges in this reversed manner (if you sort breaking ties consistently), each edge in E is in the same position as its opposite edge in E' . That is, if you scan the two arrays E and E' in parallel, then when you find edge (x, y) in E you will find edge (y, x) in E' . By comparing (x, y) to (y, x) you can determine if both x and y are marked: if x is marked, then (x, y) will be marked; if y is marked, then (y, x) will be marked.

Assume you are scanning E and E' and arrive at edge (x, y) and (y, x) and discover that both x and y are marked. In this case, if $d(x) < d(y)$ or if $d(x) = d(y)$ and $x < y$, then you unmark x . To do this, you scan through all the edges (x, \cdot) and unmark them. (If we decide to keep x marked, then do nothing; we do not need to modify E' .)

One way to think about doing this is that you have three cursors: two in E and one in E' . Whenever you begin scanning a new node in E , you leave the first cursor at the beginning, while allowing the remaining two cursors to proceed with the scan. If you discover during the scan of the node that it needs to be unmarked, then you advance the trailing cursor, unmarking each edge. When you get to the last edge of the node in E , then you advance both cursors to the beginning of the next node.

When this is done, each node in E is marked only if it was marked in the first step and did not have a larger neighbor marked. In our running example, the degree of a is smaller than the degree of c , so we unmark a . The degree of b is also smaller than the degree of c , so we unmark b .

$$\begin{aligned} E &= (a, c), (a, d), (b, c), M(c, a), M(c, b), M(c, f), (d, a), (d, e), (d, f), (e, d), (f, c), (f, d) \\ E' &= M(c, a), (d, a), M(c, b), M(a, c), M(b, c), (f, c), M(a, d), (e, d), (f, d), (d, e), M(c, f), (d, f) \end{aligned}$$

Marking edges for deletion. At the same time, we also want to ensure that both endpoints of each edge to be deleted are marked. Hence we repeat the same trick as before: we copy the array E into E'' and sort by the second component. We scan the two arrays in parallel. If as we scan (x, y) and (y, x) we discover that (x, y) is marked and (y, x) is not marked, then we mark node y for deletion. (Note that we mark all the edges (y, \cdot) for deletion in this case.) Similarly, if we discover that (x, y) is not marked and (y, x) is marked, then we mark node x for deletion.

$$\begin{aligned} E &= D(a, c), D(a, d), D(b, c), M(c, a), M(c, b), M(c, f), (d, a), (d, e), (d, f), (e, d), D(f, c), D(f, d) \\ E'' &= M(c, a), (d, a), M(c, b), (a, c), (b, c), (f, c), (a, d), (e, d), (f, d), (d, e), M(c, f), (d, f) \end{aligned}$$

At this point, if we have marked node x , then all the neighbors of y have been marked for deletion.

We then repeat the process to make sure that both pairs (x, y) and (y, x) are either deleted or not deleted. We again copy E to E''' and sort E''' by the second component. We again scan and mark edges deleted: if (x, y) is marked deleted, we mark (y, x) deleted, and if (y, x) is marked deleted then we mark (x, y) deleted. In this case, we do not delete the entire node—just the corresponding edge. The goal here is to ensure that if we decided to delete an edge in the prior step, then we also delete its partner. After copying and sorting we have an array as such:

$$\begin{aligned} E &= D(a, c), D(a, d), D(b, c), M(c, a), M(c, b), M(c, f), (d, a), (d, e), (d, f), (e, d), D(f, c), D(f, d) \\ E''' &= M(c, a), (d, a), M(c, b), D(a, c), D(b, c), D(f, c), D(a, d), (e, d), D(f, d), (d, e), M(c, f), (d, f) \end{aligned}$$

This then yields:

$$\begin{array}{llllllllllll} E & = & D(a, c), & D(a, d), & D(b, c), & M(c, a), & M(c, b), & M(c, f), & D(d, a), & (d, e), & D(d, f), & (e, d), & D(f, c), & D(f, d) \\ E''' & = & M(c, a), & (d, a), & M(c, b), & D(a, c), & D(b, c), & D(f, c), & D(a, d), & (e, d), & D(f, d), & (d, e), & M(c, f), & (d, f) \end{array}$$

At this point, we have now marked an independent set of nodes, and we have indicated for deletion all the edges adjacent to those marked nodes.

Finishing the iteration. To finish the iteration, we need to partition the array E into two parts: the independent set and the edges remaining for the next iteration. To do this, we scan the array copy the nodes with marked edges to one array, skip the deleted edges, and copy the unmarked, undeleted edges to another array. In our running example, this yields:

$$\begin{array}{ll} E & = (d, e), (e, d) \\ I & = c \end{array}$$

That is, we have chosen to add one node c to our independent set, and that leaves exactly one edge (d, e) remaining in the graph. All the other nodes (a, b, f) were neighbors of c and deleted.

In the next iteration, we will begin with the edge array E , and we will append the new independent nodes to the existing array I .

Cost. Notice that each step described above consists of $O(1)$ array copies, $O(1)$ sorts, and $O(1)$ scans. In an iteration that begins with edges E , each copy and scan takes time $O(|E|/B)$, and each sort takes time $sort(E) = O((E/B) \log_{M/B}(E/B))$. Thus, the total expected cost is:

$$\begin{aligned} E[\text{cost}] &= E \left[\sum_{t=0}^{\infty} sort(E_t) \right] \\ &= \sum_{t=0}^{\infty} E[sort(E_t)] \\ &= \sum_{t=0}^{\infty} sort(|E|(1 - \alpha/2)^t) \\ &\leq \frac{E}{B} \log_{M/B}(|E|/B) \sum_{t=0}^{\infty} (1 - \alpha/2)^t \\ &= O(sort(E)) \end{aligned}$$

Notice that the since the number of edges is decreased, in expectation, by a constant fraction, the total cost is dominated by the $sort(E)$.

Theorem 9 *Cache-efficient Luby's Algorithm has cost $O(sort(E)) = O((|E|/B) \log_{M/B}(|E|/B))$.*

3 Connected Components

Our goal is now to determine the connectivity of a graph $G = (V, E)$. Specifically, we will reorganize the graph so that each component of the graph consists of a depth-1 tree, i.e., a designated root r that is connected directly to every node in the component. If there are k components in the original graph, there will be k such trees generated; if two nodes are in the same component initially, then they will be in the same tree in the end.

We are going to assume that the graph is given simply as an array of edges E . (Unlike in Luby's, we will assume that each edge appears only once.) The goal is to transform the array into a new array where each edge is of the form (r_j, x) where r_j is the root of one of the components.

Basic Idea

The basic idea is divide-and-conquer. First, we divide the edges list E into two equal-sized parts: E_1 and E_2 . (We can do this simply scanning through the entire array once and copying it into two places.) Then, we recursively solve the problem for E_2 . We now have two edge lists: E_1 , which is a normal set of edges, and E_2 , which consists of depth-1 trees. We are going to contract E_1 with respect to edges in E_2 , recurse on E_1 , and then merge the resulting E_1 and E_2 .

Contraction

Assume we have edge list E_1 and edge list E_2 , and we have already recursively solved the problem on E_2 . Ideally, we would like to solve the problem on E_1 recursively as well. Unfortunately, in its current state, E_1 does not correctly reflect the connectivity of the original graph. For example, it may consist of edges (a, b) and (c, d) which are disconnected; by contrast, edges (b, c) may be in E_2 .

More generally, for every pair of nodes in E_1 , if they are connected in E_2 then we want to ensure they are also connected in E_1 . Luckily, E_2 has a special structure: it is a depth-1 tree. So there are only two different ways that nodes in E_1 might be connected in E_2 , each of which has four variations:

- (a, b) and (c, d) are in E_1 . Edges (r, a) and (r, c) are in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (r, a) and (r, d) are in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (r, b) and (r, c) are in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (r, c) and (r, d) are in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (a, c) is in E_2 , where a is a root in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (a, d) is in E_2 , where a is a root in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (b, c) is in E_2 , where b is a root in E_2 .
- (a, b) and (c, d) are in E_1 . Edges (b, d) is in E_2 , where b is a root in E_2 .

Basically, we need to check for each of these eight cases and modify the edges in E_1 to fix the problem.

There is a simple general way to deal with all eight cases: if edge (a, b) in E_1 intersects an edge (r, b) , then replace it with (a, r) ; similarly, if edge (a, b) intersects an edge (r, a) , then replace it with (a, b) . In either case, any edge that connects to a leaf in a tree in E_2 is re-connected to the parent of that subtree.

Notice that when this is done, every node in a leaf of a tree in E_2 is removed from E_1 . As important, any two nodes in E_1 that are connected in the original graph are now connected in E_1 also. (You can go through all eight cases to see that this works!)

We refer to this operation as contraction, i.e., we are effectively contracting all the edges in the trees in E_2 , leaving only a single node (represented by the root of the tree) for each tree in E_2 .

To perform contraction, we execute the following steps:

- Sort E_1 by the first component.
- Sort E_2 by the second component.
- Scan through E_1 and E_2 in parallel, looking for any pair of edges $(a, b) \in E_1$ and (r, a) in E_2 . Whenever such an edge is found, mark (a, b) in E_1 for deletion and add edge (r, b) to the end of E_1 .
- Sort E_1 by the second component.

- Scan through E_1 and E_2 in parallel, looking for any pair of edges $(a, b) \in E_1$ and $(r, b) \in E_2$. Whenever such an edge is found, mark (a, b) in E_1 for deletion and add edge (a, r) to the end of E_1 .
- Copy E_1 , removing all the edges marked for deletion.

Remainder of the Algorithm

At this point, we have recursively solved E_2 and contracted E_1 . We now recursively solve the problem for E_1 . Notice that the base case (for both of the recursive calls) is when all the edges fit in memory, i.e., there are $\leq M$ edges. Once this occurs, we can simply load all the edges into memory with M/B block transfers and solve the problem in memory (e.g., using DFS).

Once we have finished the recursion on E_1 , we have two sets of depth-1 trees: E_1 and E_2 . We now need to merge these two together. If we simply add all the edges of E_2 to E_1 , we will have an almost correct solution, since every step preserves the connectivity of $E_1 \cup E_2$:

- Dividing the edges in two does not change the connectivity of $E_1 \cup E_2$.
- The recursive call on E_2 does not change the connectivity of E_2 (by recursive assumption), and so it does not change the connectivity of $E_1 \cup E_2$.
- The contraction procedure *does* change the connectivity of E_1 , but only by connecting nodes in E_1 that were already connected in $E_1 \cup E_2$. Thus contraction does not connect any separate components in $E_1 \cup E_2$ (and since it does not remove any edges it does not separate any nodes). Thus contraction maintains the connectivity of $E_1 \cup E_2$.
- The recursive call on E_1 does not change the connectivity of E_1 and hence does not change the connectivity of $E_1 \cup E_2$.

However, if we simply add together the edges of E_1 and E_2 , we will get a depth-2 tree: if r_1 is a root in E_1 and r_2 is a root in E_2 , we may get edges: (r_1, r_2) in E_1 and edge (r_2, x) in E_2 , i.e., a two level tree.

Thus we want to repeat something like that contraction procedure: if we have an edge (a, b) in E_1 and an edge (b, w) in E_2 , we want to add edge (a, w) to the final answer.

We sort E_1 by the second component and E_2 by the first component. Then we scan the two array of edges at the same time. Whenever we find such edges $(a, b) \in E_1$ and $(b, w) \in E_2$, we add an edge (a, w) to E_1 and we mark $(b, w) \in E_2$ for deletion. Lastly, we can through E_2 and add every edge not marked for deletion directly to E_1 . We then return E_1 .

Cost

To compute the cost of this algorithm, we notice that it consists of two recursive calls containing at most half the edges, along with a constant number of sorts and scans. The base case occurs when all the remaining edges fit in memory, at which point the final cost is $O(M/B)$ to perform the connectivity check in memory. The resulting recurrence is thus:

$$\begin{aligned} T(|E|) &\leq 2T(|E|/2) + O(\text{sort}(E)) \\ T(M) &= M/B \end{aligned}$$

Solving this, we find that $T(|E|) = O(\text{sort}(E) \log(|E|/M))$.

4 Minimum Spanning Tree

We would like to generalize the algorithm for finding connected components to instead find a minimum spanning tree. In fact, we will focus on finding a minimum spanning *forest* (MSF), i.e., we will not assume that the graph is connected.

The key idea is that instead of dividing the edges arbitrarily into two sets E_1 and E_2 , we will sort them and divide them by size: E_1 containing the edges smaller than the median weight edge and E_2 containing edges at least as large as the median weight edge. We can then recursively find the minimum spanning forest of the edges in E_1 —and argue that every edge in the MSF for E_1 must also be in the MSF for $E_1 \cup E_2$. We can then contract the trees from E_1 in E_2 , recurse on E_2 , and then re-expand to get the final minimum spanning forest.

Recursion. First, we sort the edges by weight and find the median weight w . We then scan the edges and divide them into two sets: E_1 contains edges of weight $< w$, and E_2 contains edges of weight $\geq w$. (Assume for now that all the edge weights are distinct. If there are many edges of weight w , then we will need to be a bit more careful to ensure that E_1 and E_2 each contain a constant fraction of the edges.)

We then recursively find the minimum spanning forest of the edges in E_1 . Let T_1 be the resulting MSF. We first argue that every edge in T_1 is also in the minimum spanning forest of the original graph E . For simplicity, again, we are assuming that every edge weight is unique (though that is not necessary).

Claim 10 Assume that T_1 is an MSF for E_1 and every edge in E_1 has weight less than every edge in E_2 . Then every edge in T_1 is in the MSF of $E = E_1 \cup E_2$.

Proof Assume, for the sake of contradiction, that this is not true. Let T' be the real MSF for E . The MSF T_1 contains at least one edge e not in T' . Edge e , when combined with the edges in T' must create a cycle C . We know that e must be the heaviest edge on the cycle: if not, then the optimal MSF would include e (and discard the heaviest edge on C). Thus e must be heavier than all the other edges in C .

Some of the other edges in C may be in E_1 and some of the edges in C may be in E_2 . Since e is heavier than all the other edges in C and since $e \in E_1$, we conclude that all the other edges in C must also be in E_1 . This implies that there is some edge in E_1 that could be added to T_1 to create a cycle wherein e would be the heaviest edge on the cycle. That contradicts the assumption that T_1 is an MSF for E_1 . Thus we conclude T' must include every edge in T_1 , and hence that every edge in T_1 must be in the MSF of E . \square

Contraction. We now want to update E_2 to contract each of the trees in T_1 . The easiest way to do this is to first use the connected component algorithm to transform T_1 into a collection of depth-1 trees. Let C_1 be the set of depth-1 trees resulting from running the algorithm for finding connected components on T_1 .

We can now contract E_2 . We sort C_1 by the second component and E_2 by the first component, and we scan both arrays at the same time in search of an edge $(u, v) \in C_1$ and $(v, x) \in E_2$. If we find such edges, we replace (v, x) in E_2 with an edge (u, x) of the same weight. We then sort E_2 by the second component, and again scan both arrays in search of an edge $(u, v) \in C_1$ and $(x, v) \in E_2$. If we find such edges, we replace (x, v) in E_2 with an edge (u, x) of the same weight. In both cases, we “tag” the newly added edge with the edge that was contracted, e.g., in the examples above the edge (u, x) is tagged with (v, x) or (x, v) (respectively). This tag will allow us later to easily re-expand the contracted edge as needed.

At this point, for every tree in T_1 , there is at most one node in E_2 (i.e., the root of that tree in T_1). That is, we have contracted all the edges from T_1 in E_2 .

Recursion again. At this point, we recursively find the minimum spanning forest of E_2 . Let T_2 be the resulting MSF. Lastly, we expand the contracted edges using the tags. That is, if T_2 includes an edge (x, y) that is tagged with a label (a, b) then we replace (x, y) with (a, b) .

Claim 11 *Let T_2 be the MSF for the contracted edge set E_2 . Then every edge in T_2 is in the MSF of the original edge set $E = E_1 \cup E_2$.*

Proof Assume not. Let T' be the MSF for E . Let e be an edge in T_2 that is not in T' . Then e must form a cycle C where all the other edges in C are in T' and e is the heaviest edge on the cycle. Some of the edges in C may be in E_1 and some of the edges in C may be in E_2 . Let C' be the set of cycle edges in E_2 (before contraction). (Notice that e is one of those edges since $e \in E_2$.) The edges C' may not form a connected cycle. Next we replace each edge in C' that was contracted with the edge that it was replaced with. Now C' forms a cycle of the same weight as C in the contracted version of E_2 . This implies that there exists a cycle of edges in (contracted) E_2 where e is the heaviest edge, i.e., e is not in the MSF for (contracted) E_2 . That is a contradiction, and hence we conclude that every edge in T_2 is in the MSF for $E = E_1 \cup E_2$. \square

Hence we return the tree $T = T_1 \cup T_2$. Lastly, we argue that $T_1 \cup T_2$ is a minimum spanning forest:

Claim 12 *The tree T is a minimum spanning forest for the original edge set E .*

Proof We have already shown that every edge in T is in the minimum spanning forest for E . It remains only to show that it is a spanning forest. Assume not, i.e., that there are two nodes u and v that are connected in E but are not connected in T . In fact, assume there is some edge $(u, v) \in E$ where u and v are not connected in T . (If any u and v are connected in E and not in T , then there must be some pair of neighbors with that property—simply look at the path from u to v in E and find where it is disconnected in T .)

If $(u, v) \in E_1$, then since T_1 is an MSF of E_1 , we can conclude that u and v are connected in T_1 . Assume, then, that $(u, v) \in E_2$ and u and v are not connected in T_1 .

If $(u, v) \in E_2$ is not contracted, then we know that u and v will be connected in T_2 , since T_2 is a MSF for E_2 . Assume instead that (u, v) is contracted to some edge (u, b) (that is tagged with edge (u, v)). (Note that the symmetric case where it is contracted to (v, b) is identical.) After the recursive call, we know that u and b are connected in T_2 by some path P . We know that the reason that (u, v) was contracted to (u, b) was because there existed some edge $(b, v) \in C_1$ and hence some path from b to v in T_1 . Thus (before expansion) we know that u is connected to b by P and b is connected to v by (b, c) and so u is connected to v .

We need to show that after the edges are expanded, we maintain the connectivity of the path P . Some of the edges on path P may be expanded (because they were previously contracted and tagged). Assume that some edge (w, x) on the path P is replaced during expansion with some edge (w, y) . This means that previously edge (w, y) was contracted because there existed an edge $(x, y) \in C_1$ and hence a path P' from x to y in T_1 . Therefore the edge $(w, x) \in P$ is replaced after expansion with (w, y) , and combined with $P' \in T_1$, ensuring that w is still connected to x in T .

Thus we conclude that in the final tree T , u and v are connected, which is a contradiction. We conclude that T is a spanning forest (and hence a minimum spanning forest) of E . \square

Cost analysis. The final step is to analyze the cost. The algorithm consists of two recursive calls, each of which contains half the edges, and several scans and sorts. It also include one call to the algorithm for connected components, which has cost $O(\text{sort}(E) \log(E/M))$. Again, the base case occurs when $|E| < M$ and all the edges fit in memory.

Thus the recurrence is:

$$\begin{aligned} T(E) &\leq 2T(E/2) + \text{sort}(E) + O(\text{sort}(E) \log(E/M)) \\ &\leq O(\text{sort}(E) \log^2(E/M)) \end{aligned}$$

A naive implementation of an MST algorithm will run in time $O(E \log V)$. Thus, this algorithm will be faster when (roughly) $B > \log_{M/B}(E/B) \log(E/M)$.

Improvements. There is an unnecessary factor of $\log(E/M)$ here due to the fact that we are reducing tree T_1 to its connected components. If you think about it, that is not really necessary.

Imagine instead that each edge in T_1 were tagged with a label that indicate its connected component, e.g., if edge (u, v) and (a, b) were in the same connected component, then they would both have the same label.

Using that labeling, we could now perform the same contraction (and re-expansion) process. During the contraction, whenever we find an edge (x, y) in E_2 and an edge (y, b) in T_1 with label ℓ , we will replace edge (x, y) in E_2 with edge (x, ℓ) with the same weight. (As before, we should tag the new edge with (x, y) , i.e., the edge it has replaced.) The label ℓ attached to the connected component in T_1 becomes a fake node in E_2 . When we are done, every node in T_1 has been replaced in E_2 with such a fake node, i.e., each tree in T_1 has been contracted.

After the recursive call to find a MSF for E_2 , we can re-expand each edge as before, replacing it with the saved tag. As before, whenever we replace an edge (x, ℓ) with some (x, y) during the expansion process, it is because y was part of connected component ℓ in T_1 and hence there is a path in T_1 to every other node in the same connected component. Hence all the same proofs go through.

The final observation is that we can construct the proper labeling of trees with connected component identifiers. Sort T_1 by the connected component label, and sort E_2 by the second component. (Assume all the fake nodes ℓ in E_2 are the second component.) As you scan through in order and expand edges in E_2 that were previously contracted, whenever you replace (x, ℓ) , at the same time relabel all the nodes in T_1 with label ℓ with the label of x .

With this modification, the running time is now $O(\text{sort}(E) \log(E/M))$. In fact, it is possible to find an MST in $\text{sort}(E)$ time with high probability, however it requires a much more complicated set of techniques.