Recitation 9 — Graph Contraction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

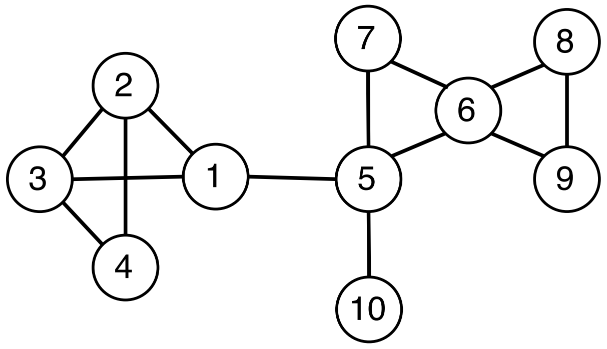
October 24, 2012

1 Announcements

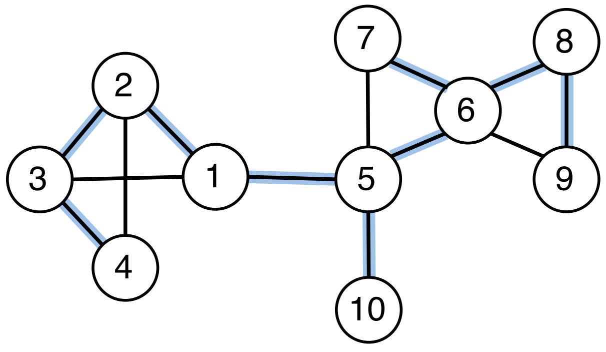
- Assignment 5 will be graded this week.
- Assignment 6 is out and due Monday. It's long so if you haven't started it, start asap.
- Today's recitation: more on DFS, Greedy MIS, labeling components, (linked list prefix sum)
- Questions about homework, class, life, universe?

2 DFS

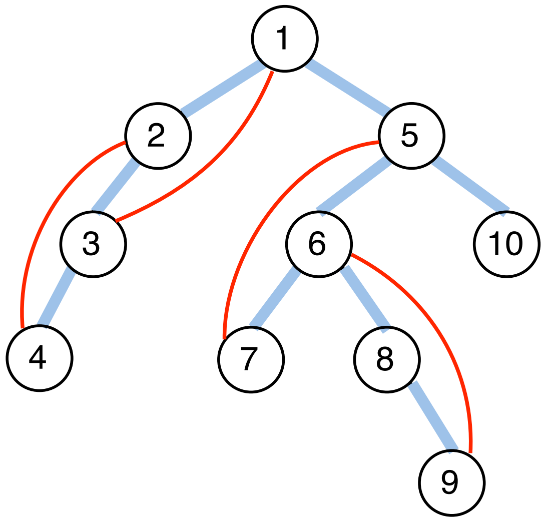
When you apply DFS to a graph, it implicitly defines a spanning tree rooted at the start vertex. If more than one tree is needed to span all vertices in the graph, then we call it a *DFS forest*, although typically we just refer to the DFS tree. Edges in the graph that correspond to edges in the DFS tree are called *tree edges*. Edges in the graph that go from a vertex v to an ancestor u in the DFS tree are called *back edges*.



Show the DFS tree on the above graph. Use the vertices in increasing order.



Redraw the tree to illustrate the back edges.



Q: Are all edges in undirected graphs either tree edges and back edges of a DFS tree?
A: Yes.
Notice that the cycle detection algorithm in the lecture notes, simply checks whether the vertex has been visited before. Why is this sufficient? Does it find all cycles?

In the homework, you are to find all bridges in an undirected graph. A *bridge* of a graph is an edge whose removal disconnects the graph. In other words, a bridge is not on a cycle.
Q: Which edges are bridges in the above graph? and in the DFS tree?

Q: Are all edges in directed graphs either tree edges and back edges of a DFS tree?
A: No. There can be *forward edges* that are non-tree edges that go from a vertex v to a descendant u in DFS tree, and *cross-edges* that are all other edges (ie, cross between two subtrees of the DFS tree)

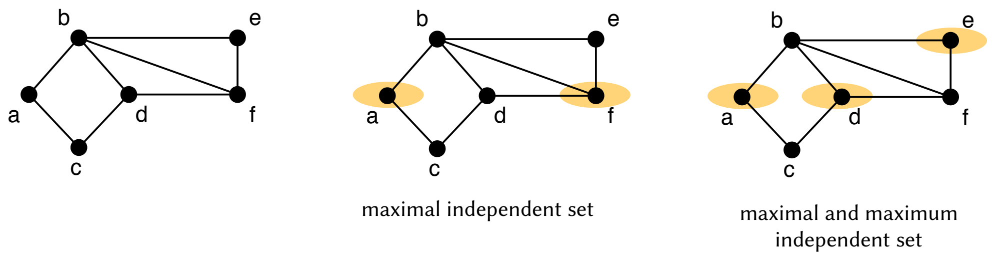
3 Maximal Independent Set (MIS)

In graph theory, an *independent set* is a set of vertices from an undirected graph that have no edges between them. More formally, let a graph $G = (V, E)$ be given. We say that a set $I \subseteq V$ is an independent set if and only if $(I \times I) \cap E = \emptyset$.

For example, if vertices in a graph represent entities and edges represent conflicts between them, an independent set is a group of non-conflicting entities, which is a natural thing to want to know. This turns out to be an important substep in several parallel algorithms since it allows one to find sets of things to do in parallel that don't conflict with each other. For this purpose, it is important to select a large independent set since it will allow more things to run in parallel and presumably reduce the span of the algorithm.

Unfortunately, the problem of finding the overall largest independent set—known as the Maximum Independent Set problem—is **NP**-hard. Its close cousin Maximal Independent Set, however, admits efficient algorithms and is a useful approximation to the harder problem.

More formally, the *Maximal Independent Set* (MIS) problem is: given an undirected graph $G = (V, E)$, find an independent set $I \subseteq V$ such that for all $v \in (V \setminus I)$, $I \cup \{v\}$ is not an independent set. Such a set I is maximal in the sense that we can't add any other vertex and keep it independent, but it easily may not be a maximum—i.e. largest—independent set in the graph.



For example, in the graph above, the set $\{a, d\}$ is an independent set, but not maximal because $\{a, d, e\}$ is also an independent set. On the other hand, the set $\{a, f\}$ is a maximal independent set because there's no vertex that we can add without losing independence. Note that in MIS, we are *not* interested in computing the overall-largest independent set: while maximum independent sets are maximal independent sets, maximal independent set are not necessarily maximum independent sets! Staying with the example above, $\{a, f\}$ is a maximal independent set but not a maximum independent set because $\{a, d, e\}$ is independent and larger.

3.1 Sequential MIS

Let's first think about how we would compute an MIS if we don't care about parallelism. We will start by thinking about the effect of picking a vertex v as part of our independent set I. By adding v to I, we know that none of v's neighbors can be added to I. This motivates an algorithm that picks an arbitrary vertex v from G, add v to I, and derive G' from G such that each vertex of G' is independent of I and can be picked in the next step. We have the following algorithm:

```
fun seqMIS($(V, E), I$) =
if $|V| = 0$ then $I$ else
  let
    val $v$ = pickAnyOne($V$)
    val $V' = V \setminus (N(v) \cup \{v\})$
    val $E' = E \cap (V' \times V')$
  in seqMIS($(V', E'), I \cup \{v\}$)
  end
```

In words, the algorithm proceeds in iterations until the whole graph is exhausted. Each iteration involves picking an arbitrary vertex, which is added to the independent set I, and removing the vertices v, together with v's neighbors $N(v)$, and edges incident on these vertices. Thus, each round picks a new vertex and removes exactly the vertices that can no longer be added to I, and nothing more. It is not difficult to convince ourselves that this algorithm indeed computes a maximal independent set of G. With a proper implementation (e.g., using arrays), this algorithm takes $O(m + n)$ work. Q: What algorithmic technique is this algorithm using? A: Greedy. The choice it makes is best at the time.

Homework 6 gives a parallel version of MIS using sets and tables. You need to reimplement it using array sequences and single-threaded sequences.

4 Connectivity

We're going to go over some examples from lecture in more detail today.

First, we're going to go over the LabelComponents problem: given a graph G, label each vertex so that two vertices have the same label iff they are in the same component (i.e. there is a path between them).

4.1 DFS

One sequential way to do this is with DFS.

```
structure DFSLabel : LABEL_COMPONENTS =
struct
  structure STSeq = STArraySequence
  structure Seq = STSeq.Seq
  open Seq

  type vertex = int
  type edge = vertex * vertex

  fun label (E, n) : labeling =
      let
        fun dfs L v label =
            case nth L v
              of SOME _ => L
               | NONE => let
                           fun doLabel (L, nbr) = dfs L nbr label
                           val L' = STSeq.update (v, label) L
                         in
                           iter doLabel L' (neighbors E x)
                         end

        val L = STSeq.fromSeq (tabulate (fn _ => NONE) n)
        val labels = iter (fn (v,L') => dfs L' v v) L (tabulate (fn v => v) n)
      in
        STSeq.toSeq labels
      end
end
```

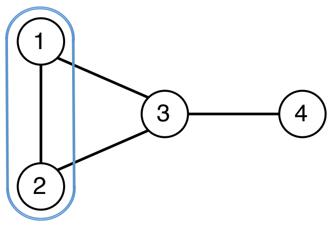
Q: What is the work/span of the above code?
A: $O(|E| + |V|)$ work and span

4.2 Union Find

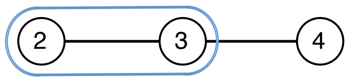
Another way is with union-find. Start with each vertex as a separate component. Then for each edge, find which components its endpoints are in and join the two components. This is slower than DFS but leads to a parallel algorithm:

```
structure UFLabel : LABEL_COMPONENTS =
struct
  structure Seq = ArraySequence
  open Seq

  type vertex = int
  type edge = vertex * vertex

  fun label (E, n) : labeling =
      let
        fun contract ((x, y), L) =
            let
              val (x', y') = (nth L x, nth L y)
              val L' = update (x', y') L
            in
              map (nth L') L' (* <-- What does this do?? *)
            end

        val L = tabulate (fn i => i) n
      in
        iter contract L E
      end
end
```

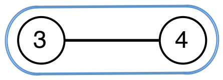
Q: What does the highlighted line do? Why do we need it? This is a technique in union-find is called *path compression*, which points each vertex to its representative vertex in its component. We can demonstrate it with the following example on 4 vertices (ignore heads/tails):



We start with L=⟨1,2,3,4⟩. If E = ⟨(1,2), (2,3), (1,3), (3,4)⟩, the first edge it contracts is (1, 2). After first round of contraction we get L'=⟨2,2,3,4⟩ and L'' is the same. That is, vertices 1 and 2 are in the same component, with 2 as its representative vertex. The graph now looks like



Contracting the edge (2,3), we get L'=⟨2,3,3,4⟩. Notice that vertex 1 is still pointing to 2, but 2 is now part of component 3. Using map (nth L') L', we *compress* the path 1− > 2− > 3 of length 2 to paths of length 1 and obtain L''=⟨3,3,3,4⟩. That gives us the following contracted graph:



Contracting the edge (1,3) has no effect, since $(x', y') = (3, 3)$. But contracting edge (3,4) we get `L'=⟨3,3,4,4⟩`. As before, vertices 1 and 2 are still pointing to 3, but 3 is now part of component 4. The `map` function results in the rest of the component 3 to point to 4 resulting in L''=`⟨4,4,4,4⟩`.

Q: What is the work/span of the above code?
A: $O(|E|^2)$ work, $O(|E|)$ span

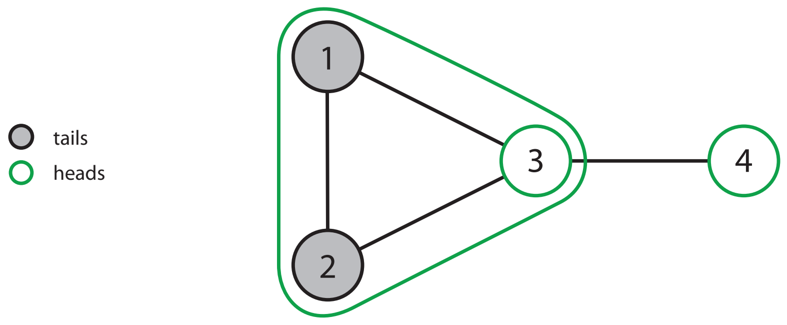
The union-find algorithm above works by contracting each edge. It is slow because it only contracts one edge at a time. Maybe we can do better with star-contraction?

4.3 Star Contraction

As in lecture, we flip a coin for each vertex, deciding if it will be the center of a star or a satellite. Then we associate each satellite with a center, and contract all of those edges. Next, we update the edges to connect to the star centers, remove new self-loops, and recurse. Except for selecting the stars to contract and updating the edges, the code is very similar to union-find above:

```
structure StarContractLabel : LABEL_COMPONENTS =
struct
  structure Seq = ArraySequence
  structure Rand = Random210
  open Seq

  type vertex = int
  type edge = vertex * vertex

  fun label (E, n) =
      let
        fun LC (L : vertex seq, E, edge seq, seed : Rand.rand) =
            if length E = 0 then L
            else let
              val F = Rand.flip seed n
              fun isHook (u,v) = nth F u = 0 andalso nth F v = 1
              val hooks = filter isHook E
              val L' = inject hooks L
              val L'' = map (nth L') L'
              val E' = map (fn (u, v) => (nth L' u, nth L' v)) E
              val E'' = filter (fn (u, v) => u <> v) E'
            in
              LC (L'', E''. Rand.next seed)
            end

        val L = tabulate (fn i => i) n
      in
        LC (L, E, Rand.fromInt 0)
      end
end
```

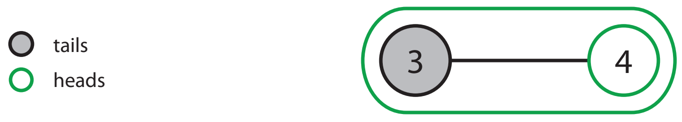
Let's run the algorithm on the example above with the coin tosses as shown:



In the first round, vertices 1 and 2 are satellites and contract to the star center, vertex 3. Vertex 4 is another star center. If `E = ⟨(1,2), (2,3), (1,3), (3,4)⟩` then after one round we have

```
hooks = ⟨(2,3), (1,3)⟩
L’ = ⟨3,3,3,4⟩
E’ = ⟨(3,3), (3,3), (3,3), (3,4)⟩
E’’ = ⟨(3,4)⟩
```

That gives us the following contracted graph:



The second and final round of `LC` contracts the graph to a single vertex and L’’=`⟨4,4,4,4⟩`.

Q: What is the work/span of the above code?
A: $O(|E| + |V|)$ work and $O(\log^2 |V|)$ span

5 Linked List Scan

We have a linked list of nodes; each node i has some data D_i. For each node, we want to compute $T_i = \sum_{\text{j after i in M}} D_j$. Let $S[i]$ be the index of the successor of node i. If there is no successor then $S[i] = i$. That is, S is a permutation of {0, 1, . . . , n-1} except that the last node points to itself.

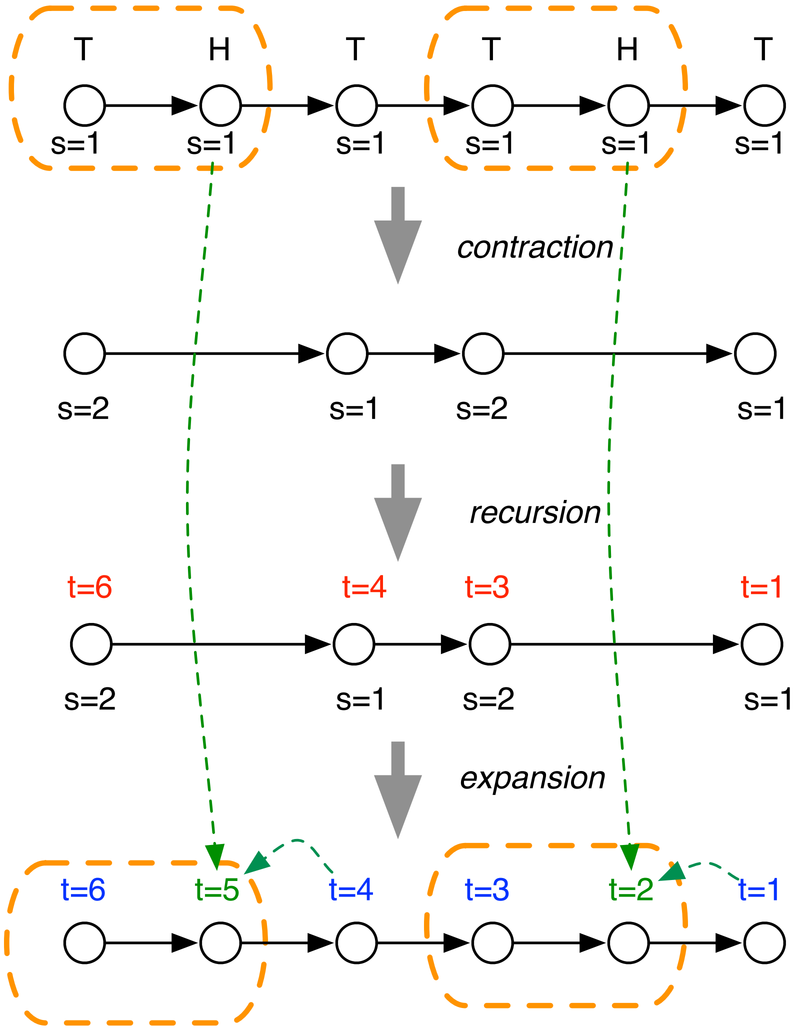
This looks like the familiar scan problem on sequences except that it computes the sum of the suffixes But it is harder than scan: we don’t know where in the list each element is. That is, the i^{th} node in the list is not necessarily at position i.

But we can use the same idea: combine pairs of adjacent nodes, recurse on the smaller list, and then expand back the list. With sequences we paired elements at even positions with the following odd elements. In linked lists, we don’t know which nodes are even and which are odd.
Q: How do we stop a node from ending up in two pairs?
A: Restrict each node to being paired with only its predecessor node or only its successor node. Then each node can only be part of one pair.

Q: How do we mark the nodes? We don't have much to go on...
A: Try random (random is usually worth a try).

Each node flips a fair coin. If the flip comes up tails then it is an "even" node; if it is head it is "odd" node. We pair only ("even","odd") adjacent nodes. That is, a node is paired with the next node if it flips tails and its successor flips heads. What is the probability that node i is paired with its successor in the list? This happens with probability 1/4 (we must marked i with heads and $S[i]$ tails, each of which occurs indepedently with probability 1/2). By linearity of expectation, we get $(n-1)/4$ pairs total in expectation (the last node can't start a pair).

Now it's just details: To contract a pair we *splice out* the second node from each pair from the list. Let (x, y) be a pair. We are deleting y from the list, so we need to make the successor to x be y's successor, that is $S'[x] = S[y]$. Now we need to add in y's data to x's data: $D'[x] = D[x] + D[y]$. Then recurse on the new list (the old list minus all the y's). The result returned is, everything that wasn't a y has the correct sum T (by IH). For each y, add in the result of its successor: $T[y] = D[y] + T[S[y]]$.



In the code below, I is sequence of indices of nodes that are still in the linked list, X are nodes that are the first of a pair, and Y are the second of a pair.

```
fun isHead (rand : Rand.rand) (v : vertex) =
      (Rand.hashInt rand v) > 0

fun listScan f b succ data =
    let
        fun listScan’ T S D I =
            if (length I) <= 2 then toSeq T
            else
            let
                fun ok i = (nth S i)!=i andalso not (isHead rand i)
                                         andalso (isHead rand (nth S i))
                val X = filter ok I
                val Y = tabulate (nth S) X
                val IS = tabulate (fn i => (i, nth S i)) X
                val ID = tabulate (fn i => (i, f (nth D i, nth D (nth S i)) X
                val T’ = listScan’ (inject IS S)  (inject ID D)
                              (difference I Y)
                val ID’ =
                    tabulate (fn i => (i, f (nth D i, nth T’ (nth S i)))) Y
            in
                inject ID’ R’
            end

        val T = tabulate (fn i => b)) (size data)
    in
        listScan’ (fromSeq T) (fromSeq succ) (fromSeq data) (tabulate (fn i => i) (length succ)
    end
```

Let’s do an example: (keep track of S, D, I at each step; I is the set of active indices)
$S = \{2, 0, 4, NONE, 3\}$
$D = \{2, 3, 5, 1, 4\}$
$I = \{0, 1, 2, 3, 4\}$

Let’s say we get $HHTTH$ on our first set of flips. Show the steps for the first round of the algorithm, do the recursive call, and then show how to reconstruct the list.

Answer:
$(0, 2)$ and $(4, 3)$ pair.

So we splice out $2, 3$ and get:
$S' = \{4, 0, 4, 3, 3\}$
$D' = \{7, 3, 5, 1, 5\}$
$I = \{0, 1, 4\}$

Now we recursively compute $T[0] = 12, T[1] = 15, T[4] = 5$
Then we can compute $T[2] = D[2] + T[4] = 10$ and $T[3] = D[3] + 0 = 1$.