Graph Contraction I

1 Introduction

So far in this course the graph algorithms we have covered have been either fully sequential, or exhibited only limited parallelism. Both DFS and Dijkstra's algorithm did not offer any opportunities for parallelism. The BFS and the Bellman-Ford algorithms are somewhat parallelizable, but they fall far short of the goal of obtaining polylogarithmic span.

In this and the following lecture we explore the technique of **graph contraction**, which offers a way to operate on multiple parts of a graph in parallel, allowing us to develop truly parallel graph algorithms and achieve polylogarithmic span.

2 What is Graph Contraction?

We've seen the concept of **contraction** before in this course. Specifically we used the technique to develop a low-span algorithm for the `scan` function. The idea is to take a big problem, and (1) contract it into a smaller instance of the same problem, (2) recursively solve the contracted problem, and (3) expand this solution to the smaller problem to a solution to the original problem. The contracted problem should be a constant factor smaller (by some measure), which leads to an efficient algorithm.

Given a graph G, the contracted graph G' is, at a high-level, constructed as follows. The vertices of G are partitioned into connected sets of vertices. Each connected set becomes a vertex of G'. If two of the sets of G have an edge between them, then there is an edge between the corresponding vertices of G'. The edges that occur within each set do not appear in G'. Graph contraction is highly parallelizable, since we can create and process the pieces of the graph in parallel.

Some of the problems that can be solved in polylogarithmic expected span with graph contraction are:

- Computing the connected components of an undirected graph
- Finding a Minimum Spanning Trees of a weighted graph
- Checking if a graph is bipartite
- Computing the bi-connected components of an undirected graph
- Computing the Min cut in a weighted graph

How can we efficiently partition the vertices of the graph? This is accomplished using one of several **contraction schemes**, which take a graph and output a partition of the vertices. This

requires the vertices to be divided up into several subsets, such that every vertex is in exactly one of the subsets. Ideally, our contraction scheme will reduce the number of vertices by a constant fraction $\alpha < 1$, so that our span recurrence can look something like:

$$S(n) = S(\alpha n) + O(\log n)$$

which is balanced, and has a polylogarithmic solution.

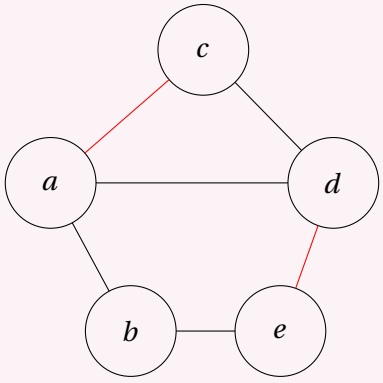
We'll discuss two contraction schemes in detail, called edge contraction and star contraction. Both of these are randomized, so we'll look at how they perform on different kinds of graphs in expectation. In doing so, we will also find out if they create the constant-factor decrease in the number of vertices that we want.

3 Edge Contraction

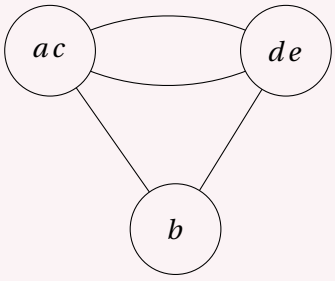
Our first method is edge contraction. In this case each of the connected sets into which we partition the vertices will consist of either (1) two vertices connected by an edge, or (1) a single vertex. This is called an **edge partition**.

Example: Edge Contraction

Let's say we start with the following graph:



We can create an edge partition from the red edges in this graph, since none of them share a vertex. The partition we would get is $\{a, c\}$, $\{b\}$, $\{d, e\}$. If we contract each of these subsets into a single vertex, we get the following graph:



Now, vertices a and c have been replaced with the super-vertex $a\,c$, and the same holds

for d and e. In this case, we allow our graph to have multiple edges between the same pair of vertices – technically, this makes it a multigraph. Whether we allow this is dependent on how we are using the contraction process in our algorithm. We could partition this new, smaller graph again, and continue contracting if we wanted to.

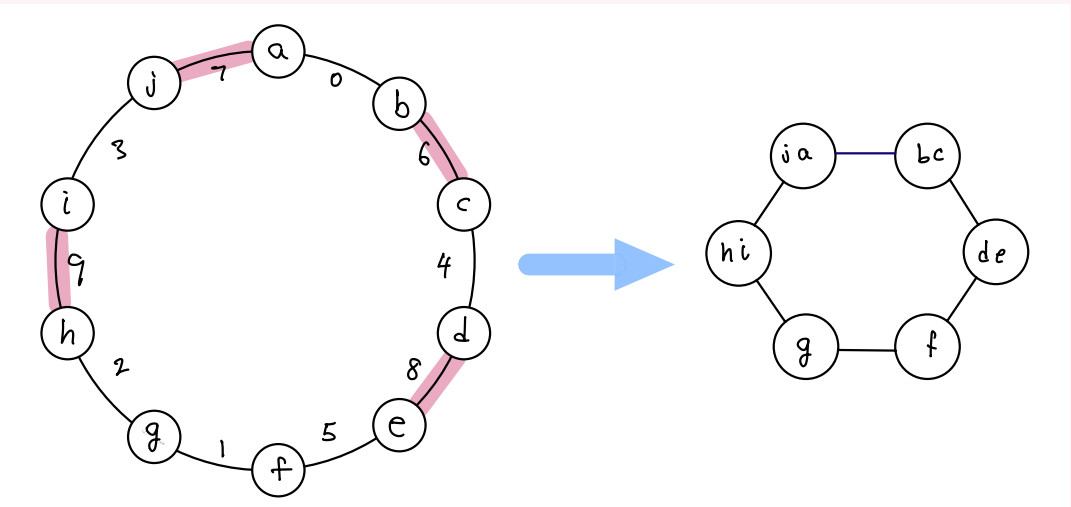
This example of an edge partition gives us some insight into what a valid edge partition looks like. We want to choose some subset of the edges in the graph such that no two selected edges share a vertex, which is otherwise known as a matching. You may have heard of matchings in 15-251, where the augmenting path algorithm was presented as a way to generate maximal matchings in a bipartite graph.

However, we will create our matchings in a simpler, randomized manner, since the augmenting path algorithm takes quadratic time. To create an edge partition, we start by assigning a random, unique priority to each edge. We then add an edge to the partition if its priority is greater than any of its neighboring edges. (Two edges are said to be neighboring if they share a vertex.)

The number of vertices we remove from the graph via contraction is then equal to the number of edges that we have selected for contraction, since each contracted edge contracts 2 vertices into 1. How well does this do, in terms of removing a constant fraction of the vertices on every round?

The effectiveness of edge contraction varies based on the structure of the graph that it is run on. Let's take a look at what happens if we run it on C_n, the undirected cycle graph on n vertices, where we simply have n vertices arranged in one large cycle. Every vertex has degree 2, and as such, every edge has two neighboring edges. Let's look at the probability that a single edge gets contracted. For this to happen, the edge needs to receive a higher priority than either of its neighbors, which happens with probability $\frac{1}{3}$ since priorities are randomly assigned. Each of the n edges gets contracted with probability $\frac{1}{3}$, so it follows that $\frac{n}{3}$ edges are contracted in expectation on every round – a constant fraction!

Example: Edge Contraction on a Cycle



The graph on the left is contracted to the one on its right via edge contraction. The edges that are selected for contraction are the ones whose priorities are greater than their two neighbors. The selected edges form a matching of size four.

Now, let's look at a more general case. Say we have an edge e that connects two vertices u and v. What is the probability that it gets contracted? This is equivalent to the probability that it has a greater priority than any of its neighbors. u has $d(u)-1$ edges other than e, and v has $d(v)-1$ edges other than e. It follows that e has $d(u)+d(v)-2$ neighbors. To get contracted, it must have the highest priority out of $d(u)+d(v)-1$ edges, which happens with probability $\frac{1}{d(u)+d(v)-1}$.

A bad scenario comes on the star graph on n vertices, which is a central vertex that has $n-1$ edges coming out of it – each going to a different vertex. All vertices other than the center are connected to the center only. What is the probability we contract an edge here? Each edge is connected to a vertex with degree 1 and a vertex with degree $n-1$. Applying the formula, we get that each edge gets contracted with probability $\frac{1}{n-1}$. That means that every round removes one vertex in expectation, which is not a constant fraction of the vertices.

As it turns out, the cycle graph was a special case where edge contraction does work. Edge contraction is not effective in general, since the number of vertices that get contracted in expectation is highly dependent on the structure of the graph. In many common cases the number of contracted vertices is small.

But the bad case for edge-contraction, a star graph, can lead us to a technique that will prove to be much more general – star contraction.

4 Star Contraction

Edge contraction provided us a way to contract along edges, provided that no two chosen edges shared a vertex. However, this caused problems in graphs where vertices had high degree. Thus, we introduce **star contraction**, which eliminates this problem. Instead of choosing certain edges to contract, we instead select some vertices in our graph as "star centers". Some of the vertices we did not select – the "satellites" – then get contracted into neighboring star centers.

More formally, we define a star partition on a graph as a partition of the graph into subsets of vertices, such that every vertex is in exactly one subset. Each subset must consist of one vertex, which we refer to as a "star center", and all other vertices in that subset must be neighbors of the star center. This means that every part of the partition forms a star subgraph of the original graph. We can then contract all of the vertices in the star partition into the center vertex.

4.1 Generating Star Partitions

Like edge partitions, we will generate star partitions using a randomized process. The nature of star partitions gives us a fairly natural pathway to creating them – first, we will select which ver-

tices are the star centers, and afterward, we will decide which adjacent star center each satellite will be contracted into.

To determine which vertices are star centers, we will flip a coin for each vertex. If the coin comes up heads, the vertex is a star center. Otherwise, the vertex is a satellite (or maybe a star center, read ahead). Then, each satellite will choose an adjacent star center to be the star center that it is contracted into. If multiple star centers are adjacent to a satellite, it can choose any of them to be its star center. If no star center is adjacent to a satellite, then it becomes a star center.

Lemma: Star Contraction

Suppose that star contraction is applied to a graph with n_1 non-isolated vertices and n_2 isolated vertices. The expected number of vertices after star contraction is at most $\frac{3}{4}n_1 + n_2$.

Proof. We need to find the probability that each vertex gets contracted into another vertex. If this event occurs, then the number of vertices goes down by 1. A vertex gets contracted if it flips tails, and at least one of its neighbors flips heads. Flipping tails happens with probability $\frac{1}{2}$. If we assume the vertex has at least one neighbor, then the probability of having a neighbor with a head is at least $\frac{1}{2}$. This means that each non-isolated vertex is removed with probability at least $\frac{1}{4}$. Isolated vertices do not get removed. Thus the expected number of vertices that remain is at most $\frac{3}{4}n_1 + n_2$. □

Notice that this analysis did not rely on any properties of the graph to show that the number of vertices (at least, the non-isolated ones) is going down by a constant factor on every round! That means that star contraction is effective on general graphs.

4.2 Informal Implementation of Star Contraction and Example

Algorithm: Informal Implementation of Star Partitioning

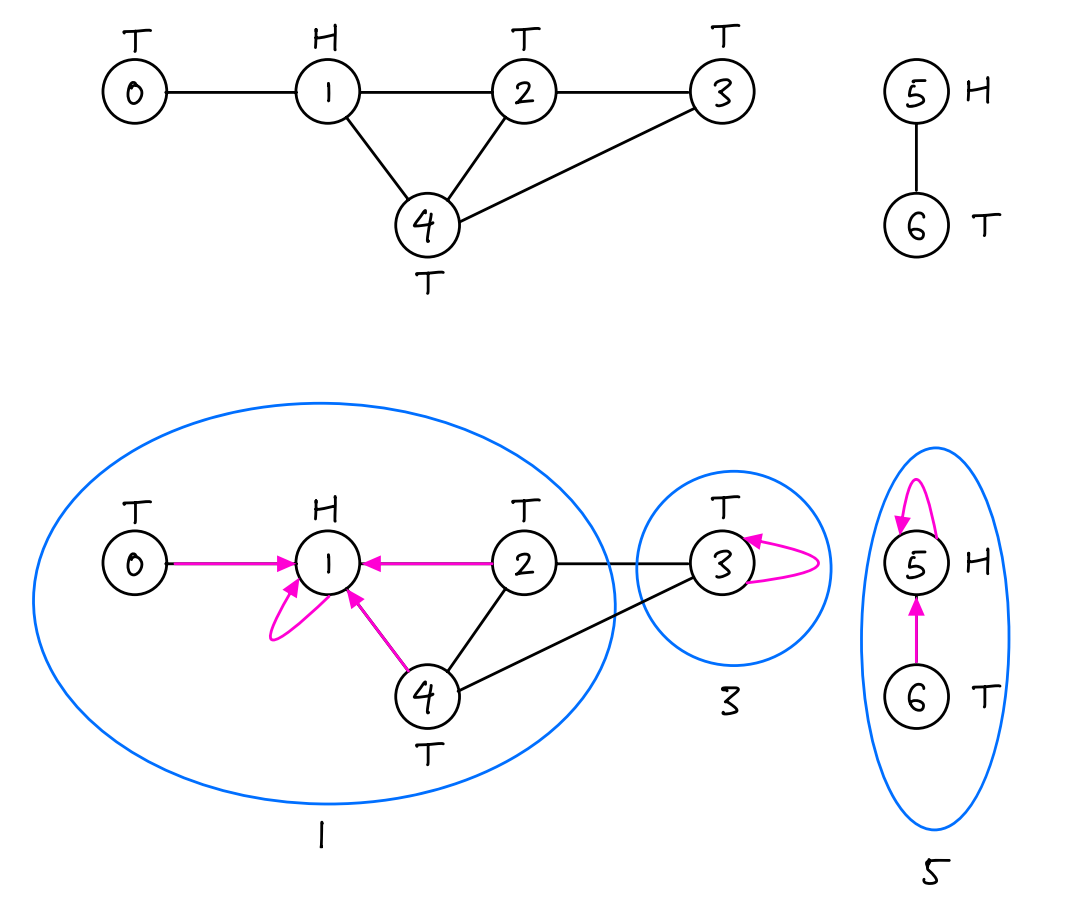
The input to the algorithm is (V, E, n). These parameters represent a graph. The vertices are numbered range $[0, n-1]$. V is a sequence containing the vertices, which are a subset of $\{0, \ldots, n-1\}$. The edges are represented by E. This is a sequence of pairs (u, v) where u and v are vertices. There are no self-loops, and if the edge (u, v) occurs in E then so does (v, u).

The algorithm returns V_c, the vertices that become star centers, and E_c the edges that remain between the star centers, and finally a sequence P which maps each vertex of V to the star center wth which it is associated.

fun `starPartition`(V, E, n):

1. Generate `heads`, a sequence of n random boolean values which are true with probability $1/2$.
2. Initialize a sequence of parent values $P = [0, 1, \ldots, n-1]$. Each vertex is initially its own parent.
3. For every vertex u with `heads`$[u]$ = `false` if it has an edge (u, v) where `heads`$[v]$ = `true` then $P[u] \leftarrow v$. (If there is more than one such v then pick arbitrarily.)
4. Compute the center vertices V_c, represented as a sequence of centers, namely the vertices for which $P[v] = v$.
5. Compute the edges E_c of the star partitioning. Start with a copy of E, and filter out (remove) the edges (u, v) where $P[u] = P[v]$. (This removes the contracted edges.) Now for all edges (u, v) that remain, replace (u, v) by $(P[u], P[v])$, so all of the edges are between vertices of V_c.
6. return (V_c, E_c, P).

Example: Star Partitioning



The input graph is shown first, along with the coin flips on each vertex. The diagram below is a star partitioning computed by the algorithm. (In this case, given the coin flips, there are no other alternatives.) The red edges in the lower graph show the parent pointers of P. Notice that there are three vertices in the contracted graph corresponding to the center vertices 1, 3, and 5. And there is one edge (1,3) in the contracted graph. But note that the algorithm does not bother to remove duplicate edges.

The input to this example is:

$V = [0,1,2,3,4,5,6]$
$E = [(0,1),(1,0),(1,2),(2,1),2,3),(3,2),(1,4),(4,1),(4,2),(2,4),(4,3),(3,4),(5,6),(6,5)]$
$n = 7$.

The output computed by the algorithm is:

$V_c = [1,3,5]$
$E_c = [(1,3),(3,1),(1,3),(3,1)]$
$P = [1,1,1,3,1,5,5]$

4.3 Implementing Star Contraction

Now that we have an algorithm for star contraction, we need to dive into the details to implement it. So we're going to go on a tangent and develop a few functions that we will need to do this.

The function `inject_disjoint(S,u)` takes as input a sequence `S` and another sequence `u` of updates to apply. Each element of the update sequence is of the form `(i,x)`. And it simply does the assignment `S[i]` ← `x` for each element of `u`. The assumption is that the indices being updated are disjoint. This means that we can do the updates in parallel with constant span and constant work per update and no data races. Here is the pseudocode for this function.

Algorithm: inject_disjoint

```
fun inject_disjoint(S : sequence<T>, u:<int,T>) -> sequence<T>:
  n = length(u)
  parallel For j=0...n-1:
    (i,x) = u[j]
    S[i] ← x
  return S
```

We're going to need a more powerful version of `inject_disjoint` called `inject` that can handle non-disjoint assignments. The algorithm will handle duplicates by arbitrarily picking one of the duplicate assignments to do, and ignoring all the others. The way we're going to implement this is simply by removing duplicate indices from `u`, then applying `inject_disjoint`.

The function we will use to remove the duplicates is `remove_duplicates(u)`. Here is the pseudo-code for this algorithm. It simply sorts the pairs `(i,x)` and removes all but the first pair beginning with a specific value of `i`.

Algorithm: remove_duplicates

```
fun remove_duplicates(u : sequence<int,T>) -> sequence<int,T>:
  u' = sort(compare,u)
  return filterIdx(fn (j,_)=> j=0 || first(u'[j-1])≠first(u'[j]), u')
```

If we let n be the length of `u`, then the work of this algorithm is $O(n \log n)$, and the span is $O(\log^2 n)$.

Now it's easy now to implement the required `inject(S,u)` function in the same work and span of `remove_duplicates(u)`.

Algorithm: inject

```
fun inject(S: sequence<T>, u : sequence<int,T>) -> sequence<T>:
  return inject_disjoint(S,remove_duplicates(u))
```

With these tools in hand, we can now give a full implementation of star partitioning.

Algorithm: starPartition

```
fun starPartition(V: sequence<int>, E: sequence<int,int>, n: int)
          -> (sequence<int>, sequence<int,int>, sequence<int>):
  heads = tabulate(fn i => random_bool(), n)
  P = [0,1,...n-1]
  TH = filter (fn (u,v) => heads[u]==false && heads[v]==true, E)
  P = inject (P, TH)
  Vc = filter (fn j => P[j] == j, [0...n-1])
  E' = filter (fn (u,v) => P[u] ≠ P[v], E)
  Ec = map (fn (u,v) => (P[u], P[v]), E')
  return (Vc, Ec, P)
```

Now let's analyze the work and span of this algorithm.

The first two lines are $O(n)$ work and $O(1)$ span. (Assuming we have an efficient source of random bits.)

The third line `TH = filter...` is $O(m)$ work and $O(\log m)$ span.

The fourth line `P = inject...` is $O(m \log m)$ work and $O(\log^2 m)$ span.

The fifth line `Vc = filter...` is $O(n)$ work and $O(\log n)$ span.

The sixth line `E' = filter...` is $O(m)$ work and $O(\log m)$ span.

The seventh line `Ec = map...` is $O(m)$ work and $O(\log m)$ span.

Since we don't know which of m or n is larger we can summarize this analysis by saying that the total work is $O((m+n)\log m)$ and the total span is $O(\log^2(n+m))$.

In the next lecture we will show how to apply star contraction to give an efficient parallel algorithm for computing the connected components of an undirected graph.