Graph Contraction II: Connectivity and Bipartiteness

1 Introduction

In the last lecture we defined the *star partitioning* algorithm. Given an undirected graph, this algorithm picks (using randomness) a subset of the vertices called the *star centers*. Every vertex is either a star center or the neighbor of one. The non star centers are called *satellites*. Each satellite picks one of its neighboring star centers. A *star* consists of a center and all of its satellite vertices. The vertices of the graph are thus partitioned into a collection of disjoint stars.

Given a graph G star partitioning creates a new contracted graph G', where each vertex of G' corresponds to a star in G. And there is an edge (x, y) in G' iff there is an edge in G between any of the vertices of the star corresponding to x and the star corresponding to y.

Two beautiful things about star partitioining are: (1) The number of non-isolated vertices decreases (in expectation) by a factor of $\frac{3}{4}$ for each application of the algorithm. And (2) if the graph is of size $s = n+m$ then the work of star partitioning is $O(s \log s)$, and the span is $O(\log^2(s))$.

In the *star contraction* framework, to solve a problem on a graph G, we contract G to G', then solve the problem on G'. Then we extend the solution for G' to a solution G, in a process called *expansion*. Because the size decreases geometrically, the number of levels of recursion is logarithmic with high probability.

This is a general framework for solving graph problems efficiently in parallel. In this lecture we give the details of two applications of this algorithm. We use it to give efficient parallel algorithms for computing the connected components of a graph, and also determine if a graph is bipartite.

For reference, here is the `starPartition` algorithm from the last lecture.

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, V)
  E' = filter (fn (u,v) => P[u] ≠ P[v], E)
  Ec = map (fn (u,v) => (P[u], P[v]), E')
  return (Vc, Ec, P)
```

Note that the representation of the edges of these graphs is a sequence of pairs of vertices (x, y). Each edge is included twice (once in each direction). The code above does not eliminate duplicate copies of the same edge. The sequence P records how vertices are classified. A vertex x is a star center iff $P[x] = x$. A vertex $y \neq x$ is a satellite of star center x iff $P[y] = x$.

2 The General Star Contraction Framework

We implement a generic sort of star contraction, which takes two functions – `base` and `expand`. The `base` function is is applied to the graph after it cannot be further contracted (i.e. it has no edges). The `expand` function is used to convert the answer `R` (to the problem we're solving on the contracted graph) the answer for the original graph. (The return type of `starContract` depends on the return type of `base` and `expand`.)

Algorithm: starContract

```
fun starContract(V: sequence<int>, E: sequence<int,int>, n: int):
  if |E| = 0 then return base(V)
  (V', E', P) = starPartition (V,E,n)
  R = starContract(V', E', n)
  return expand(V,E,V',P,n,R)
```

When this algorithm is given a graph with no edges, it will simply call the base case on the set of vertices. Otherwise, it will use our star partition function to get the new set of vertices `V'`, as well as the new edges `E'`, and a mapping from vertices to their star centers `P`. Finally, we recursively call on the new set of vertices and edges, expanding the recursive result if necessary.

2.1 Applying the Framework to Graph Connectivity

In the contracted graph G' a single vertex represents a connected set of vertices and edges of the original graph G. Thus a connected component in G becomes a connected component in G'. This is true inductively. So, when the base case is reached, there are no edges and one vertex for each connected component of the original graph.

Suppose we are just interested in counting the number of connected components in a graph. Then we can use the following `base` and `expand` functions:

```
fun base (V,n):
  return length(V)

fun expand (V,E,V',P,n,R):
  return R
```

All that happens in this case is that the base function returns the number of vertices when there are no edges left. And this number is just passed through unchanged by expand and is returned at top level.

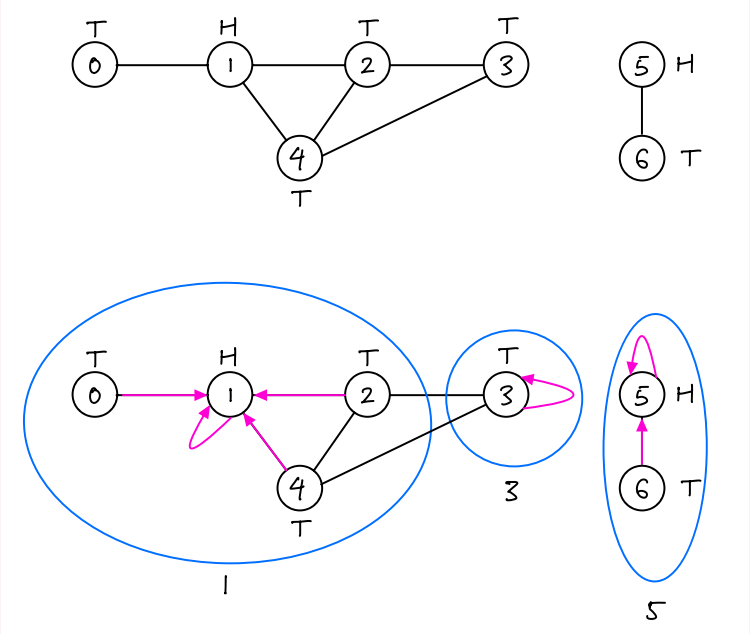
It might be more useful to actually compute the connected components. Namely the partitioning of the vertices into connected components. This can be done with the following functions:

```
fun base (V,n):
  return (V, tabulate((fn i => i), n)

fun expand (V,E,V',P,n,(_,M)):
  return (V, tabulate((fn i => M[P[i]]), n))
```

To illustrate what this does, consider the example of star partitioning from the last lecture.

Example: Star Partitioning



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:

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

Suppose that in the next (and final) contraction vertex 3 is the satellite of 1. Then the M map returned is $[0,1,2,1,4,5,6]$. This is because the $(1,3)$ edge is the only one contracted. Thus when the tabulate of the final call to `expand` is executed, it returns the map $M[P[i]]$ which is $[1,1,1,1,1,5,5]$. This indicates that vertices $\{0,1,2,3,4\}$ are in one component and $\{5,6\}$ are in the other.

In general what is going on is this. Consider the final call to `expand` which is returning the final answer. At this point in time P is the very first (top-level) map of vertices to their centers. And M is the map of vertices of all subsequent calls to `starPartition` to their *final* isolated vertex at the bottom. So to build the final map we apply P first, then M.

3 Bipartiteness

Another application of star contraction is in determining if a graph is bipartite. Recall that a bipartite graph is one in which the vertices can be partitioned into two sets A and B such that all the edges are between A and B. (Every edge has one endpoint in A and the other in B.)

Note that if a graph is bipartite, then there is a way of coloring each vertex from a set of two colors, such that there are no monochromatic edges (i.e. an edge both of whose vertices have the same color). So this coloring is a natural certificate of it being bipartite.[1]

There are very simple linear time algorithms using BFS and DFS to determine if a graph is bipartite, and finding a valid coloring if it is. The problem for us is that these algorithms are highly sequential – their span is high. Our goal here is polylogarithmic span.

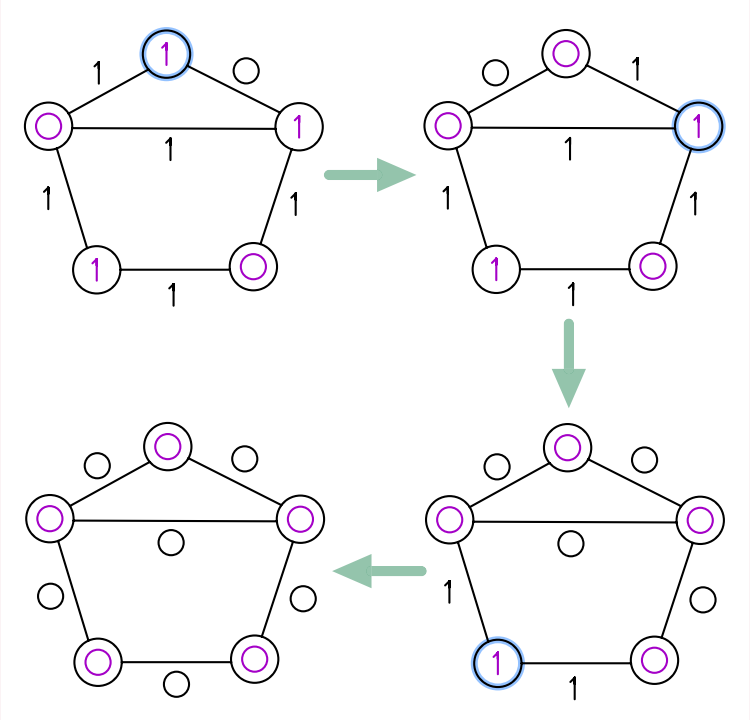
There is a beautiful way of using star partitioning to solve this problem. But the first thing we're going to need to do is to *strengthen* the problem. That is, we need to define a slightly more general problem. Because we're using recursion, and it turns out that the most natural solution involves asking a slightly more general question on the recursive calls.

So we'll call the problem *generalized bipartiteness*. In this problem we're given a graph G where each edge is labeled with a number from the set $\{0,1\}$. And we're also going to color the vertices with 0 and 1. And the rule is that for an edge labeled with 1, its two endponts must be different colors. And if the edge is labeled with 0 the two endpoints must be the same color. The problem is still solved trivially using BFS or DFS. But it's going to be a useful generalization as you'll see.

Another useful observation about this setup is the following *toggling rule*. Consider two graphs G and G'. Here G' was obtained from G by picking some vertex v and toggling the labels of all the edges incident to v. It turns out that this does not materially change the problem. There's a valid 2-coloring of G iff there is one of G'. Any valid coloring for G can be converted to one for G' by taking whatever color G uses for v and toggling $v's$ color in G', and leaving rest of the coloring unchanged.

[1] There is also a certificate that the graph is non-bipartite, namely it has an odd-length cycle.

Example: Examples of generalized bipartiteness and the toggling rule



In the upper left corner is a graph whose edges are labeled with zeros and ones, along with a valid coloring of the vertices. Note that the colors are the same at both ends of the edge labeled zero, and different at the ends of the edges labeled one. Applying the toggling rule to the top vertex results in the labeled graph shown in the upper right corner. A valid solution is obtained for that labeled graph from the previous valid solution by toggling the color of the top vertex.

Two more applications of the rule lead to the graph in the lower left corner. (In general, starting from a valid labeling, if you toggle all the vertices labeled one, you will end up with a graph where all the edges and vertices are labeled with zero.)

3.1 The Algorithm

Algorithm: checkBipartite (Informal Description)

This is a recursive algorithm which makes use of `starPartition`, like `starContract`. The input to the algorithm is (V, E, n). These parameters represent a graph. The vertices are numbered in the range $[0, n-1]$. V is a sequence of the vertices which are a subset of $\{0, \ldots, n-1\}$. The edges are represented by E. This is a sequence of pairs of the form (x, y, b) where (x, y) the pair of vertices for the edge, and b is a bit (the edge label referred to earlier). There are no self-loops. If the edge (x, y) occurs, then so does (y, x). (Both instances have the same bit label.)

The algorithm returns `NONE` if it is impossible to color the graph. Otherwise it returns `SOME colors`, where `colors` is a sequence of n colors of the corresponding vertices.

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

1. If there are no edges then return `SOME colors` where `colors` is a sequence of all 0s. (A valid coloring of a graph with no edges.)
2. $(V', P) =$ `starPartition`(V, E, n). Notice we are using a version of `starPartition` that does not generate the new edges (we do that below).
3. Now we begin the process of building the contracted graph for which we will call `checkBipartite` recursively. The key idea is to apply the toggling rule to certain vertices to ensure that the edges being contracted are all labeled with 0. More specifically, each satellite vertex whose edge to its center vertex is labeled with 1 is toggled. We remember this with a sequence `to_toggle` of n booleans. All the edges incident to a toggled vertex are toggled (Some may be toggled twice which does nothing).
4. If two different satellite vertices to the same center have an edge between them, that edge must be labeled with 0 (after step 3). If it's a 1, it means that there is no solution (so return `NONE`).
5. Now we finish the work to construct the non-contracted edges, which we call E_c. If there is an edge (x, y) and $P[x] \neq P[y]$ it means we want to change that edge to $(P[x], P[y])$. Its label remains the same. This process could create multi-edges. We will eliminate them by sorting and filtering. But while we are doing that we check to make sure that the multi-edges being eliminated have the same label. For if they don't we know there is no solution (so return `NONE`).
6. The next step is to recursively call `checkBipartite`. If it returns `NONE` we return `NONE`. If not we will receive a valid coloring of the contracted graph. We fill in the colors of the vertices that were contracted using the color of their center. Then we correct this coloring by toggling the colors of the vertices marked by `to_toggle`. Then return the new color vector.

Just as with `starPartition` all these steps can be done in $O(s \log s)$ work and $O(s \log^2(s))$ span. (Here s is the size of the graph, or $n+m$.) Since the expected recursion depth is $O(\log s)$ w.h.p. this means that the expected work is $O(s \log^2(s))$ and the expected span is $O(\log^3(s))$.

The following is a translation of the above description into code.

Algorithm: Check Bipartiteness

In the code below we have handled the case of a non-bipartite graph by throwing an exception when that is discovered, rather than using an option return type. This exception may be thrown in the function `edges_after_contraction`.

```
fun checkBipartite (V : sequence<int>, E : sequence<int, int, int>,
                    n : int) -> sequence<int>:
  if |E| = 0 then return tabulate(fn _ => 0, n)
  (V', P) = starPartition (V, E, n) // (E' is constructed below)
  to_toggle = compute_to_toggle(E, P, n)
  E' = edges_after_contraction(E, P, to_toggle)
  color' = checkBipartite(V', E', n)
  color = tabulate(fn i => if to_toggle[i] then 1-color'[P[i]]
                                           else color'[P[i]], n)
  return color


fun compute_to_toggle(E, P, n) -> sequence<int>:
   E1 = filter (fn (x,y,b) => P[x]==y && P[y]==y && b==1, E)
   update_sequence = map (fn (x,_,_) => (x,true), E1)
   to_toggle = tabulate (fn _ => false, n)
   return (inject_disjoint (to_toggle, update_sequence))


fun edges_after_contraction(E,P,to_toggle)->sequence<int,int,int>:
  E1 = map (fn (x,y,c) => if to_toggle[x] == to_toggle[y]
                          then (x,y,c) else (x,y,1-c), E)
     // E1 is all the edges, but with the necessary toggling
  E2 = filter (fn (x,y,c) -> P[x]==P[y] && c==1, E1)
     // E2 is the edges to be contracted that have color 1
  if length(E2) > 0 then raise OddCycle
  Ec1 = filter (fn (x,y,c) => P[x] != P[y], E1)
  Ec2 = map(fn (x, y, c) => (P[x], P[y], c), Ec1)
     // Ec2 is all the remaining edges after contraction,
     // with multi-edges, but no self loops
  if exists (x,y,c) and (x,y,d) in Ec2 with c!=d then raise OddCycle
  return unique(sort(Ec2))  // get rid of multi-edges
```

For completeness it is useful to explain how to implement the following expression using the standard sequence operations:

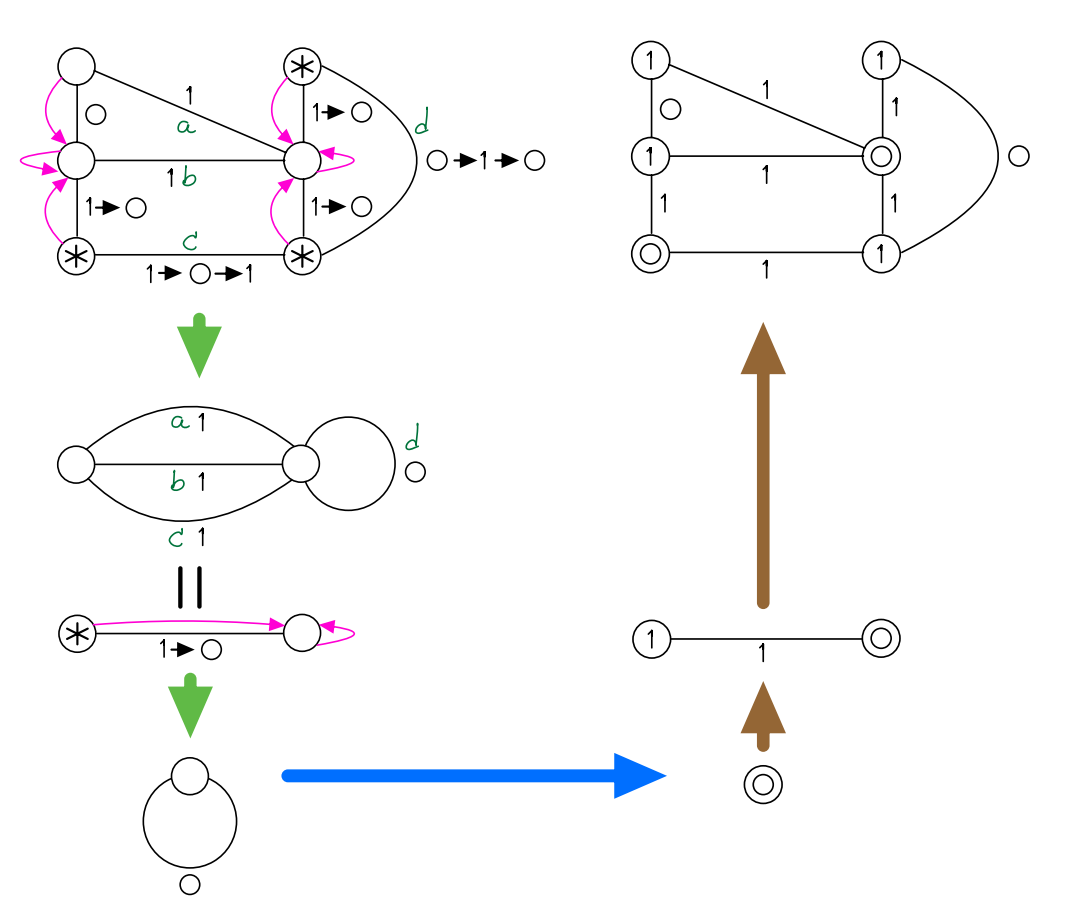
```
if exists (x,y,c), (x,y,d) in Ec2 with c!=d then raise OddCycle
```

Here's one way to do it:

```
Ec3 = sort(Ec2)
violation = tabulate (fn i ->
                        let (x0,y0,c) = Ec3[i]
                        let (x1,y1,d) = Ec3[i+1]
                        (x0,y0) == (x1,y1) && c != d,
                      length(Ec3)-1)
if reduce(fn (a,b) -> a||b, false, violation) then raise OddCycle
```

A sample run of the algorithm appears on the next page.

Example: Sample run of the algorithm



The above diagrams illustrates a run of the algorithm. The original input graph is on the upper right (just ignore the vertex color labels). The upper left diagram shows the parent pointers (in pink) and the `to_toggle` bits (asterisks inside the nodes). Toggling is required on satellite nodes whose edges to their center are labeled with a 1. The labels on the edges change due to the application of the toggling rule to the vertices with stars. (For example edge c changes from 1 to 0 and then back to 1 because it's toggled twice, once by each of its endpoints.)

The large green arrows show contraction. So the original graph has six vertices. When contracted it has two. Four of the original edges (labeled a, b, c, and d) are preserved in the contracted graph. Then the self loop d is removed, and the multi-edges a, b and c are replaced by just one edge. (Note that the multi-edges all have the same label. And the self loop is a 0. Were this not the case the algorithm would terminate due to the discovery of an odd cycle.)

The large blue arrow represents the base case. And the brown ones show the expansion process. In the expansion step, the color of a satellite vertex is always inherited from its center vertex, and then toggled if indicated by the `to_toggle` bit (shown as asterisks in the diagram).