Parallel And Sequential Data Structures and Algorithms

Star Contraction, Connectivity, Bipartite

Announcements

- **ShortLab due today at 11:59pm**
- **PowerLab out tomorrow, due Monday 13th (after carnival)**

Learning Objectives

- Understand Star Contraction via Star Partition
- Apply Star Contraction to various problems
  - Connected Component
  - Parallel Bipartite Graph Check

Star Contraction

Recall...

Star Partition

Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a "star center." If it comes up tails, then connect it to a neighboring star center!

Input: (V, E, n) representing a graph

Output:

- V_c : the vertices that become star centers ;
- E_c : the edges that remain between the star centers ;
- P : a sequence which maps each vertex of V to its star .

Recall...

Star Partition (Psudocode)

Algorithm (Star Partitioning)

```
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)
```

Recall...

Star Partition

Star Partition Properties:

1. The number of non-isolated vertices decreases (in expectation) by a factor of $\frac{3}{4}$ for each application of the algorithm.
2. The work is $O\big((m+n)log(n)\big)$, and the span is $O\big(log^2(n)\big)$ for a graph of m edges and n vertices.

Star Contraction

Algorithm (Star Contraction): Given a graph $G = (V, E, n)$,

- Apply star partition, get centers (V_c) and satellites
- Contract/replace each center-satellites group by the center
- Result in $G' = (V_c, E_c, n' = |V_c|)$
- Recursively apply on G'
- Expand solution to G' to solution of G

Star Contraction

Algorithm (Generic Star Contraction)

```
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)
```

base: used on graphs with no edges (cannot be contracted any further).

expand: used to expand the result from the contracted case.

Star Contraction

Star Contraction Properties:

1. The number of recursion level is $O\big(log(n)\big)$ with high probability given graph with n vertices. (Star Partition $\frac{3}{4}$ decrease + Skittles Lemma)
2. The work is $O\big((m+n)\ log^2 n\big)$, and the span is $O\big(log^3(n)\big)$ for a graph of m edges and n vertices with constant expansion.

Star Contraction

Star Contraction Cost Bound: The work is $O\big((m+n)\ log^2\ n\big)$, and the span is $O\big(log^3(n)\big)$ for graph of m edges and n vertices with constant expansion.

$$W(n) = W\left(\frac{3}{4}n\right) + O((m+n)log(n))$$

$$S(n) = S\left(\frac{3}{4}n\right) + O\big(log^2(n)\big)$$

Connectivity

Counting Connected Component

- Suppose we are interested in counting the number of connected components in a graph.
- What base and expand should we use?

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

Algorithm (Generic Star Contraction)

```
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)
```

Compute Connected Component

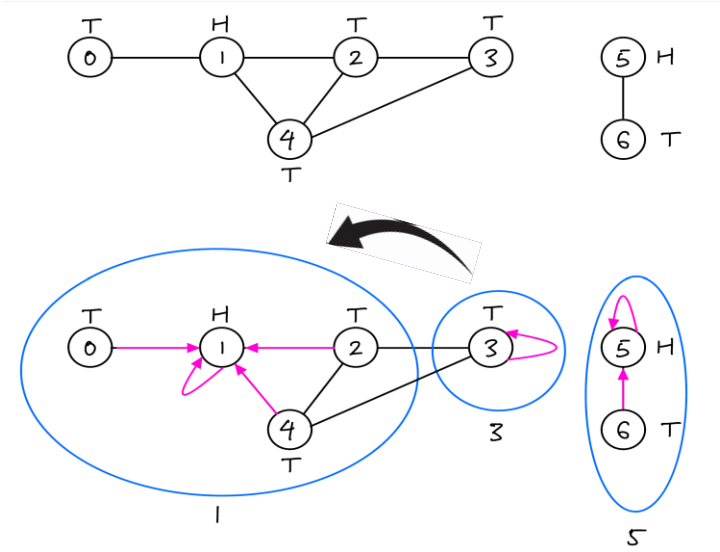
- Now we are interested in compute the connected components in a graph.
- What base and expand should we use?

```
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))
```

Algorithm (Generic Star Contraction)

```
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)
```

Compute Connected Component



Algorithm (Generic Star Contraction)

```
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)
```

```
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))
```

After 1 round of partition:
V' = [1,3,5]
E' = [(1,3),(3,1),(1,3),(3,1)]
P = [1,1,1,3,1,5,5]

After 2 round of partition:
V' = [1,5]
E' = []
P = [0,1,2,1,4,5,6]

1st round contraction returns:
(V=[0,1,…,6}, [1,1,1,1,1,5,5])

2nd round contraction returns:
(V=[1,3,5], M=[0,1,2,1,4,5,6])

Base returns:
(V=[1,5], M=[0,1,2,3,4,5,6])

Bipartite Graph

Recall...

Bipartite Graph

Definition (Bipartite Graph): a graph is bipartite if its vertices can be partitioned into two sets ***A*** and ***B*** such that all the edges are between ***A*** and ***B*** (i.e. has one endpoint in ***A*** and the other in ***B***).

Bipartite Graph Coloring: given a bipartite graph ***G***, there is a way of coloring each vertex from a set of two colors, such that there are no edges with both vertices having the same color (i.e. monochromatic edges).

Parallel Bipartite Check

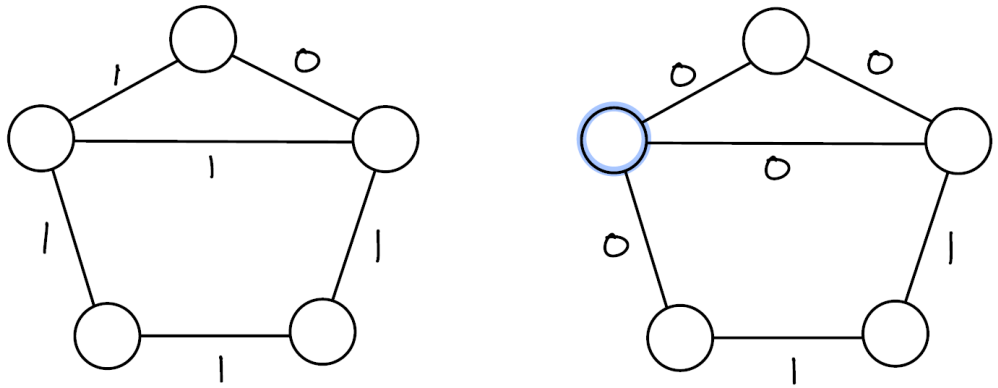
- There exists simple linear time algorithms using BFS and DFS to determine if a graph is bipartite
  - Highly sequential with high span
- Our goal here is polylogarithmic span
- Star Partition!!!

Parallel Bipartite Check

Definition (Generalized Bipartiteness): given a graph G where each edge is labeled with a number from the set $\{0, 1\}$, it is generalized bipartite if there exists a coloring for the vertices with $\{0, 1\}$, such that for an edge with label 1, its two endpoints must be **different** colors, and if for an edge with label 0 the two endpoints must be the **same** color.

Parallel Bipartite Check

Generalized Bipartite Toggling Rule: Consider G' obtained from G by picking a vertex v and toggling the labels of all the edges incident to v. There is a valid 2-coloring of G *if and only if* 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.

Parallel Bipartite Check

```
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
```

Parallel Bipartite Check

```
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))
```

“Each satellite vertex whose edge to its center vertex is labeled with 1 is toggled.”

Parallel Bipartite Check

"Check if multi-edges have consistent labels before returning the mapped set of edges."

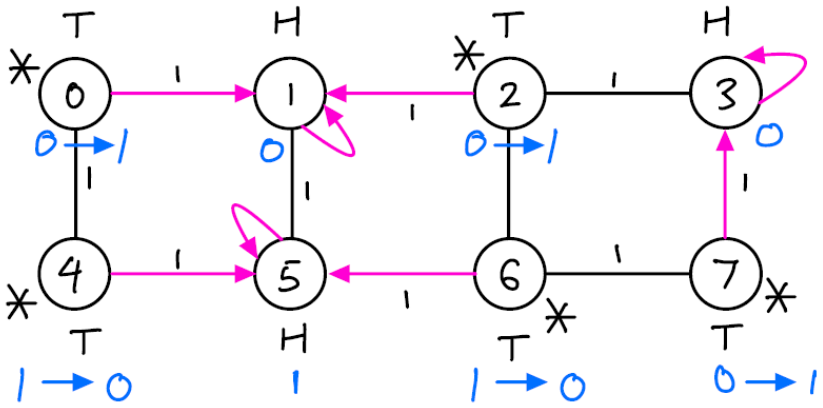
```
fun edges_after_contraction(E, P, to_toggle)-> sequence<int,int,int>:
  E1 = map (fn (x,y,c) => if to_toggle[x] == to_toggle[y]
                          then (x,y,c) else (x,y,1-c), E)
     // E1 is all the edges, but with the necessary toggling
  E2 = filter (fn (x,y,c) -> P[x]==P[y] && c==1, E1)
     // 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 ∃ (x,y,c) and (x,y,d) in Ec2 with c!=d then raise OddCycle
  return unique(sort(Ec2)) // get rid of multi-edges
```

Parallel Bipartite Check

Parallel Bipartite Check Work/Span:

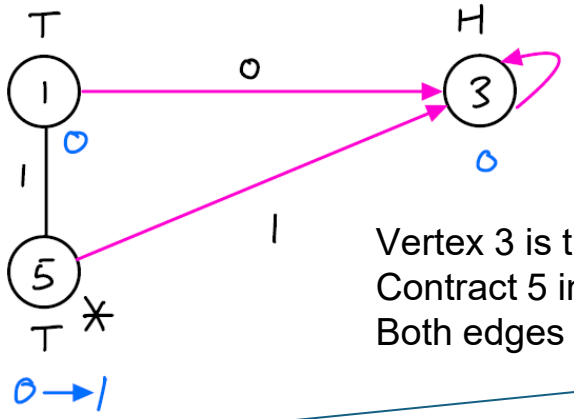
1. The number of recursion level is $O\big(log(n)\big)$ with high probability.
2. For each level, the work is $O\big((m+n)\ log\ n\big)$ and span is $O\big(log^2(n)\big)$ whp (from Star Partition).
3. The overall work is $O\big((m+n)\ log^2\ n\big)$ and span is $O\big(log^3(n)\big)$, whp.

Example Run



Move up another level:
The colors of 0, 2, 6, 4, and 7 are inherited from their centers. But then they're all toggled.

Move up 1 level:
Vertices 1 and 5 inherit the color of their center (0).
Then the color of vertex 5 is toggled to 1.



Next level down of recursion:
No edges → returns a coloring with vertex 3 colored 0.

Vertex 3 is the star center and vertices 1 and 5 are the satellites.
Contract 5 into 3 → toggling rule to vertex 5.
Both edges into 5 to have their labels changed to 0.

Summary

- **Star Contraction** is cool !
- Can be applied to problems to achieve polylogarithmic span