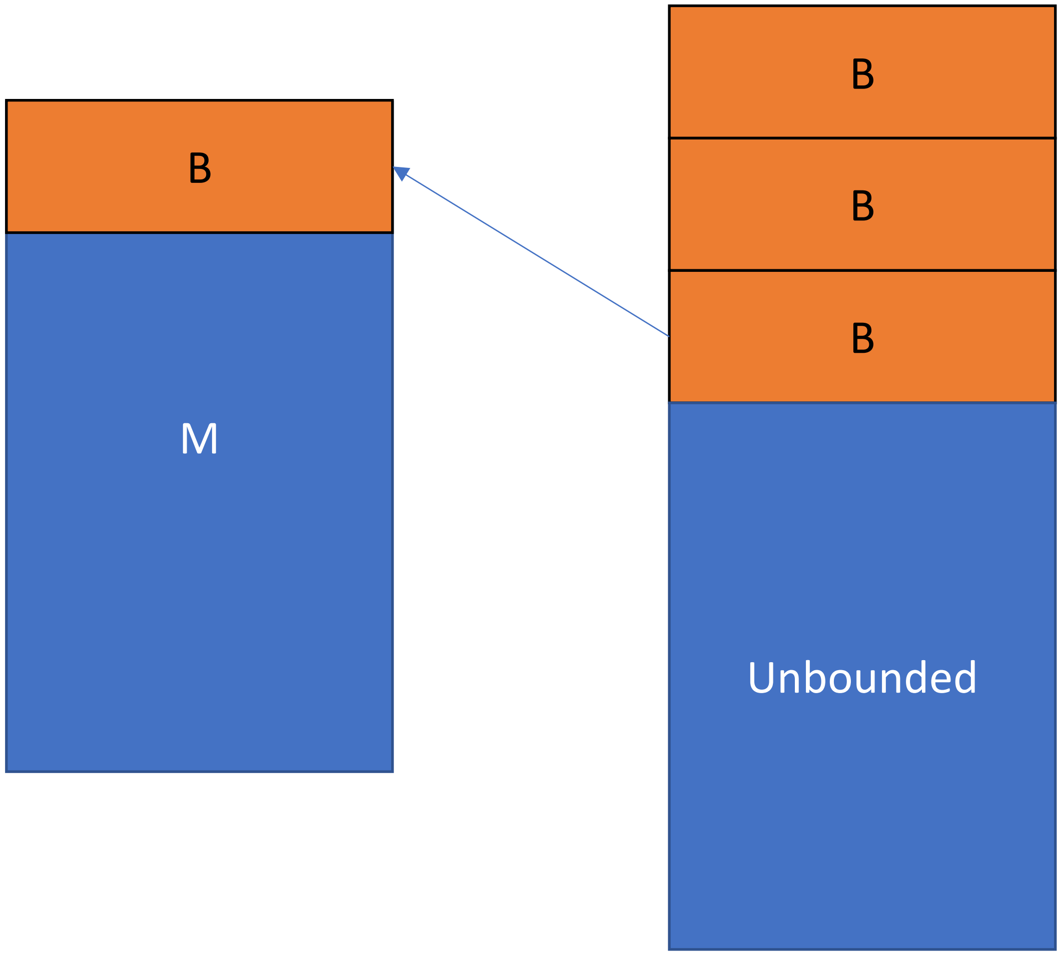
A Functional Approach to External Graph Algorithms

J. Abello, A. L. Buchsbaum, and J. R. Westbrook

Algorithmica 2002 | DOI: 10.1007/s00453-001-0088-5

External Memory model

- N = number of items in the instance
- M = number of items that can be fit in main memory
- B = number of items per block



- M/B-way merge sort Complexity: $O((N/B) \log_{M/B} (N/B))$
- Scan Complexity: $O(N/B)$

Motivation

- What is a functional model?
  - Inputs cannot be changed (Lisp)
- Why a functional approach?
  - Checkpointing
    - Only have to change the file-descriptor table, no copying required
    - Increases reliability
  - General programming language optimizations can be applied
  - Random memory accesses are also reduced
- PRAM Simulation (Chiang et al.)
  - Simulating PRAM steps using only one processor and an external disk
  - Impractical, requires both a practical PRAM algorithm and an implementation of the external memory simulation
- Buffer data structures (Arge, Kumar & Schwabe, Fadel, etc.)
  - Maintain buffer tree, operation is performed by adding to the root node of a buffer
  - Hard to implement and checkpointing is expensive

Problems

- Connected Components
  - Maximal set of vertices such that each pair of vertices is connected by some path
- Minimum Spanning Forests
  - MST for disconnected graphs
- Bottleneck minimum spanning forests
  - Minimize maximum edge weight
- Maximal matching
  - Maximal set of edges such that no two edges share a common vertex
- Maximal independent set
  - Maximal set of vertices such that no two vertices are adjacent

Functional Graph Transformations

Selection, Relabeling, Contraction, Vertex/Edge deletion

Selection

Select(I, k) returns the k’th biggest element from I
Briefly: identical to median finding algorithm from 6.046

1. Partition I into cM-element subsets, for some $0 < c < 1$.
2. Determine the median of each subset in main memory. Let S be the set of medians of the subsets. <-- One scan
3. $m \leftarrow \texttt{Select}(S, \lceil S/2 \rceil)$.
4. Let I_1, I_2, I_3 be the sets of elements less than, equal to, and greater than m, respectively. <-- One scan
5. If $|I_1| \geq k$, then return $\texttt{Select}(I_1, k)$. k’th largest element is in I1
6. Else if $|I_1| + |I_2| \geq k$, then return m. m is k’th largest element
7. Else return $\texttt{Select}(I_3, k - |I_1| - |I_2|)$. k’th largest element is in I3

<-- Recursive scans only run on at most ¾ elements in I

$$T(|I|) \leq 2 \cdot scan(I) + T(|I|/cM) + T(3|I|/4); \text{ by induction, } T(|I|) = O(scan(|I|)),$$

Relabeling - $O(sort(|I|) + sort(|F|))$

- Given a rooted forest *F* as an unordered sequence of oriented tree edges {(p(v), v), …} and an edge set *I* (not necessarily the same edges in F), the relabel operation replaces each edge (u, v) with its respective parent (if it exists) in *F*.

1. Sort F by source vertex, v. — $sort(|F|)$ I/Os
2. Sort I by second component. — $sort(|I|)$ I/Os
3. Process F and I in tandem. — $scan(|F| + |I|)$ I/Os
   (a) Let $\{s, h\} \in I$ be the current edge to be relabeled.
   (b) Scan F starting from the current edge until finding $(p(v), v)$ such that $v \geq h$.
   (c) If $v = h$, then add $\{s, p(v)\}$ to I''; otherwise, add $\{s, h\}$ to I''.
4. Repeat steps 2 and 3, relabeling first components of edges in I'' to construct I'.

Relabeling

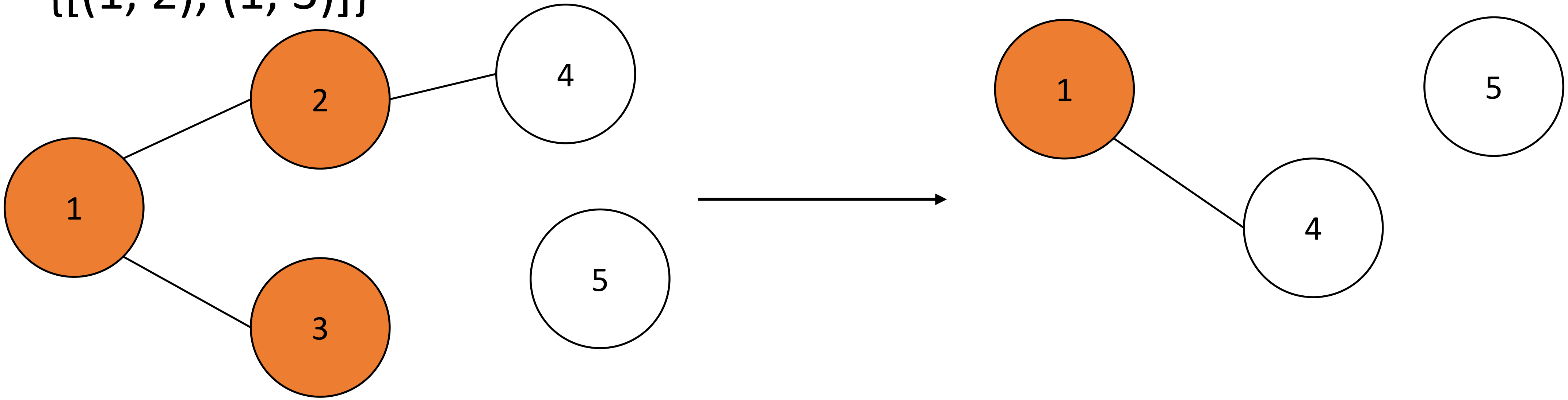
```python
def relabel(F, I):
    # 1
    F.sort(key = lambda x : x[1])
    # 2
    I.sort(key = lambda x : x[1])
    # 3
    Iii = []
    for s, h in I: # a
        flag = 0
        for p, v in F: # b
            if v == h:
                Iii.append((s, p)) # c
                flag = 1
                break
        if flag == 0:
            Iii.append((s, h)) # c

    # 4
    Iii.sort(key = lambda x : x[0])
    Ii = []
    for s, h in Iii:
        flag = 0
        for p, v in F:
            if v == s:
                Ii.append((p, h))
                flag = 1
                break
        if flag == 0:
            Ii.append((s, h))
    return Ii
```

1. Sort F by source vertex, v. — Sort($|F|$) I/Os
2. Sort I by second component. — Sort($|I|$) I/Os
3. Process F and I in tandem. — Scan($|F| + |I|$) I/Os
   (a) Let $\{s, h\} \in I$ be the current edge to be relabeled.
   (b) Scan F starting from the current edge until finding $(p(v), v)$ such that $v \geq h$.
   (c) If $v = h$, then add $\{s, p(v)\}$ to I''; otherwise, add $\{s, h\}$ to I''.
4. Repeat steps 2 and 3, relabeling first components of edges in I'' to construct I'.

In English: iterate through all edges in I. For each edge (u, v) check if u or v have valid parents in F. If they do, replace u, v with their respective parents. If not, don’t replace.

Contraction

- A subcomponent is a collection of edges among vertices in the same connected component of G that aren't necessarily maximal. A contraction of G by C is G/C, the vertices of each subcomponent are contracted into a supervertex.
- {[(1, 2), (1, 3)]}

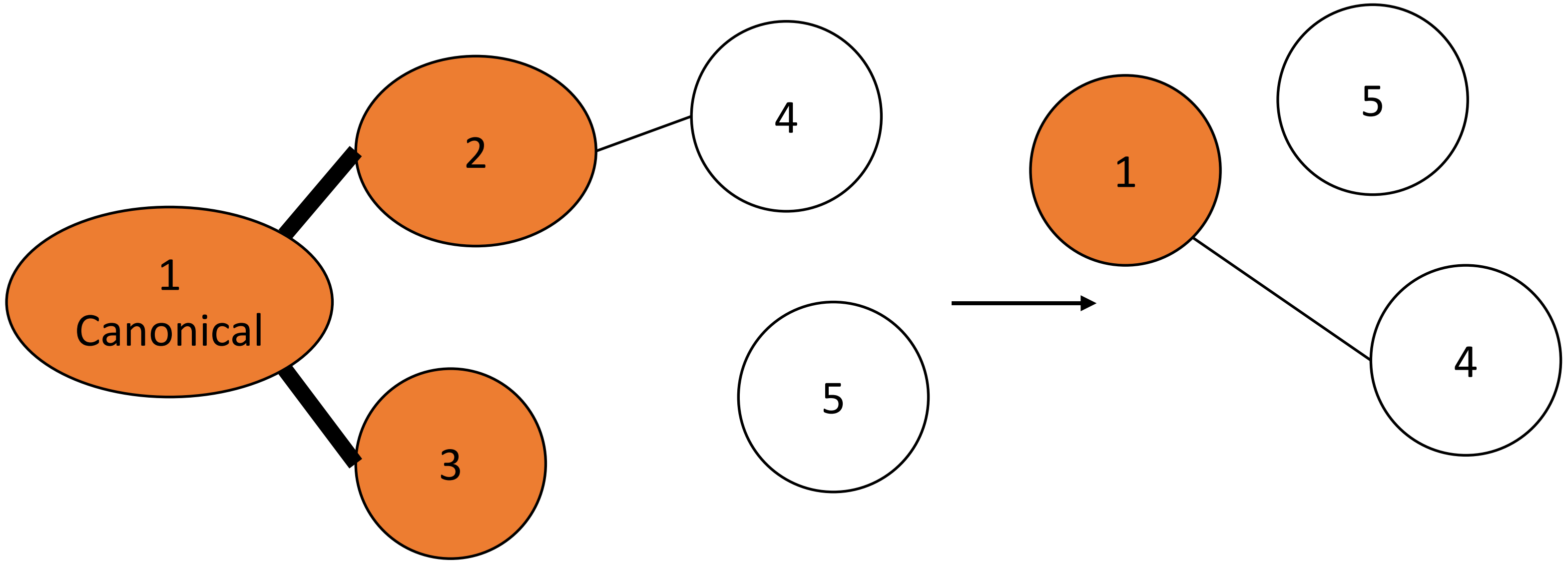


```
def contract(graph, edges):
  # create subcomponents from edges
  subcomponent_map = {}
  subcomponents = []
  for edge in edges:
    x = min(edge[0], edge[1])
    y = max(edge[0], edge[1])
    if x in subcomponent_map:
      subcomponent_map[x].append(y)
    else:
      subcomponent_map[x] = [y]

  # create subcomponent maps
  for x in subcomponent_map:
    vertex = []
    for y in subcomponent_map[x]:
      vertex.append((x, y))
    subcomponents.append(vertex)

  # create R_i
  relabelling_forest = set([])
  for component in subcomponents:
    canonical_vertex = component[0][0]
    for edge in component:
      relabelling_forest.add((canonical_vertex, edge[0]))
      relabelling_forest.add((canonical_vertex, edge[1]))

  # relabel
  rl = relabel(list(relabelling_forest), graph.edges)

  # remove self edges
  contract = []
  for edge in rl:
    if edge[0] != edge[1]:
      contract.append(edge)

  return Graph(contract)
```

Contraction – O(*scan*(|*I*|)) I/Os

1. For each $C_i = \{\{u_1, v_1\}, \ldots\}$:
   (a) $R_i \leftarrow \emptyset$.
   (b) Pick u_1 to be the canonical vertex.
   (c) For each $\{x, y\} \in C_i$, add (u_1, x) and (u_1, y) to relabeling R_i.
2. Apply relabeling $\bigcup_i R_i$ to I, yielding the contracted edge list I'.

RL(forest = {(1, 1), (1, 2), (1, 3)}, I = all the edges)



Vertex/Edge deletion

- Edge deletion:
  - I \ D: sort I and D lexicographically
  - trivial filter: $sort(|I|)$ + sort(N) I/Os
- Vertex deletion:
  - Create edge list from vertex list: $I'' = \{\{u, v\} \in I : u \notin U \wedge v \notin U\}$
  - Same as before, sort and filter: $sort(|I|)$ + sort(N) I/Os

Creating algorithms with this framework

Deterministic Algorithms

Connected Components

```
def CC(G):
    if len(G.edges) == 1:
        return [G.edges[0]]

    #1
    G1 = Graph(G.edges[:len(G.edges)//2])
    #2
    cc_g1 = CC(G1)

    #3
    g_prime = contract(G, cc_g1)
    remaining_edges = []
    for edge in g_prime.edges:
        if edge not in G.edges[:len(G.edges)//2]:
            remaining_edges.append(edge)

    G2 = Graph(remaining_edges)
    #4
    cc_g_prime = CC(G2)
    #5
    return edge_union(cc_g_prime, relabel(cc_g_prime, cc_g1))
```

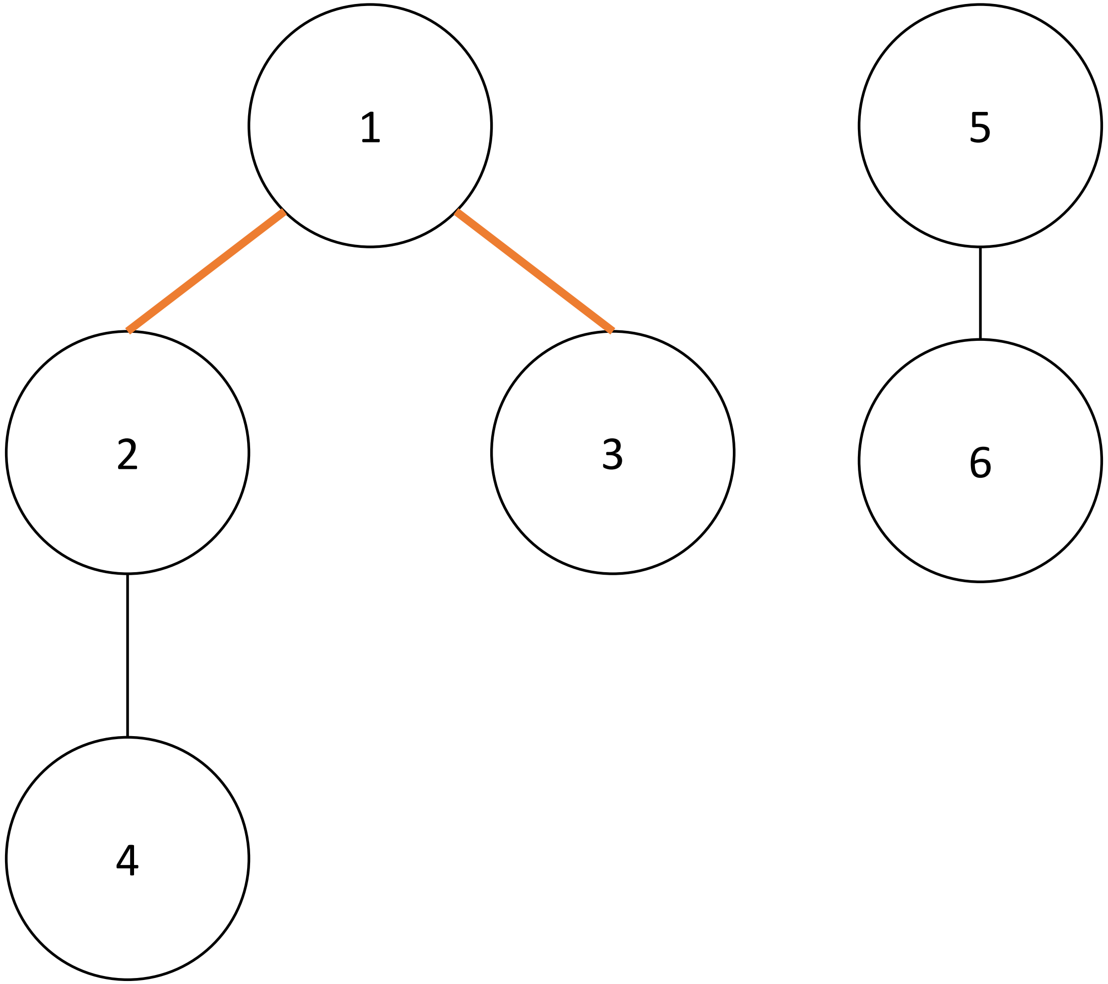
Algorithm CC

1. Let E_1 be any half of the edges of G; let $G_1 = (V, E_1)$.
2. Compute $CC(G_1)$ recursively.
3. Let $G' = G/CC(G_1)$.
4. Compute $CC(G')$ recursively.
5. $CC(G) = CC(G') \cup RL(CC(G'), CC(G_1))$.

Step 1: $O(scan(|E|))$
Step 3: $O(sort(|E|))$
Step 5: $O(sort(|E|))$

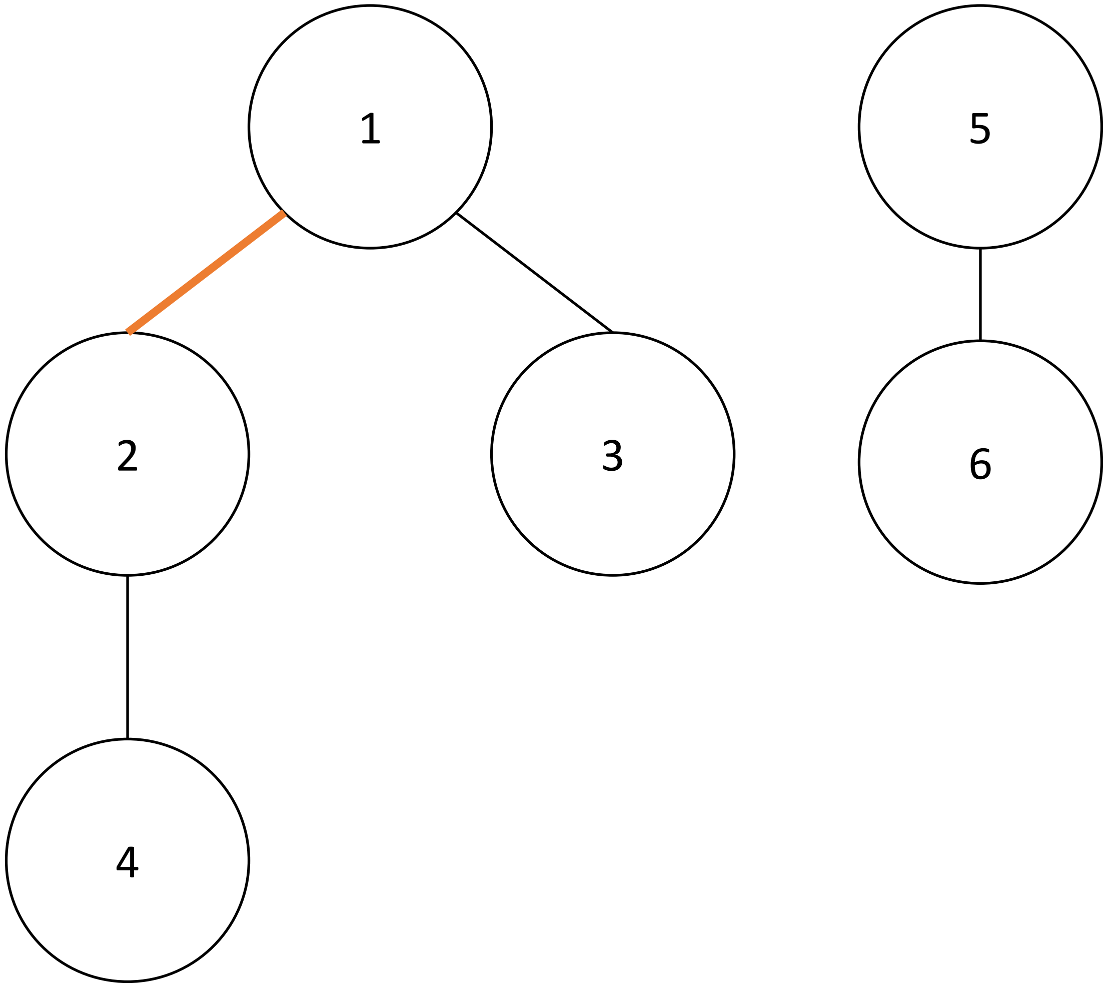
$T(E) <= O(sort(|E|)) + 2T(E/2)$
$T(E) = O(sort(|E|\ log_2\ (E/M))$

Example: Level 1



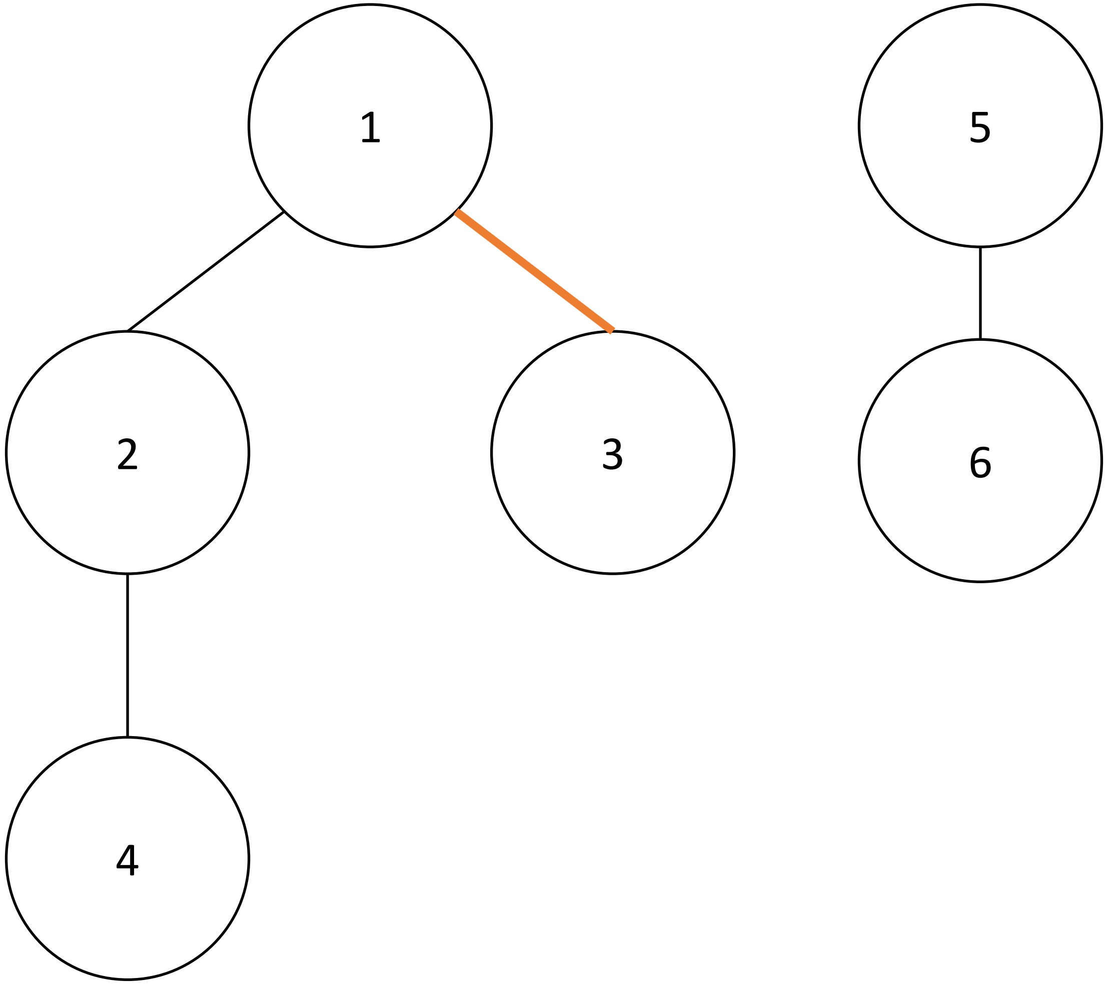

```python
def CC(G):
  if len(G.edges) == 1:
    return [G.edges[0]]

  #1
  G1 = Graph(G.edges[:len(G.edges)//2])
  #2
  cc_g1 = CC(G1)

  #3
  g_prime = contract(G, cc_g1)
  remaining_edges = []
  for edge in g_prime.edges:
    if edge not in G.edges[:len(G.edges)//2]:
      remaining_edges.append(edge)

  G2 = Graph(remaining_edges)
  #4
  cc_g_prime = CC(G2)
  #5
  return edge_union(cc_g_prime, relabel(cc_g_prime, cc_g1))
```

Example: Level 2



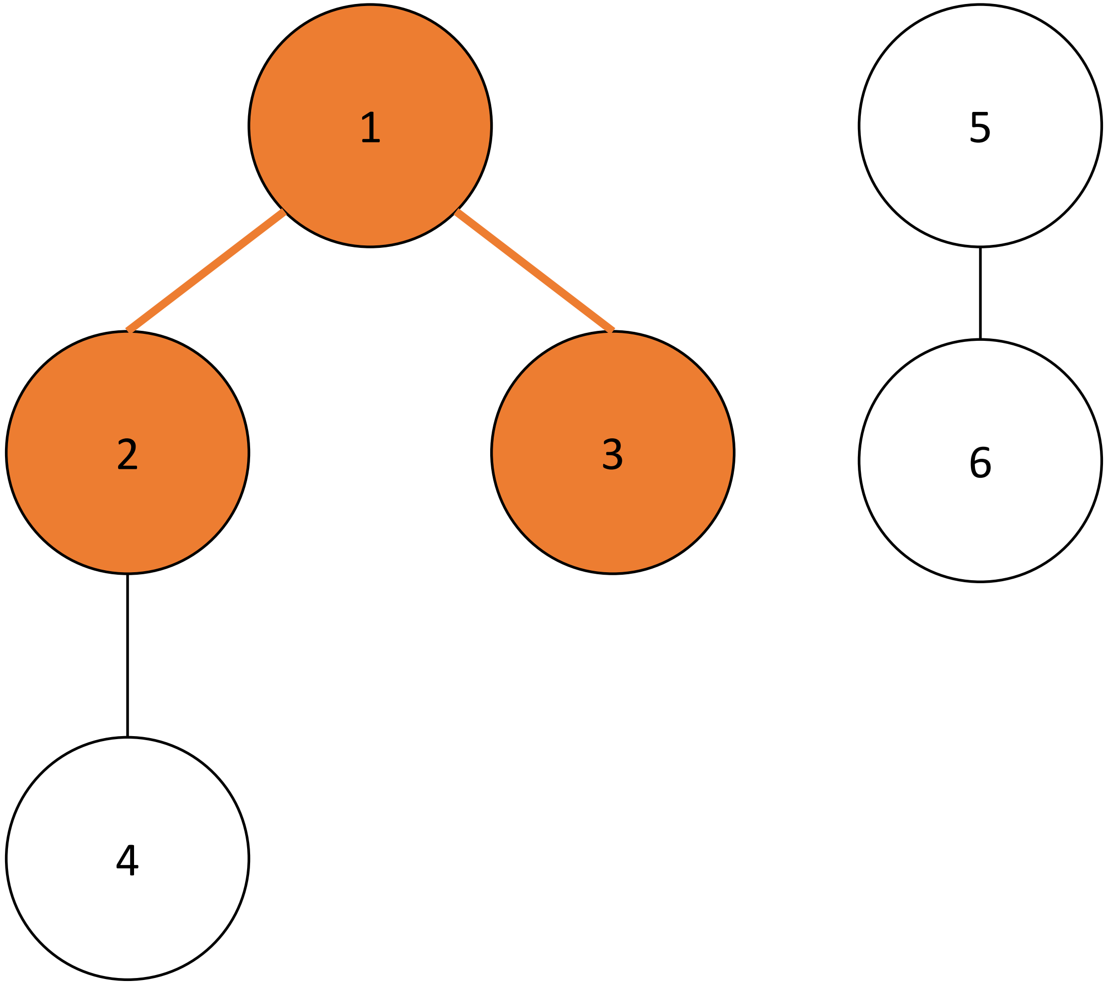

```
def CC(G):
    if len(G.edges) == 1:
        return [G.edges[0]]

    #1
    G1 = Graph(G.edges[:len(G.edges)//2])
    #2
    cc_g1 = CC(G1)

    #3
    g_prime = contract(G, cc_g1)
    remaining_edges = []
    for edge in g_prime.edges:
        if edge not in G.edges[:len(G.edges)//2]:
            remaining_edges.append(edge)

    G2 = Graph(remaining_edges)
    #4
    cc_g_prime = CC(G2)
    #5
    return edge_union(cc_g_prime, relabel(cc_g_prime, cc_g1))
```

Example: G’, CC(G’)



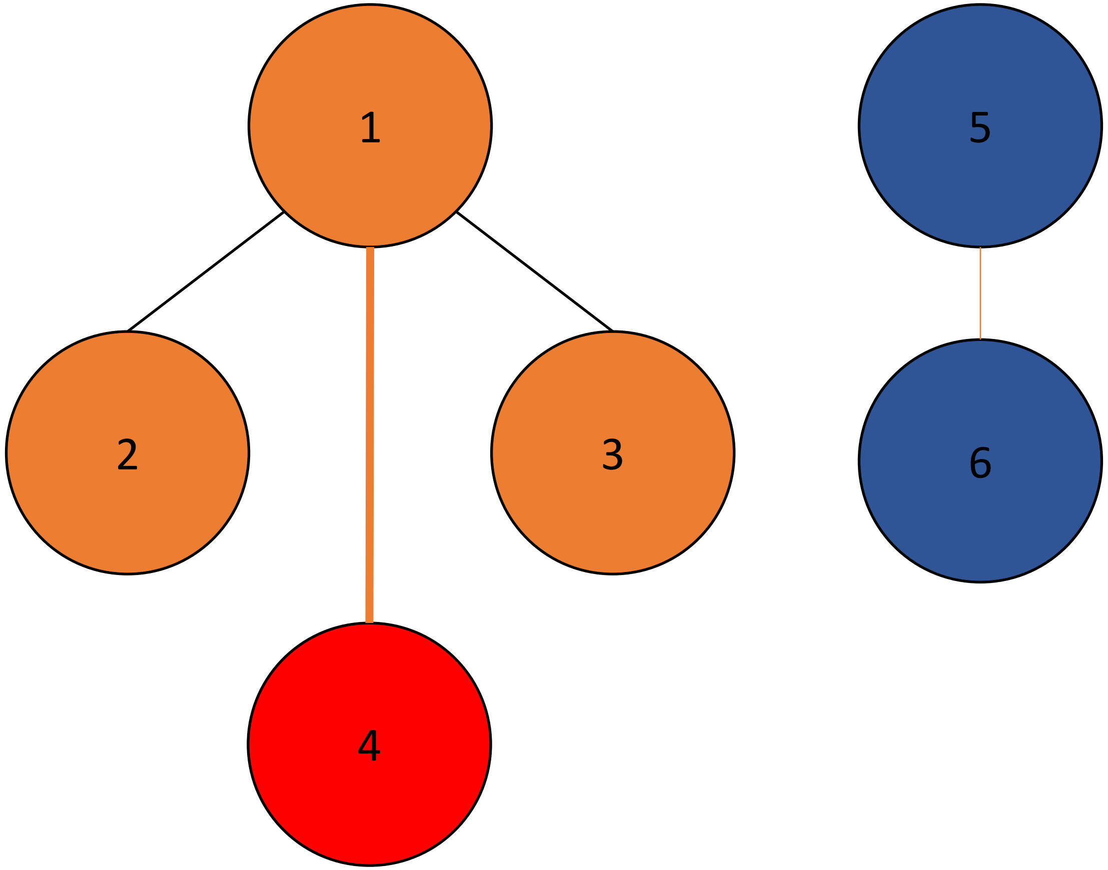

```python
def CC(G):
  if len(G.edges) == 1:
    return [G.edges[0]]

  #1
  G1 = Graph(G.edges[:len(G.edges)//2])
  #2
  cc_g1 = CC(G1)

  #3
  g_prime = contract(G, cc_g1)
  remaining_edges = []
  for edge in g_prime.edges:
    if edge not in G.edges[:len(G.edges)//2]:
      remaining_edges.append(edge)

  G2 = Graph(remaining_edges)
  #4
  cc_g_prime = CC(G2)
  #5
  return edge_union(cc_g_prime, relabel(cc_g_prime, cc_g1))
```

Example: CC(G') U RL(CC(G'), CC(G1))




```
def CC(G):
  if len(G.edges) == 1:
    return [G.edges[0]]

  #1
  G1 = Graph(G.edges[:len(G.edges)//2])
  #2
  cc_g1 = CC(G1)

  #3
  g_prime = contract(G, cc_g1)
  remaining_edges = []
  for edge in g_prime.edges:
    if edge not in G.edges[:len(G.edges)//2]:
      remaining_edges.append(edge)

  G2 = Graph(remaining_edges)
  #4
  cc_g_prime = CC(G2)
  #5
  return edge_union(cc_g_prime, relabel(cc_g_prime, cc_g1))
```

Example: G’ after contraction




```
def CC(G):
    if len(G.edges) == 1:
        return [G.edges[0]]

    #1
    G1 = Graph(G.edges[:len(G.edges)//2])
    #2
    cc_g1 = CC(G1)

    #3
    g_prime = contract(G, cc_g1)
    remaining_edges = []
    for edge in g_prime.edges:
        if edge not in G.edges[:len(G.edges)//2]:
            remaining_edges.append(edge)

    G2 = Graph(remaining_edges)
    #4
    cc_g_prime = CC(G2)
    #5
    return edge_union(cc_g_prime, relabel(cc_g_prime, cc_g1))
```

Sparsification

- Partition E into E/V lists of no more than V edges each.
- Then, we get from this: $O(sort(E)\log_2(E/M))\ I/Os$
- To: $O((E/V)sort(V)\log_2(V/M))$

- This is better since the number of edges is usually way more than the number of vertices

MSF, MM, MIS

1. $G_1 \leftarrow S(G)$;
2. $G_2 \leftarrow T_1(G, f_{\mathcal{P}}(G_1))$;
3. $f_{\mathcal{P}}(G) = T_2(G, G_1, G_2, f_{\mathcal{P}}(G_1), f_{\mathcal{P}}(G_2))$.

Algorithm **MM**

1. Let E_1 be any non-empty, proper subset of edges of G; let $G_1 = (V, E_1)$.
2. Compute $MM(G_1)$ recursively.
3. Let $E' = E \backslash V(MM(G_1))$; let $G' = (V, E')$.
4. Compute $MM(G')$ recursively.
5. $MM(G) = MM(G') \cup MM(G_1)$.

Algorithm **MSF**

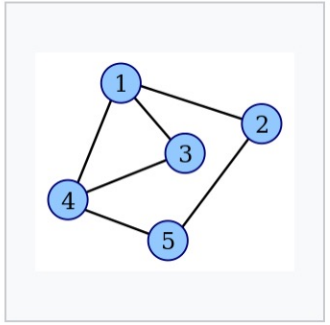
1. Let E_1 be any lowest-cost half of the edges of G; i.e., every edge in $E \backslash E_1$ has weight at least that of the edge of greatest weight in E_1. Let $G_1 = (V, E_1)$.
2. Compute $MSF(G_1)$ recursively.
3. Let $G' = G/MSF(G_1)$.
4. Compute $CC(G')$ recursively.
5. $MSF(G) = EX(MSF(G')) \cup MSF(G_1)$.

Algorithm **CC**

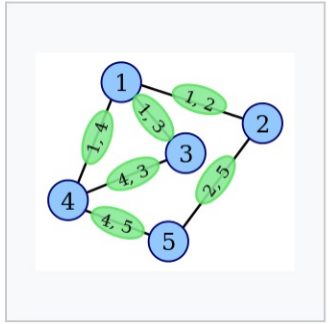
1. Let E_1 be any half of the edges of G; let $G_1 = (V, E_1)$.
2. Compute $CC(G_1)$ recursively.
3. Let $G' = G/CC(G_1)$.
4. Compute $CC(G')$ recursively.
5. $CC(G) = CC(G') \cup RL(CC(G'), CC(G_1))$.

Maximal Independent Set / Maximal Matching

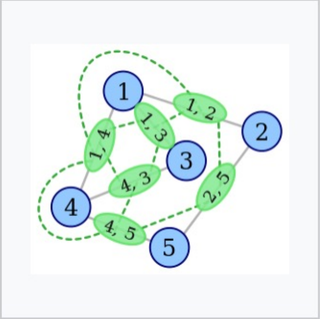
MIS problems can be converted into a MM problem



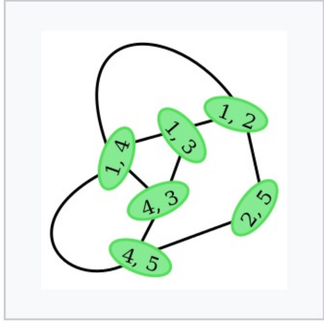
Graph G



Vertices in L(G) constructed from edges in G



Added edges in L(G)



The line graph L(G)

BMSF (Bottleneck MSF)

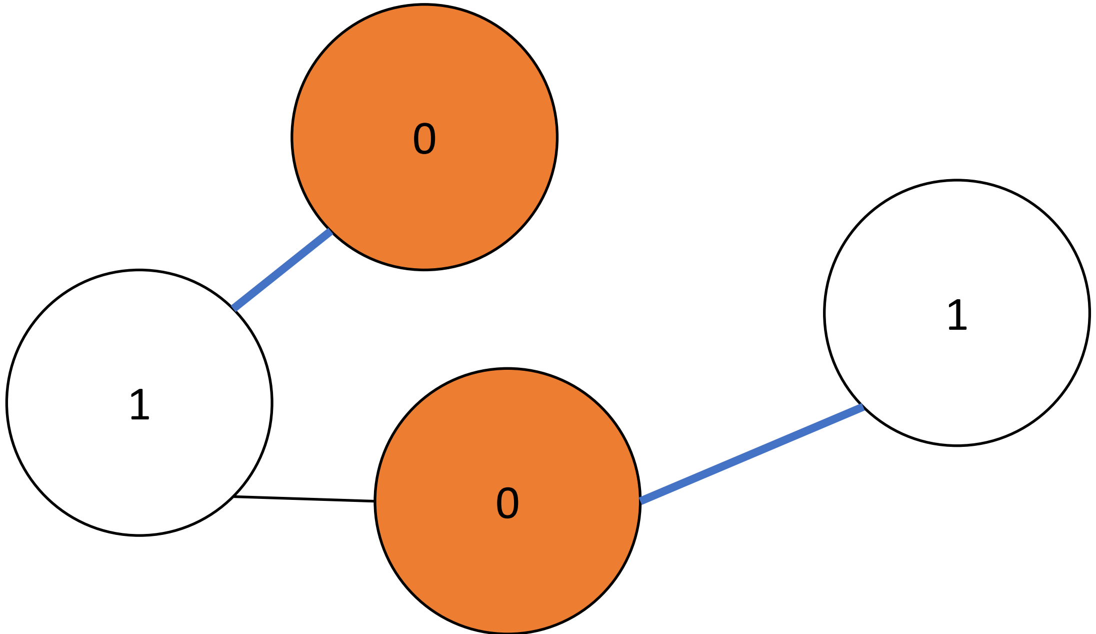
- Computed similarly to MSF, if lower-weighted half of edges span graph then it contains a BMSF.
- Otherwise, it contains an edge from the upper half so the lower half can be contracted
- Divide & conquer again!

Randomized Variants

- Minimum Spanning Forest
- Connected Components
- Maximal Independent Set
- Maximal Matching

Randomized MM

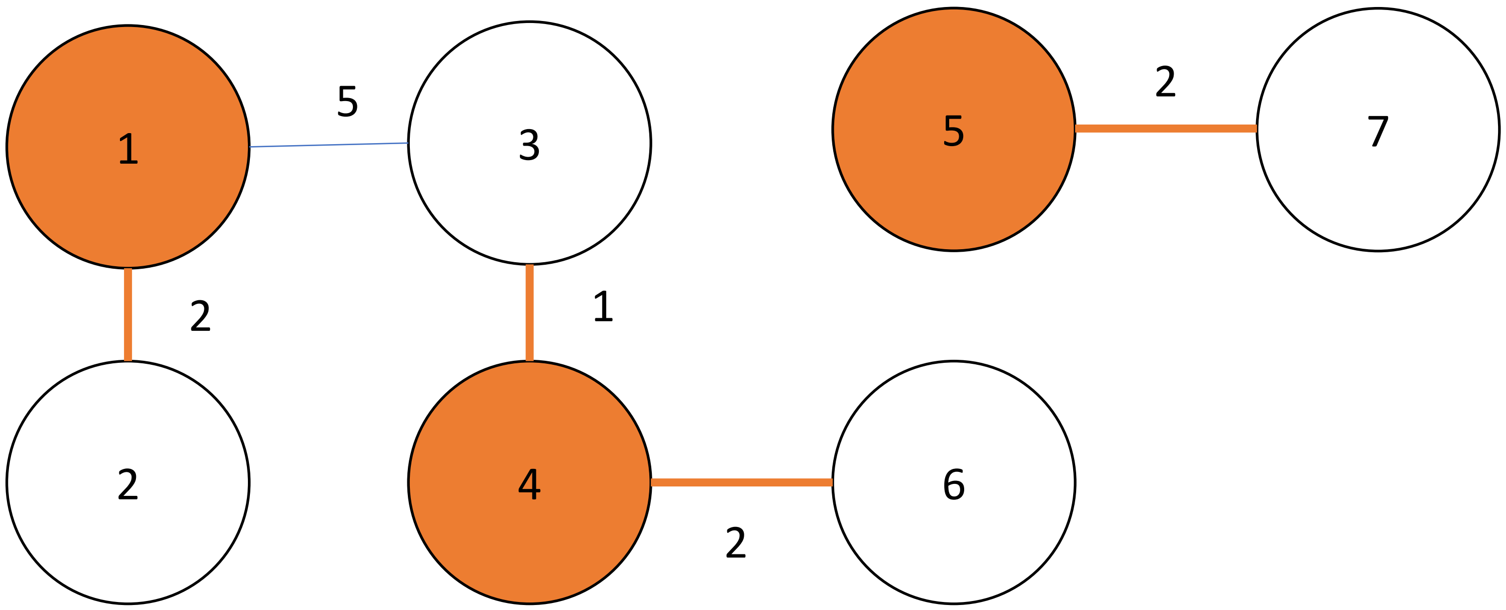
1. $\mathcal{M} \leftarrow \emptyset$.
2. Set the label of v to 0 with probability $1/2$ and to 1 with probability $1/2$, $\forall v \in V$. O(sort(V))
3. For each $u \in V$ such that u is labeled 1, pick any adjacent v such that v is labeled 0. (If u has no adjacent 0-labeled vertex, then u makes no choice.) Let E' be the resulting set of $\{u, v\}$ edges.
4. Let V' be the 0-labeled vertices among the edges in E'. For each $v \in V'$, pick any one incident edge $\{v, w\} \in E'$. (Note that w is labeled 1.) Let E'' be the resulting set of $\{v, w\}$ edges. O(sort(V))
5. $\mathcal{M} \leftarrow \mathcal{M} \cup E''$.
6. $E \leftarrow E \setminus E''$.
7. If $E \neq \emptyset$, repeat from step 2.



Reduces the number of edges by at least $\frac{1}{4}(1 - e^{-1/3})$ and they show that it works in O(sort(E)) with probability 1 - ε

Boruvka Step

- Selects and contracts the edge of the minimum weight incident on each vertex
  - Sort by first component of edge, scan to select minimum weight edge/vertex
  - Sort by second and do the same



Karger Linear-time Randomized MSF/CC

Same as deterministic MSF, divide and conquer on a contracted subgraph except now we expect G’’ to have about *V/4* and *V/8* vertices. We also expect H to have *V/2* vertices

1. Perform two Borůvka steps, which reduces the number of vertices by at least a factor of four. Call the contracted graph G'.
2. Choose a subgraph H of G' by selecting each edge independently with probability $1/2$.
3. Apply the algorithm recursively to find the MSF F of H.
4. Delete from G' each edge $\{u, v\}$ such that (1) there is a path, $P(u, v)$, from u to v in F and (2) the weight of $\{u, v\}$ exceeds that of the maximum-weight edge on $P(u, v)$. Call the resulting graph G''.
5. Apply the algorithm recursively to G'', yielding MSF F'.
6. Return the edges contracted in step 1 together with those in F'.

$O(sort(E))$ *I/Os with probability* $1 - e^{-\Omega(E)}$

Semi-External Problems

- $V <= M, E > M$
- Vertices can fit into main memory but edges can't
- MSF with dynamic trees to maintain internal forest (Kruskal's algorithm)
- CC = label edges by components in one scan and sort edge list to arrange edges by component
- Fast sorting & record/key grouping if number of keys are small
  - $O(scan(N) \log_{M/B} K)$ I/Os

Previous Results

- CC:
  - $O(sort(E) \log_2 (V/M))$ I/Os - Chiang et al. (PRAM)
  - $O(V + sort(E) \log_2 (M/B))$ – Kumar and Schwabe (Buffer Tree)
    - *Abello et al. performs better when* $V < M^2 / B$
  - $O(\max\{1, \log\log VBP/E\} (E/V) sort(V))$ – Munagala and Ranade (Multiset)
    - P is number of parallel disks, performs better than our deterministic one
- *MSF:*
  - $O(sort(E) \log_2 (V/M))$ I/Os - Chiang et al.
  - $O(sort(E) \log_2 (B) + scan(E) \log_2 (V))$ – Kumar and Schwabe
    - *Abello et al. performs better when* $V < M B$
- *MM:*
  - $O(sort(E) \log_2^3 V)$ - Israeli and Shiloach

Results

Problem	Deterministic	Randomized	
	I/O bound	I/O Bound	With probability
Connected components	$O(sort(E) + \frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - e^{\Omega(E)}$
MSFs	$O(sort(E) + \frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - e^{\Omega(E)}$
BMSFs	$O(sort(E) + \frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - e^{\Omega(E)}$
Maximal matchings	$O(\frac{E}{V} sort(V) \log_2 \frac{V}{M})$	$O(sort(E))$	$1 - \varepsilon$ for any fixed ε
Maximal independent sets		$O(sort(E))$	$1 - \varepsilon$ for any fixed ε