Graphs III

Minimum Spanning Trees (MST)

Laura Toma
Algorithms (csci2200), Bowdoin College

Minimum Spanning Tree (MST)

Problem: Given an undirected, weighted, connected graph G, compute a spanning tree of minimum weight.

where

- Spanning tree: a subgraph of G that is a tree and contains all vertices of G.
- The weight of a tree T is the sum of the weights of its edges:

$$w(T) = \sum_{(u,v) \in T} w_{u,v}$$

Note: G needs to be connected to admit a ST. If not, first find its CCs, and then find an MST for each component → minimum spanning forest (MSF).

Minimum Spanning Tree (MST)

Two approaches, both greedy:

Minimum Spanning Tree (MST)

Two approaches, both greedy:

- Kruskal's algorithm:
  Start with an empty tree T. Consider the edges in G in increasing order of weight. Add edges to T in order of weight unless doing so would create a cycle.

Minimum Spanning Tree (MST)

Two approaches, both greedy:

- Kruskal's algorithm:
  Start with an empty tree T. Consider the edges in G in increasing order of weight. Add edges to T in order of weight unless doing so would create a cycle.

- Prim's algorithm:
  Start with an empty tree T. Greedily grow T one edge at a time. At each step, add the edge of minimum weight that has exactly one endpoint in T.

Prim's MST algorithm

Idea:

- Start with a tree T containing an arbitrary vertex r and no edges
- Grow T by repeatedly adding minimum-weight edge connecting a vertex in the current T with a vertex not in T

Prim's MST algorithm

Idea:

- Start with a tree T containing an arbitrary vertex r and no edges
- Grow T by repeatedly adding minimum-weight edge connecting a vertex in the current T with a vertex not in T

Implementation:

- To find mimimum-weight edge connected to current T we maintain a priority queue on vertices not in T.
- The priority of a vertex is the weight of the mimimum-weight edge connecting v to the tree.

Prim's MST algorithm

- pick arbitrary vertex r
- Initialize:
  For each $v \in V$, Insert(PQ, (v,∞)).
  Decrease-Key(PQ, r, 0).
- while PQ not empty do:
  - u = Delete-Min(PQ)
  - output the edge $(u, visit(u))$ as part of MST
  - for each $(u, v) \in E$ do:
    - if $v \in PQ$ and $w_{u,v} < key(v)$ then
      $visit(v) = u$
      Decrease-Key(PQ, v, w_{uv})

Prim's MST algorithm

- pick arbitrary vertex r
- Initialize:
  For each $v \in V$, Insert(PQ, (v,∞)).
  Decrease-Key(PQ, r, 0).
- while PQ not empty do:
  - u = Delete-Min(PQ)
  - output the edge $(u, visit(u))$ as part of MST
  - for each $(u, v) \in E$ do:
    - if $v \in PQ$ and $w_{u,v} < key(v)$ then
      $visit(v) = u$
      Decrease-Key(PQ, v, w_{uv})

Analysis: $|V|$ Insert, $|V|$ Delete-Min, $|E|$ Decrease-Key

Prim's MST algorithm

- pick arbitrary vertex r
- Initialize:
  For each $v \in V$, Insert(PQ, (v,∞)).
  Decrease-Key(PQ, r, 0).
- while PQ not empty do:
  - u = Delete-Min(PQ)
  - output the edge $(u, visit(u))$ as part of MST
  - for each $(u, v) \in E$ do:
    - if $v \in PQ$ and $w_{u,v} < key(v)$ then
      $visit(v) = u$
      Decrease-Key(PQ, v, w_{uv})

Analysis: $|V|$ Insert, $|V|$ Delete-Min, $|E|$ Decrease-Key
$\rightarrow O(E \lg V)$ with a heap

Kruskal's MST algorithm

Idea:

- Start with $|V|$ trees, each consisting of one vertex and no edges.
- Consider edges in E in increasing order of weight; add an edge if it connects two trees

Kruskal's MST algorithm

Idea:

- Start with $|V|$ trees, each consisting of one vertex and no edges.
- Consider edges in E in increasing order of weight; add an edge if it connects two trees

Kruskal's MST algorithm

Idea:

- Start with $|V|$ trees, each consisting of one vertex and no edges.
- Consider edges in E in increasing order of weight; add an edge if it connects two trees

Implementation:

- sort E by weight
- How to decide if an edge (u, v) creates a cycle, or connects two trees?

Kruskal's MST algorithm

Idea:

- Start with $|V|$ trees, each consisting of one vertex and no edges.
- Consider edges in E in increasing order of weight; add an edge if it connects two trees

Implementation:

- sort E by weight
- How to decide if an edge (u, v) creates a cycle, or connects two trees?
  Comes down to checking if vertices u, v are in the same tree, or not.

Kruskal's MST algorithm

- Initialize: T consists of all vertices, and no edges
- Sort E by weight
- For each edge (u, v) in order do:
  - if u, v in the same "tree" (i.e. connected in T): skip
  - else: add edge (u, v) to T

Kruskal's MST algorithm

- Initialize: T consists of all vertices, and no edges
- Sort E by weight
- For each edge (u, v) in order do:
  - if u, v in the same "tree" (i.e. connected in T): skip
  - else: add edge (u, v) to T

How to check if u, v are in the same CC of T?

Kruskal's MST algorithm

How to check if u, v are in the same CC of T?

Kruskal's MST algorithm

How to check if u, v are in the same CC of T?
Ideas:
Could run BFS/DFS on T every time.... too slow.
Or...

Kruskal's MST algorithm

How to check if u, v are in the same CC of T?
Ideas:
Could run BFS/DFS on T every time.... too slow.
Or...

We need a data structure that supports:

- Make-Set(v): create set containing v
- Union-Set(u, v): unite set containing u and set containing v
- Find-Set(v): return unique representative for set containing v

Called Union-Find data structure

Kruskal's MST algorithm

- Initialize:
  T consists of all vertices, and no edges
  For each $v \in V$, Make-Set(v)
- Sort E by weight
- For each edge (u, v) in order do:
  - if Find-Set(u) $\neq$ Find-Set(v):
    add edge (u, v) to T
    Union-Set(u, v)

Kruskal's MST algorithm

- Initialize:

  T consists of all vertices, and no edges

  For each $v \in V$, Make-Set(v)
- Sort E by weight
- For each edge (u, v) in order do:
  - if Find-Set$(u) \neq$ Find-Set(v):

    add edge (u, v) to T

    Union-Set(u, v)

Analysis: $E \lg E$ to sort; $|V|$ Make-Set, $2|E|$ Find-Set, $|V| - 1$ Union-Set

A Union-Find structure

- Make-Set(v): create set containing v
- Union-Set(u, v): unite set containing u and set containing v
- Find-Set(v): return unique representative for set containing v

A Union-Find structure

- Make-Set(v): create set containing v
- Union-Set(u, v): unite set containing u and set containing v
- Find-Set(v): return unique representative for set containing v

Simple solution:

A Union-Find structure

- Make-Set(v): create set containing v
- Union-Set(u, v): unite set containing u and set containing v
- Find-Set(v): return unique representative for set containing v

Simple solution:
Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative)

A Union-Find structure

- Make-Set(v): create set containing v
- Union-Set(u, v): unite set containing u and set containing v
- Find-Set(v): return unique representative for set containing v

Simple solution:
Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative)

Find-Set(u) runs in $O(1)$ time, but Union-Set(u, v) needs $O(|V|)$ time.

A Union-Find structure

- Make-Set(v): create set containing v
- Union-Set(u, v): unite set containing u and set containing v
- Find-Set(v): return unique representative for set containing v

Simple solution:
Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative)

Find-Set(u) runs in $O(1)$ time, but Union-Set(u, v) needs $O(|V|)$ time.
$\Rightarrow$ Kruskal's algorithm runs in $O(E \lg V + V^2)$. Too slow.

A Union-Find structure

Supports:

- Make-Set(v), Union-Set(u, v), Find-Set(v)

Refined solution: Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative), and in a union-set, always link the smaller list after the longer list.

A Union-Find structure

Supports:

- Make-Set(v), Union-Set(u, v), Find-Set(v)

Refined solution: Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative), and in a union-set, always link the smaller list after the longer list.

Analysis: We'll count the total nb of pointers that are changed in all calls to Union-Set(). Assume we start with $|V|$ sets.

A Union-Find structure

Supports:

- Make-Set(v), Union-Set(u, v), Find-Set(v)

Refined solution: Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative), and in a union-set, always link the smaller list after the longer list.

Analysis: We'll count the total nb of pointers that are changed in all calls to Union-Set(). Assume we start with $|V|$ sets. How many times can a pointer of an element change?

A Union-Find structure

Supports:

- Make-Set(v), Union-Set(u, v), Find-Set(v)

Refined solution: Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative), and in a union-set, always link the smaller list after the longer list.

Analysis: We'll count the total nb of pointers that are changed in all calls to Union-Set(). Assume we start with $|V|$ sets. How many times can a pointer of an element change? Every time an element changes its pointer, it belongs to a set of at least double the size than what it was before.

A Union-Find structure

Supports:

- Make-Set(v), Union-Set(u, v), Find-Set(v)

Refined solution: Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative), and in a union-set, always link the smaller list after the longer list.

Analysis: We'll count the total nb of pointers that are changed in all calls to Union-Set(). Assume we start with $|V|$ sets. How many times can a pointer of an element change? Every time an element changes its pointer, it belongs to a set of at least double the size than what it was before. Thus, the pointer of an element can change at most $\lg V$ times.

A Union-Find structure

Supports:

- Make-Set(v), Union-Set(u, v), Find-Set(v)

Refined solution: Maintain elements in the same set in a linked list with each element having a pointer to the first element in the list (unique representative), and in a union-set, always link the smaller list after the longer list.

Analysis: We'll count the total nb of pointers that are changed in all calls to Union-Set(). Assume we start with $|V|$ sets. How many times can a pointer of an element change? Every time an element changes its pointer, it belongs to a set of at least double the size than what it was before. Thus, the pointer of an element can change at most $\lg V$ times. Overall, $|V| - 1$ Union-Set() calls will run in $O(V \lg V)$ time.

Union-Find structure

Actually, a much better bound for a Union-Find structure can be obtained:

- use rooted trees (instead of lists)
- the representative of the set containing u: the root of the tree that contains u
- Find-Set(u): go up the path from u to its root
- at Union-Set: connect the smaller tree as a child of the larger tree
- at Find-set: link all nodes on the path to the root as children of the root.

Can be shown $|V|$ operations run in $O(V\alpha(V|)$ time, which is practically $O(|V|)$ time.

MST algorithms

Correctness:

Theorem

Let V_1, V_2 be a partition of V into two disjoint sets, $V_1 \cup V_2 = V$. Consider all edges with one endpoint in V_1 and another one in V_2. Then there is a MST that includes the minimum-weight such edge e.

MST algorithms

Correctness:

Theorem

Let V_1, V_2 be a partition of V into two disjoint sets, $V_1 \cup V_2 = V$. Consider all edges with one endpoint in V_1 and another one in V_2. Then there is a MST that includes the minimum-weight such edge e.

Proof.

Let T be an MST of G. Assume by contradiction T does not include e, then adding e to T creates a cycle. There must be another edge on this cycle that has one endpoint in V_1 and one in V_2. It has weights $\geq e$. Remove it from T and add e instead; this gives a ST of weight $\leq$ than before — contradiction. $\square$

MST algorithms

Theorem

Let V_1, V_2 be a partition of V into two disjoint sets, $V_1 \cup V_2 = V$. Consider all edges with one endpoint in V_1 and another one in V_2. Then there is a MST that includes the minimum-weight such edge e.

Correctness:

Argue that Prim's and Kruskal's algorithms are correct by using the theorem, and chosing the partition appropriately.