CS 473ug: Algorithms

Chandra Chekuri
chekuri@cs.uiuc.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2007

Part I

Problems

Why Algorithms?

World is full of algorithmic problems.

- decision problems (example: given n, *is* n prime?)
- search problems (example: given n, *find a* factor of n if it exists)
- optimization problems (example: find the *fewest* class rooms to schedule all classes)

Why Algorithms?

World is full of algorithmic problems.

- decision problems (example: given n, *is* n prime?)
- search problems (example: given n, *find a* factor of n if it exists)
- optimization problems (example: find the *fewest* class rooms to schedule all classes)

Some problems are provably difficult (example: does my program halt?) and some are easy (example: sorting)

Why Algorithms?

World is full of algorithmic problems.

- decision problems (example: given n, *is* n prime?)
- search problems (example: given n, *find a* factor of n if it exists)
- optimization problems (example: find the *fewest* class rooms to schedule all classes)

Some problems are provably difficult (example: does my program halt?) and some are easy (example: sorting)

Given a new or unfamiliar problem how do we know whether it is easy or not?

Why Algorithms?

World is full of algorithmic problems.

- decision problems (example: given n, *is* n prime?)
- search problems (example: given n, *find a* factor of n if it exists)
- optimization problems (example: find the *fewest* class rooms to schedule all classes)

Some problems are provably difficult (example: does my program halt?) and some are easy (example: sorting)

Given a new or unfamiliar problem how do we know whether it is easy or not?
We don't! In fact one can formally show that this meta-problem is difficult.

What do we do?

What do we do?

- take an algorithms class to learn
  - some standard methods: greedy, divide and conquer, dynamic programming, optimization, reductions, ...
  - some standard problems to use as templates and for use in reductions
  - some standard methods to prove intractability or difficulty of problems (lower bounds) so that we do not waste time looking for an algorithm when there is none

What do we do?

- take an algorithms class to learn
  - some standard methods: greedy, divide and conquer, dynamic programming, optimization, reductions, ...
  - some standard problems to use as templates and for use in reductions
  - some standard methods to prove intractability or difficulty of problems (lower bounds) so that we do not waste time looking for an algorithm when there is none
- creativity in devising new ideas/algorithms

What do we do?

- take an algorithms class to learn
  - some standard methods: greedy, divide and conquer, dynamic programming, optimization, reductions, ...
  - some standard problems to use as templates and for use in reductions
  - some standard methods to prove intractability or difficulty of problems (lower bounds) so that we do not waste time looking for an algorithm when there is none
- creativity in devising new ideas/algorithms
- pay someone else to do it

Problem

What is a problem?

Problem

What is a problem?

Language Decision Problem

- fix some alphabet Σ (say binary).
- problem $\Pi \subseteq \Sigma^*$ (essentially a language)
- Goal: given $x \in \Sigma^*$, is $x \in \Pi$?

Problem

What is a problem?

Language Decision Problem

- fix some alphabet Σ (say binary).
- problem $\Pi \subseteq \Sigma^*$ (essentially a language)
- Goal: given $x \in \Sigma^*$, is $x \in \Pi$?

Example: given an ascii string, is it a valid Java program?

Search and Optimization Problems

- problem Π is a subset of Σ^* (essentially a language)
- given a string $I \in \Sigma^*$, I is an *instance* of Π if $I \in \Pi$
- assumption: given $I \in \Sigma^*$ there is an efficient algorithm to tell if I is an instance or not
- each instance I has a set $sol(I)$ - set of all *feasible solutions* for I
- implicit assumption: given instance I and $y \in \Sigma^*$, some reasonable efficient way to check if $y \in sol(I)$

Decision problem: given I, is $sol(I)$ empty?
Search problem: given I find *some* $y \in sol(I)$.

Optimization Problem

more information!

- a *valuation function* v that for each $y \in sol(I)$ assigns a number $v(y)$
- minimization problem: given I, find $\min_{y \in sol(I)} v(y)$
- maximization problem: given I, find $\max_{y \in sol(I)} v(y)$

Continuous vs Discrete Problems

Computers: discrete input only

Nevertheless, $sol(I)$ can be an infinite continuous set (example: linear programming)

Discrete/Combinatorial problems: $sol(I)$ is a discrete set (potentially infinite)

Combinatorial Optimization Problems

A typical problem:

- instance I consists of a set of objects N
- each object i may have a weight/value w_i
- $sol(I) \subseteq P(N)$ (powerset) — some subsets of N are solutions
- goal: find a set $S \in sol(I)$ to minimize/maximize $w(S) = \sum_{i \in S} w_i$

There is always an exponential time algorithm for such problems. Why?

Part II

Greedy Algorithms via a Strong Exchange Property

A Really Simple Problem

Informal

- Given n items each with a non-negative weight w_i
- Pick at most k items to maximize their total weight

A Really Simple Problem

Informal

- Given n items each with a non-negative weight w_i
- Pick at most k items to maximize their total weight

Formal

- instance I: $(n, w_1, w_2, \ldots, w_n)$ properly encoded as a string
- $sol(I)$: $S \in sol(I)$ iff $S \subseteq \{1, 2, \ldots, n\}$ and $|S| \leq k$
- $v(S) = w(S) = \sum_{i \in S} w_i$

A Really Simple Problem

Informal

- Given n items each with a non-negative weight w_i
- Pick at most k items to maximize their total weight

Formal

- instance I: $(n, w_1, w_2, \ldots, w_n)$ properly encoded as a string
- $sol(I)$: $S \in sol(I)$ iff $S \subseteq \{1, 2, \ldots, n\}$ and $|S| \leq k$
- $v(S) = w(S) = \sum_{i \in S} w_i$

Algorithm?

A Really Simple Problem

Informal

- Given n items each with a non-negative weight w_i
- Pick at most k items to maximize their total weight

Formal

- instance I: $(n, w_1, w_2, \ldots, w_n)$ properly encoded as a string
- $sol(I)$: $S \in sol(I)$ iff $S \subseteq \{1, 2, \ldots, n\}$ and $|S| \leq k$
- $v(S) = w(S) = \sum_{i \in S} w_i$

Algorithm? Sort and pick the k items of largest weight

A Really Simple Problem

Informal

- Given n items each with a non-negative weight w_i
- Pick at most k items to maximize their total weight

Formal

- instance I: $(n, w_1, w_2, \ldots, w_n)$ properly encoded as a string
- $sol(I)$: $S \in sol(I)$ iff $S \subseteq \{1, 2, \ldots, n\}$ and $|S| \leq k$
- $v(S) = w(S) = \sum_{i \in S} w_i$

Algorithm? Sort and pick the k items of largest weight

Why does it work? Assume that weights are distinct

A Really Simple Problem

Informal

- Given n items each with a non-negative weight w_i
- Pick at most k items to maximize their total weight

Formal

- instance I: $(n, w_1, w_2, \ldots, w_n)$ properly encoded as a string
- $sol(I)$: $S \in sol(I)$ iff $S \subseteq \{1, 2, \ldots, n\}$ and $|S| \leq k$
- $v(S) = w(S) = \sum_{i \in S} w_i$

Algorithm? Sort and pick the k items of largest weight

Why does it work? Assume that weights are distinct
Exchange argument: if one of the heaviest k items is not in an optimal solution, put it in and remove some lighter element

Exchange argument in more detail

$sol(I)$ has the property: for every $S \in sol(I)$ and item i

- either $S + i \in sol(I)$ or
- there is some $j \in S$ such that $S + i - j$ is in $sol(I)$.

The weights do not play a role in this exchange property and hence algorithm works for any given weights on the items.

This is true only for a class of problems related to matroids (out of scope for this class)

Note the difference with the interval selection problem

A More Interesting Problem

- Given n items N each with a non-negative weight w_i
- N partitioned into sets $N_1, N_2, \ldots, N_\ell$
- Goal: pick at most k items overall but at most k_j items from N_j for $1 \leq j \leq \ell$.

Example: $k = 4$, $k_1 = 1$, $k_2 = 1$, $k_3 = 3$.

	N_1		N_2			N_3		
Item	1	2	3	4	5	6	7	8
w_i	5	3	10	2	9	1	3	2

A More Interesting Problem

- Given n items N each with a non-negative weight w_i
- N partitioned into sets $N_1, N_2, \ldots, N_\ell$
- Goal: pick at most k items overall but at most k_j items from N_j for $1 \leq j \leq \ell$.

Example: $k = 4$, $k_1 = 1$, $k_2 = 1$, $k_3 = 3$.

	N_1		N_2			N_3		
Item	1	2	3	4	5	6	7	8
w_i	5	3	10	2	9	1	3	2

Optimal solution: $S = \{1, 3, 7, 8\}$ and $w(S) = 20$

A More Interesting Problem: Algorithm

Algorithm?

A More Interesting Problem: Algorithm

Algorithm?

$S \subseteq N$ is *feasible* if $|S| \leq k$ and $|S \cap N_j| \leq k_j$ for $1 \leq j \leq \ell$

A More Interesting Problem: Algorithm

Algorithm?

$S \subseteq N$ is *feasible* if $|S| \leq k$ and $|S \cap N_j| \leq k_j$ for $1 \leq j \leq \ell$

```
sort items and assume w_1 > w_2 > ... > w_n
S = ∅
for i = 1 to n do
    if S + i is feasible then
        S = S + i
end for
return S
```

A More Interesting Problem: Algorithm

Algorithm?

$S \subseteq N$ is *feasible* if $|S| \leq k$ and $|S \cap N_j| \leq k_j$ for $1 \leq j \leq \ell$

```
sort items and assume w_1 > w_2 > ... > w_n
S = ∅
for i = 1 to n do
    if S + i is feasible then
        S = S + i
end for
return S
```

Claim: algorithm gives an optimum solution

A More Interesting Problem: Algorithm

Algorithm?

$S \subseteq N$ is *feasible* if $|S| \leq k$ and $|S \cap N_j| \leq k_j$ for $1 \leq j \leq \ell$

```
sort items and assume w_1 > w_2 > ... > w_n
S = ∅
for i = 1 to n do
    if S + i is feasible then
        S = S + i
end for
return S
```

Claim: algorithm gives an optimum solution

Proof: exchange argument

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Proof.

- Let $i_1, i_2, \ldots, i_p$ be picked by Greedy in order.

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Proof.

- Let $i_1, i_2, \ldots, i_p$ be picked by Greedy in order.
- Fix optimum solution O and suppose O is not $S = \{i_1, i_2, \ldots, i_p\}$.

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Proof.

- Let $i_1, i_2, \ldots, i_p$ be picked by Greedy in order.
- Fix optimum solution O and suppose O is not $S = \{i_1, i_2, \ldots, i_p\}$.
- Let i_j be the first item from S not in O.

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Proof.

- Let $i_1, i_2, \ldots, i_p$ be picked by Greedy in order.
- Fix optimum solution O and suppose O is not $S = \{i_1, i_2, \ldots, i_p\}$.
- Let i_j be the first item from S not in O.
- If $O + i_j$ is feasible, contradicts optimality of O

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Proof.

- Let $i_1, i_2, \ldots, i_p$ be picked by Greedy in order.
- Fix optimum solution O and suppose O is not $S = \{i_1, i_2, \ldots, i_p\}$.
- Let i_j be the first item from S not in O.
- If $O + i_j$ is feasible, contradicts optimality of O
- Lemma: can add i_j to O and remove a lighter item from O

$\square$

Optimality Proof

Theorem

If weights are distinct then there is a unique optimum solution and the Greedy algorithm finds it.

Proof.

- Let $i_1, i_2, \ldots, i_p$ be picked by Greedy in order.
- Fix optimum solution O and suppose O is not $S = \{i_1, i_2, \ldots, i_p\}$.
- Let i_j be the first item from S not in O.
- If $O + i_j$ is feasible, contradicts optimality of O
- Lemma: can add i_j to O and remove a lighter item from O
- Contradicts optimality of O

□

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.
- Suppose $i_j \in N_r$.

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.
- Suppose $i_j \in N_r$.
- If $A \cap N_r \neq \emptyset$ pick t from $A \cap N_r \neq \emptyset$, otherwise pick any t from A.

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.
- Suppose $i_j \in N_r$.
- If $A \cap N_r \neq \emptyset$ pick t from $A \cap N_r \neq \emptyset$, otherwise pick any t from A.
- Claim: $O' = O - t + i_j$ is feasible.

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.
- Suppose $i_j \in N_r$.
- If $A \cap N_r \neq \emptyset$ pick t from $A \cap N_r \neq \emptyset$, otherwise pick any t from A.
- Claim: $O' = O - t + i_j$ is feasible.
- $|O'| = |O| \leq k$.

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.
- Suppose $i_j \in N_r$.
- If $A \cap N_r \neq \emptyset$ pick t from $A \cap N_r \neq \emptyset$, otherwise pick any t from A.
- Claim: $O' = O - t + i_j$ is feasible.
- $|O'| = |O| \leq k$.
- Can it be that $|O' \cap N_r| > k_r$?

Proof of Lemma

Lemma

Can add i_j to O and remove a lighter item from O.

Proof.

- Let $A = O - \{i_1, i_2, \ldots, i_{j-1}\}$. $A \neq \emptyset$.
- Suppose $i_j \in N_r$.
- If $A \cap N_r \neq \emptyset$ pick t from $A \cap N_r \neq \emptyset$, otherwise pick any t from A.
- Claim: $O' = O - t + i_j$ is feasible.
- $|O'| = |O| \leq k$.
- Can it be that $|O' \cap N_r| > k_r$?
- No. Exercise.

Part III

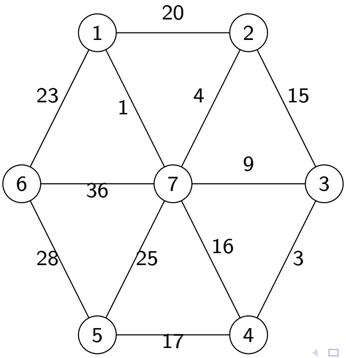
Greedy Algorithms: Minimum Spanning Tree

Minimum Spanning Tree

Input Connected graph $G = (V, E)$ with edge costs

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost of all edges in T is smallest

- T is the minimum spanning tree (MST) of G

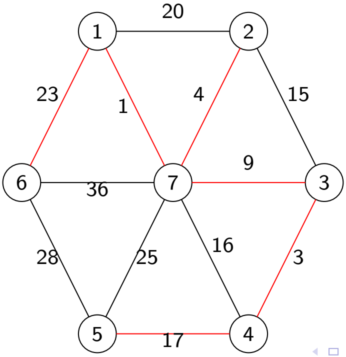


Minimum Spanning Tree

Input Connected graph $G = (V, E)$ with edge costs

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost of all edges in T is smallest

- T is the minimum spanning tree (MST) of G



Applications

- Network Design
  - Designing networks with minimum cost but that satisfy connectivity requirements
- Approximation algorithms
  - Can be used to bound the optimality of algorithms to approximate Travelling Salesman Problem, Steiner Trees, etc.
- Cluster Analysis

Greedy Template

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty
    choose i ∈ E
    if (T+i is feasible)
        add i to T
end while
return the set T
```

Main Task: In what order should edges be processed?

KA PA

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

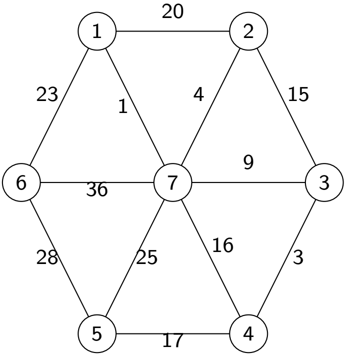


Figure: Graph G

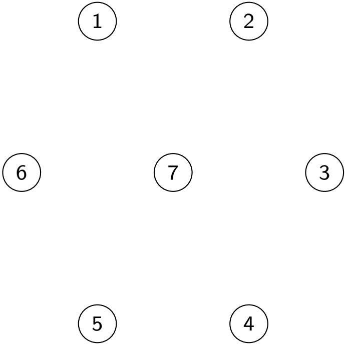


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

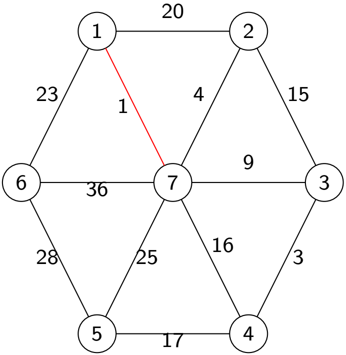


Figure: Graph G

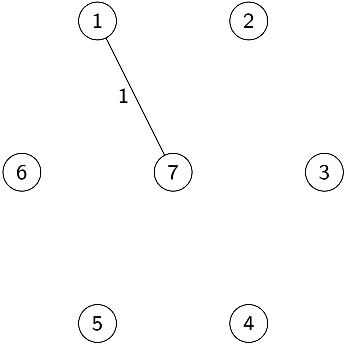


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

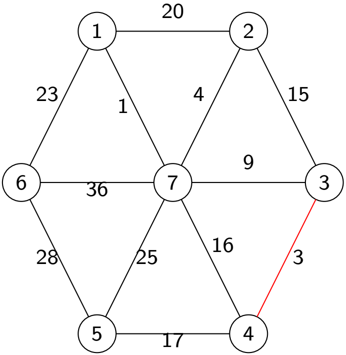


Figure: Graph G

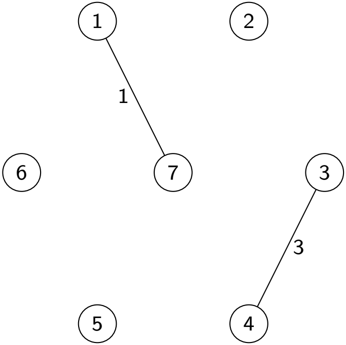


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

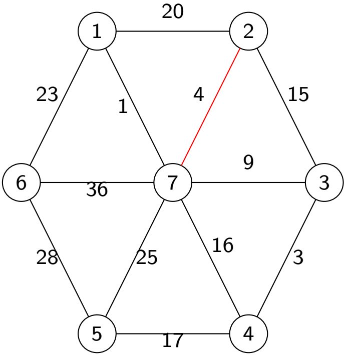


Figure: Graph G

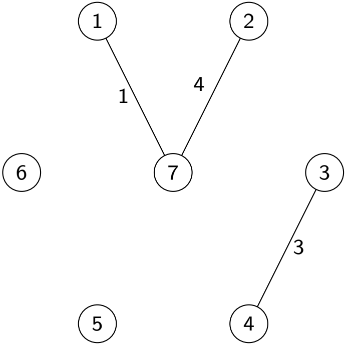


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

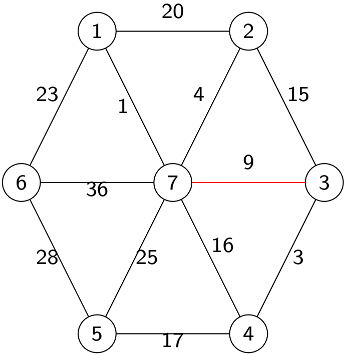


Figure: Graph G

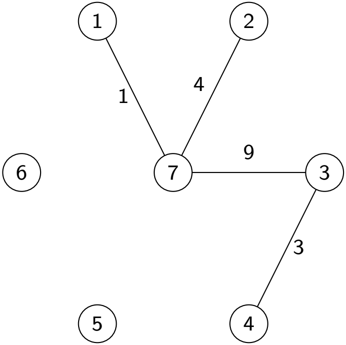


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

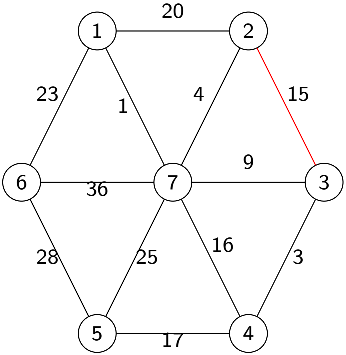


Figure: Graph G

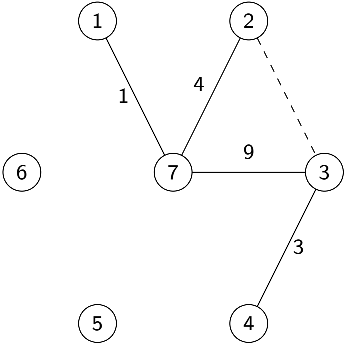


Figure: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

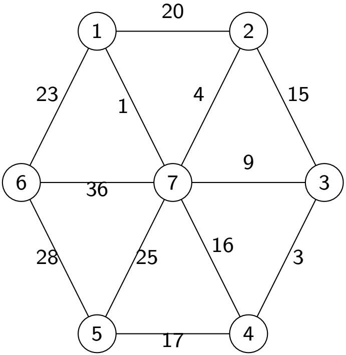


Figure: Graph G

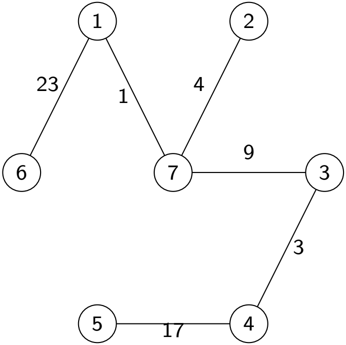


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

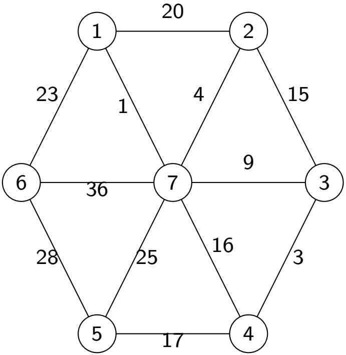


Figure: Graph G

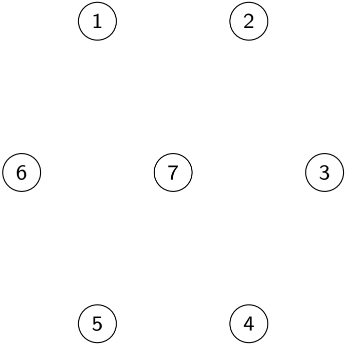


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

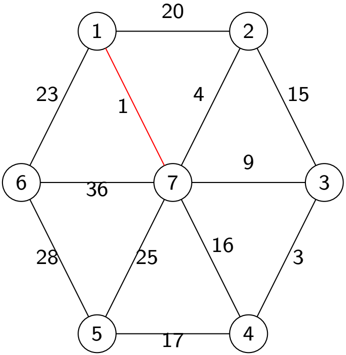


Figure: Graph G

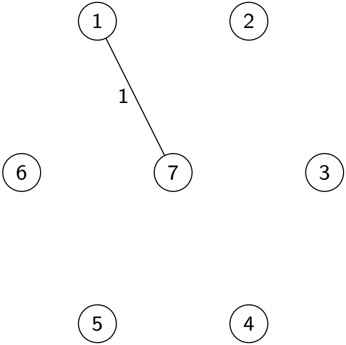


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

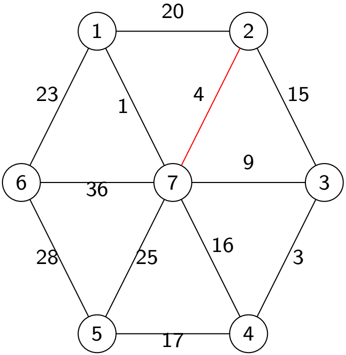


Figure: Graph G

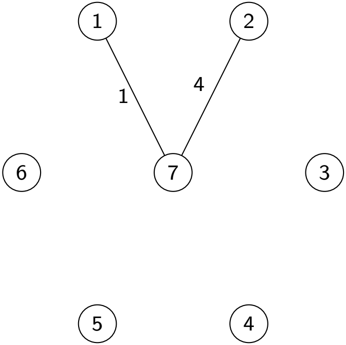


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

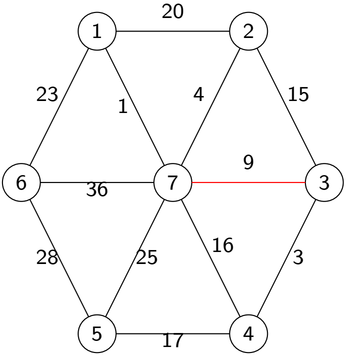


Figure: Graph G

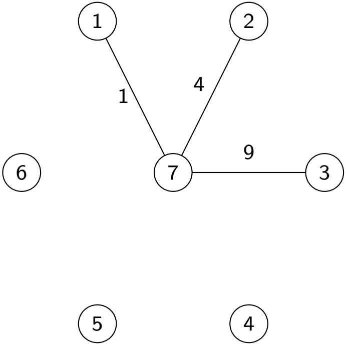


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

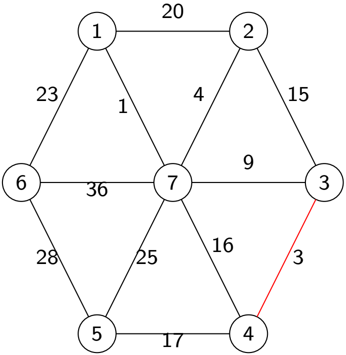


Figure: Graph G

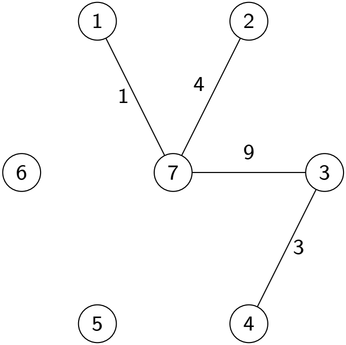


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

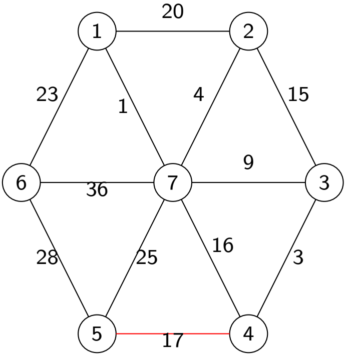


Figure: Graph G

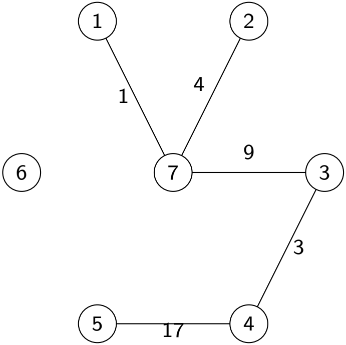


Figure: MST of G

Prim's Algorithm

T maintained by algorithm will be a tree. In each iteration, pick edge with least attachment cost to T.

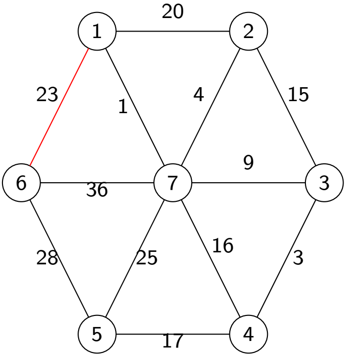


Figure: Graph G

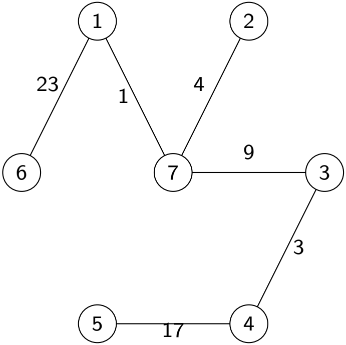


Figure: MST of G

Kruskal's Algorithm

```
Sort edges by weight and assume w_1 ≤ w_2 ≤ ... ≤ w_m
T is empty (* T will store edges of a MST *)
for i = 1 to m do
    if (T+i is feasible (does not contain a cycle))
        add i to T
end while
return the set T
```

Feasibility

Lemma

If G is connected, algorithm outputs a spanning tree.

Feasibility

Lemma

If G is connected, algorithm outputs a spanning tree.

Proof.

Suppose not. More than one connected component. $V_1, V_2, \ldots, V_r$ vertices in connected components. $r > 1$ by assumption.

Feasibility

Lemma

If G is connected, algorithm outputs a spanning tree.

Proof.

Suppose not. More than one connected component. $V_1, V_2, \ldots, V_r$ vertices in connected components. $r > 1$ by assumption.

Consider V_1. Since G is connected, there is an edge $e = (u, v)$ in G with $u \in V_1$, $v \in V \setminus V_1$.

Feasibility

Lemma

If G is connected, algorithm outputs a spanning tree.

Proof.

Suppose not. More than one connected component. $V_1, V_2, \ldots, V_r$ vertices in connected components. $r > 1$ by assumption.

Consider V_1. Since G is connected, there is an edge $e = (u, v)$ in G with $u \in V_1$, $v \in V \setminus V_1$.

Why did algorithm not add e? □

And for now . . .

Assumption

No two edge costs are equal, that is $w_1 < w_2 < \ldots < w_m$.

Optimality Proof

Theorem

Kruskal's algorithm outputs the unique optimum solution when edge weights are distinct.

Proof.

- Let $i_1, i_2, \ldots, i_{n-1}$ be the edges added by algorithm in order.

Optimality Proof

Theorem

Kruskal's algorithm outputs the unique optimum solution when edge weights are distinct.

Proof.

- Let $i_1, i_2, \ldots, i_{n-1}$ be the edges added by algorithm in order.
- Suppose T' is an optimum solution and $T \neq T'$.

Optimality Proof

Theorem

Kruskal's algorithm outputs the unique optimum solution when edge weights are distinct.

Proof.

- Let $i_1, i_2, \ldots, i_{n-1}$ be the edges added by algorithm in order.
- Suppose T' is an optimum solution and $T \neq T'$.
- Let i_j be the first edge in T not in T'.

Optimality Proof

Theorem

Kruskal's algorithm outputs the unique optimum solution when edge weights are distinct.

Proof.

- Let $i_1, i_2, \ldots, i_{n-1}$ be the edges added by algorithm in order.
- Suppose T' is an optimum solution and $T \neq T'$.
- Let i_j be the first edge in T not in T'.
- Lemma: there is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

□

Optimality Proof

Theorem

Kruskal's algorithm outputs the unique optimum solution when edge weights are distinct.

Proof.

- Let $i_1, i_2, \ldots, i_{n-1}$ be the edges added by algorithm in order.
- Suppose T' is an optimum solution and $T \neq T'$.
- Let i_j be the first edge in T not in T'.
- Lemma: there is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.
- Since i_j is lighter than e, contradicts optimality of T'.

$\square$

Proof of Lemma

Lemma

There is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

- Let $i_j = (u, v)$.

Proof of Lemma

Lemma

There is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

- Let $i_j = (u, v)$.
- There is a path P from u to v in T' since T' is a spanning tree.

Proof of Lemma

Lemma

There is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

- Let $i_j = (u, v)$.
- There is a path P from u to v in T' since T' is a spanning tree.
- $P + i_j$ is a cycle.

Proof of Lemma

Lemma

There is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

- Let $i_j = (u, v)$.
- There is a path P from u to v in T' since T' is a spanning tree.
- $P + i_j$ is a cycle.
- There must be $e \in P$ such that $e \notin \{i_1, i_2, \ldots, i_{j-1}\}$. Why?

Proof of Lemma

Lemma

There is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

- Let $i_j = (u, v)$.
- There is a path P from u to v in T' since T' is a spanning tree.
- $P + i_j$ is a cycle.
- There must be $e \in P$ such that $e \notin \{i_1, i_2, \ldots, i_{j-1}\}$. Why?
- If P contains only edges in $\{i_1, i_2, \ldots, i_{j-1}\}$ then $P + i_j$ contains a cycle and algorithm will not pick i_j.

Proof of Lemma

Lemma

There is an edge $e \in T' \setminus \{i_1, i_2, \ldots, i_{j-1}\}$ such that $T' - e + i_j$ is a spanning tree.

- Let $i_j = (u, v)$.
- There is a path P from u to v in T' since T' is a spanning tree.
- $P + i_j$ is a cycle.
- There must be $e \in P$ such that $e \notin \{i_1, i_2, \ldots, i_{j-1}\}$. Why?
- If P contains only edges in $\{i_1, i_2, \ldots, i_{j-1}\}$ then $P + i_j$ contains a cycle and algorithm will not pick i_j.
- Easy claim: $T - e + i_j$ is a spanning tree. Why?

Illustration for Proof

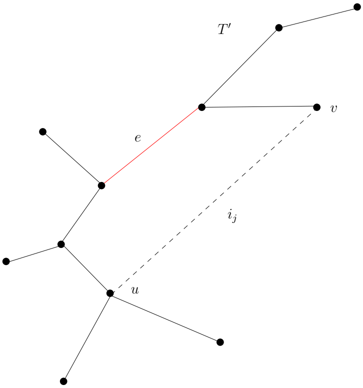


Figure: $e \not\in \{i_1, i_2, \ldots, i_j\}$. $T' + i_j - e$ is cheaper than T'

When edge costs are not distinct

Order edges lexicographically to break ties

- $i \prec j$ if either $w_i < w_j$ or $(w_i = w_j$ and $i < j)$

Assume edge weights distinct. Saw proof that MST is unique (this is not obvious btw).

Assume edge weights distinct. Saw proof that MST is unique (this is not obvious btw).

Questions:

- When is an edge e in all possible MSTs?
- When is an edge e in no MST?

Cut Property

Lemma

An edge $e = (u, v)$ is in every MST iff the following is true. There is some set $S \subset V$ with $u \in S, v \in V \setminus S$ such that $w(e)$ is the unique smallest weight edge between *S and $V \setminus S$.*

Proof Sketch.

Exchange property!

- If T does not contain e, add e to T to form cycle C.
- C must contain edge e' that crosses cut $(S, V \setminus S)$.
- $T - e' + e$ has strictly less weight than T (since e is strictly lighter than e' by assumption).
- Contradicts optimality of T.

$\square$

Cycle Property

Lemma

An edge $e = (u, v)$ is no MST iff there is a cycle C containing e such that e is the unique maximum weight edges on C.

Proof Sketch.

- Suppose e is in some MST T.
- Removing e from T generates two components S and $V \setminus S$.
- Pick $e' \in C$ *not* in T that has one end point in S and the other in $V \setminus S$.
- Why should such an edge e' exist? Exercise.
- $T - e + e'$ is strictly smaller weight than T, T cannot be an MST.

$\square$

Other MST Algorithms

Many variants of MST algorithms.

All of them rely essentially on the exchange property via the Cut or Cycle Properties.

To be done: implementation issue for Kruskal/Prim's algorithm.