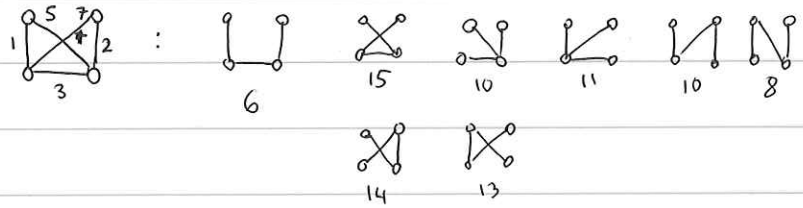
Minimum Spanning Trees (MSTs) (CLRS §23)

G is an undirected, connected graph, with a weight $w(u,v)$ associated to each edge.

Goal: find a spanning tree $T \subseteq E$ (acyclic and connects all vertices) with smallest total weight

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



Generic MST(G, w):

$A = \emptyset$

while A is not a spanning tree → { Loop invariant: A is a subset of some MST

find $(u,v) \in E$ safe for A

$A = A \cup \{(u,v)\}$ → { (u,v) is safe for A if $A \cup \{(u,v)\}$ is also a subset of some MST

return A

Note: by loop invariant, $\exists$ some safe edge

Definitions:

- A cut of $G = (V,E)$ is a partition $(S, V \setminus S)$.
- Edge (u,v) crosses cut $(S, V \setminus S)$ if $u \in S$ and $v \in V \setminus S$ or vice versa.
- Cut respects A if no edge in A crosses the cut.
- (u,v) is a light edge crossing a cut if it has minimum weight of any edge entering the cut.

THM 23.1: Let $A \subseteq E$ be included in some MST, $(S, V \setminus S)$ be a cut respecting A, and (u,v) a light edge crossing $(S, V \setminus S)$. Then (u,v) is safe for A.

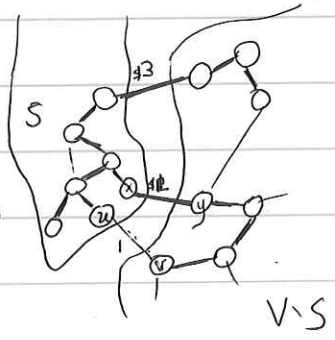
Proof: Let T be an MST containing A. If $(u,v) \in T$, we're done.

So assume $(u,v) \notin T$.

Since T is a spanning tree, there is a path in T connecting u to v.

Since $u \in S$ and $v \in V \setminus S$, there exists an edge (x,y) on that path that crosses $(S, V \setminus S)$.

A - yellow
T - orange



Because (u,v) is a light edge,

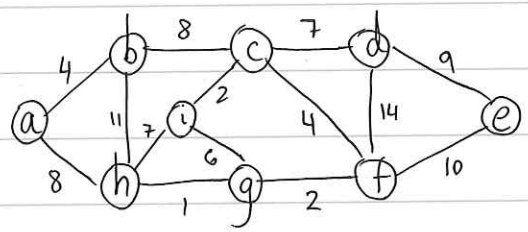
$$w(u,v) \leq w(x,y).$$

Consider $T' = T \setminus \{(x,y)\} \cup \{(u,v)\}$. It is a spanning tree. Moreover,

$$w(T') = w(T) - w(x,y) + w(u,v)$$
$$\leq w(T)$$

but T was MST, so T' is also an MST. Therefore (u,v) is safe. ∎

KRUSKAL'S ALG



Consider edges by non-decreasing weights. Any edge that connects two connected components (i.e., trees in the forest) is added to A.

Correctness: such an edge is a light edge for the cut $(C_1, V - C_1)$, (since no edge leaving C_1 can have lower weight) and is therefore safe.

MST-KRUSKAL (G, w)

1. $A = \emptyset$
2. FOR EACH $v \in G.V$
3. MAKESET(v) → each vertex is its own connected component $O(V)$
4. Sort E in non-decreasing weight order → $O(E \log E) = O(E \log V)$ (since $|V| \leq |E|^2$)
5. FOR $(u,v) \in E$
6. IF FINDSET(u) ≠ FINDSET(v) → if u and v are in different conn. comp.
7. $A = A \cup \{(u,v)\}$
8. UNION(u,v) → merge components $O(E \log V)$.
9. RETURN A

Runtime: $|V| \times$ MAKESET; sort $|E|$ numbers; $|E| \times$ FINDSET; $|V| \times$ UNION

Need a disjoint set data structure (CLRS §21.3)

MAKESET	$O(1)$
FINDSET	$O(\log V)$
UNION	$O(\log V)$

Total time = $O(V + E \log V + E \log V) = O(E \log V)$

Remark: Using improvement in CLRS §21.4, runtime of 6-8 is only $O(E\alpha(V))$ where α is a very slowly growing func. Total is still $O(E \log V)$ due to sort.

PRIM'S ALG

[Same graph example] [Start from a] [Also draw priority queue of all waiting vxs]
vx labels outside — b, k, ...

Similar to Dijkstra. We grow a tree, and each time find the vertex outside the tree that is connected by the lightest edge (which is therefore safe).

MST-PRIM (G, w, r)

1. FOR $u \in V$
2. $\quad u.key = \infty$
3. $\quad u.\pi = NIL$
4. r. key = 0
5. Q = V
6. while $Q \neq \emptyset$
7. $\quad u = EXTRACT\ MIN(Q)$ ← u is added to tree with $(u.\pi, u)$
8. $\quad$ FOR $v \in ADJ[u]$
9. $\quad\quad$ if $v \in Q$ and $w(u,v) < v.key$
10. $\quad\quad\quad v.\pi = u$
11. $\quad\quad\quad v.key = w(u,v)$ // Implicit DECREASEKEY

Implicitly builds

$$A = \{ (u.\pi, u) \mid u \in V - \{r\} - Q \}$$

Runtime: $|V| \times$ INSERT (line 5); $|V| \times$ EXTRACT-MIN (line 7); $|E| \times$ DECREASEKEY (line 11)

Using a priority queue based on binary heap, all operations are $O(\log V)$, total runtime

$$O(\underset{\substack{1\text{-}5 \\ \text{(using MAKEHEAP/BUILD)} \\ \text{otherwise } V\log V}}{V} + \underset{7}{V\log V} + \underset{8\text{-}11}{E\log V}) = O(E \log V).$$

With Fibonacci heaps, EXTRACT MIN is still $O(\log V)$ (but amortized) and DECREASEKEY is only $O(1)$ (amortized). So total $O(V + V\log V + E) = O(E + V\log V)$.

Single source shortest paths (SSSP)

Input: a directed graph $G=(V,E)$ with non-negative weights $w: E \to \mathbb{R}^{\geq 0}$ on the edges, and a source vertex $s \in V$.

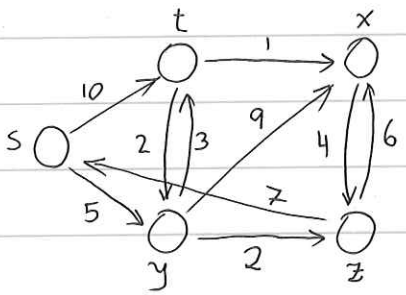
Goal: find a shortest path from s to any $v \in V$.

Remark: We might only care about shortest path between $s \in V$ and one $t \in V$ (as opposed to between s and all V). The running time of algorithms for this "single pair" variant is not asymptotically faster.

Remark One can also consider negative weights, and there are algorithms for this extension, like Bellman-Ford, running in time $O(|V| \cdot |E|)$

Remark The case of all weights being 1 (i.e., unweighted) was solved by BFS.

Dijkstra's algorithm (§24.3)



DIJKSTRA (G, w, s)

```
1. FOR EACH v ∈ V
2.     v.d = ∞
3.     v.π = NIL
4. s.d = 0
5. Q = V          // initialize priority queue
6. while Q ≠ ∅
7.     u = EXTRACT-MIN(Q)
8.     FOR EACH v ∈ Adj[u]
9.         IF v.d > u.d + w(u,v)
10.            v.d = u.d + w(u,v)     // Implicit DECREASEKEY
11.            v.π = u
```

Runtime analysis: $|V|\times$ INSERT (line 5); $|V|\times$ EXTRACT-MIN (line 7); $|E|\times$ DECREASEKEY (line 10).
So, just like Prim's, using binary heap we get runtime of $O((|V|+|E|)\log|V|)$.
If we use Fibonacci heaps, runtime is $O(|V|\log|V| + |E|)$.

		Fibonacci Heap
(5)	$\|V\|\times$ INSERT	$O(\|V\|\log\|V\|)$
(7)	$\|V\|\times$ EXTRACT	$O(\|V\|\log\|V\|)$
(10)	$\|E\|\times$ DECKEY	$O(\|E\|)$

Let $\delta(u,v)$ be weight of shortest path from u to v.

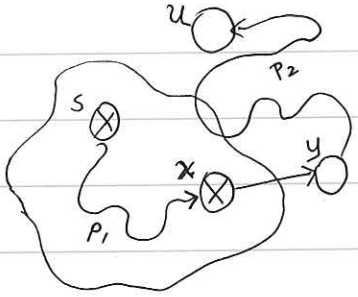
THM 24.6 Dijkstra's algorithm terminates with $u.d = \delta(s,u)$ for all $u \in V$.

Proof sketch:

Loop invariant: each vertex v that is no longer in Q (i.e., is black) satisfies $v.d = \delta(s,v)$.

Initialization trivial.

Maintenance: assume the invariant holds for all black vertices and that we're about to add u. Consider a shortest path from s to u:



and let x and y be two consecutive vertices along the path such that x is black and y is white.

① ~~When u removed from Q~~, $x.d = \delta(s,x)$: Because x is black and our assumption.

② ~~When u removed from Q~~, $y.d = \delta(s,y)$: That's because ~~xxxx~~ $\delta(s,y) = \delta(s,x) + w(x,y)$, and because when we explored x's neighbors, ~~xx~~ in lines 9-10, we made sure $y.d \leq x.d + w(x,y)$. Therefore,

$$y.d \leq x.d + w(x,y) \overset{①}{=} \delta(s,x) + w(x,y) = \delta(s,y).$$

Moreover, $y.d$ can never be smaller than $\delta(s,y)$, and so $y.d = \delta(s,y)$.

③ $u.d = \delta(s,v)$: Because u was chosen from the priority queue,

$$u.d \leq y.d = \delta(s,y) \leq \delta(s,u)$$

and so $u.d = \delta(s,u)$ (again, because $u.d$ never goes below $\delta(s,u)$).

All Pairs Shortest Paths (APSP)

The goal is to compute the shortest paths (or just their cost) between any two nodes $u, v \in V$.

If all weights are non-negative, we can simply run Dijkstra from each $u \in V$. Runtime is $O(V^2 \log V + VE)$.

If, however, there are negative weights, we need to use Bellman-Ford instead of Dijkstra, and the running time becomes the much worse $O(V^2 \cdot E)$. We will improve this to $O(V^3)$ using the Floyd-Worshall algorithm. (See the textbook for Johnson's algorithm, achieving the better $O(V^2 \log V + VE)$, just like in the case of non-negative weights)

Remark We assume there are no negative cycles, as otherwise some distances are undefined. With this assumption, shortest paths are always simple, and in particular have at most $n-1$ edges.

We will assume input is given as adjacency matrix. So for each $u, v \in V$,

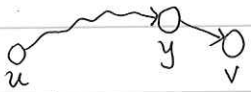
$$w(u,v) = \begin{cases} 0 & \text{if } u = v \\ \text{weight of edge } (u,v) & \text{if } u \neq v \text{ and } (u,v) \in E \\ \infty & \text{if } u \neq v \text{ and } (u,v) \notin E \end{cases}$$

We will use dynamic programming.

Attempt #1: (CLRS 25.1)

Subproblems: for each $u, v \in V$ and $k \geq 1$, the cost of the shortest path from u to v using at most k edges.

Guess: next-to-last vertex



Recurrence: $DP(u,v,0) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{o.w.} \end{cases}$ $\quad DP(u,v,1) = w(u,v)$

$$DP(u,v,k) = \max_{y \in V} (DP(u,y,k-1) + w(y,v))$$

Final output: $DP(u,v,n-1)$ for all $u, v \in V$ (since the shortest path has at most $n-1$ edges)

Runtime:

$O(|V|^3)$ subproblems $\times$ $O(|V|)$ time/subproblem $= O(|V|^4)$ time

Attempt #2:

Guess middle vertex

Recurrence:

$$DP(u,v,k) = \max_{y \in V} (DP(u,y,k/2) + DP(y,v,k/2)) \qquad \text{(assume } k \text{ is power of 2)}$$

$$DP(u,v,1) = w(u,v)$$

Runtime:

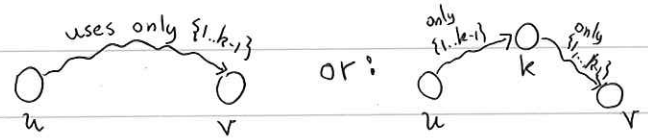
$O(|V|^2 \log |V|)$ subproblems $\times$ $O(|V|)$ time/subproblem $= O(|V|^3 \log V)$ time

FLOYD-WARSHALL (CLRS 25.2)

Different DP, achieving $O(|V|^3)$ runtime. Assume $V = \{1, 2, \ldots, n\}$.

Subproblems: $\forall u, v \in V$, $k \in \{0, \ldots, n\}$, the minimum weight of a path from u to v that only uses vertices from $\{1, \ldots, k\}$ as intermediate nodes.

Such a path looks like either:



Recurrence:

$$DP(u,v,k) = \begin{cases} w(u,v) & \text{if } k = 0 \\ \min\left(DP(u,v,k-1),\ DP(u,k,k-1) + DP(k,v,k-1)\right) & \text{if } k \geq 1 \end{cases}$$

Final output: $DP(u,v,n)$ for all $u, v \in V$.

Runtime: $O(|V|^3)$ subproblems $\times$ $O(1)$ time/subproblem $= O(|V|^3)$

P, NP, NP-completeness

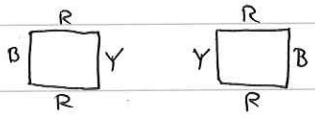
All the algorithms we saw in the course had polynomial running time: some linear $O(n)$ like BFS, some quadratic $O(n^2)$ like Longest Common Subseq. etc... Many more important poly-time algorithms were discovered: optimization algorithms (linear programming), Fast Fourier Transform, ... Problems that can be solved in poly time form the complexity class P.

Do all problems have poly. time algorithms?

No!

Some problems cannot be solved by <u>any</u> algorithm, no matter how slow. They are said to be <u>uncomputable</u>.

One example: given a set of Wang tiles, can they tile the plane?



The original example by Turing (1940s) was the halting problem: given code P for a program, and a string x, does P halt when run on x?

<u>Proof of uncomputability</u>: Assume towards contradiction that there exists a program HALT that solves the halting problem. Then consider:

```
def Z(P):
    if HALT(P,P):
        Loop forever
    else:
        Return.
```

Does Z halt on input Z?

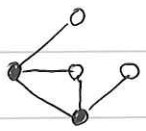
- If yes, then no, contradiction..
- If no, then yes, contradictio...

Luckily, almost all problems encountered in real life are computable. However, many don't (seem to) have polynomial time algorithms.

How to deal with that?

1. Heuristics
2. Simplify problem, maybe assume something about input
3. Approximate
4. Buy more hardware (only gets you that far...)

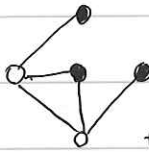
DEF A vertex cover of an undirected graph $G=(V,E)$ is a subset $S \subseteq V$ that touches all vertices.



Possible story: Need to install monitoring stations on all fibers.

The vertex cover problem (VC) asks to find the smallest vertex cover.

DEF An independent set of an undirected graph $G=(V,E)$ is a subset $S \subseteq V$ that contains no edges inside it.



Possible story: each vertex is a possible message to a satellite, and edges connect two messages that can be confused.

The independent set problem (IS) asks to find the largest independent set.

The best known algorithms for both VC and IS take exponential time.

Claim $VC \leq IS$, i.e., there is a reduction from VC to IS

Proof Given a graph G, find largest ind. set S, and output $V \setminus S$.

This is the smallest vertex cover.

Similarly $IS \leq VC$.

Another example: a 3SAT formula looks like:

OR AND NOT

$$(X \vee Y \vee \bar{Z}) \wedge (X \vee \bar{Y} \vee W) \wedge (\bar{X} \vee \bar{Y} \vee \bar{W}) \wedge (Y \vee W \vee Z) \wedge (\bar{X} \vee \bar{W} \vee Z)$$

$n = 4$ variables
$m = 5$ clauses

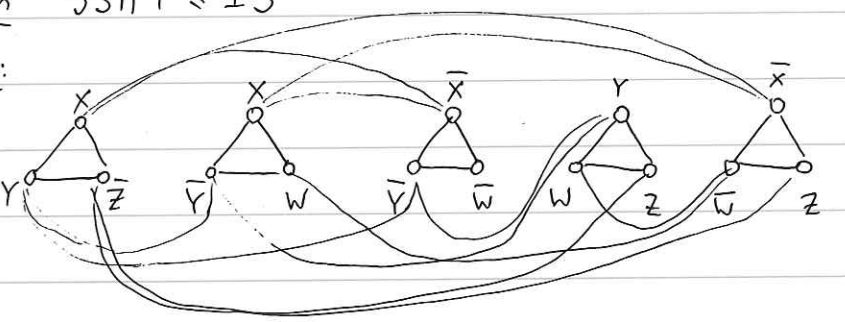
This formula is satisfied by $X = T, Y = F, W = T, Z = T$.

3SAT problem: given a 3SAT formula on n vars, is it satisfiable?

Best algorithm is exponential time.

Claim $3SAT \leq IS$

Proof:



The graph has an ind. set of size m if and only if the formula is satisfiable. ∎

DEF The class NP consists of YES/NO problems ("decision problems") for which there is an efficient (= poly time) "verifier algorithm" that can check that an instance of the problem is a YES instance when given an appropriate witness.

Ex • Soduku $\in$ NP

• 3SAT $\overset{\in NP}{:}$ given a satisfying assignment, easy to check that satisfiable.

• VC $\overset{\in NP}{:}$ to make it a YES/NO question we ask if there is a vertex cover of size $\leq k$? Given witness S, check ① S is a vertex cover ② $|S| \leq k$.

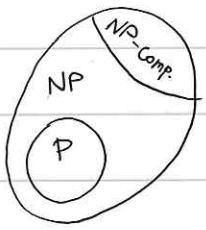
• Similarly IS $\in$ NP.

DEF A problem $A \in NP$ is called NP complete if it is hardest in NP, i.e., $\forall B \in NP$, $B \leq A$.

THM [Cook Levin 1971] 3SAT is NP-complete

COR VC, IS are NP-complete.

There are 1000s more NP-complete problems. If you solve any one of them, you solve all of them. This is the famous $P \overset{?}{=} NP$ question, one of the "millenium problems" with a $1M prize.

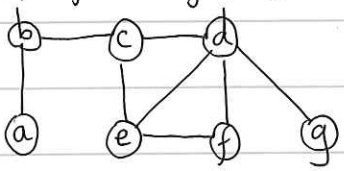


It is widely believed that $P \neq NP$ but proving it is beyond reach. If unexpectedly it turned out that $P = NP$ then we would have efficient algorithms for lots of important problems (but also no cryptography!).

Approximating VC (§35.1)

Even though VC is NP-complete (and likely requires exponential time), we can hope to efficiently find an approximate solution that is guaranteed to be not much worse than optimal.

(In practice, one would also try some heuristics)
(Taking vertex of highest degree sometimes gives very bad results!)



Optimal solution: 3 vertices $\{b, d, e\}$

Approximation algorithm:

1. Couple the vertices until we can't (this gives a so-called "maximal matching")
2. Output all coupled vertices

The result is a vertex cover since otherwise we could couple more vertices
Example: we couple $b-c$, then $e-d$, Output $\{b, c, d, e\}$.

Thm This algorithm gives a 2-approximation
Proof The optimal solution must include at least one of each couple, whereas we took both.

Remark Getting better approximation than 2 is a hard problem
(based on Khot's Unique Games conjecture [KR02])