1 Graph

1.1 Breadth-first search (BFS)

The *distance* between two nodes is the length of the shortest path between them. How do we find the shortest paths from s to all other vertices?

```
function BFS($G, s$)
    for all $u \in V$ do
        dist($u$) $\leftarrow \infty$
    dist($s$) $\leftarrow 0$
    $Q \leftarrow [s]$                              ▷ queue containing $s$
    while $Q$ is not empty do
        u $\leftarrow$ pop($Q$)
        for all edges $(u, v) \in E$ do
            if dist($v$) $\leftarrow \infty$ then
                push($Q, v$)
                dist($v$) $\leftarrow$ dist($u$)+1
```

1.1.1 Correctness

Use induction. For each $d = 0, 1, 2, \cdots$ there is a moment at which

- All nodes at distance $\leq d$ from s have distances correctly set
- All other nodes have their distances set to ∞
- The queue contains exactly the nodes at distance d

1.1.2 Analysis

Each vertex is put on the queue exactly once $\rightarrow 2|V|$ queue operations. `for` loop looks at each edge once (in directed graphs) or twice (in undirected graphs) $\rightarrow O(|E|)$ time. Therefore, $O(|V| + |E|)$.

1.2 Weighted graphs

Can we adapt BFS to a more general graph $G = (V, E)$ whose edge lengths are positive integers?

1.2.1 Priority Queue

Data structure supporting the following operations:

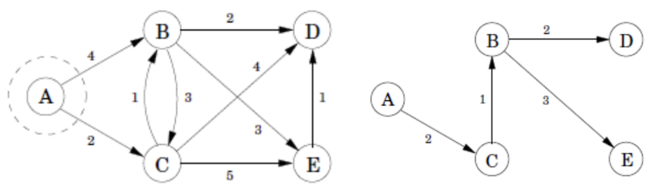
- `insert`: Add a new element to the set.
- `decreasekey`: Accommodate the decrease in key value of a particular element
- `deletemin`: Return the element with the smallest key, and remove it from the set.
- `makequeue`: Build a priority queue out of the given elements, with the given key values. (In many implementations, this is faster than inserting the elements one by one.)

1.2.2 Binary heap (Implementation)

- Complete Binary Tree: All levels are completely filled except possibly the lowest, which is filled from the left up to a point.
- The value of each node $\leq$ value of its children. (min-heap)

1.3 Dijkstra's algorithm

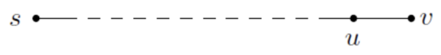
Using `prev`, we can construct shortest-path tree.



```
function DIJKSTRA($G, l, s$)
    for all $u \in V$ do
        dist($u$) $\leftarrow \infty$
        prev($u$) $\leftarrow$ nil
    dist($s$) $\leftarrow 0$
    $H \leftarrow$ makequeue($V$)
    while $H$ is not empty do
        u $\leftarrow$ deletemin($H$)
        for all edges $(u, v) \in E$ do
            if dist($v$)>dist($u$)+$l(u, v)$ then
                dist($v$)$\leftarrow$ dist($u$)+$l(u, v)$
                prev($v$) $\leftarrow u$
                decreasekey ($H, v$)    ▷ Set key of $v$ to dist($v$)
```

1.3.1 Correctness

Starting from s, we expand the *known region* R of the graph where shortest paths are known. What is the next vertex v to add to R?– The node outside R that is closest to s. Consider the shortest path from s to v.



Let u be the node before v on this path. Since all edge lengths are positive, u must be closer to s than v is. Thus, u is in R. (Since v is the closest node to s outside R.) So, the shortest path from s to v is a known shortest path extended by a single edge. v is the node outside R for which the smallest value of dist(s, u)+$l(u, v)$ is attained, as u ranges over R.

Following this idea, we prove the correctness using induction. At the end of each iteration of the while loop, the following conditions hold:

- There is a value dsuch that all nodes in R are at distance $\leq d$ from s and all nodes outside R are at distance $\geq d$ from s
- For every node u, the value dist(u) is the length of the shortest path from s to u whose intermediate nodes are constrained to be in R (if no such path exists, the value is ∞).

1.3.2 Analysis

1. $|V|$ `deletemin` operations
2. $|V| + |E|$ `insert`/`decreasekey` operations

	1	2	Total
Array	$O(\|V\|)$	O(1)	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\| \cdot d + \|E\|)\frac{\log \|V\|}{\log d})$
Fib. heap	$O \log(\|V\|)$	$O(1)$	$O(\|V\| \log \|V\| + \|E\|)$

Table 1. Running time by implementation of priority queue

1.4 Update

> Other way to prove correctness of Dijkstra's algorithm

We can consider Dijkstra's algorithm as performing a sequence of the following update procedure.

```
function UPDATE($u, v$)
    dist($v$) $\leftarrow$ min {dist($v$), dist($u$)+$l(u, v)$}
```

This UPDATE operation uses the fact that the distance to v cannot be more than the distance to $u + l(u, v)$.

Property 2 (Safeness)

Initializing $\text{dist}(s) = 0$ and $\text{dist}(v) = \infty\ \forall v \in V - \{s\}$ establishes $\text{dist}(v) \geq$ distance to v for all $v \in V$. This invariant is maintained over any sequence of UPDATE's.

Proof. Suppose not. Let v be the first vertex for which $\text{dist}(v)<$ distance to v, and let u be the vertex that caused $\text{dist}(v)$ to change: $\text{dist}(v) = \text{dist}(u) + l(u, v)$. Then,

$$\begin{aligned} \text{dist}(v) &< \text{distance to } v && \text{(supposition)} \\ &\leq \text{distance to } u + \text{distance from } u \text{ to } v && \text{(triangle inequality)} \\ &\leq \text{distance to } u + l(u, v) && \text{(shortest} \leq \text{specific)} \\ &\leq \text{dist}(u) + l(u, v) && (v \text{ is first violation)} \end{aligned}$$

Contradiction. □

Property 1

Let u be v's predecessor on a shortest path from s to v. Then, if $\text{dist}(u)$=distance to u, after UPDATE(u, v), $\text{dist}(v) =$ distance to v.

Proof. distance to v = distance to $u + l(u, v)$. Suppose $\text{dist}(v)>$distance to v before UPDATE. (Otherwise, by Property 2, $\text{dist}(v) =$ distance to v.) Then, $\text{dist}(v) > \text{dist}(u) + l(u, v)$ because $\text{dist}(v)>$distance to v=distance to $u+l(u, v)$=$\text{dist}(u) + l(u, v)$. By UPDATE(u, v), $\text{dist}(v) = \text{dist}(u) + l(u, v) =$ distance to v. □

Theorem 1

Dijkstra's algorithm terminates with $\text{dist}(v) =$ distance to $v \forall v \in V$.

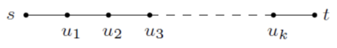
Proof. It suffices to show that $\text{dist}(v) =$ distance to v for every $v \in V$ when v is added to R. Suppose u is the first vertex added to R for which $\text{dist}(u) >$ distance to u. Let y be the first vertex in $V - R$ along a shortest path from s to u, and let x be its predecessor. Since u is the first vertex violating the claimed invariant, we have $\text{dist}(x)$= distance to x. When x was added to R, we UPDATE the edge (x, y), which implies that $\text{dist}(y) =$ distance to $y \leq$ distance to $u < \text{dist}(u)$. But $\text{dist}(u) \leq \text{dist}(y)$ by our choice of u. Contradiction. □

1.5 Bellman-Ford algorithm

```
function BELLMAN-FORD($G, l, s$)
    for all $u \in V$ do
        $\text{dist}(u) \leftarrow \infty$
        $\text{prev}(u) \leftarrow$ nil
    $\text{dist}(s) \leftarrow 0$
    for $|V| - 1$ times do
        for all $e \in E$ do
            UPDATE($e$)
```

1.5.1 Negative edges

Consider a shortest path from s to t.



This path can have at most $|V| - 1$ edges. If the sequence of UPDATES performed includes $(s, u_1), (u_1, u_2), \cdots, (u_k, t)$ in that order (not necessarily consecutively), then by Property 1, the distance to t will be correctly computed. So simply UPDATE *all edges* $|V| - 1$ times.

1.6 Negative cycles

If a graph contains a negative-weight cycle, some shortest paths may not exist. Instead of stopping after $|V| - 1$ iterations, perform one extra round. There is a negative cycle if and only if some dist value is reduced during this final round.

1.7 Shortest paths in dags

We need to perform a sequence of UPDATES that includes every shortest path as a subsequence. In any path of a dag, the vertices appear in increasing linearized order.

```
function DAG-SHORTEST-PATHS($G, l, s$)
    for all $u \in V$ do
        $\text{dist}(u) \leftarrow \infty$
        $\text{prev}(u) \leftarrow$ nil
    $\text{dist}(s) \leftarrow 0$
    Linearize $G$
    for each $u \in V$, in linearized order do
        for all $(u, v) \in E$ do
            UPDATE($u, v$)
```

2 Greedy algorithms

2.1 Minimum spanning tree

The minimum spanning tree (MST) is connected tree with minimum cost.

- Input: Undirected graph $G = (V, E)$, edge weights w_e
- Output: Tree $T = (V, E'), E' \subseteq E$that minimizes $\text{weight}(T) = \sum_{e \in E'} w_e$

Since removing a cycle edge cannot disconnect a graph, the solution must be connected and acyclic.

> Undirected connected acyclic graphs = trees

2.1.1 Properties of trees

- A tree on n nodes has $n - 1$ edges
- Any connected, undirected graph with $|E| = |V| - 1$ is a tree
- An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

2.2 Kruskal's algorithm

1. Starts with the empty graph
2. Repeatedly add the lightest edge that doesn't produce a cycle.

2.2.1 Cut property

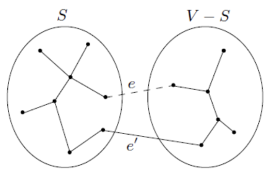
Theorem 2

Suppose edges X are part of a MST of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then, $X \cup \{e\}$ is part of some MST.

Proof. If e is in T, done. Otherwise, add e to T. It creates a cycle. This cycle must have another edge e' across the cut $(S, V - S)$. Remove e', then we have new spanning tree $T' = T \cup \{e\} - \{e'\}$.

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e')$$

Since e is the lightest edge crossing the cut, $w(e) \leq w(e')$. Thus $\text{weight}(T') \leq \text{weight}(T)$. Since T is a MST, T' is also a MST. □



2.2.2 Correctness

At any given moment, the edges already chosen form a partial solution, a collection of connected components (trees). The next edge e to be added connects two of these components; call them T_1, T_2. Since e is the lightest edge that doesn't produce a cycle, it is certain to be the lightest edge between T_1 and $V - T_1$. Therefore, it satisfies the cut property.

2.2.3 Implementation

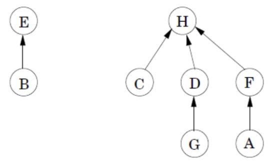
We use *disjoint-set* data structure to test candidate edge $u - v$ if they lie in different components, not producing a cycle.

- `makeset`(x): create a singleton set containing just x
- `find`(x): which set does x belong?
- `union`(x, y): merge the sets containing x and y

```
function KRUSKAL($G, w$)
    for $u \in V$ do makeset($u$)
    $X \leftarrow \{\}$
    Sort the edges $E$ by weight
    for edges $\{u, v\} \in E$, in increasing order of weight do
        if find($u$) $\neq$ find($v$) then
            $X \leftarrow X \cup \{u, v\}$
            union($u, v$)
```

→ $|V|$ `makeset`, $2|E|$ `find`, $|V| - 1$ `union` operations

- Store a set as a directed tree.
- Nodes of the tree are elements of the set, in no particular order.
- Each has parent pointers π that lead up to the root of the tree.
- The root is a representative, or name, for the set.
- The root has a parent pointer π pointing itself.
- The rank represents the height of the subtree from the node.



```
function MAKESET($x$)
    $\pi(x) = x$
    rank($x$) = 0
```

Constant-time operation

```
function FIND($x$)
    while $x \neq \pi(x)$ do $x = \pi(x)$
    return $x$
```

O(height of the tree) algorithm

Make the root of the shorter tree point to the root of the taller tree. Then, the overall height increases only if the two trees being merged are equally tall.

```
function UNION($x, y$)
    $r_x \leftarrow$ FIND($x$), $r_y \leftarrow$ FIND($y$)
    if $r_x = r_y$ then return
    if rank($r_x$)>rank($r_y$) then
        $\pi(r_y) \leftarrow r_x$
    else
        $\pi(r_x) \leftarrow r_y$
        if rank($r_x$)=rank($r_y$) then
            rank($r_y$) $\leftarrow$ rank($r_x$)+1
```

2.2.4 Analysis

- For any x, rank(x)<rank($\pi(x)$)
- Any root node of rank k has at least 2^k nodes in its tree
- If there are n elements overall, there can be at most $n/2^k$ nodes of rank k

The maximum rank is $\log n$. Thus all the trees have height $\leq \log n$, and this is an upper bound on the running time of `find` and `union`.

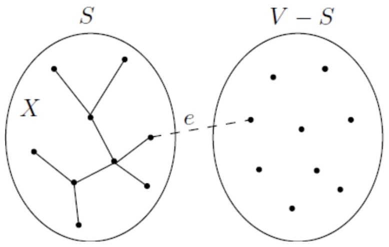
- $O(|E| \log |V|)$ to sort the edges ($\because \log |E| = \Theta(\log |V|)$)
- $O(|E| \log |V|)$ for `find` and `union` operations

→ Kruskal's algorithm is $O(|E| \log |V|)$.

2.3 Prim's algorithm

The cut property suggests that the following greedy schema works to find MST.

1. X is edges picked so far
2. Pick a set $S \in V$ for which X has no edges between $S, V - S$
3. Let $e \in E$ be the minimum weight edge between $S, V - S$
4. $X = X \cup \{e\}$



In Prim's algorithm, the intermediate set of edges X always forms a subtree. Let S is the set of this tree's vertices. Use *priority queue* to find the lightest edge between a vertex in S and a vertex outside S, then grow S to include the vertex $v \notin S$ of smallest cost: cost(v) = $\min_{u \in S} w(u, v)$.

```
function PRIM($G, w$)
    for $u \in V$ do
        cost($u$) $\leftarrow \infty$
        prev($u$) $\leftarrow$ nil
    $u_0 \leftarrow$ any initial node
    cost($u$) $\leftarrow 0$
    $H$=makequeue($V$)
    while $H$ is not empty do
        $v \leftarrow$ deletemin($H$)
        for $\{v, z\} \in E$ do
            if cost($z$)>$w(v, z)$ then
                cost($z$) $\leftarrow w(v, z)$
                prev($z$) $\leftarrow v$
                decreasekey($H, z$)
```

2.4 Huffman encoding

Consider problem of designing a binary code where each symbol is represented by a unique binary string. Given the frequencies for each symbol, using variable-length code can compress data considerably by giving frequent symbols short codewords.

2.4.1 Prefix-free encoding

If codewords are {0,01,11,001}, the decoding of 001 is ambiguous. The *prefix-free encoding* is an encoding that no codeword is a prefix of another codeword. An optimal code is represented by a *full* binary tree, which any internal node has two children.

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Symbol	Codeword
A	0
B	100
C	101
D	11

2.4.2 Cost of tree

How do we find the optimal coding tree, given the frequencies $f_1, f_2, \cdots, f_n$ of n symbols? We want to minimize the

$$\text{cost of tree} = \sum_{i=1}^{n} f_i \cdot (\text{depth of } i\text{th symbol in tree})$$

The frequency of an internal node is defined as the sum of the frequencies of its descendant leaves–the number of times the internal node is visited during decoding. So the total number of bits = cost of tree = **the sum of the frequencies of all leaves and internal nodes** except the root.

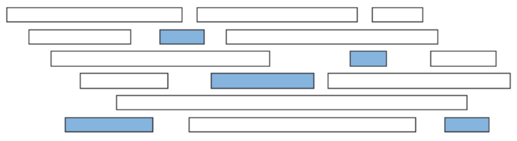
2.4.3 Greedy algorithm

1. The two symbols with the smallest frequencies f_1, f_2 must be at the bottom of the optimal tree, as children of the lowest internal node. (two since the tree is full).
2. Any tree in which f_1, f_2 are sibling has cost $f_1 + f_2$+the cost for a tree with $n-1$ leaves of frequencies $(f_1 + f_2), f_3, \cdots, f_n$

```
function HUFFMAN(f)
    $H \leftarrow$ a priority queue of integers ordered by $f$
    for $i = 1$ to $n$ do
        insert$(H, i)$
    for $k = n + 1$ to $2n - 1$ do
        $i \leftarrow$ deletemin$(H)$, $j \leftarrow$ deletemin$(H)$
        Create a node numbered $k$ with children $i, j$
        $f[k] \leftarrow f[i] + f[j]$
        insert$(H, k)$
```

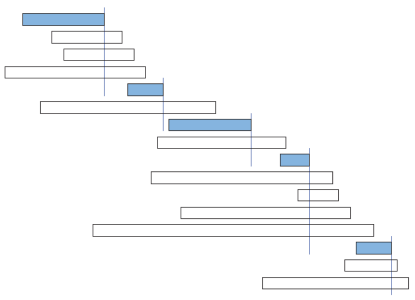
2.5 Scheduling problem

Given two arrays $S[1...n]$ and $F[1...n]$ listing the start and finish times of each class, choose the largest subset $X \in \{1, \cdots n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$


A maximal conflict-free schedule for a set of classes.

2.5.1 Greedy algorithm

1. Sort classes by finish times.
2. Scan classes in the sorted order and choose the next class that does not conflict with the latest class.



```
function GREEDYSCHEDULE($S[1..n]$, $F[1..n]$)
    Sort $F$ and permute $S$ to match
    count $\leftarrow 1$
    $X$[count] $\leftarrow 1$
    for $i \leftarrow 2$ to $n$ do
        if $S[i] > F[X[\text{count}]]$ then
            count $\leftarrow$ count+1
            $X$[count] $\leftarrow i$
```

2.5.2 Correctness

Lemma 1

At least one maximal conflict free schedule includes the class that finishes first.

Proof. Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set $X - \{g\}$. Thus, the schedule $X' = X \cup \{f\} - \{g\}$ is also conflict-free. Since X' has the same size as X, it is also maximal. □

Theorem 3

The greedy schedule is an optimal schedule.

Proof. Let f be the class that finishes first, and let L be the subset of classes that start after f finishes. The lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for L. The greedy algorithm, by the inductive hypothesis, computes an optimal schedule from L. □

We use an *inductive exchange argument*.

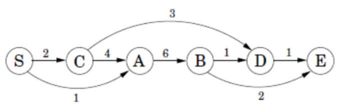
- Assume that there is an optimal solution that is different from the greedy solution.
- Find the *first* difference between the two solutions.
- Argue that we can exchange the optimal choice for the greedy choice without degrading the solution.
- This argument implies by induction that there is an optimal solution that contains the entire greedy solution.

3 Dynamic Programming

To solve the original problem, define a collection of subproblems $L(j) : 1 \leq j \leq n$ with the key property: There is an *ordering* on the subproblems, and a *relation* that shows how to solve a subproblem given the answers to smaller subproblems.

3.1 Shortest paths in dags

Consider the shortest path in linearized dag.



$\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\}$

If we compute dist in the left-to-right order, when we get to a node v, we already have all the information we need to compute dist(v).

The algorithm solves a collection of subproblems, $\{\text{dist}(u) : u \in V\}$, starting from dist$(s)$, then solve *larger* subproblems.

```
initialize all dist(·) values to $\infty$
dist($s$) $\leftarrow 0$
for each $v \in V - \{s\}$, in linearized order do
    dist($v$) $\leftarrow \min_{(u,v)\in E}\{\text{dist}(u) + l(u,v)\}$
```

3.2 Longest increasing subsequences

- Input: a sequence of numbers $a_1, \cdots, a_n$
- A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, \cdots, a_{i_k}$ where $1 \le i_1 < i_2 \cdots < i_k \le n$
- Find the increasing subsequence of greatest length.
- It is finding the longest path in the dag!

Subproblem: Let $L(j)$ the length of the longest path ending at j

```
for $j = 1, \cdots, n$ do
    $L(j) \leftarrow 1 + \max\{L(i) : (i,j) \in E\}$        ▷ Relation
return max $L(j)$
```

L values only tells us the length. To construct the subsequence, while computing $L(j)$, record prev(j), the previous node on the longest path to j.

3.2.1 Running time

- To compute $L(j)$: $O(\text{in-degree}(j))$
- Total $O(|E|) \rightarrow O(n^2)$

3.2.2 Comparison with recursive algorithm

The formula of $L(j)$ suggests an alternative, recursive algorithm (top-down). Then $L(j) = 1 + \max\{L(1), \cdots, L(j-1)\}$. The tree for $L(n)$ has exponential size with many repeated nodes. There are only small number of distinct subproblems.

3.3 Edit distance

Edit distance of two strings is the cost of their best possible alignment. Cost is the number of columns in which the letters differ. We may place any number of gaps (-).

- Input: Two strings $x[1..m], y[1..n]$
- Subproblem: $E(i,j)$ defined as prefixes $x[1..i], y[1..j]$
- Goal $E(m,n)$

The rightmost column of the best alignment can be one of:

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

$$E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), \text{diff}(i,j) + E(i-1,j-1)\}$$

where diff(i,j)= 0 if $x[i] = y[j]$ and 1 otherwise. Base case is $E(i,0) = i, E(0,j) = j$

The answers to all subproblems $E(i,j)$ form a 2-dimensional table. So the running time is $O(mn)$.

3.4 Common subproblems

- The input is $x_1, \cdots, x_n$ and subproblem is $x_1, \cdots, x_i \rightarrow O(n)$
- The input is $x_1, \cdots, x_n$, $y_1, \cdots y_m$ and subproblem is $x_1, \cdots, x_i$ and $y_1, \cdots, y_j \rightarrow O(mn)$
- The input is $x_1, \cdots, x_n$ and subproblem is $x_i, \cdots, x_j \rightarrow O(n^2)$
- The input is rooted tree and subproblem is a rooted subtree.

3.5 Knapsack

Given a knapsack of capacity W, n items of weight $w_1, \cdots, w_n$ and value $v_1, \cdots v_n$, choose the most valuable combination of items. Two versions:

3.5.1 Knapsack with repetition

Define $K(w)$ = maximum value achievable with a knapsack of capacity w. If the optimal solution to $K(w)$ includes item i, then removing it leaves an optimal solution to $K(w - w_i)$. We don't know which i, so try all possibilities.

```
$K(0) \leftarrow 0$
for $w = 1$ to $W$ do
    $K(w) \leftarrow \max_{i \text{ s.t. } w_i \le w}\{K(w - w_i) + v_i\}$
return $K(W)$
```

This algorithm fills in a 1-d table of length $W + 1$, in left-to-right order. Each entry can take up to $O(n)$ time to compute. $\rightarrow O(nW)$.

3.5.2 Knapsack w/o repetition

Define $K(w,j)$ = maximum value achievable using a knapsack of capacity w and item $1, \cdots, j$. Express $K(w,j)$ in terms of smaller subproblems considering whether item j is needed or not.

```
$K(0,\cdot) \leftarrow 0, K(\cdot,0) \leftarrow 0$
for $j = 1$ to $n$ do
    for $w = 1$ to $W$ do
        if $w_j > w$ then
            $K(w,j) \leftarrow K(w,j-1)$
        else
            $K(w,j) \leftarrow \max\{K(w - w_j, j-1) + v_j, K(w,j-1)\}$
return $K(W,n)$
```

The algorithm fills out a 2-d table, with $W + 1$ rows and $n + 1$ columns. Each table entry takes constant time. $\rightarrow O(nW)$.

3.6 Memoization

In DP, we use a recursive formula to fill out a table of solution values in a bottom-up manner, from smallest subproblem to largest. The formula also suggests a recursive algorithm, but we should use memoization to record the result of subproblem:

```
if $w$ is in hash table return $K(w)$
···                                              ▷ algorithm body
Insert $K(w)$ into hash table with key $w$
```

3.7 Shortest reliable paths

Given a graph G with edge lengths, two nodes s and t and an integer k, we want the shortest path from s to t that uses at most k edges.

For each vertex v and each integer $i \le k$, define dist(v,i) = the length of the shortest path from s to t that uses i edges. Base case is dist(s, 0) = 0,dist(v, 0) = ∞, for all vertices except s.

$$\text{dist}(v,i) = \min_{(u,v)\in E}\{\text{dist}(u, i-1) + l(u,v)\}$$

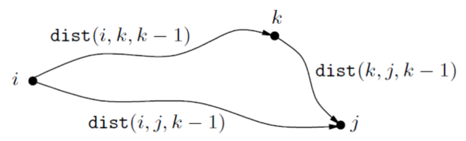
3.8 Floyd-Warshall algorithm

How to find the shortest path between all pairs of vertices? Running single-source algorithm $|V|$ times, once for each starting node takes $|V|\times$ Bellman-Ford $= O(|V|^2|E|)$. $\rightarrow$ Can we do better?

Consider $V = \{1, \cdots, n\}$ and subproblem dist(i,j,k) = the length of the shortest path from i to j in which only nodes $\{1, \cdots, k\}$ can be used as intermediate nodes (*permissible intermediate nodes*).

$$\text{dist}(i,j,k) = \min\{\text{dist}(i,k,k-1) + \text{dist}(k,j,k-1), \text{dist}(i,j,k-1)\}$$

$\rightarrow O(n^3)$ algorithm, which is much efficient than $O(|V|^2|E|)$ especially in dense graph.



```
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        dist$(i, j, 0) \leftarrow \infty$
for all $(i, j) \in E$ do
    dist$(i, j, 0) \leftarrow l(i, j)$
for $k = 1$ to $n$ do
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            ···                                ▷ Relation equation above
```

3.9 Independent set

A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V, E)$ if there are no edges between them. Finding the largest independent set in a graph is intractable, *i.e.* no polynomial time algorithm.

3.9.1 Independent set in trees

Start by rooting the tree at any node r. Now, each node defines a subtree–the one hanging from it. Define $I(u)$ = size of largest independent set of subtree hanging from u. Suppose we know $I(w)$ for all descendants w of u. If the independent set

- Include u: cannot include children; move on to grandchildren.
- Don't include u: move on to children.

$$I(u) = \max\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w) + \sum_{\text{children } w \text{ of } u} I(w)\}$$

The number of subproblems is exactly the number of vertices. $\rightarrow O(|V| + |E|)$.

3.10 Traveling salesman problem (TSP)

Given n cities and the matrix of intercity distances $D = (d_{ij})$, find a tour that starts and ends at node 1, includes all other cities exactly once, and has minimum total length. Brute-force algorithm is trying all possible tour $\rightarrow O(n!)$

For a subset of cities $S \subseteq \{1, \cdots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

```
$C(\{1\}, 1) \leftarrow 0$
for $s = 2$ to $n$ do
    for all subsets $S \subseteq \{1, \cdots, n\}$ of size $s$ and containing 1 do
        $C(S, 1) \leftarrow \infty$                  ▷ cannot both start and end at 1
        for all $j \in S, j \neq 1$ do
            $C(S, j) \leftarrow \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}$   ▷ second-to-last $i$
return $\min_j C(\{1, \cdots, n\}, j) + d_{j1}$
```

There are at most $2^n \cdot n$ subproblems, and each subproblem takes $O(n) \rightarrow$ Total $O(n^2 2^n)$

3.11 Coin change problem

Given a set of denominations $D = d_1, \cdots, d_k$, find the minimum number of coins that sums up to n. Assume each d_i is an integer and $d_1 > \cdots > d_k$ and $d_k = 1$ so that there is always a solution.

Greedy algorithm repeatedly chooses the largest coin less than or equal to the remaining sum, until the desired sum is obtained.

- For $D = \{25, 10, 5, 1\}$, greedy algorithm works.
- For $D = \{25, 10, 1\}$, greedy does not work.

3.11.1 Dynamic programming

Define $C[j]$ to be the minimum number of coins for j cents. denom$[j]$ is the denomination of a coin used for j cents (for recording)

```
$C[0] \leftarrow 0$
for $j = 1$ to $n$ do
    $C[j] \leftarrow \infty$
    for $i = 1$ to $k$ do
        if $j \geq d_i$ and $1 + C[j - d_i] < C[j]$ then
            $C[j] \leftarrow 1 + C[j - d_i]$
            denom$[j] \leftarrow d_i$
```

$\rightarrow O(nk)$

3.12 Greedy vs. dynamic programming

Greedy-choice property: optimal solution includes greedy choice.

Fractional knapsack problem ($\leftrightarrow$ 0-1 knapsack) can take fraction of an item. Greedy algorithm works for fractional knapsack problem.

4 NP-Completeness

4.1 Hardness of problems

We say that an algorithm is efficient if its running time is polynomial–$O(n^k)$ for some constant k.

- *Tractable/easy*: solvable by polynomial-time algorithms
- *Intractable/hard*: require superpolynomial time algorithms

NP-complete problems are *currently* status unknown:

- No polynomial time algorithm has yet been discovered
- No proof that no polynomial-time algorithm can exist.

4.2 P, NP, co-NP

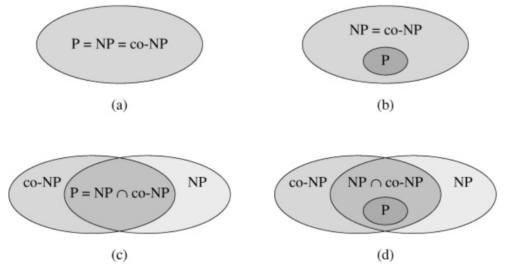
A *decision problem* is a problem whose output is YES or NO.

- **P**: the set of decision problems that are solvable in polynomial time
- **NP**: the set of decision problems that are *verifiable* in polynomial time–if the answer is YES, then there is a *certificate* that can be checked in polynomial time.
- **co-NP**: the opposite of NP. If the answer is NO, then there is a *certificate* that can be checked in polynomial time.

Every decision problem in P is also in NP. Every decision problem in P is also in co-NP. **Open questions**:

Is P ≠ NP? / Is NP ≠ co-NP?

4.2.1 Four cases



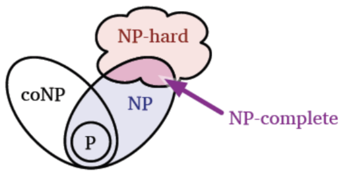
(a) P=NP=co-NP (most unlikely)
(b) If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP.
(c) P=NP ∩ co-NP, but NP is not closed under complement.
(d) NP ≠ co-NP and P ≠ NP ∩ co-NP (most likely)

4.3 NP-hard, NP-complete

A problem Π is *NP-hard* if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for every problem in NP.

> Π is NP-hard $\Leftrightarrow$ If Π can be solved in polynomial time, P=NP.

A problem is *NP-complete* if it is both NP-hard and in NP. NP-complete problems are the hardest problems in NP.



What we *think* the world looks like

4.4 Circuit satisfiability problem

Consider a boolean circuit (a collection of AND, OR, and NOT gates connected by wires).

- Input: a set of m boolean values $x_1, \cdots, x_m$
- Output: a single boolean value.

Given specific input, we can calculate the output in linear time using DFS, since we can compute the output of a k-input gate in $O(k)$. The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output TRUE, or conversely, the circuit always outputs FALSE (*unsatisfiable*).

- Nobody solved this problem faster than trying all 2^m possible input, nor proved that this is the best we can do.
- It is in NP. If the answer is YES, then any set of m input values that produces TRUE output is a certificate. We can verify the certificate in polynomial time by evaluating the circuit.
- It is NP-complete. (Cook-Levin Theorem)

4.5 Tautology

The formula ϕ is a *tautology* if it evaluates to 1 for every input. The problem of deciding whether a formula is a tautology is in co-NP.

4.6 NP-completeness proofs

To prove that a problem A is NP-complete,

- Prove A is NP in by polynomial-time verification for certificate
- Prove A is NP-hard by reduction argument
  1. Show a polynomial-time transformation from input x of B to input $f(x)$ of A
  2. Explain $f(x)$ is YES for A if and only if x is YES for B

4.6.1 Reduction

To prove that problem A is NP-hard, use *reduction argument*, *i.e.* reduce a known NP-hard problem B to A ($B \le_p A$). If any subroutine for task Q can also be used to solve P, we say ***P* reduces to *Q*** ($P \le_p Q$). For example, the longest path in a dag reduces to the shortest path in a dag by negating all edge weights.

4.7 Formula satisfiability problem (SAT)

Let ϕ be a boolean formula constructed from the boolean input variables $x_1, \cdots, x_k, \neg, \wedge, \vee, \Rightarrow, \cdots$, and parentheses.

The formula satisfiability problem (SAT) asks whether it is possible to assign boolean values to the variables so that the formula evaluates to TRUE.

4.7.1 SAT is NP

If the formula is satisfiable, we can verify the ceritficate in polynomial (linear) time by simply replacing each variable in the formula with its assigned value.

4.7.2 SAT is NP-hard

Theorem 4

Circuit SAT $\le_p$ SAT

Proof. Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate including final output and then writing down the list of gates separated by ANDs. The original circuit is satisfiable if and only if the resulting formula is satisfiable.

($\rightarrow$) Given a satisfying input to the circuit, we can get a satisfying assignment for the formula by computing the output of every gate.

($\leftarrow$) Given a satisfying assignment for the formula, we can get a satisfying input to the circuit by just ignoring internal and output variables.

We can transform any boolean circuit into a formula in linear time using depth-first search. The size of the resulting formula is O(size of the circuit). Thus, we have a polynomial-time reduction from circuit satisfiability to SAT. □

$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{boolean formula}} \arrow[d, Rightarrow, "\text{SAT}"] \\
\boxed{\text{TRUE or FALSE}} & \boxed{\text{TRUE or FALSE}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{SAT}}(O(n)) \implies T_{\text{SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

4.8 3-SAT

Call a boolean variable or its negation a *literal*. A boolean formula is in *conjunctive normal form* (CNF) if it is a sequence of clauses connected by $\wedge$, and each clause is a sequence of literals connected by $\vee$. A 3-CNF formula is a CNF formula with exactly three distinct literals per clause. We have trivial reduction 3-SAT $\le_p$ SAT.

4.8.1 3-SAT is NP-hard

Lemma 2

We can transform one- and two-literal clauses to three literal.

Proof.

$$\begin{aligned}
a &= (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \\
a \vee b &= (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\
a \wedge b &= \text{use formula for one-literal twice}
\end{aligned}$$

□

Lemma 3

We can transform secondary operations $\rightarrow, \leftrightarrow, \oplus$ to CNF.

Proof.

$$\begin{aligned}
x \rightarrow y &= \bar{x} \vee y \\
x \leftrightarrow y &= (x \vee \bar{y}) \wedge (\bar{x} \vee y) \\
x \oplus y &= (x \vee y) \wedge (\bar{x} \vee \bar{y})
\end{aligned}$$

□

Theorem 5

SAT $\leq_p$ 3-SAT

Proof. First, we construct a binary *parse* tree for the input formula ϕ, with literals as leaves and connectives as internal nodes. Should the input formula contain a clause such as the OR of several literals, we use associativity to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. We introduce a variable y_i for the output of each internal node.

Then, we rewrite ϕ as ϕ', the AND of the root variable y_1 and a conjunction of clauses describing the operation of each node, ($y_i \leftrightarrow$ node operation). Using Lemmas 2,3, we can transform ϕ' to 3-CNF formula ϕ''.

3-CNF formula ϕ'' is satisfiable if and only if ϕ' is satisfiable. Similar to proof of Theorem 4, satisfiability is conserved from ϕ to ϕ', ϕ'' and ϕ' is algebraically equivalent. So construction of formula can easily be accomplished in polynomial time. □

Theorem 6

Circuit SAT $\leq_p$ 3-SAT

Proof. First, make every AND, OR gate has only two inputs. If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ two-input gates. Write down the circuit as a formula, with one clause per gate. Change every gate clause into a CNF formula ϕ:

$$a = b \wedge c \mapsto (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b) \wedge (\neg a \vee c)$$
$$a = b \vee c \mapsto (\neg a \vee b \vee c) \wedge (a \vee \neg b) \wedge (a \vee \neg c)$$
$$a = \neg b \mapsto (a \vee b) \wedge (\neg a \vee \neg b)$$

By Lemma 2, we can transform ϕ to 3-CNF formula ϕ'. Circuit is satisfiable if and only if ϕ' is satisfiable (See proof of Theorem 4). □

4.9 Optimization problem

We want to find a feasible solution with the best value. NP-completeness applies to decision problem, so cast optimization problem to decision problem. For example of MST, ask whether there is a spanning tree with a cost at most k.

We can solve decision problem by solving optimization problem. So, decision problem is at least no harder than optimization problem. By showing a decision problem is NPC, we provide an evidence that the optimization problem is hard.

4.10 Independent set (IND-SET)

Let $G = (V, E)$ be an undirected graph. A subset $W \subseteq V$ is independent if none of the vertices in W are adjacent.

- Optimization: Size of the largest independent set in G
- Decision: Is there an independent set of k or more vertices?

4.10.1 IND-SET is NP

Given certificate of size k, check if there is any pair among the set is adjacent $\to O(k^2)$.

4.10.2 IND-SET is NP-hard

Theorem 7

3-SAT $\leq_p$ IND-SET

Proof. We will transform a 3-CNF formula with k clauses into a graph that has an independent set of a k if and only if the formula is satisfiable. The graph has one vertex for each instance of each literal in the formula. Two nodes are connected by an edge if they are

(i) Literals in the same clause
(ii) A variable and its inverse

($\rightarrow$) If we have a satisfying assignment, then we can choose one literal in each clause that is TRUE. Those literals form an independent set in the graph because (i) they are in different clause, (ii) and cannot be inverse of each other since they are all TRUE.

($\leftarrow$) If the graph has an independent set of k vertices, then each vertex must come from a different clause. Assign TRUE to each literal in the independent set. This assignment is consistent since contradictory literals are connected by edges. Assign any value to variables that have no literal in the independent set. The resulting assignment satisfies the original 3-CNF formula.

The reduction from 3-CNF formula to graph takes polynomial time and the graph size is O(size of 3-CNF formula). □

4.11 Clique problem (CLIQUE)

A *clique* is a complete subgraph of undirected graph $G = (V, E)$, that is, for a subset $V' \subseteq V$ of vertices, each pair of V' is connected by an edge in E. The size of a clique is the number of vertices it contains.

- Optimization: Size of the largest clique in G
- Decision: Is there a clique of size k?

4.11.1 CLIQUE is NP-hard

Theorem 8

3-SAT $\leq_p$ CLIQUE

Proof. We will transform a 3-CNF formula with k clauses into a graph that has an clique of size k if and only if the formula is satisfiable. The graph has one vertex for each instance of each literal in the formula. Two nodes are connected by an edge if they are

(i) Literals **not** in the different clause
(ii) **Not** a variable and its inverse

($\rightarrow$) If we have a satisfying assignment, there is at least one true literal in each clause. The true literals form a clique.

($\leftarrow$) If the graph has clique of size k, it covers all clauses and thus implies a satisfying truth assignment. □

4.12 Vertex cover problem (VERTEX-COVER)

Given a graph $G = (V, E)$, a subset $V' \subseteq V$ is a *vertex cover* if every edge has at least one endpoint in V'.

- Optimization: Size of the smallest vertex cover in G
- Decision: Is there a vertex cover of size k?

4.12.1 VERTEX-COVER is NP-hard

Theorem 9

CLIQUE $\leq_p$ VERTEX-COVER

Proof. G has a clique of size k if and only if the complement of G has a vertex cover of size $|V| - k$.

($\rightarrow$) Suppose G has a clique $V' \subseteq V$ with $|V'| = k$. Let (u, v) be any edge in the complement of G. So at least one of u or v does not belong to V', which means edge (u, v) is covered by $V - V'$. Thus, $V - V'$ forms a vertex cover of the complement of G with size $|V| - k$.

($\leftarrow$) Suppose that the complement of G has a vertex cover $V' \subseteq V$ where $|V'| = |V| - k$. Then, $\forall u, v \in V$, if (u, v) is an edge in the complement of G, then $u \in V'$ or $v \in V'$ or both. The contrapositive is that $\forall u, v \in V$, if $u \notin V', v \notin V'$, then $(u, v) \in E$. Thus, $V - V'$ is a clique and its size is $|V| - |V'| = k$. □

Theorem 10

1. V' is a clique of G.
2. V' is an independent set of the complement of G.
3. $V - V'$ is a vertex cover of the complement of G.

Above are equivalent for $G = (V, E)$ and subset V' of V.

4.13 Hamiltonian cycles

A hamiltonian cycle in a graph is a cycle that visits every vertex exactly once. The graph G is hamiltonian if it has a hamiltonian cycle. The *hamiltonian cycle problem* (HAM) asks whether a given graph is hamiltonian. Given two vertices s, t, the *hamiltonian path problem* (HAM-PATH) asks whether G has a path from s to t that goes every vertex exactly once.

Theorem 11

HAM-PATH $\le_p$ HAM

Proof. Map an instance $G = (V, E), s, t$ of HAM-PATH to an instance $G' = (V', E')$ of the HAM where $V' = V \cup \{x\}$ and $E' = E \cup \{\{s, x\}, \{x, t\}\}$ G' has a Hamiltonian cycle if and only if G has a hamiltonian path from s to t. ($\rightarrow$) Delete edges $\{s, x\}, \{x, t\}$ ($\leftarrow$) Add edges $\{s, x\}, \{x, t\}$ □

4.13.1 HAM is NP

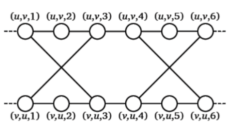
The certificate is the sequence of $|V|$ vertices that make up the cycle. We can check wether the sequence contains each vertex in V once and forms a cycle in polynomial time.

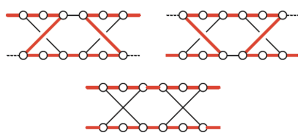
4.13.2 HAM is NP-hard

Theorem 12

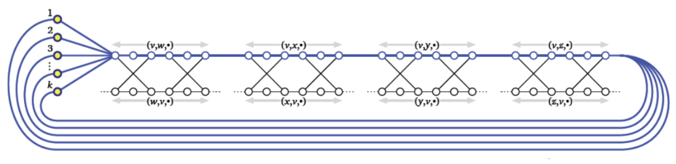
VERTEX-COVER $\le_p$ HAM

Proof. Given a graph G and integer k, we will transform it to graph G'. For each edge $(u, v) \in G$, we have an *edge gadget* in G' consisting of 12 vertices and 14 edges (left). 4 corner vertices $(u, v, 1), (u, v, 6), (v, u, 1), (v, u, 6)$ have an edge leaving the gadget. A hamiltonian cycle can only pass through an edge gadget in only 3 ways (right).

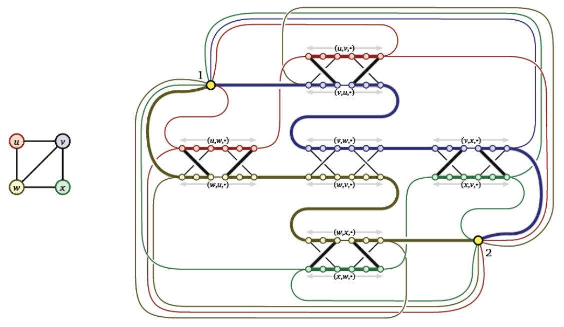




G' also contains k *cover vertices*, simply numbered 1 through k. For each vertex u in G, we string together all the edge gadgets for edges (u, v) into single *vertex chain*, and then connect edges of the chain to all cover vertices. Suppose vertex u has d neighbors $v_1, \cdots, v_d$. Then, G' has $d - 1$ edges between $(u, v_i, 6)$ and $(u, v_{i+1}, 1)$, plus k edges between the cover vertices and $(u, v_1, 1)$, and finally k edges between cover vertices and $(u, v_d, 6)$.



For example, the original graph with vertex cover $\{v, w\}$ (left) can be transformed to graph G' (right) with a corresponding hamiltonian cycle. Chains are colored to match their corresponding vertices.



G' has a hamiltonian cycle if and only if G has a vertex cover of size k.

($\leftarrow$) Consider a vertex cover of G, $\{v_1, \cdots, v_k\}$. Hamiltonian cycle of G' starts at cover vertex 1, traverses the vertex chain for v_1, then visits cover vertex 2, then traverses the vertex chain for v_2, and so forth, finally returns to cover vertex 1.

($\rightarrow$) Consider a hamiltonian cycle C in G'. C alternates between cover vertices and vertex chains, and the vertex chains correspond to the k vertices in a vertex cover of G.

The size of G' is polynomial in the size of G, and hence we can construct G' in polynomial time in the size of G (in $O(n^2)$ time). □

4.14 Traveling salesman problem (TSP)

Assume a complete graph. Each edge has a non-negative integer weight. The traveling salesman problem (TSP) asks whether there is a permutation of the vertices s.t. the sum of edges connecting contiguous vertices (and the last vertex to the first) is k or less.

4.14.1 TSP is NP

The verification algorithm checks that the tour sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is $\le k$, which can be done in polynomial time.

4.14.2 TSP is NP-hard

Theorem 13

HAM $\le_p$ TSP

Proof. For a graph G in hamiltonian cycle problem, construct an instance of TSP by forming the complete graph $G' = (V, E')$ where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$. If there is an edge $(i, j) \in G$, assign 0 as edge weight, otherwise 1. G has a hamiltonian cycle h if and only if G' has a TSP tour with weight 0.

($\leftarrow$) Suppose that G has a hamiltonian cycle h. Then, each edge in h has weight 0 in G'. Thus, h is a tour in G' with weight 0.

($\rightarrow$) Suppose that G' has a TSP tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of h' is 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. Thus, h' is a hamiltonian cycle in G. □

4.15 Other NP-hard problems

- SUBSET-SUM: Given finite set S of positive integers and an integer target $t > 0$, is there a subset $S' \subseteq S$ whose element sum to t?
- 3-COLOR: Given a graph, can it be 3-colored? (2-COLOR $\in P$)

4.16 Hierarchy of complexity classes

The set of decision problems that can be solved

- PSACE: using polynomial space
- EXP: in exponential time
- NEXP: in nondeterministic exponential time; for every YES instance, there is a certificate of this fact that can be checked in exponential time.
- EXPSPACE: using exponential space

P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXP $\subseteq$ NEXP $\subseteq$ EXPSPACE

5 Approximation algorithms

How to deal with intractable problems?

- For small input size, just use exponential algorithm
- Special subclasses of hard problems can have polynomial-time algorithms. You can find an optimal vertex cover for a tree in linear time. Also, DFN-SAT, 2-CFN-SAT $\in P$.

Or we can use *approximation algorithms*, a polynomial-time algorithm to find near-optimal solutions.

5.1 Approximation ratio

Consider optimization problems whose solutions have a positive cost. An algorithm has an approximation ratio $\rho(n)$ if, for every input of size n, the cost C of the produced solution satisfies $\max\{C/C^*, C^*/C\} \leq \rho(n)$ where C^* is the cost of the optimal solution. For minimization problems, $C^* \leq C$, for maximization problems, $C \leq C^*$. An algorithm with approximation ratio $\rho(n)$ is a *$\rho(n)$-approximation algorithm.*

5.2 Approximation algorithm for VERTEX-COVER

```
function Approx-Vertex-Cover($G$)
    $C \leftarrow \emptyset$
    $E' \leftarrow E[G]$
    while $E' \neq \emptyset$ do
        Let $(u, v)$ be an arbitrary edge of $E'$
        $C \leftarrow C \cup \{u, v\}$
        Remove from $E'$ every edge incident on either $u$ or $v$
    return $C$
```

Theorem 14

Approx-Vertex-Cover is a polynomial-time 2-approximation algorithm.

Proof. Approx-Vertex-Cover runs in $O(V + E)$ time. C is a vertex cover since the algorithm loops until every edge $E[G]$ has been covered by some vertex in C.

Let A be the set of edges that were picked. In order to cover the edges in A, any vertex cover (including optimal cover C^*) must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked, all other edges incident on its endpoints are deleted. Thus, no two edges in A are covered by the same vertex from C^*, $|C^*| \geq |A|$. Therefore, $|C| = 2|A| \leq 2|C^*|$ □

5.3 Approximation algorithm for TSP

5.3.1 TSP with triangle inequality

Given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ for each edge $(u, v) \in E$, find a hamiltonian cycle (a tour) of G with minimum cost. Cost function c satisfies the triangle inequality if, for all vertices $u, v, w \in V$, $c(u, w) \leq c(u, v) + c(v, w)$. Many practical applications satisfy the triangle inequality. TSP is NP-complete even if we require that the cost function satisfies the triangle inequality. The proof is similar to proof of Theorem 13, but set the cost to 1 and 2 so that it satisfies the triangle inequality.

```
function Approx-TSP-Tour($G, c$)
    root $\leftarrow$ some vertex $\in V[G]$
    $T \leftarrow$ Prim($G, c, r$)
    $H \leftarrow$ list of vertices ordered according to when they are first
visited in a preorder tree walk of $T$
        return hamiltonian cycle $H$
```

Theorem 15

Approx-TSP-Tour is a polynomial-time 2-approximation algorithm with the triangle inequality.

Proof. Approx-TSP-Tour runs in $O(V^2)$ time since Prim is $O(V^2)$ time algorithm and preorder tree walk takes $O(V)$.

Consider an optimal tour H^* and the MST for given graph, T, which is obtained by removing any edge from H^*. Therefore, $c(T) \leq c(H^*)$. Consider a full walk of T, W. We visit each edge twice, so $c(W) = 2c(T)$. Thus, $c(W) \leq 2c(H^*)$. The Approx-TSP-Tour returns H, the cycle corresponding to the preorder walk. Thus, $c(H) \leq c(W) \leq 2c(H^*)$. □

5.3.2 General TSP

Theorem 16

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time ρ-approximation algorithm for the general TSP.

Proof. Suppose that there exists a polynomial-time ρ-approximation algorithm A for a constant ρ. We can use A to solve HAM in polynomial time.

Let $G = (V, E)$ be an instance of HAM and $G' = (V, E')$ be the complete graph on V. $E' = \{(u, v) : u, v \in V \text{ and } u \neq v\}$.

$$\begin{cases} c(u, v) = 1 & \text{if } (u, v) \in E \\ \rho|V| + 1 & \text{otherwise} \end{cases}$$

Consider the TSP (G', c). If the graph G has a hamitonian cycle H, then (G', c) contains a tour of cost $|V|$. If G does not contain a hamiltonian cycle, then a tour has a cost of at least $(\rho|V| + 1) + (|V| - 1) > \rho|V|$. So, we can use A to solve the hamiltonian-cycle problem in polynomial time. Since HAM is NP-complete, if we can solve it in polynomial time, P=NP. Contradiction. □

A CLRS

A.1 Dynamic programming

A.1.1 Matrix chain multiplication

Let us define $m[i, j]$ as the number of computation required to multiply matrices $A_i, \cdots, A_j$.

```
function MATRIX-CHAIN-ORDER($p$)
    Let $m[1..n, 1..n]$
    Let $s[1..n-1, 2..n]$          ▷ record optimal parenthesization
    for $i = 1$ to $n$ do
        $m[i, i] \leftarrow 0$
    for $l = 2$ to $n$ do                          ▷ chain length
        for $i = 1$ to $n - l + 1$ do
            $j \leftarrow i + l - 1$
            $m[i, j] \leftarrow \infty$
            for $k = i$ to $j - 1$ do
                $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$
                if $q < m[i, j]$ then
                    $m[i, j] \leftarrow q$
                    $s[i, j] \leftarrow k$
    return $m, s$
```

A.1.2 Longest common subsequence (LCS)

Given two sequences X, Y, we say that a sequence Z is a common subsequence (not necessarily consecutive) of X, Y if Z is a subsequence of both. **Let us define** $c[i, j]$ as the length of LCS of $X[1..i]$ and $Y[1..j]$.

```
function LCS-LENGTH($X, Y$)
    Let $c[0..m, 0..n]$
    Let $b[1..m, 1..n]$          ▷ record optimal character choice
    for $i = 1$ to $m$ do
        $c[i, 0] \leftarrow 0$
    for $j = 0$ to $n$ do
        $c[0, j] \leftarrow 0$
    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            if $X_i = Y_i$ then
                $c[i, j] \leftarrow c[i-1, j-1] + 1$
                $b[i, j] \leftarrow$ hit
            else if $c[i-1, j] \geq c[i, j-1]$ then
                $c[i, j] \leftarrow c[i-1, j]$
                $b[i, j] \leftarrow$ decrement $X$
            else
                $c[i, j] \leftarrow c[i, j-1]$
                $b[i, j] \leftarrow$ decrement $Y$
    return $c, b$
```

A.1.3 Optimal BST

Given a sequence $K =< k_1, \cdots, k_n >$ of n distinct keys in ascending order. For each key k_i, we have a probability p_i that a search will be for k_i. Then the expected cost of a search in T is

$$e(T) = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

We wish to construct a binary search tree whose expected search cost is smallest. **Let us define** $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \cdots, k_j$.

$$e[i, j] = \min_{i \leq r \leq j}(e[i, r-1] + e[r+1]) + \sum_{i \leq k \leq j} p_k$$

```
function OPTIMAL-BST-COST($p$)
    Let $e[1..n, 1..n]$
    Let $w[0..n]$
    Let root$[1..n, 1..n]$          ▷ record optimal root
    for $i = 0$ to $n$ do
        $e[i, i] \leftarrow 0$
        $w[i] \leftarrow w[i-1] + p[i]$
    for $l = 1$ to $n$ do
        for $i = 1$ to $n - l$ do
            $j \leftarrow i + l$
            for $r = i$ to $j$ do
                val $\leftarrow e[i, r-1] + e[r+1, j] + (w[j] - w[i-1])$
                if val $< e[i, j]$ then
                    $e[i, j] \leftarrow$ val
                    root$[i, j] \leftarrow r$
```