Video 4.1
Sampath Kannan

Shortest Paths

- How does Google Maps plan your route?
- How is email sent to its destination on the internet?

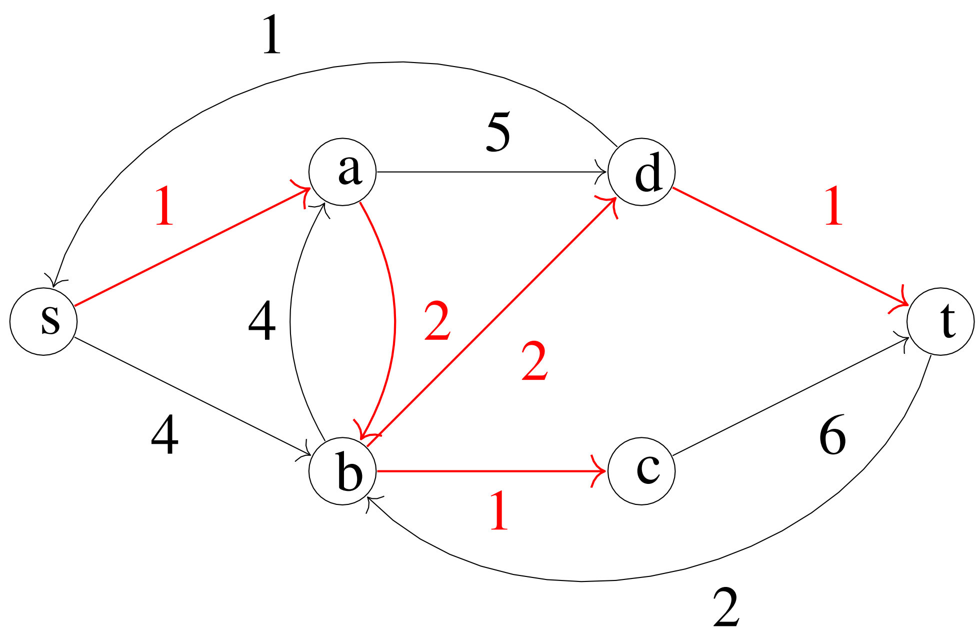
Shortest Paths

- How does Google Maps plan your route?
- How is email sent to its destination on the internet?
- Weighted graphs! Weight = Distance.

Shortest Paths

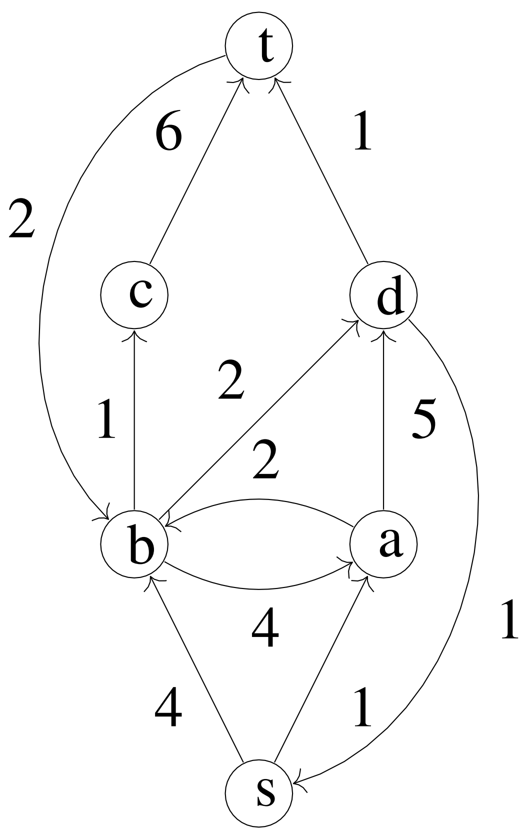
- How does Google Maps plan your route?
- How is email sent to its destination on the internet?
- Weighted graphs! Weight = Distance.
- But edges are directed now (one-way streets, asymmetric links, etc)
- (if a link is two-way we can always draw two one-way links)

Single-Source Shortest Path Problem



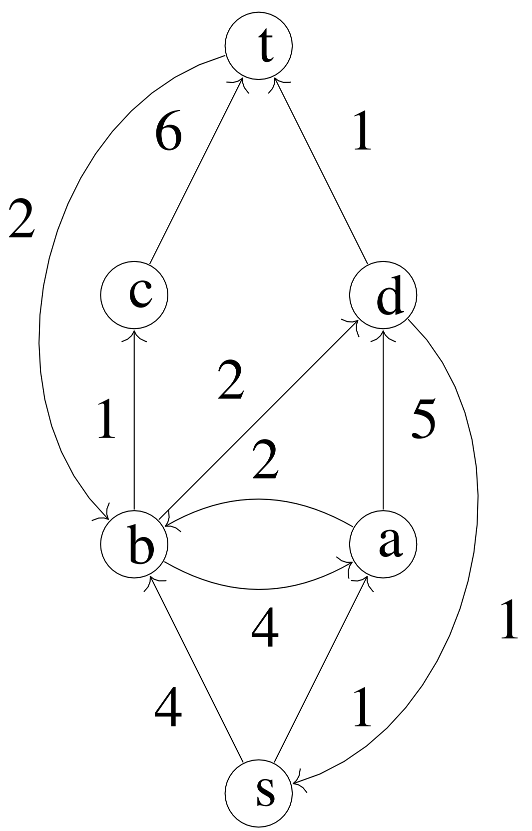
- Given a weighted, directed graph and a vertex s, find the shortest paths from s to all other vertices
- We will assume the weights are positive

Greedy Approach



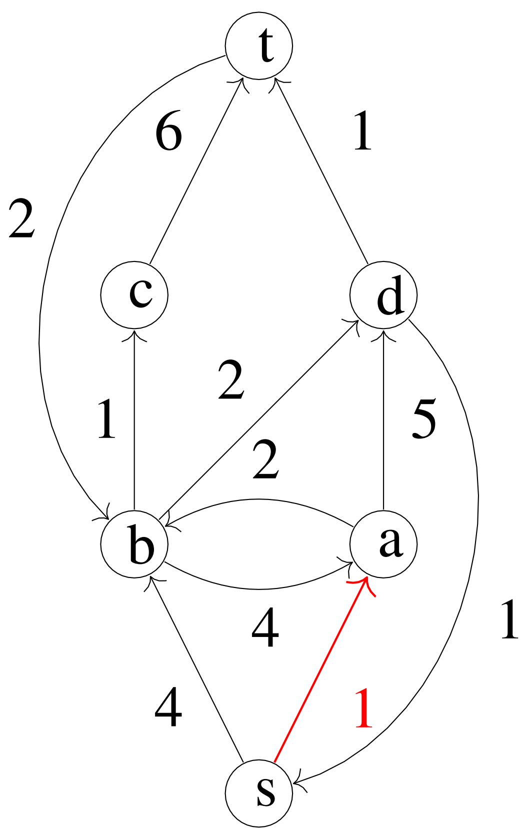
- What is a good greedy approach? Can we make any decision "for sure" right now?
- Problem: shortest path from s to a vertex v could be a single edge, or a path of multiple edges.

Greedy Approach



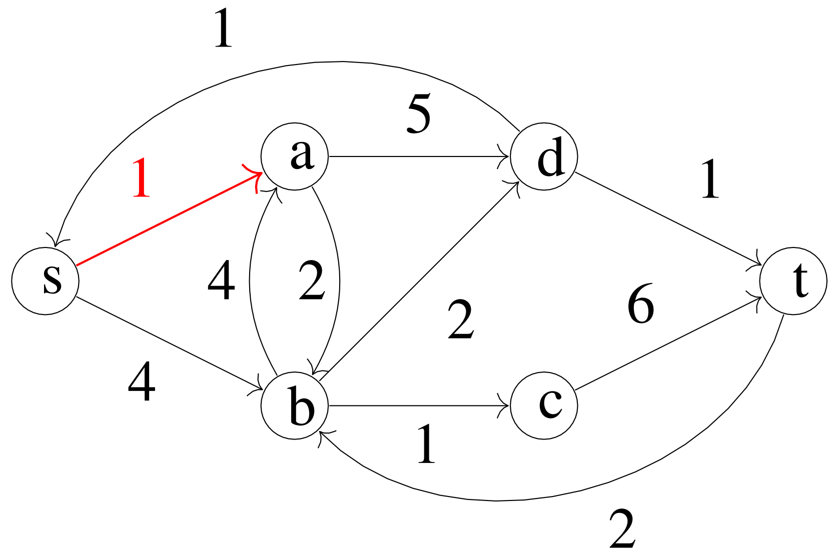
- What is a good greedy approach? Can we make any decision "for sure" right now?
- Problem: shortest path from s to a vertex v could be a single edge, or a path of multiple edges.
- Except the the shortest path to s. It is of length and cost 0.
- For what neighbor of s can we be sure the shortest path is a single edge?

Greedy Approach

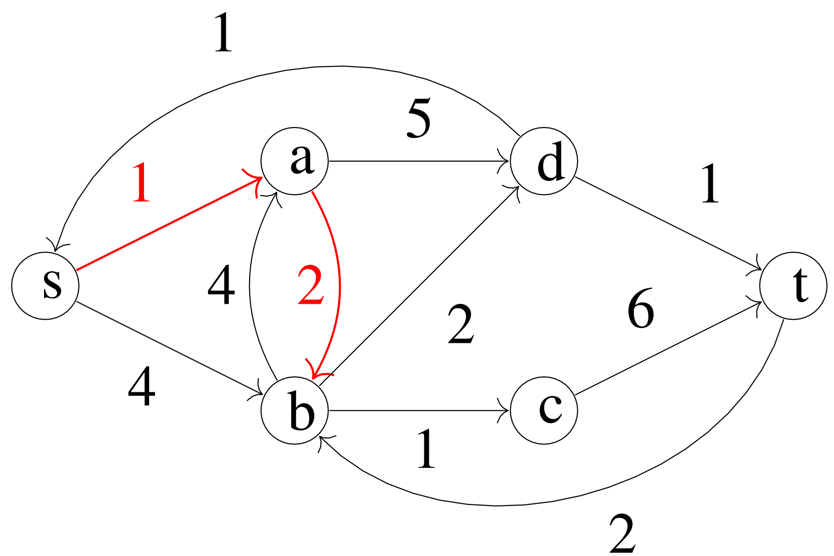


- What is a good greedy approach? Can we make any decision "for sure" right now?
- Problem: shortest path from s to a vertex v could be a single edge, or a path of multiple edges.
- Except the the shortest path to s . It is of length and cost 0.
- For what neighbor of s can we be sure the shortest path is a single edge?
- For the neighbor a that is closest to s. Let w(s, a) = d(a) be the cost of this path
- There can be no shorter path to a. Even the first edge from s on some other path costs more than d(a).

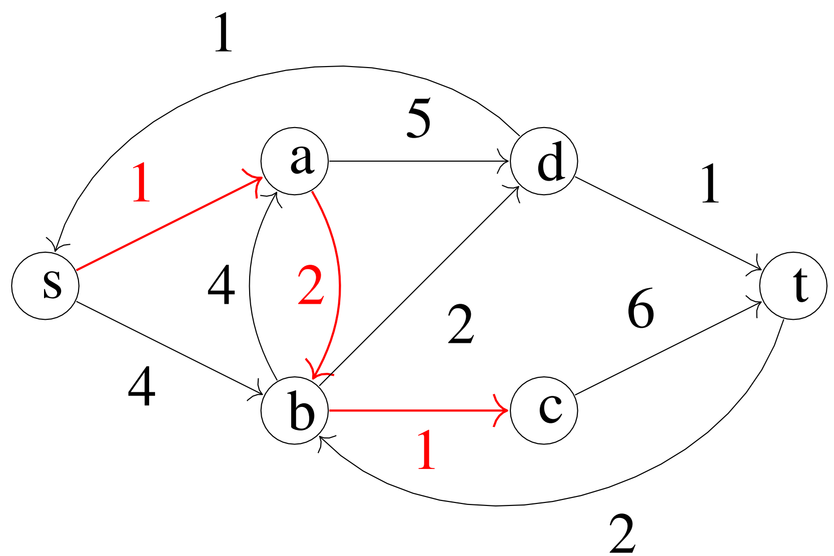
What is the next vertex for which we know the shortest path?



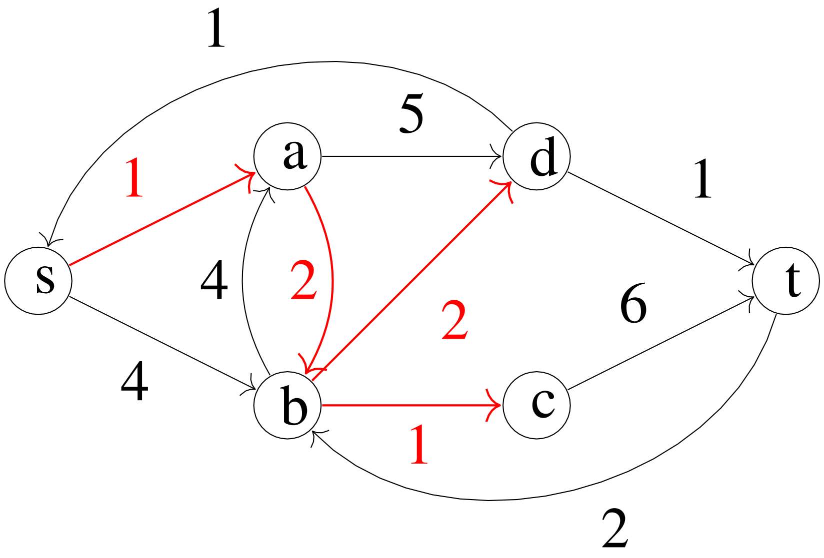
What is the next vertex for which we know the shortest path?



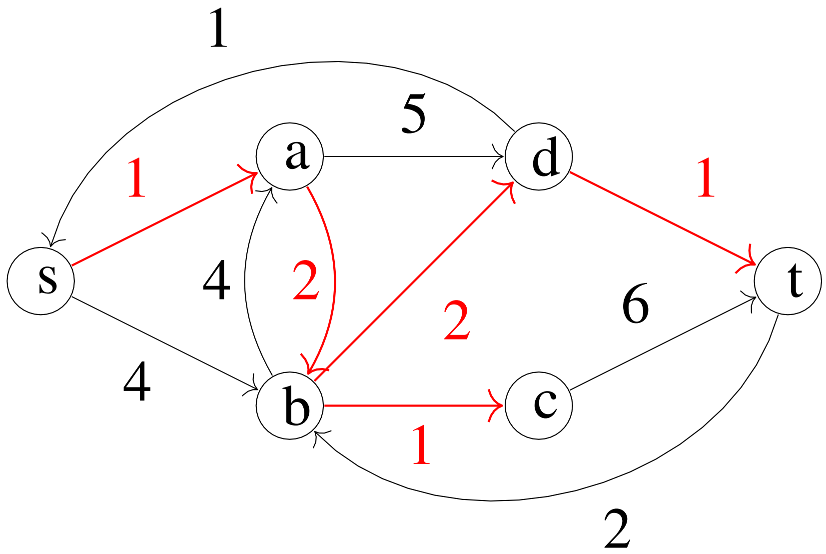
What is the next vertex for which we know the shortest path?



What is the next vertex for which we know the shortest path?



What is the next vertex for which we know the shortest path?



Dijktra's Algorithm

- Dijkstra's algorithm is a greedy one that repeatedly chooses the next vertex to which the shortest path is known.

Dijktra's Algorithm

- Dijkstra's algorithm is a greedy one that repeatedly chooses the next vertex to which the shortest path is known.
- It maintains a set S of such vertices, initially just {s}.
- For all vertices $v \in V - S$ it maintains a distance $d(v)$ which is the length of the shortest path to v passing only though vertices in S.

Dijktra's Algorithm

- Dijkstra's algorithm is a greedy one that repeatedly chooses the next vertex to which the shortest path is known.
- It maintains a set S of such vertices, initially just $\{s\}$.
- For all vertices $v \in V - S$ it maintains a distance $d(v)$ which is the length of the shortest path to v passing only though vertices in S.
- At each stage it brings the vertex u with minimum $d(u)$ into S and updates the values of $d(v)$ for all vertices v that are still in $V - S$.

Video 4.2

Sampath Kannan

Dijkstra's Pseudocode

```
dijkstra(G, w, s):
    S = {s}
    d(s) = 0
    for all u in G.out-neighbor(s):
        d(u) = w(s, u):
    for all u != s and u not in G.out-neighbor(s):
        d(u) = infinity
    while S != G.V:
        u = argmin d(v) over all v in G.V-S
        add u to S
        for each v in G.adj(u):
            d(v) = min(d(v), d(u) + w(u,v))
            //This maintains the property that d(w) is
            //length of s->v path going only through
            //vertices in S
```

Why does this work?

- **Lemma 1**: For any vertex v, d(v) cannot increase as the algorithm progresses.

Why does this work?

- **Lemma 1**: For any vertex v, d(v) cannot increase as the algorithm progresses.
- **Lemma 2**: If u is brought into S before v, then d(v) ≥ d(u).

Why does this work?

- **Lemma 1**: For any vertex v, $d(v)$ cannot increase as the algorithm progresses.
- **Lemma 2**: If u is brought into S before v, then $d(v) \geq d(u)$. **Proof by contradiction**: Suppose $d(v) < d(u)$. Consider the shortest path from s to v and let w be the first vertex on the path brought in after u. We have $d(w) < d(v) < d(u)$. So the algorithm should have brought in w before u.

  Contradiction!

Why does this work? cont.

- **Theorem**: At all points in the algorithm the following are true:
  - For any $v \in S$, $d(v)$ is the length of the shortest path from s to v
  - For any $v \in V - S$, $d(v)$ is the minimum length of an $s--v$ path with intermediate vertices only in S.
- **Base Case**:
  - Initially $S = \{s\}$ and $d(s) = 0$, which is the length of the shortest path from s to s.
  - The only paths with all intermediate vertices in S are exactly the length-1 paths, for which $d(u)$ is correctly set. For all other vertices there are no such paths and $d(v) = \infty$.

Why does this work? cont.

- **Theorem**: At all points in the algorithm the following are true:
  - For any $v \in S$, $d(v)$ is the length of the shortest path from s to v
  - For any $v \in V - S$, $d(v)$ is the minimum length of an $s - -v$ path with intermediate vertices only in S.
- **Inductive Hypothesis**: Assume these statements are true just before bringing v into S. What happens after?

Why does this work? cont.

- **Theorem**: At all points in the algorithm the following are true:
  - For any $v \in S$, $d(v)$ is the length of the shortest path from s to v
  - For any $v \in V - S$, $d(v)$ is the minimum length of an $s - -v$ path with intermediate vertices only in S.
- **Inductive Hypothesis**: Assume these statements are true just before bringing v into S. What happens after?
- **Inductive Step**:
  - At this point $d(v)$ is the length of the shortest path to v. A future vertex u coming into s has $d(u) > d(v)$ and will not be on the shortest path to v . Thus $d(v)$ is the true length of the shortest path from s to v in the graph.
  - For any vertex u in $V - S$, either the shortest path to u with all intermediate vertices in S passes through v or it doesn't. If it doesn't then we already have its length in $d(u)$, if it does it's length is $d(v) + w(u, v)$.

Correctness of Dijkstra's

- The theorem in the previous slide ensures that Dijkstra's algorithm is correct.

Correctness of Dijkstra's

- The theorem in the previous slide ensures that Dijkstra's algorithm is correct.
- Since it brings in one vertex into S at each iteration, it will bring in all vertices in $n - 1$ iterations
- At this point, by the theorem we proved for all vertices v, $d(v)$ is the length of the shortest path to v.

Correctness of Dijkstra's

- The theorem in the previous slide ensures that Dijkstra's algorithm is correct.
- Since it brings in one vertex into S at each iteration, it will bring in all vertices in $n - 1$ iterations
- At this point, by the theorem we proved for all vertices v, $d(v)$ is the length of the shortest path to v.
- With a little more bookkeeping, we can compute the actual shortest path

Running time of Dijkstra's

- Run times depends on what kind of structure we use for $d(v)$. If we use an array, finding the argmin takes $O(n)$ each round, leading to $O(n^2)$ time overall. The updates are then constant time and there are $O(m)$ of them which is also $O(n^2)$.

Running time of Dijkstra's

- Run times depends on what kind of structure we use for $d(v)$. If we use an array, finding the argmin takes $O(n)$ each round, leading to $O(n^2)$ time overall. The updates are then constant time and there are $O(m)$ of them which is also $O(n^2)$.
- What if we use heaps instead?

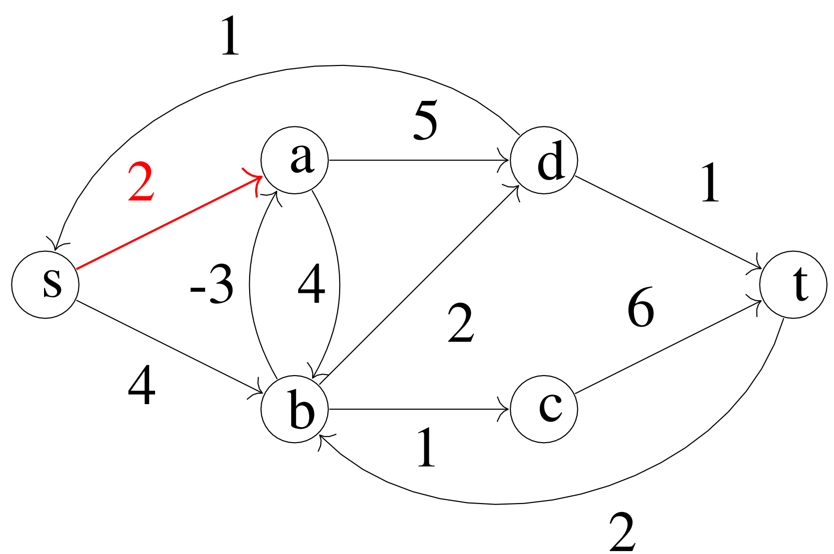
Running time of Dijkstra's

- Run times depends on what kind of structure we use for $d(v)$. If we use an array, finding the argmin takes $O(n)$ each round, leading to $O(n^2)$ time overall. The updates are then constant time and there are $O(m)$ of them which is also $O(n^2)$.
- What if we use heaps instead?
- $O(n \log n)$ to extract-min n times
- Updates are $O(\log n)$ per update which is $O(m \log n)$

Running time of Dijkstra's

- Run times depends on what kind of structure we use for d(v). If we use an array, finding the argmin takes O (n) each round, leading to O (n^2) time overall. The updates are then constant time and there are O(m) of them which is also $O(n^2)$.
- What if we use heaps instead?
- O(n log n) to extract-min n times
- Updates are O(log n) per update which is O(m log n)
- Which running time is better: (n^2) or O(m log n)?. Depends on whether the graph is dense or sparse.

Negative edge weights



The algorithm will start by assigning d(a) = 2 but the path s $\rightsquigarrow$ b $\rightsquigarrow$ a has length 1!

Video 4.3
Sampath Kannan

All Pairs Shortest Path Problem

- Problem: Given a weighted, directed graph $G = (V, E)$ with weight function w, find the shortest paths between every pair of vertices u, v.

All Pairs Shortest Path Problem

- Problem: Given a weighted, directed graph $G = (V, E)$ with weight function w, find the shortest paths between every pair of vertices u, v.
- The first solution we will cover represents G by a matrix A which is like the adjacency matrix except:

$$A[i,j] = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \text{ is an edge} \\ \infty & \text{otherwise} \end{cases}$$

All Pairs Shortest Path Problem

- Problem: Given a weighted, directed graph $G = (V, E)$ with weight function w, find the shortest paths between every pair of vertices u, v.
- The first solution we will cover represents G by a matrix A which is like the adjacency matrix except:

$$A[i,j] = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \text{ is an edge} \\ \infty & \text{otherwise} \end{cases}$$

- Thus $A[i,j]$ represents the weight of the shortest path between any pair of vertices using at most 1 edge.

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?
- If there is a path from i to j that uses 2 edges, it must go through some intermediate vertex k.

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?
- If there is a path from i to j that uses 2 edges, it must go through some intermediate vertex k.
- The weight of the length 2 path $i \rightsquigarrow k \rightsquigarrow j$ is $A[i,k]+A[k,j]$. If we consider every possible k we would have considered every way of going from i to j in (at most) two hops.

$$\begin{array}{cc} & \begin{array}{cccc} k & & & \\ \downarrow & & & \end{array} \\ \begin{array}{c} \\ \\ \\ i \rightarrow \end{array} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix} \end{array} \quad \begin{array}{cc} & \begin{array}{cccc} & & j & \\ & & \downarrow & \end{array} \\ \begin{array}{c} k \rightarrow \\ \\ \\ \\ \end{array} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix} \end{array}$$

Penn Engineering

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?
- If there is a path from i to j that uses 2 edges, it must go through some intermediate vertex k.
- The weight of the length 2 path $i \rightsquigarrow k \rightsquigarrow j$ is $A[i,k] + A[k,j]$. If we consider every possible k we would have considered every way of going from i to j in (at most) two hops.

$$
\begin{array}{cc}
 & \begin{array}{cccc} & k & & \\ & \downarrow & & \end{array} \\
\begin{array}{c} \\ \\ \\ i \rightarrow \end{array} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
\qquad
\begin{array}{cc}
 & \begin{array}{cccc} & & j & \\ & & \downarrow & \end{array} \\
\begin{array}{c} \\ k \rightarrow \\ \\ \\ \end{array} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
$$

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?
- If there is a path from i to j that uses 2 edges, it must go through some intermediate vertex k.
- The weight of the length 2 path $i \rightsquigarrow k \rightsquigarrow j$ is $A[i,k] + A[k,j]$. If we consider every possible k we would have considered every way of going from i to j in (at most) two hops.

$$
\begin{array}{cc}
 & \begin{array}{cccc} & & k & \\ & & \downarrow & \end{array} \\
\begin{array}{c} \\ \\ \\ i \rightarrow \end{array} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
\quad
\begin{array}{cc}
 & \begin{array}{cccc} & & j & \\ & & \downarrow & \end{array} \\
\begin{array}{c} \\ \\ k \rightarrow \\ \\ \end{array} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
$$

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?
- If there is a path from i to j that uses 2 edges, it must go through some intermediate vertex k.
- The weight of the length 2 path $i \rightsquigarrow k \rightsquigarrow j$ is $A[i,k]+A[k,j]$. If we consider every possible k we would have considered every way of going from i to j in (at most) two hops.

$$
\begin{array}{rc}
 & \begin{matrix} & & & k \\ & & & \downarrow \end{matrix} \\
\begin{matrix} \\ \\ \\ i \rightarrow \end{matrix} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
\qquad
\begin{array}{rc}
 & \begin{matrix} & & j & \\ & & \downarrow & \end{matrix} \\
\begin{matrix} \\ \\ \\ k \rightarrow \end{matrix} & \begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
$$

All Pairs Solution 1

- How do we find the weight of the shortest path using at most 2 edges?
- If there is a path from i to j that uses 2 edges, it must go through some intermediate vertex k.
- The weight of the length 2 path $i \rightsquigarrow k \rightsquigarrow j$ is $A[i, k] + A[k, j]$. If we consider every possible k we would have considered every way of going from i to j in (at most) two hops.

$$
\begin{array}{cc}
 & \\
 & \\
 & \\
i \rightarrow &
\end{array}
\begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\qquad
\begin{array}{c}
\begin{array}{cccc} & & j & \\ & & \downarrow & \end{array} \\
\begin{bmatrix} 0 & \infty & 2 & 1 \\ 7 & 0 & 3 & 2 \\ \infty & \infty & 0 & 1 \\ 2 & \infty & 5 & 0 \end{bmatrix}
\end{array}
$$

- This looks like matrix multiplication except. . .

All Pairs Solution 1 cont.

- In normal matrix multiplication we multiply corresponding entries and add the products.
- Here we add corresponding entries, and take the minimum.

All Pairs Solution 1 cont.

- In normal matrix multiplication we multiply corresponding entries and add the products.
- Here we add corresponding entries, and take the minimum.
- Denoting the matrix we get by this kind of multiplication as A^2, we can repeat this to get $A^3, A^4, \ldots$ until we get A^{n-1}.
- $A^{n-1}[i, j]$ is the weight of the shortest path from i to j of length at most $n - 1$. Thus it is the shortest weight simple path from i to j.

Non-simple Paths

- Could a non-simple path be shorter? Why would we want to loop around?

Non-simple Paths

- Could a non-simple path be shorter? Why would we want to loop around?
- Only if there are negative-weight cycles

Non-simple Paths

- Could a non-simple path be shorter? Why would we want to loop around?
- Only if there are negative-weight cycles
- If there are, then in A^n, some diagonal entry becomes negative.

Non-simple Paths

- Could a non-simple path be shorter? Why would we want to loop around?
- Only if there are negative-weight cycles
- If there are, then in A^n, some diagonal entry becomes negative.
- The negative-weight cycle has length at most n, and if i is on the cycle $A^n[i, i]$ will be negative.

Non-simple Paths

- Could a non-simple path be shorter? Why would we want to loop around?
- Only if there are negative-weight cycles
- If there are, then in A^n, some diagonal entry becomes negative.
- The negative-weight cycle has length at most n, and if i is on the cycle $A^n[i,i]$ will be negative.
- If there are negative-weight cycles you can shorten paths that pass through it infinitely. So there is no shortest path between some vertices.

Non-simple Paths

- Could a non-simple path be shorter? Why would we want to loop around?
- Only if there are negative-weight cycles
- If there are, then in A^n, some diagonal entry becomes negative.
- The negative-weight cycle has length at most n, and if i is on the cycle $A^n[i,i]$ will be negative.
- If there are negative-weight cycles you can shorten paths that pass through it infinitely. So there is no shortest path between some vertices.
- Examining the diagonals of A^n will detect the presence of negative-weight cycles and can abort the algorithm if they are found

Faster Multiplication

- Run time to "multiply" two $n \times n$ matrices: $O(n^3)$
- n multiplications gives run time of $O(n^4)$.

Faster Multiplication

- Run time to "multiply" two $n \times n$ matrices: $O(n^3)$
- n multiplications gives run time of $O(n^4)$.
- Instead of "multiplying", repeatedly "square" A until we get to a power that is at least n. That is we compute $A, A^2, A^4, \ldots, A^p$ where $n \leq p < 2n$.

Final Running Time

- $O(\log n)$ "squaring" operations for a total of $O(n^3 \log n)$. Close to the best!
- Note that "squaring" like multiplication is done in a strange way: we add corresponding entries and take the minimum.
- If A does not have negative weight cycles, all the shortest paths are simple and $A^p = A^{n-1}$ for any $p \geq n - 1$.

Video 4.4
Sampath Kannan

Floyd-Warshall Algorithm

- We already saw an $O(n^3 \log n)$ algorithm for computing all pairs shortest paths
- We can use dynamic programming to get an $O(n^3)$ algorithm.

Figure 1: Robert Floyd and Stephen Warshall

Subproblem

- This algorithm will use intermediate vertices like the previous one, but will consider subproblems by restricting which ones can be used.

Subproblem

- This algorithm will use intermediate vertices like the previous one, but will consider subproblems by restricting which ones can be used.
- Number the vertices from 1 to n arbitrarily and let $D^k[i, j]$ be the weight of the shortest path from i to j where all intermediate nodes have number $\leq k$.
- $D^n[i, j]$ has no restriction on intermediate nodes, so this is what we wish to compute.

Base Case

- Base Case: $D^0[i, j]$, the shortest path from i to j using no intermediate nodes.

Base Case

- Base Case: $D^0[i,j]$, the shortest path from i to j using no intermediate nodes.

$$D^0[i,j] = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Base Case

- Base Case: $D^0[i,j]$, the shortest path from i to j using no intermediate nodes.

$$D^0[i,j] = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

- So D^0 are the "smallest" subproblems which we can use to compute D^k for increasing values of k.

Recurrence

- Given $D^k[i, j]$, how do we compute $D^{k+1}[i, j]$?

Recurrence

- Given $D^k[i,j]$, how do we compute $D^{k+1}[i,j]$?
- The only difference is that now we consider paths include vertex $k+1$

Recurrence

- Given $D^k[i, j]$, how do we compute $D^{k+1}[i, j]$?
- The only difference is that now we consider paths include vertex $k+1$
- If a path in D^{k+1} uses $k+1$, it goes through it only once and all other nodes on the path are numbered $\leq k$.

Recurrence cont.

$$D^{k}[i,j] = \min(D^{k}[i,j], D^{k}[i,k+1] + D^{k}[k+1,j])$$

- This is a minimum over the two types of shortest path from i to j that only use vertices numbered $\leq k+1$:

Recurrence cont.

$$D^k[i,j] = \min(D^k[i,j], D^k[i,k+1] + D^k[k+1,j])$$

- This is a minimum over the two types of shortest path from i to j that only use vertices numbered $\leq k+1$:
  1. Paths that only go through nodes numbered $\leq k$

Recurrence cont.

$$D^k[i,j] = \min(D^k[i,j], D^k[i,k+1] + D^k[k+1,j])$$

- This is a minimum over the two types of shortest path from i to j that only use vertices numbered $\leq k+1$:
  1. Paths that only go through nodes numbered $\leq k$
  2. Paths that go through $k+1$ exactly once.

Running Time

- Number of subproblems: Each D^i has n^2 entries to compute, and there are n value of i. So $O(n^3)$ subproblems.

Running Time

- Number of subproblems: Each D^i has n^2 entries to compute, and there are n value of i. So $O(n^3)$ subproblems.
- Computing each sub-problem requires taking the minimum of two previously computed entries so it is $O(1)$.

Running Time

- Number of subproblems: Each D^i has n^2 entries to compute, and there are n value of i. So $O(n^3)$ subproblems.
- Computing each sub-problem requires taking the minimum of two previously computed entries so it is $O(1)$.
- Therefore the total running time is $O(n^3)$.

Video 4.5
Sampath Kannan

Efficiency Definition

- What does it mean for an algorithm to be efficient?

Efficiency Definition

- What does it mean for an algorithm to be efficient?
- We define running time as a function of the input length n
- When an algorithm has running time $O(n^2)$ it means that for long enough inputs, the algorithm takes no more than quadratic time.

Efficiency Definition

- What does it mean for an algorithm to be efficient?
- We define running time as a function of the input length n
- When an algorithm has running time $O(n^2)$ it means that for long enough inputs, the algorithm takes no more than quadratic time.
- In general an algorithm is efficient if its running time is polynomial. More precisely a running time is polynomial when it is $O(n^c)$ for some constant c.
  - Polynomial: $n^2, n^{100}, n \log n$
  - Non Polynomial: $2^n, n!, n^{\log n}$

P

- A decision problem is one where the answer is either a YES or a NO
  - Examples: Is N prime? Do sequences x and y have a common subsequence of length > k? Does the graph G have a path from s to t of length at most k?

P

- A decision problem is one where the answer is either a YES or a NO
  - Examples: Is N prime? Do sequences x and y have a common subsequence of length > k? Does the graph G have a path from s to t of length at most k?
  - For most problems there is an associated decision problem. An efficient algorithm for the decision problem gives rise to an efficient algorithm for the original problem.

P

- A decision problem is one where the answer is either a YES or a NO
  - Examples: Is N prime? Do sequences x and y have a common subsequence of length > k? Does the graph G have a path from s to t of length at most k?
  - For most problems there is an associated decision problem. An efficient algorithm for the decision problem gives rise to an efficient algorithm for the original problem.
- A decision problem with a polynomial time algorithm is said to be in the class P.
- Decision problems associated with minimum spanning trees, shortest paths, and the dynamic programming and greedy examples are in P.

P

- A decision problem is one where the answer is either a YES or a NO
  - Examples: Is N prime? Do sequences x and y have a common subsequence of length > k? Does the graph G have a path from s to t of length at most k?
  - For most problems there is an associated decision problem. An efficient algorithm for the decision problem gives rise to an efficient algorithm for the original problem.
- A decision problem with a polynomial time algorithm is said to be in the class P.
- Decision problems associated with minimum spanning trees, shortest paths, and the dynamic programming and greedy examples are in P.
- What about testing if N is prime?
  - Simple Algorithm: Try dividing N by all numbers between 2 And $\sqrt{N}$. If some i is a factor of N, output 'NOT PRIME'; If no such i exists, output 'PRIME'.

P

- A decision problem is one where the answer is either a YES or a NO
  - Examples: Is N prime? Do sequences x and y have a common subsequence of length > k? Does the graph G have a path from s to t of length at most k?
  - For most problems there is an associated decision problem. An efficient algorithm for the decision problem gives rise to an efficient algorithm for the original problem.
- A decision problem with a polynomial time algorithm is said to be in the class P.
- Decision problems associated with minimum spanning trees, shortest paths, and the dynamic programming and greedy examples are in P.
- What about testing if N is prime?
  - Simple Algorithm: Try dividing N by all numbers between 2 And $\sqrt{N}$. If some i is a factor of N, output 'NOT PRIME'; If no such i exists, output 'PRIME'.
  - But this is not poly-time! We only require $n = \log N$ bits to represent N and $\sqrt{N} = 2^{n/2}$

NP

- Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y?)

NP

- Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y ?)
- There are many decision problems where we can efficiently verify solutions, but can't efficiently find them.
  - Example: Hamiltonian Cycle: In the graph G is there a simple cycle that goes through every vertex?

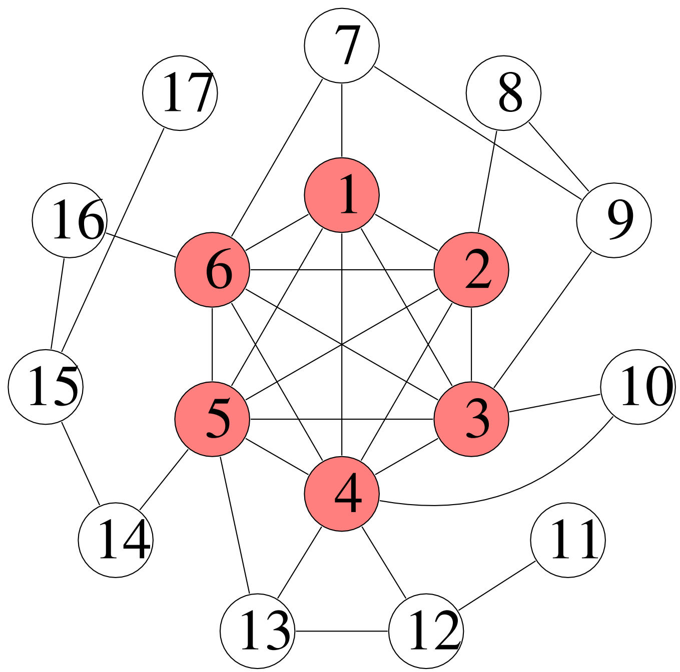
NP

- Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y ?)
- There are many decision problems where we can efficiently verify solutions, but can't efficiently find them.
  - Example: Hamiltonian Cycle: In the graph G is there a simple cycle that goes through every vertex?
- There is no known poly-time algorithm for this but it is easy to verify if a given cycle is a Hamiltonian cycle.

NP

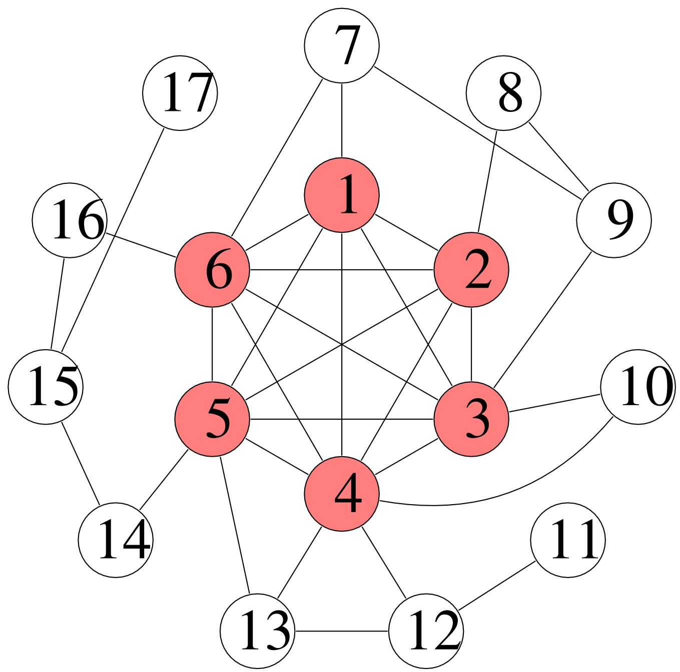
- Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y ?)
- There are many decision problems where we can efficiently verify solutions, but can't efficiently find them.
  - Example: Hamiltonian Cycle: In the graph G is there a simple cycle that goes through every vertex?
- There is no known poly-time algorithm for this but it is easy to verify if a given cycle is a Hamiltonian cycle.
- NP is the class of decision problems where if the answer is YES then there is a short "solution" which can be easily verified.

Maximum Clique



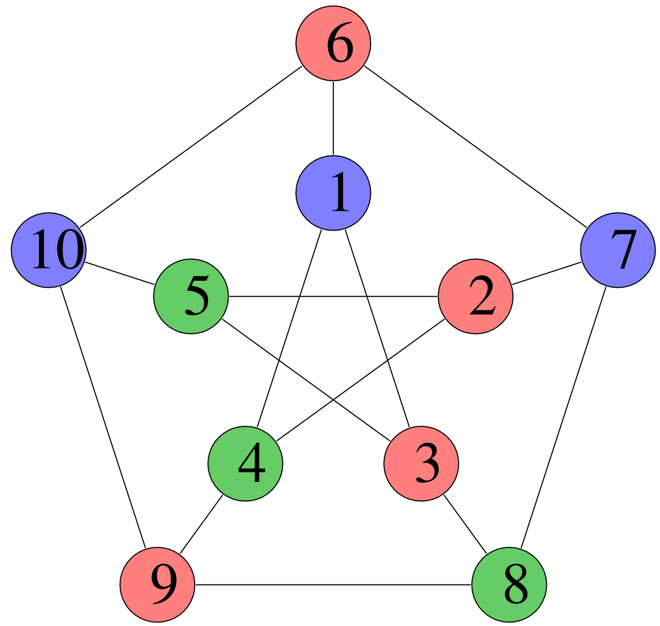
- Given a graph G and a number K, are there K vertices in G that are all pairwise adjacent (this is called a clique)?

Maximum Clique



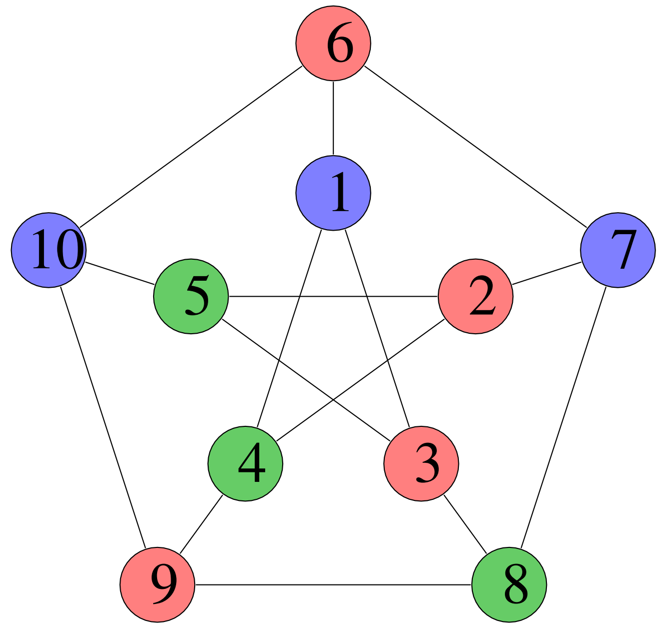
- Given a graph G and a number K, are there K vertices in G that are all pairwise adjacent (this is called a clique)?
- Easy to verify since given the vertices we only need check that they are all adjacent.
- Useful for finding groups of mutual friends in social networks.

3-Coloring



- Given a graph G, can we assign 3 colors to its vertices so that any pair of adjacent vertices have different colors?

3-Coloring



- Given a graph G, can we assign 3 colors to its vertices so that any pair of adjacent vertices have different colors?
- Easy to verify a coloring by examining all edges so it is in NP.
- Useful for allocating transmission frequencies to radio stations to avoid interference.

Partition Problem

1	5	2	4	3	7
1	5	2	4	3	7

1	5	9	4	3	8	10	2	6

?

- Given n numbers, can they be partitioned into 2 sets such that the sums of the numbers in the sets are equal

Partition Problem

1	5	2	4	3	7
1	5	2	4	3	7

1	5	9	4	3	8	10	2	6

?

- Given n numbers, can they be partitioned into 2 sets such that the sums of the numbers in the sets are equal
- Easy to verify given the two sets, so it is in NP.

Video 4.6
Sampath Kannan

P v NP

- Recall that for a decision problem in NP, if the answer is yes for a given input then the "solution" can be verified efficiently

P v NP

- Recall that for a decision problem in NP, if the answer is yes for a given input then the "solution" can be verified efficiently
- But can we compute the solution efficiently?

P v NP

- Recall that for a decision problem in NP, if the answer is yes for a given input then the "solution" can be verified efficiently
- But can we compute the solution efficiently?
- We don't know! This is the $P \stackrel{?}{=} NP$ question.

Hard Problems

- One approach to settling $P \stackrel{?}{=} NP$: Identify the "hardest" problems in NP and focus on solving them in poly-time

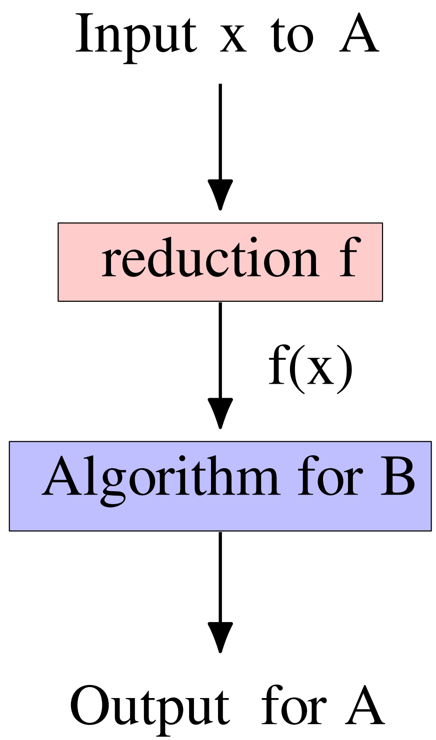
Hard Problems

- One approach to settling $P \stackrel{?}{=} NP$: Identify the "hardest" problems in NP and focus on solving them in poly-time
- But how do we know which problems are hard?

Hard Problems

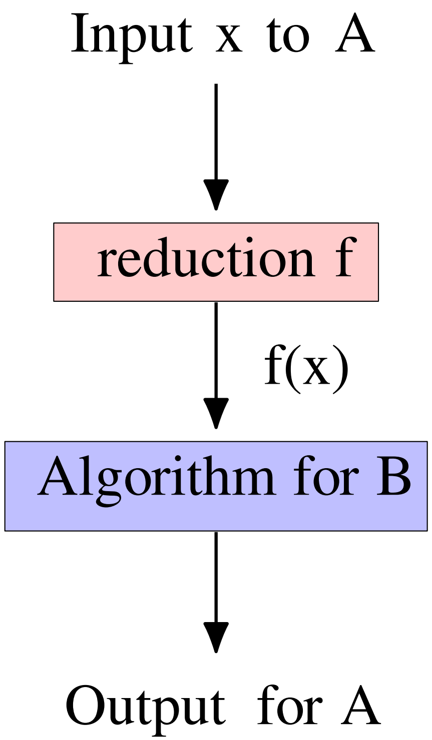
- One approach to settling P $\stackrel{?}{=}$ NP: Identify the “hardest” problems in NP and focus on solving them in poly-time
- But how do we know which problems are hard?
- Idea: Reductions

Reductions



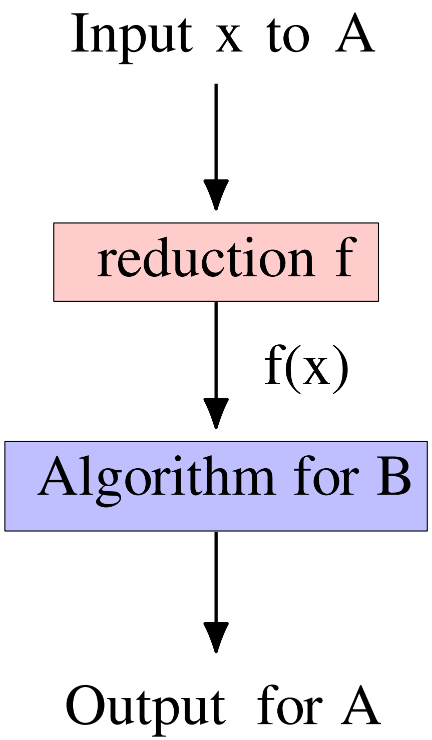
- We can reduce problem A to problem B if we can use a solution to B to solve A.

Reductions



- We can reduce problem A to problem B if we can use a solution to B to solve A.
- Example: Problem A is finding the median of n elements and Problem B is sorting n elements.

Reductions



- We can reduce problem A to problem B if we can use a solution to B to solve A.
- Example: Problem A is finding the median of n elements and Problem B is sorting n elements.

  A reduces to B since we can take the input to A, sort it using the solution to B, and then recover the solution to A by looking at the middle element in sorted order.

Reductions cont.

- The median and sorting example illustrates reductions, but it is a bad example since medians can be computed directly faster than sorting.

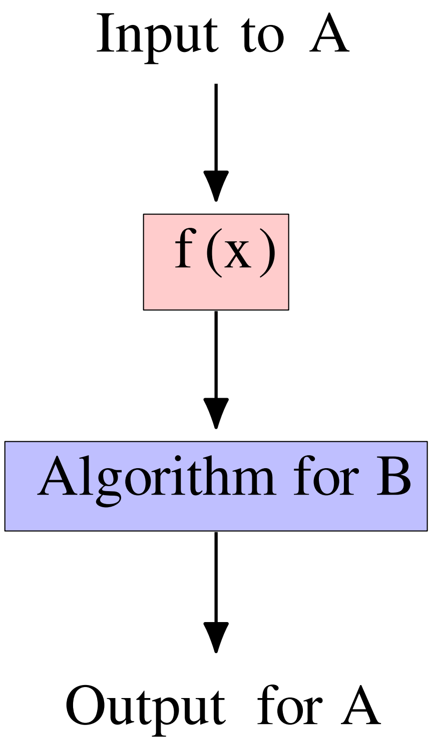
Reductions cont.

- The median and sorting example illustrates reductions, but it is a bad example since medians can be computed directly faster than sorting.
- However if we go the other way we can use the median finding algorithm to make an efficient sorting algorithm.

Reductions cont.

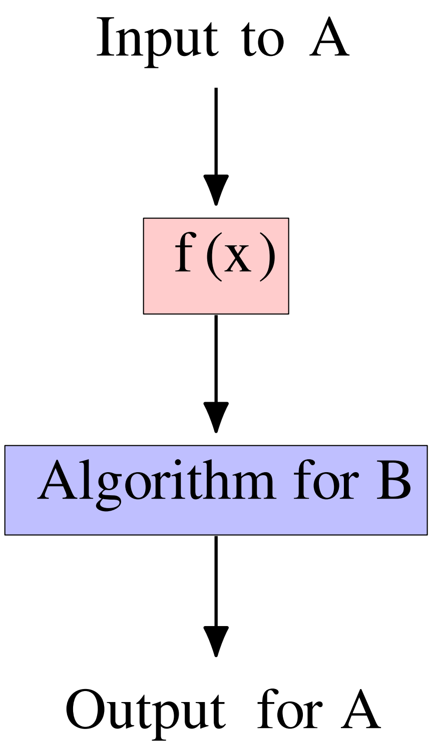
- The median and sorting example illustrates reductions, but it is a bad example since medians can be computed directly faster than sorting.
- However if we go the other way we can use the median finding algorithm to make an efficient sorting algorithm.
  - Sort n elements, compute the median using the black box for A
  - Use the median as a pivot like in quicksort and recurse. We will always have a perfect partition so the algorithm will be efficient.

Reduction Definition



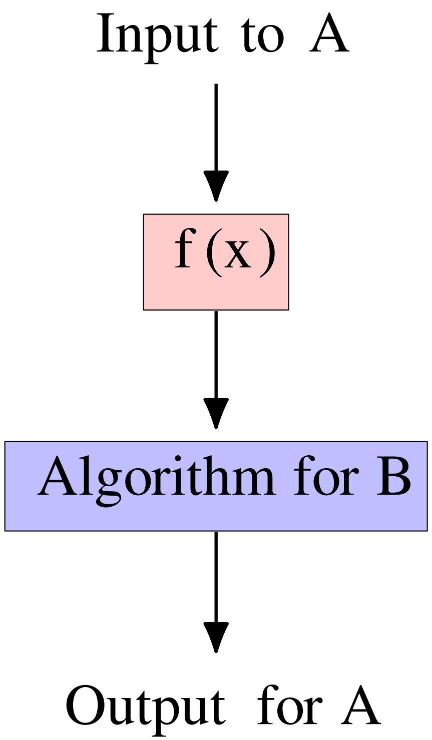
- We need a more rigorous definition of a reduction to identify the hardest problems in NP.

Reduction Definition



- We need a more rigorous definition of a reduction to identify the hardest problems in NP.
- We say Decision Problem A reduces to Decision problem B if there is a function f mapping inputs of A to inputs of B such that:
  - If x is a YES input for A, then f(x) is a YES input for B
  - IF x is a NO input for A, then f(x) is a NO input for B

Reduction Definition



- We need a more rigorous definition of a reduction to identify the hardest problems in NP.
- We say Decision Problem A reduces to Decision problem B if there is a function f mapping inputs of A to inputs of B such that:
  - If x is a YES input for A, then f(x) is a YES input for B
  - IF x is a NO input for A, then f(x) is a NO input for B
- The reduction is f itself.

Poly-Time Reductions

- If $f(x)$ can be computed in polynomial time, we say it is a polynomial-time reduction.
- Note: If f is a polynomial-time reduction, then $|f(x)|$ is polynomial in the length of x.

Poly-Time Reductions

- If $f(x)$ can be computed in polynomial time, we say it is a polynomial-time reduction.
- Note: If f is a polynomial-time reduction, then $|f(x)|$ is polynomial in the length of x.
- Median finding and sorting are not decision problems, but otherwise the median finding to sorting reduction fits this definition.
  - What is $f(x)$? Is it in computable poly-time?
- However, the other direction does not fit since we repeatedly use the median finding algorithm

Poly-Time Reduction Implications

- What does it meant if there is a poly-time reduction from A to B?

Poly-Time Reduction Implications

- What does it mean if there is a poly-time reduction from A to B?
- If B is solvable in poly time, then so is A. Just map the input to A to an input to B, and use the method for solving B

(recall that if $f(x)$ and $g(x)$ are polynomials then $f(g(x))$ is also a polynomial).

Poly-Time Reduction Implications

- What does it mean if there is a poly-time reduction from A to B?
- If B is solvable in poly time, then so is A. Just map the input to A to an input to B, and use the method for solving B

(recall that if $f(x)$ and $g(x)$ are polynomials then $f(g(x))$ is also a polynomial).

- If A is not solvable in polynomial time, then neither is B. This statement is actually equivalent to the original one.

Example

- Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.

Example

- Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.
- Using our language, the problem of traveling back to the past reduces to the problem of traveling faster than the speed of light

Example

- Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.
- Using our language, the problem of traveling back to the past reduces to the problem of traveling faster than the speed of light
- If we manage to build a faster-than-light vehicle, then we can go back to the past
- But if we prove that is impossible to travel back in time, then we immediately know it is impossible to build a faster-than-light vehicle.

Video 4.7
Sampath Kannan

NP – Completeness Definition

- Suppose we have a decision problem A in NP such that for every problem in NP there is a polynomial time reduction to A.

NP − Completeness Definition

- Suppose we have a decision problem A in NP such that for every problem in NP there is a polynomial time reduction to A.
- If we knew A is solvable in polynomial time, then every problem in NP can be solved in polynomial time.

Equivalently, $A \in P \implies P = NP$.

NP – Completeness Definition

- Suppose we have a decision problem A in NP such that for every problem in NP there is a polynomial time reduction to A.
- If we knew A is solvable in polynomial time, then every problem in NP can be solved in polynomial time. Equivalently, $A \in P \implies P = NP$.
- In a sense, A is a "hardest" problem in NP.
- We say a problem A is NP-complete if
  - $A \in NP$
  - Every problem in NP reduces to A

Satisfiability

- How could we every show that every problem in NP reduces to a problem A?

Satisfiability

- How could we every show that every problem in NP reduces to a problem A?
- Cook and Levin did exactly this for the Satisfiability problem.
- Satisfiability: Given a boolean formula, is there a way to set the variables such that the formula evaluates to true?

Satisfiability

- How could we every show that every problem in NP reduces to a problem A?
- Cook and Levin did exactly this for the Satisfiability problem.
- Satisfiability: Given a boolean formula, is there a way to set the variables such that the formula evaluates to true?

For example, $((\neg a \vee \neg b \vee c) \wedge (a \vee c) \wedge (\neg c \vee b)) \vee (\neg a \wedge b \wedge c)$

is satisfied by $a \longmapsto 1, b \longmapsto 0, c \longmapsto 0$.

Satisfiability

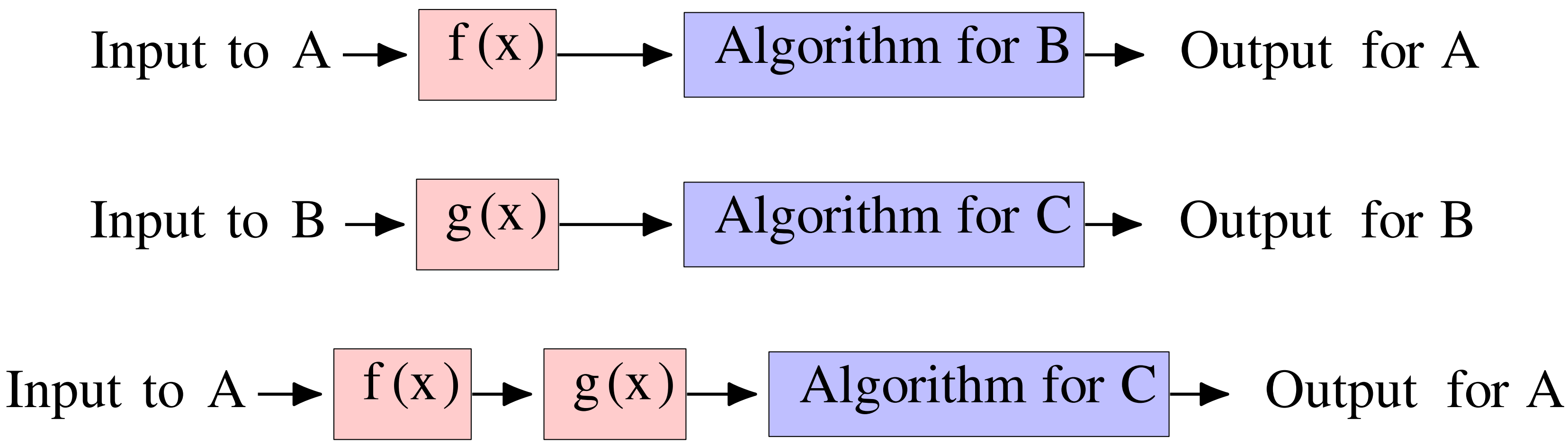
- How could we every show that every problem in NP reduces to a problem A?
- Cook and Levin did exactly this for the Satisfiability problem.
- Satisfiability: Given a boolean formula, is there a way to set the variables such that the formula evaluates to true?

For example, $((\neg a \vee \neg b \vee c) \wedge (a \vee c) \wedge (\neg c \vee b)) \vee (\neg a \wedge b \wedge c)$

- is satisfied by $a \longmapsto 1, b \longmapsto 0, c \longmapsto 0$.

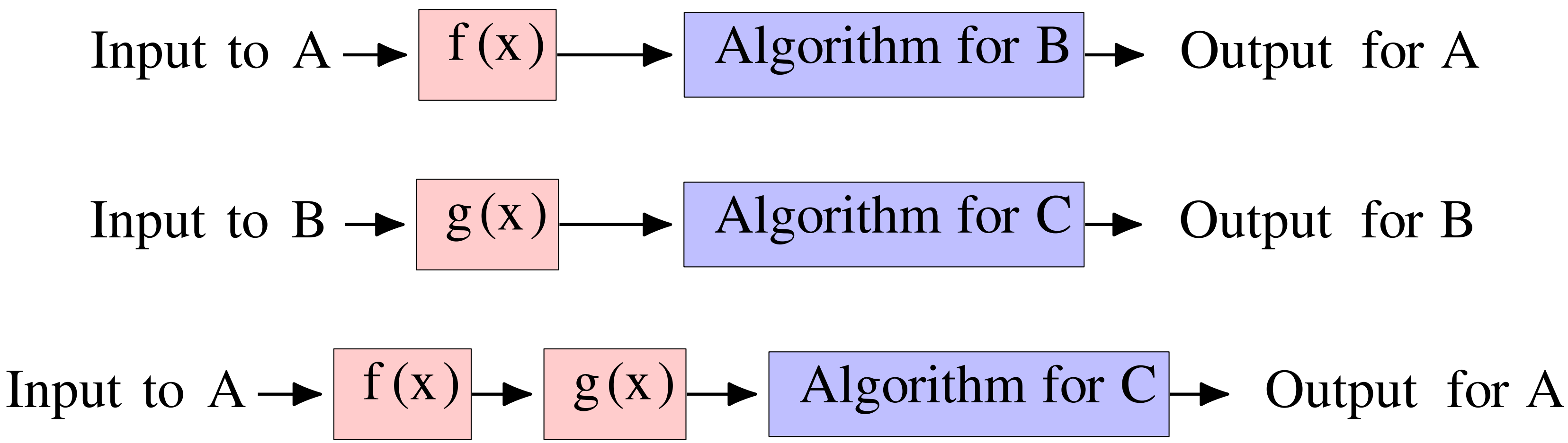
Cook-Levin uses the fact that every problem in NP has a poly-time verifier to construct a formula that is satisfiable if and only if there is a "solution" that the verifier will accept.

Properties of Reductions



- Polynomial-time reductions are transitive: If A reduces to B and B reduces to C, then A reduces to C

Properties of Reductions



- Polynomial-time reductions are transitive: If A reduces to B and B reduces to C, then A reduces to C
- After Cook-Levin, to show a problem X is NP-complete we need only show that $X \in NP$ and that Satisfiability or another NP-complete problem reduces to X.

Example Reduction

- The Non-Unique Satisfiability problem is to determine whether or not a given boolean formula φ has at least 2 unique assignments that satisfy it.

Example Reduction

- The Non-Unique Satisfiability problem is to determine whether or not a given boolean formula φ has at least 2 unique assignments that satisfy it.
- NUS is in NP since a "solution" could present two assignments and a verifier can simply plug them in and evaluate the formula.

Example Reduction

- The Non-Unique Satisfiability problem is to determine whether or not a given boolean formula φ has at least 2 unique assignments that satisfy it.
- NUS is in NP since a "solution" could present two assignments and a verifier can simply plug them in and evaluate the formula.
- Now given an input to Satisfiability φ, we let the input to NUS be $\psi = \varphi \wedge (x \vee \neg x)$.

Example Reduction

- The Non-Unique Satisfiability problem is to determine whether or not a given boolean formula φ has at least 2 unique assignments that satisfy it.
- NUS is in NP since a "solution" could present two assignments and a verifier can simply plug them in and evaluate the formula.
- Now given an input to Satisfiability φ, we let the input to NUS be $\psi = \varphi \wedge (x \vee \neg x)$.
- $\Longrightarrow$: Suppose φ were satisfiable. Then given the satisfying assignment B for φ we know the assignments $B, x \longmapsto 0$ and $B, x \longmapsto 1$ satisfy ψ

Example Reduction

- The Non-Unique Satisfiability problem is to determine whether or not a given boolean formula φ has at least 2 unique assignments that satisfy it.
- NUS is in NP since a "solution" could present two assignments and a verifier can simply plug them in and evaluate the formula.
- Now given an input to Satisfiability φ, we let the input to NUS be $\psi = \varphi \wedge (x \vee \neg x)$.
- $\Longrightarrow$: Suppose φ were satisfiable. Then given the satisfying assignment B for φ we know the assignments B , $x \longmapsto 0$ and B, $x \longmapsto 1$ satisfy ψ
- $\Longleftarrow$: Suppose $\psi = \varphi \wedge (x \vee \neg x)$ is non-unique satisfiable. Then it has a satisfying assignment, and by the definition of $\wedge$ this means φ has one as well.

Example Reduction

- The Non-Unique Satisfiability problem is to determine whether or not a given boolean formula φ has at least 2 unique assignments that satisfy it.
- NUS is in NP since a "solution" could present two assignments and a verifier can simply plug them in and evaluate the formula.
- Now given an input to Satisfiability φ, we let the input to NUS be $\psi = \varphi \wedge (x \vee \neg x)$.
- $\Longrightarrow$: Suppose φ were satisfiable. Then given the satisfying assignment B for φ we know the assignments B , $x \longmapsto 0$ and $B, x \longmapsto 1$ satisfy ψ
- $\Longleftarrow$: Suppose $\psi = \varphi \wedge (x \vee \neg x)$ is non-unique satisfiable. Then it has a satisfying assignment, and by the definition of $\wedge$ this means φ has one as well.
- Therefore the function $f(\varphi) = \varphi \wedge (x \vee \neg x)$ is a polynomial time reduction from Satisfiability and NUS is NP-Complete.

Video 4.8
Sampath Kannan

3-SAT

- 3-SAT is a variant of the Satisfiability problem that asks if a given boolean formula in 3-Conjunctive Normal Form is satisfiable.

3-SAT

- 3-SAT is a variant of the Satisfiability problem that asks if a given boolean formula in 3-Conjunctive Normal Form is satisfiable.
- A formula is in Conjunctive Normal Form (CNF) when it is the AND of m clauses where each clause is an OR of some number of variables. For example

  $(\neg x \vee y) \wedge z \wedge (\neg z \vee y \vee a \vee b)$ is in CNF.

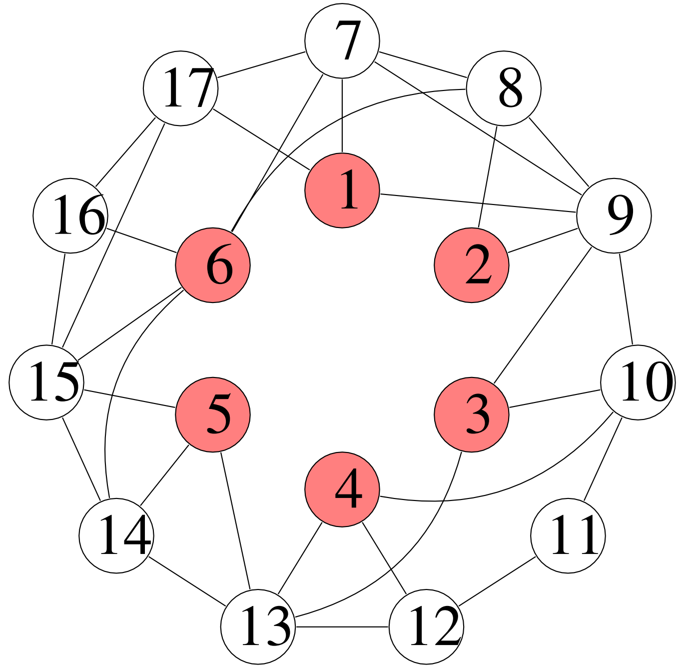
3-SAT

- 3-SAT is a variant of the Satisfiability problem that asks if a given boolean formula in 3-Conjunctive Normal Form is satisfiable.
- A formula is in Conjunctive Normal Form (CNF) when it is the AND of m clauses where each clause is an OR of some number of variables. For example $(\neg x \vee y) \wedge z \wedge (\neg z \vee y \vee a \vee b)$ is in CNF.
- A formula is is in 3-CNF when each clause has at most 3 literals. For example $(\neg x \vee y) \wedge (\neg z \vee y \vee a)$ is in 3-CNF.

3-SAT

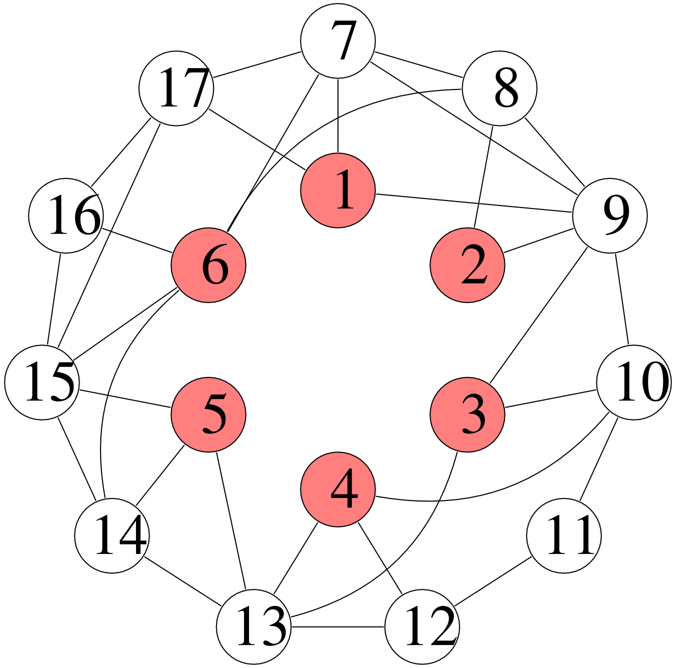
- 3-SAT is a variant of the Satisfiability problem that asks if a given boolean formula in 3-Conjunctive Normal Form is satisfiable.
- A formula is in Conjunctive Normal Form (CNF) when it is the AND of m clauses where each clause is an OR of some number of variables. For example $(\neg x \vee y) \wedge z \wedge (\neg z \vee y \vee a \vee b)$ is in CNF.
- A formula is is in 3-CNF when each clause has at most 3 literals. For example $(\neg x \vee y) \wedge (\neg z \vee y \vee a)$ is in 3-CNF.
- We will omit the details but Satisfiability can be reduced to 3-SAT, implying that 3-SAT is NP-Complete.

Independent Set



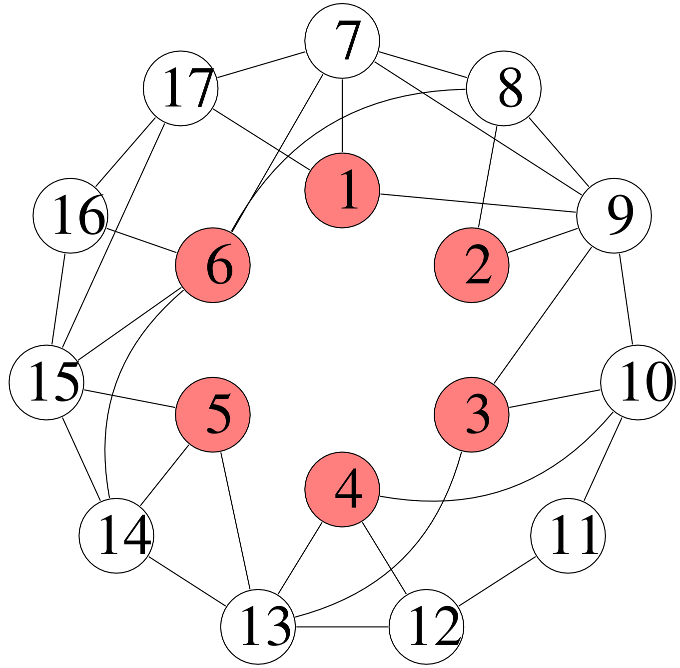
- An *independent set* in a graph is a set of vertices, no two of which are adjacent.

Independent Set



- An independent set in a graph is a set of vertices, no two of which are adjacent.
- Independent Set Decision Problem (ISDP): Given a graph G and integer K, does G have an independent set of size K?

Independent Set

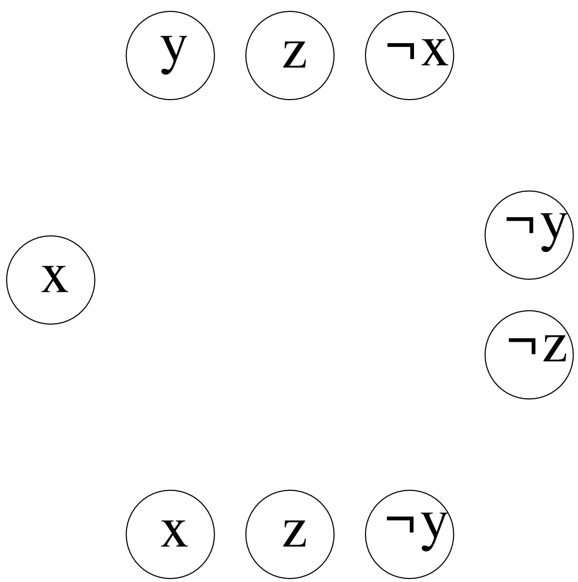


- An independent set in a graph is a set of vertices, no two of which are adjacent.
- Independent Set Decision Problem (ISDP): Given a graph G and integer K, does G have an independent set of size K?
- Clearly ISDP is in NP since we can easily verify if a given set is independent. We will now reduce 3-SAT to ISDP to show that ISDP is NP-Complete.

Reduction

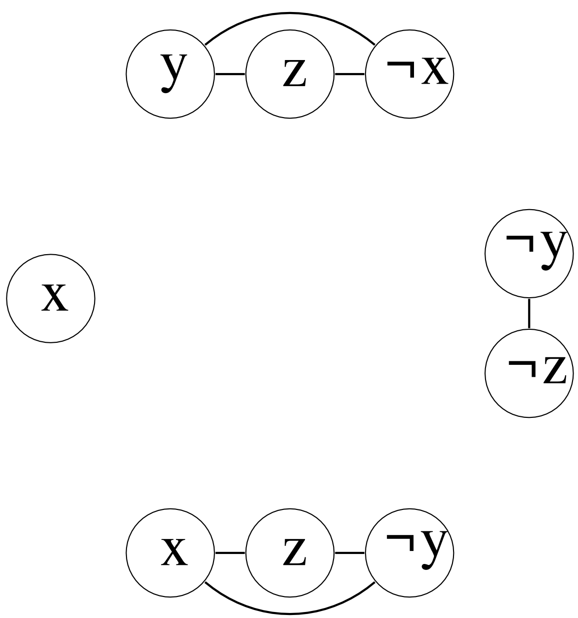
$$x \wedge (y \wedge z \wedge \neg x) \wedge (\neg y \wedge \neg z) \wedge (x \vee z \vee \neg y)$$

- For a 3-CNF formula φ with n variables and m clauses, create a vertex for each occurrence of each literal.



Reduction

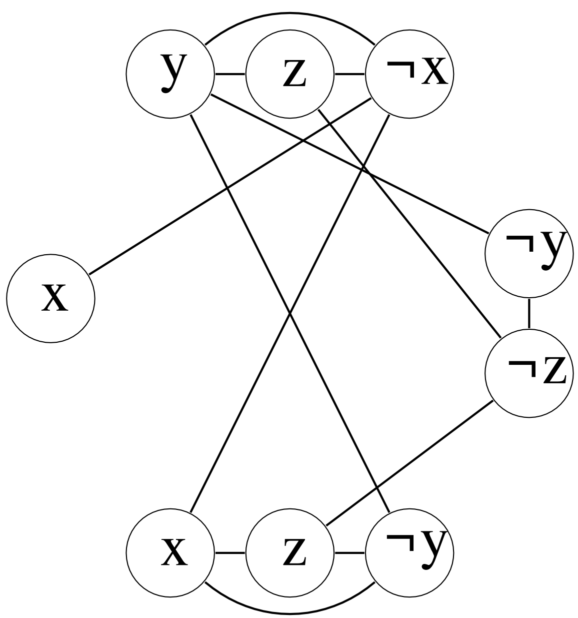
$$x \wedge (y \wedge z \wedge \neg x) \wedge (\neg y \wedge \neg z) \wedge (x \vee z \vee \neg y)$$



- For a 3-CNF formula φ with n variables and m clauses, create a vertex for each occurrence of each literal.
- Join every pair of literals from the same clause.

Reduction

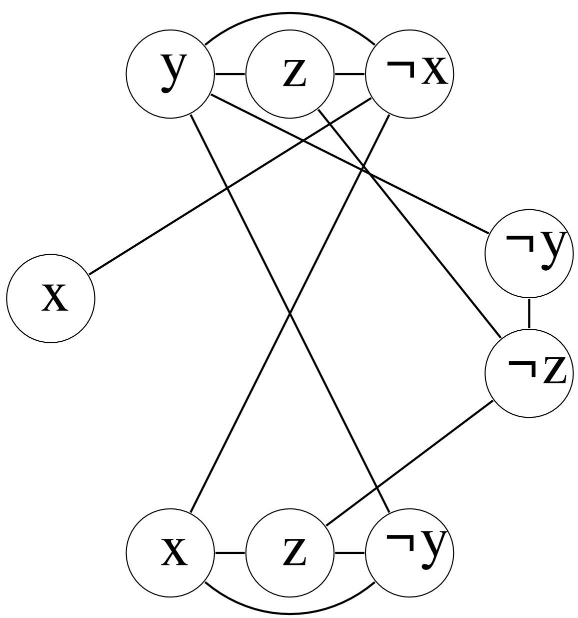
$$x \wedge (y \wedge z \wedge \neg x) \wedge (\neg y \wedge \neg z) \wedge (x \vee z \vee \neg y)$$



- For a 3-CNF formula φ with n variables and m clauses, create a vertex for each occurrence of each literal.
- Join every pair of literals from the same clause.
- Between any node representing x and any node representing ¬x draw an edge.

Reduction

$x \wedge (y \wedge z \wedge \neg x) \wedge (\neg y \wedge \neg z) \wedge (x \vee z \vee \neg y)$



- For a 3-CNF formula φ with n variables and m clauses, create a vertex for each occurrence of each literal.
- Join every pair of literals from the same clause.
- Between any node representing x and any node representing ¬x draw an edge.
- The input to ISDP will be this graph and K = m.

Proof

- $\Longrightarrow$: If φ is satisfiable then using the satisfying assignment we can pick one literal from each clause that evaluates to true and we notice that no two of these literals will be negations of each other. The vertices corresponding to these literals will be an independent set of size m.

Proof

- $\Longrightarrow$: If φ is satisfiable then using the satisfying assignment we can pick one literal from each clause that evaluates to true and we notice that no two of these literals will be negations of each other. The vertices corresponding to these literals will be an independent set of size m.
- $\Longleftarrow$: If G has an independent set S of size m, then one vertex from each of the m clauses is in S, since all vertices in the same clause are adjacent. We also have that no two of the vertices in S are negations of each other since an edge connects all such pairs. Therefore we can make an assignment such that the literal corresponding to each vertex in S evaluates to true, which is a satisfying assignment for φ. Therefore φ is satisfiable.

Conclusion

- Currently we do not know if $P = NP$. But we have tens of thousands of NP-complete problems. If any of them is solved in polynomial time, then $P = NP$ and every problem in NP can be solved efficiently.

Conclusion

- Currently we do not know if P = NP. But we have tens of thousands of NP-complete problems. If any of them is solved in polynomial time, then P = NP and every problem in NP can be solved efficiently.
- Practical implication: If a mathematical theorem has a short proof, then an efficient algorithm can find it!

Conclusion

- Currently we do not know if P = NP. But we have tens of thousands of NP-complete problems. If any of them is solved in polynomial time, then P = NP and every problem in NP can be solved efficiently.
- Practical implication: If a mathematical theorem has a short proof, then an efficient algorithm can find it!
- Less formal implication: Appreciating beautiful music is easier than creating beautiful music (it seems). But appreciation is like an algorithm that verifies if a piece of music is beautiful, so creating beautiful music is like an NP-problem, that has efficient verification.

Conclusion

- Currently we do not know if $P = NP$. But we have tens of thousands of NP-complete problems. If any of them is solved in polynomial time, then $P = NP$ and every problem in NP can be solved efficiently.
- Practical implication: If a mathematical theorem has a short proof, then an efficient algorithm can find it!
- Less formal implication: Appreciating beautiful music is easier than creating beautiful music (it seems). But appreciation is like an algorithm that verifies if a piece of music is beautiful, so creating beautiful music is like an NP-problem, that has efficient verification.
- If $P = NP$ then music creation is no more difficult than music appreciation! Likewise for art creation, so $P = NP$ implies that creativity can be automated!

Technical Conclusion

- We learned about data structures, algorithm design techniques, and some important graph algorithms. We also learned that some interesting problems may not have efficient solutions at all.

Technical Conclusion

- We learned about data structures, algorithm design techniques, and some important graph algorithms. We also learned that some interesting problems may not have efficient solutions at all.
- When confronted with a problem:
  - Try to see if some of the techniques you learned lead to an efficient solution

Technical Conclusion

- We learned about data structures, algorithm design techniques, and some important graph algorithms. We also learned that some interesting problems may not have efficient solutions at all.
- When confronted with a problem:
  - Try to see if some of the techniques you learned lead to an efficient solution
  - If they don't, perhaps the problem is too hard. Try to show it is NP-complete to avoid wasting your time looking for an efficient algorithm that probably doesn't exist.

Further Topics

There are many issues in Algorithm Design and Analysis not covered in this short course.

1. Analysis of resources other than time (memory space is an important one!).

Further Topics

There are many issues in Algorithm Design and Analysis not covered in this short course.

1. Analysis of resources other than time (memory space is an important one!).
2. Dealing with NP-complete and harder problems by finding heuristics and showing they work well, or by finding algorithms that provably find an approximately optimal solution.

Further Topics

There are many issues in Algorithm Design and Analysis not covered in this short course.

1. Analysis of resources other than time (memory space is an important one!).
2. Dealing with NP-complete and harder problems by finding heuristics and showing they work well, or by finding algorithms that provably find an approximately optimal solution.
3. Many other algorithm design techniques and heuristic approaches.

Further Topics

There are many issues in Algorithm Design and Analysis not covered in this short course.

1. Analysis of resources other than time (memory space is an important one!).
2. Dealing with NP-complete and harder problems by finding heuristics and showing they work well, or by finding algorithms that provably find an approximately optimal solution.
3. Many other algorithm design techniques and heuristic approaches.
4. Different models of computing and algorithms for those models.