Recovering Shortest Paths

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i, j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i, k] + f[k, j] < f[i, j]$ **then**
                $f[i, j] \leftarrow f[i, k] + f[k, j]$, $\pi[i, j] \leftarrow k$

Recovering Shortest Paths

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$

print-path(i, j)

if $\pi[i,j] = \perp$ **then**
    print(i, j)
else
    print-path$(i, \pi[i,j])$, print-path$(\pi[i,j], j)$

Detecting Negative Cycles

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$

Detecting Negative Cycles

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \bot$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                report "negative cycle exists" and exit

Summary of Shortest Path Algorithms

algorithm	graph	weights	SS?	running time
Simple DP	DAG	$\mathbb{R}$	SS	$O(n+m)$
Dijkstra	U/D	$\mathbb{R}_{\geq 0}$	SS	$O(n \log n + m)$
Bellman-Ford	U/D	$\mathbb{R}$	SS	$O(nm)$
Floyd-Warshall	U/D	$\mathbb{R}$	AP	$O(n^3)$

- DAG = directed acyclic graph U = undirected D = directed
- SS = single source AP = all pairs

CSE 431/531: Algorithm Analysis and Design (Fall 2024)
NP-Completeness

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

Q: Why do we study negative results?

NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

Q: Why do we study negative results?

- A given problem X cannot be solved in polynomial time.
- Without knowing it, you will have to keep trying to find polynomial time algorithm for solving X. All our efforts are doomed!

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then k is small, say 4
- A good cut separating problems: for most natural problems, either we have a polynomial time algorithm, or the best algorithm runs in time $\Omega(2^{n^c})$ for some c
- Do not need to worry about the computational model

Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
7. Summary of Studies 2024 Spring

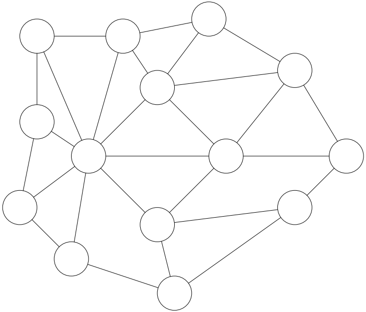
Example: Hamiltonian Cycle Problem

Def. Let G be an undirected graph. A Hamiltonian Cycle (HC) of G is a cycle C in G that passes each vertex of G exactly once.

Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle



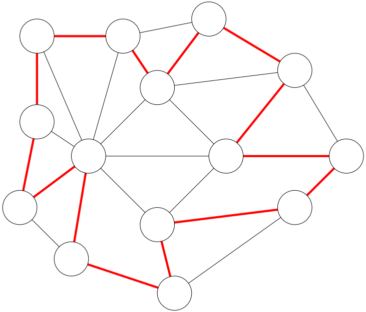
Example: Hamiltonian Cycle Problem

Def. Let G be an undirected graph. A Hamiltonian Cycle (HC) of G is a cycle C in G that passes each vertex of G exactly once.

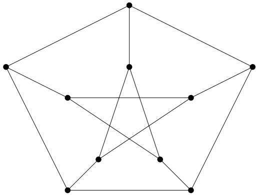
Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle



Example: Hamiltonian Cycle Problem



- The graph is called the Petersen Graph. It has no HC.

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

Input: graph $G = (V, E)$

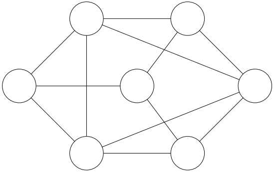
Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time
- HC is NP-hard: it is unlikely that it can be solved in polynomial time.

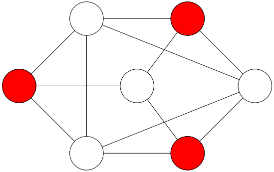
Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



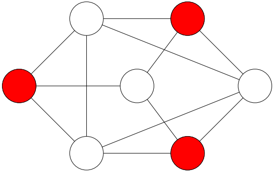
Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Maximum Independent Set Problem

Input: graph $G = (V, E)$

Output: the size of the maximum independent set of G