CS 401

Computational Complexity

Xiaorui Sun

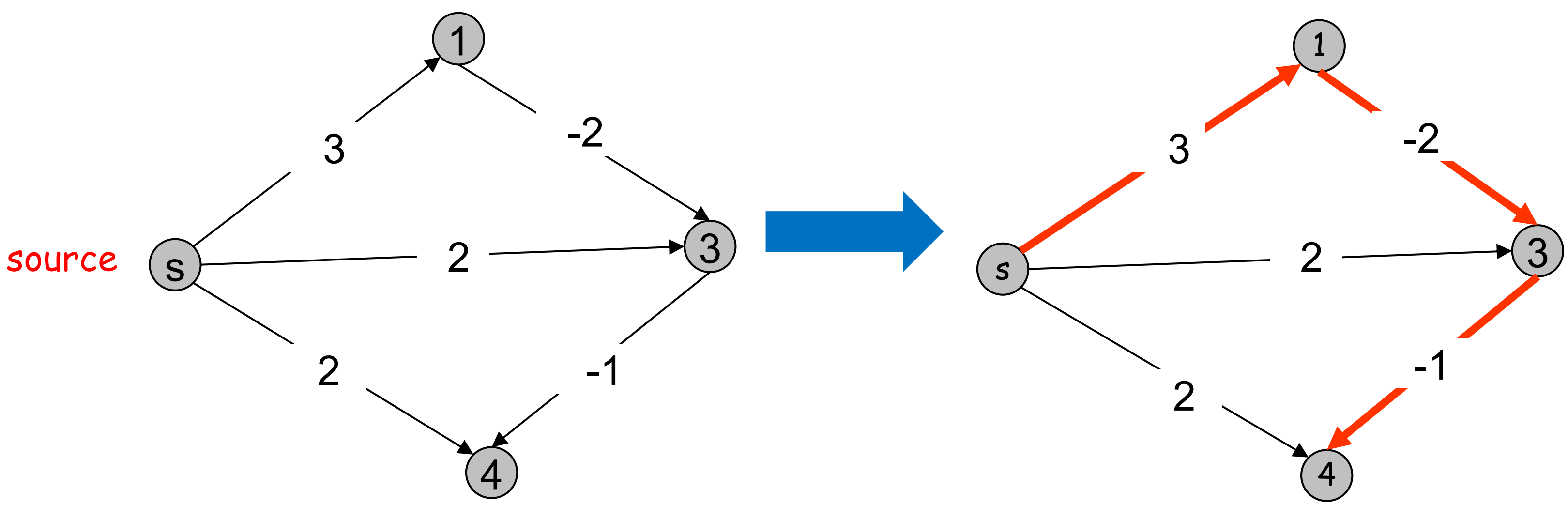
Shortest Paths with Negative Edge Weights

Shortest Paths with Neg Edge Weights

Given a weighted directed graph $G = (V, E)$ and a source vertex s, where the weight of edge (u,v) is $c_{u,v}$ **(that can be negative)**

Goal: Find the shortest path from s to all vertices of G.

Recall that Dikjstra's Algorithm fails when weights are negative

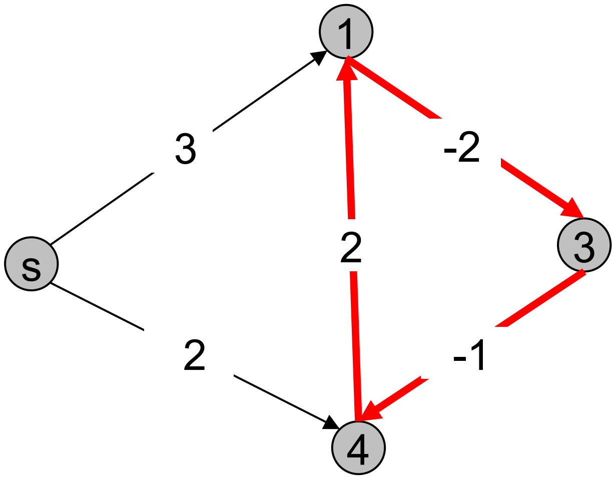


Impossibility on Graphs with Neg Cycles

Observation: No solution exists if G has a negative cycle.

This is because we can minimize the length by going over the cycle again and again.

So, suppose G does not have a negative cycle.



DP for Shortest Path (First Attempt)

Def: Let $OPT(v)$ be the length of the shortest s - v path

$$OPT(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u:(u,v) \text{ an edge}} OPT(u) + c_{u,v} & \end{cases}$$

The formula is correct. But it is not clear how to compute it.

DP for Shortest Path

Def: Let $OPT(v, i)$ be the length of the shortest s - v path with at most i edges.

Let us characterize $OPT(v, i)$.

Case 1: $OPT(v, i)$ path has less than i edges.

- Then, $OPT(v, i) = OPT(v, i - 1)$.

Case 2: $OPT(v, i)$ path has exactly i edges.

- Let $s, v_1, v_2, \dots, v_{i-1}, v$ be the $OPT(v, i)$ path with i edges.
- Then, $s, v_1, \dots, v_{i-1}$ must be the shortest s - v_{i-1} path with at most $i - 1$ edges. So,

$$OPT(v, i) = OPT(v_{i-1}, i - 1) + c_{v_{i-1}, v}$$

DP for Shortest Path

Def: Let $OPT(v, i)$ be the length of the shortest s - v path with at most i edges.

$$OPT(v,i) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s, i = 0 \\ \min\left(OPT(v, i-1), \min\limits_{u:(u,v) \text{ an edge}} OPT(u, i-1) + c_{u,v}\right) & \end{cases}$$

So, for every v, $OPT(v, ?)$ is the shortest path from s to v.

But how long do we have to run?

Since G has no negative cycle, it has at most $n - 1$ edges. So, $OPT(v, n-1)$ is the answer.

Bellman Ford Algorithm

```
for v=1 to n
    if v ≠ s then
        M[v,0]=∞
M[s,0]=0.

for i=1 to n-1
    for v=1 to n
        M[v,i]=M[v,i-1
        for every edge
            M[v,i]=min(
```

	Complexity	Author
	$O(n^4)$	Shimbel (1955) [30]
	$O(Wn^2m)$	Ford (1956) [14]
*	$O(nm)$	Bellman (1958) [1], Moore (1959) [25]
	$O(n^{\frac{3}{4}} m \log W)$	Gabow (1983) [9]
	$O(\sqrt{n}m \log(nW))$	Gabow and Tarjan (1989) [10]
*	$O(\sqrt{n}m \log(W))$	Goldberg (1993) [12]
*	$\tilde{O}(Wn^{\omega})$	Sankowski (2005) [27] Yuster and Zwick (2005) [35]
*	$\tilde{O}(m^{10/7} \log W)$	Cohen, Madry, Sankowski, Vladu (2016)

Table 1: The complexity results for the SSSP problem with negative weights (* indicates asymptotically the best bound for some range of parameters).

$m^{1+o(1)} \log W$ algorithm
By Bernstein, Nanongkai, and Wulff-Nilsen;
Chen, Kyng, Liu, Peng, Gutenberg, Sachdeva
2022

Running Time: $O(nm)$
Can we test if G has negative cycles?
Yes, run for i=1…3n and see if the M[v,n-1] is different from M[v,3n]

DP Techniques

Principle:

- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem
- Carefully define subproblems. Typically, only a polynomial number of subproblems
- Parameterization/Memorization

Recipe:

- Find optimal substructure by investigating the optimal solution
- Find out additional variables/subproblems that you need to do the induction
- Strengthen the hypothesis and define new subproblems

Dynamic programming techniques.

- Adding a new variable: knapsack
- Order subproblems in the right way: RNA secondary structure

Computational Complexity

Algorithm Design Patterns and Anti-Patterns

Algorithm design patterns. Ex.

- Greed. $O(n \log n)$ interval scheduling.
- Divide-and-conquer. $O(n^2)$ edit distance.
- Dynamic programming.
- Reductions.
- Local search.
- Randomization.

Algorithm design anti-patterns.

- NP-completeness. $O(n^k)$ algorithm unlikely.
- PSPACE-completeness. $O(n^k)$ certification algorithm unlikely.
- Undecidability. No algorithm possible.

Computational Complexity

Goal: Classify problems according to the amount of computational resources used by the best algorithms that solve them

Here we focus on time complexity

Recall: worst-case running time of an algorithm

- **max** # steps algorithm takes on any input of size **n**

Relative Complexity of Problems

- Want a notion that allows us to compare the complexity of problems
- Want to be able to make statements of the form

  "If we could solve problem **B** in polynomial time then we can solve problem **A** in polynomial time"

  "Problem **B** is at least as hard as problem **A**"

Polynomial Time Reduction

Def $A \leq_P B$: if there is an algorithm for problem A using a 'black box' (subroutine) that solve problem B s.t.,

- Algorithm uses only a polynomial number of steps
- Makes only a polynomial number of calls to a subroutine for **B**

Example

```
Algorithm for A:
Int i=0, i'=0;
Int j=0, j'=0;
…..
i=i+j;
(computation on i, j, i', j')
…..
Int x = B(i, j)
Int y = B(i', j')
…..
(compute z based on x and y)
Return z
```

Polynomial Time Reduction

Def $A \leq_P B$: if there is an algorithm for problem A using a 'black box' (subroutine) that solve problem B s.t.,

- Algorithm uses only a polynomial number of steps
- Makes only a polynomial number of calls to a subroutine for **B**

Question: Is the following polynomial time reduction correct?

Interval Scheduling $\leq_P$ Max Independent Set

- Yes. Without the blackbox of max independent set, we still have a polynomial time algorithm for interval scheduling.
- If problem A can be solved in polynomial time, then $A \leq_P B$ holds for any problem B

Polynomial Time Reduction

Def $A \leq_P B$: if there is an algorithm for problem A using a 'black box' (subroutine) that solve problem B s.t.,

- Algorithm uses only a polynomial number of steps
- Makes only a polynomial number of calls to a subroutine for **B**

So,

B is Polynomial time solvable $\Rightarrow$ A is Polynomial time solvable

Conversely,

No efficient Algorithm for A $\Rightarrow$ No efficient Algorithm for B

In words,

- Problem A is polynomial-time reducible to problem B
- B is as hard as A (it can be even harder)
- Informally, A is a special case of B

Polynomial Time Reduction

Purpose. Classify problems according to relative difficulty.

Design algorithms. If $A \leq_P B$ and B can be solved in polynomial-time, then A can also be solved in polynomial time.

Establish intractability. If $A \leq_P B$ and A cannot be solved in polynomial-time, then B cannot be solved in polynomial time.

Establish equivalence. If $A \leq_P B$ and $B \leq_P A$, we use notation $A \equiv_P B$.

up to cost of reduction