Chapter 20.3 Dijkstra's Algorithm & Chapter 34 NP-Completeness

Joon Soo Yoo

June 10, 2025

Assignment

34

- Read §22.3, §34.1
- Problems
  - §22.3 - 2

Chapter 22: Single-Source Shortest Paths

BFS

- Chapter 22.1: The Bellman-Ford Algorithm
- Chapter 22.2: Single-Source Shortest Paths in Directed Acyclic Graphs
- **Chapter 22.3: Dijkstra's Algorithm**
- Chapter 22.4: Difference Constraints and Shortest Paths
- Chapter 22.5: Proofs of Shortest-Paths Properties

Relaxation in Shortest-Paths Algorithms

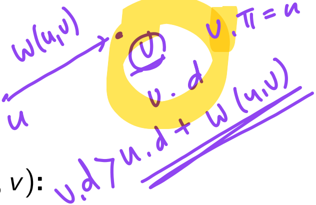
Core Concept: Relaxation

- Each vertex v maintains a **shortest-path estimate** $v.d$
  - $v.d$ is an upper bound on the weight of the shortest path from source s to v
- The goal is to iteratively **reduce** $v.d$ to the correct shortest-path weight $\delta(s, v)$

Initialization:

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
        v.d = ∞
        v.π = NIL
s.d = 0
```

The RELAX Operation



Relaxing an edge (u, v) with weight $w(u, v)$:

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Effect:

- Updates $v.d$ if a better path through u is found
- Updates $v.\pi$ to point to u

The RELAX Example

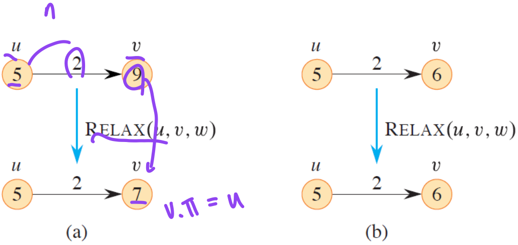
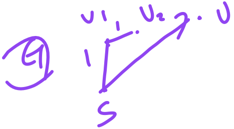


Figure 22.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Since we have $v.d \leq u.d + w(u, v)$ before relaxing the edge, the relaxation step leaves $v.d$ unchanged.

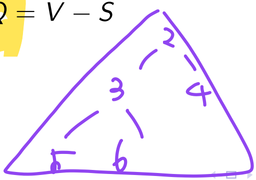
Dijkstra's Algorithm: Overview



Goal: Find the shortest paths from a source vertex s to all other vertices in a weighted, directed graph $G = (V, E)$, with $\mathbf{w(u, v) \geq 0}$.

Key Concepts:

- Generalizes BFS to weighted graphs
- Uses a **greedy strategy** to iteratively build a shortest-paths tree
- Maintains:
  - Set S: vertices whose shortest-path weights are known
  - Min-priority queue $Q = V - S$

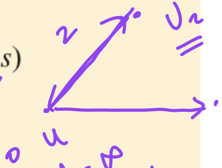


Dijkstra's Algorithm: Pseudocode

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = ∅
    for each vertex u ∈ G.V
        INSERT(Q, u)
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v in G.Adj[u]
            RELAX(u, v, w)
            if the call of RELAX decreased v.d
                DECREASE-KEY(Q, v, v.d)
```

$|V|$

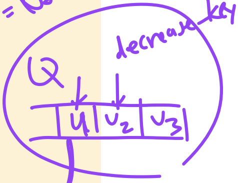
$|V|$



$d = 2$
$\pi = u$

$d = \infty$
$\pi = NIL$

$d = \infty$
$\pi = NIL$



min-Queue

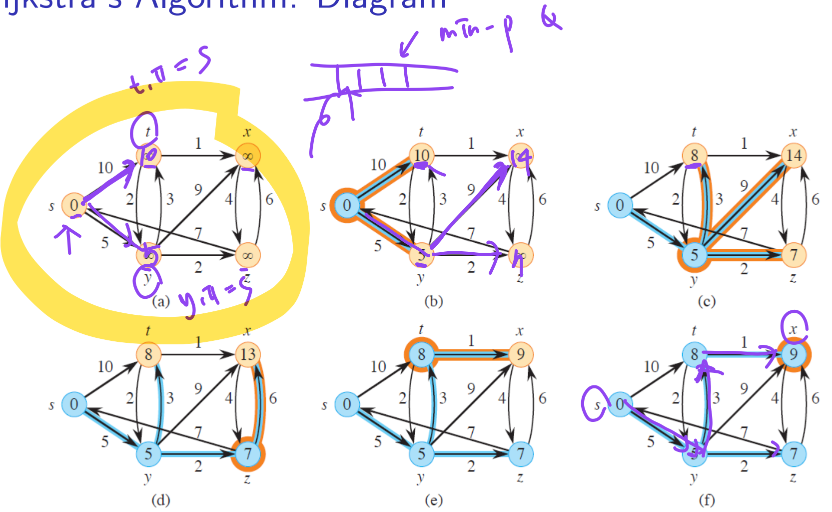
$O(1)$

$O(E)$

v_1

u

Dijkstra's Algorithm: Diagram



Dijkstra's Algorithm: Step-by-Step (I)

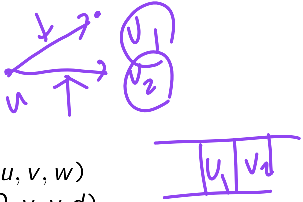
Input: Directed graph $G = (V, E)$ with $w(u, v) \geq 0$ for all $(u, v) \in E$ and source s

Initialization:

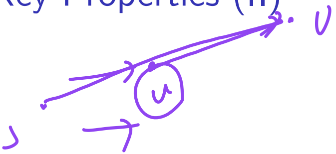
1. For each $v \in V$: set $v.d \leftarrow \infty$, $v.\pi \leftarrow$ NIL; then set $s.d \leftarrow 0$
2. Set $S \leftarrow \emptyset$ (vertices with finalized shortest-path weights)
3. Insert all $v \in V$ into a min-priority queue Q, keyed by $v.d$

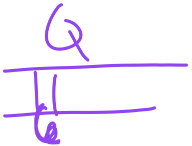
Main Loop:

- While $Q \neq \emptyset$:
  - $u \leftarrow$ `EXTRACT-MIN`(Q)
  - Add u to S
  - For each $v \in \mathrm{Adj}[u]$, perform `RELAX`(u, v, w)
  - If $v.d$ changes, call `DECREASE-KEY`$(Q, v, v.d)$



Dijkstra's Algorithm: Key Properties (II)





Greedy Strategy:

- Always selects vertex $u \in V - S$ with the smallest $u.d$
- Once u is added to S, $u.d = \delta(s, u)$ is guaranteed

Invariant:

- $Q = V - S$ at the start of each iteration
- Each vertex is extracted from Q exactly once
- Each edge (u, v) is relaxed at most once

Output: For all $v \in V$, $v.d = \delta(s, v)$ and $v.\pi$ gives the predecessor on the shortest path

Correctness of Dijkstra's Algorithm

BFS

Theorem (Theorem 22.6)

Let $G = (V, E)$ be a weighted, directed graph with nonnegative edge weights $w(u, v) \geq 0$ and source vertex $s \in V$. Then, after Dijkstra's algorithm terminates, for every vertex $u \in V$, we have:

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

u

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

That is, the algorithm correctly computes the shortest-path distance from s to u.

Corollary (Corollary 22.7)

The predecessor subgraph G_π forms a shortest-paths tree rooted at s.

Proof Sketch of Theorem 22.6 (1/2)

Strategy: Use induction on the number of iterations of the `while` loop (i.e., size of S).

Base case:

- When $S = \emptyset$, no vertex is added, so the claim trivially holds.
- When $S = \{s\}$, we have $s.d = 0 = \delta(s, s)$.

Inductive step:

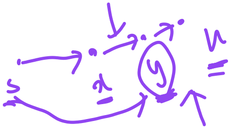
- Assume for all $v \in S$, $v.d = \delta(s, v)$.
- Let u be the next vertex extracted from $Q = V - S$ (minimum d).
- Show: $u.d = \delta(s, u)$.

Key idea:

- Let y be the first vertex on a shortest path from s to u that is not in S.
- Let $x \in S$ be the predecessor of y on this path.

Proof Sketch of Theorem 22.6 (2/2)

$y.d \geq \delta(s,y)$



$u.d \geq y.d$

Argument:

- From the inductive hypothesis: $x.d = \delta(s, x)$.
- Relaxation of edge (x, y) ensures: $y.d = \delta(s, y)$ at the time x was added to S.
- Because u was chosen as $\texttt{EXTRACT-MIN}(Q)$ and y is on the path to u, we have:

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

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

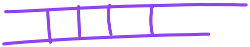
and since $y.d = \delta(s, y)$, all inequalities collapse:

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

Conclusion: By induction, when the algorithm terminates, $u.d = \delta(s, u)$ for all $u \in V$.

Time Complexity of Dijkstra's Algorithm (Simple Array Implementation)

Assumption: Min-priority queue implemented using an *unsorted array*.



Operation Costs:

- `INSERT(v)`: $\mathcal{O}(1)$
- `EXTRACT-MIN()`: $\mathcal{O}(|V|)$
- `DECREASE-KEY(v)`: $\mathcal{O}(1)$

Total Number of Calls:

- $|V|$ calls to `INSERT` and `EXTRACT-MIN`
- At most $|E|$ calls to `DECREASE-KEY` ($\because$ every edge relaxes once)

Total Time:

$$\mathcal{O}(|V| \cdot |V| + |E| \cdot 1) = \mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$$

Use case: Acceptable for *dense graphs*

Time Complexity of Dijkstra's Algorithm (Binary Min-Heap Implementation)

min-heap

Assumption: Min-priority queue implemented using a *binary min-heap* with vertex-to-index mapping.

Q

insert

Operation Costs:

- `INSERT(v)`: $\mathcal{O}(\log |V|)$ — insert into heap
- `EXTRACT-MIN()`: $\mathcal{O}(\log |V|)$ — remove min and heapify
- `DECREASE-KEY(v)`: $\mathcal{O}(\log |V|)$ — heapify

$O(\log(n))$

Total Number of Calls:

- $|V|$ calls to `INSERT` and `EXTRACT-MIN`
- At most $|E|$ calls to `DECREASE-KEY`

Total Time:

$$\mathcal{O}(|V| \log |V| + |E| \log |V|) = \mathcal{O}((|V| + |E|) \log |V|)$$

Use case: Efficient for *sparse graphs*

Chapter 34: NP-Completeness

- Chapter 34.1: Polynomial Time
- Chapter 34.2: Polynomial-Time Verification
- Chapter 34.3: NP-Completeness and Reducibility
- Chapter 34.4: NP-Completeness Proofs
- Chapter 34.5: NP-Complete Problems

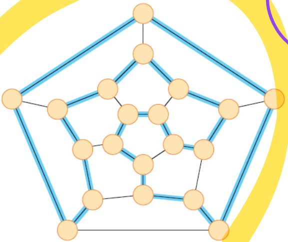
P, NP, and NP-Completeness: Informal Overview

- **P:** Problems solvable in polynomial time (e.g., $O(n^k)$ for some constant k).
- **NP:** Problems verifiable in polynomial time. Given a solution (*certificate*), it can be verified quickly.
- **NP-Complete (NPC):** Problems in NP that are as hard as any other problem in NP.

If any NP-complete problem is in P, then $\mathbf{P} \overset{?}{=} \mathbf{NP}$.

Examples of NP Verifiability

NP-complete



(a)

Hamiltonian Cycle:

- Input: Graph $G = (V, E)$
- Certificate: Sequence $\langle v_1, v_2, \ldots, v_{|V|} \rangle$
- Verify: Each node appears once; all edges exist; $v_{|V|} \rightarrow v_1$

P vs NP

- Every problem in **P** is in **NP**.
- Open question: Is $\mathbf{P} \subsetneq \mathbf{NP}$?
- If any NP-complete problem is in **P**, then $\mathbf{P} = \mathbf{NP}$.

$P \overset{?}{\neq} NP$

Most researchers believe $\mathbf{P} \neq \mathbf{NP}$.

What is NP-Completeness?

A problem Q is NP-complete if:

1. $Q \in$ **NP** (verifiable in poly time)
2. Every problem in **NP** can be *reduced* to Q in poly time

If any NP-complete problem is solvable in poly time, all of NP is.

$$NP_1 \leq_p \cdots \leq_p NP_k \leq Q$$

Why NP-Completeness Matters

- Proving NP-completeness shows likely intractability
- Better to design:
  - Approximation algorithms
  - Heuristics
  - Algorithms for special cases
- Many natural problems turn out to be NP-complete

Optimization vs Decision Problems

Optimization: Find best solution (e.g., shortest path, min cost).
Decision: Yes/No form. E.g.,

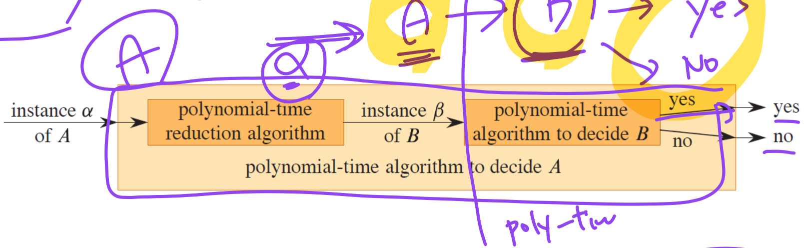
- PATH: Is there a path from u to v with $\leq k$ edges?

We study NP-completeness through decision problems.

Reductions: Relating Problem Hardness

- Given problems A and B
- If you can reduce A to B in poly time:
  - $A \leq_p B$: A is *no harder* than B
- Use known hard problems to show new problems are hard

Reduction Example: Visual



Goal: Show that problem A is solvable in polynomial time.

Algorithm A' for solving A:

1. Input: instance α of problem A
2. Compute $\beta = f(\alpha)$ in polynomial time
3. Run the known poly-time algorithm for B on β
4. Output the same answer for α

$B \le_p A$

NP-complete

Conclusion: A is solvable in polynomial time.

Proving Intractability via Reductions

Suppose:

- Problem A is known to be hard (no poly algorithm)
- You reduce A to B (i.e., A $\leq_p$ B)

Then:

- If B had a poly-time algorithm, so would A — contradiction!
- So B must also be hard

This is how we prove NP-completeness.

Chapter 34: NP-Completeness

- **Chapter 34.1: Polynomial Time**
- Chapter 34.2: Polynomial-Time Verification
- Chapter 34.3: NP-Completeness and Reducibility
- Chapter 34.4: NP-Completeness Proofs
- Chapter 34.5: NP-Complete Problems

Abstract Problems: Formal Definition

Abstract problem Q is defined as a **binary relation** between:

- A set I of **problem instances**, and
- A set S of **problem solutions**

Example: `SHORTEST-PATH`

- Instance: (G, u, v) — a graph and two vertices
- Solution: A path from u to v (or empty if none exists)
- Relation: Matches each instance with a valid shortest path

Note: There may be multiple valid solutions — shortest paths are not necessarily unique.

From Abstract Problems to Decision Problems

Why focus on decision problems?

- NP-completeness theory deals with problems having **yes/no answers**
- These are easier to define and analyze formally

A decision problem is a function:

$$f : I \to \{0, 1\}$$

- $f(i) = 1$ (yes) if the instance satisfies the condition
- $f(i) = 0$ (no) otherwise

Example: `PATH`

- Input: (G, u, v, k)
- Output: 1 if there exists a path from u to v with at most k edges; otherwise 0

Optimization vs. Decision Problems

Many problems are optimization problems:

- Find the shortest path, largest flow, minimum cut, etc.

But we can often transform them into decision problems:

- Ask: *"Is there a solution of cost $\leq k$?"*
- This transformation is key in NP-completeness proofs

Why this works:

- The decision version is *no harder* than the optimization version
- We can recover optimal values using repeated queries (e.g., binary search)

Optimization vs. Decision: Example

Problem: SHORTEST PATH

Optimization Version

- Given a graph G and vertices u, v
- **Goal:** Find the *length of the shortest path* from u to v

Decision Version

- Given a graph G, vertices u, v, and integer k
- **Question:** Is there a path from u to v with length $\leq k$?
- Answer is **Yes/No**

Both versions deal with the same underlying problem, but the decision version is used for NP-completeness analysis.

What is an Encoding?

Encoding is how we represent abstract problem instances as binary strings that a computer can understand.

Formal Definition: An *encoding* is a function:

$$e : S \to \{0, 1\}^*$$

that maps abstract objects S to binary strings.

Examples:

- $e(17) = 10001$
- $e(\text{A}) = 01000001$ (ASCII)
- A graph G: encoded as adjacency list, matrix, or edge list

Compound objects (sets, graphs, programs) are encoded by combining the encodings of their parts.

From Abstract to Concrete Problems

To run on a computer, an abstract decision problem must be encoded.

Concrete Problem:

- Input: a binary string $x \in \{0, 1\}^*$
- Output: $f(x) \in \{0, 1\}$

Size of instance: $n = |x|$ (length of binary string)

Polynomial-time algorithm: solves any input of length n in $O(n^k)$ for some constant k.

Encoding Can Affect Complexity

Encoding affects whether an algorithm appears to run in polynomial or exponential time.

Example: Integer k given as input

- Unary encoding: $111\ldots1$ (length $n = k$) $\rightarrow$ runtime $\Theta(k) = \Theta(n) \rightarrow$ polynomial
- Binary encoding: 10001 (length $n = \lfloor \log_2 k \rfloor + 1$) $\rightarrow$ runtime $\Theta(k) = \Theta(2^n) \rightarrow$ exponential

Conclusion: The *choice of encoding* can drastically change the apparent complexity.

Polynomially Related Encodings

Goal: We want our notion of "efficient" (polynomial-time) to be encoding-independent.

Definition: Encodings e_1 and e_2 are *polynomially related* if:

- There exist functions f_{12}, f_{21} computable in polynomial time
- Such that $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$ for all instances i

This ensures we can convert between encodings without changing the complexity class.

Lemma 34.1: Encoding-Invariant Polynomial Time

Lemma: Let Q be an abstract decision problem.
If encodings e_1 and e_2 are polynomially related, then:

$$e_1(Q) \in \mathbf{P} \iff e_2(Q) \in \mathbf{P}$$

Implication:

- Polynomial-time solvability doesn't depend on which *reasonable* encoding you choose
- We can now safely talk about abstract problems being in **P**, assuming a standard encoding

Practical Takeaways on Encodings

- You must encode instances as binary strings to analyze complexity formally
- Bad encodings (e.g., unary) can make easy problems look hard
- As long as we use **reasonable encodings** (e.g., binary, base-3, ASCII), complexity classes are preserved
- We usually assume a *standard encoding* (e.g., $\langle G \rangle$ for a graph)

This allows us to focus on the *problem itself*, not the details of its representation.

A Formal-Language Framework for Decision Problems

Alphabet: A finite set of symbols (denoted Σ)

Language: A set of strings over Σ (i.e., $L \subseteq \Sigma^*$)

Examples:

- $\Sigma = \{0, 1\}$
- $L = \{10, 11, 101, 111, 10001, \ldots\}$ (binary encodings of prime numbers)

Useful notation:

- ϵ: empty string
- Σ^*: set of all binary strings

Viewing Decision Problems as Languages

Decision problem $Q : \Sigma^* \to \{0,1\}$ maps binary strings to yes/no.

We define the associated language:

$$L_Q = \{x \in \Sigma^* \mid Q(x) = 1\}$$

This is the set of inputs for which the answer is **yes**.

Example: PATH

- Input: $x = \langle G, u, v, k \rangle$ (encoded as a binary string)
- $x \in L_{\text{PATH}}$ if G contains a path from u to v of length $\leq k$

This lets us study decision problems as formal languages, enabling complexity class definitions.

Accepting vs. Deciding a Language

Algorithm A **accepts** x if $A(x) = 1$ (i.e., it outputs yes).
Language accepted by A**:**

$$L = \{x \in \Sigma^* \mid A(x) = 1\}$$

But: A might not halt on $x \notin L$ (it might loop forever).
Algorithm A **decides** L if:

- $A(x) = 1$ for $x \in L$
- $A(x) = 0$ for $x \notin L$

Polynomial-Time Acceptance vs. Decision

Accepted in polynomial time:

- A outputs 1 for $x \in L$ in $O(n^k)$ time
- No guarantee for $x \notin L$ (may loop forever)

Decided in polynomial time:

- A halts on all inputs
- $A(x) = 1$ if $x \in L$, and $A(x) = 0$ otherwise

Example: PATH can be both accepted and decided in polynomial time

Defining **P** via Formal Languages

Complexity Class P:

$\mathbf{P} = \{L \subseteq \Sigma^* \mid \text{there exists an algorithm that decides } L \text{ in } O(n^k) \text{ time}\}$

Theorem 34.2:

- **P** is the set of languages that are *accepted* by a polynomial-time algorithm
- Proof: Any polynomial-time accepting algorithm can be converted to a deciding algorithm via bounded simulation

Question?