Longest Path Problem

Ahnaf Faisal, 1505005
Raihanul Alam, 1505010
Mahim Mahbub, 1505022
Zahin Wahab, 1505031
Bishal Basak Papan, 1505043

Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology

October 5, 2020

Outline:

Where we left off

- We have already introduced the **Longest Path Problem** in our last presentation.

Where we left off

- We have already introduced the **Longest Path Problem** in our last presentation.
- Today we are going to extend our analysis and focus on **exact exponential algorithms** for solving this problem and **exact polynomial algorithm** for solving a variation of this problem.

Where we left off

- We have already introduced the **Longest Path Problem** in our last presentation.
- Today we are going to extend our analysis and focus on **exact exponential algorithms** for solving this problem and **exact polynomial algorithm** for solving a variation of this problem.
- But before we proceed any further, let's just shed some light to our previous discussion.

What is a Longest Path Problem?

Previously we used **decision version** of this problem to analyse its complexity.

What is a Longest Path Problem?

Previously we used **decision version** of this problem to analyse its complexity.

Decision Version (Weighted)

Given a graph G, and an integer k, does the graph G have a longest path of *total weight* **at least** k ?

What is a Longest Path Problem?

Previously we used **decision version** of this problem to analyse its complexity.

Decision Version (Weighted)

Given a graph G, and an integer k, does the graph G have a longest path of *total weight* **at least** k ?

We are more concerned about optimization version for this week.

What is a Longest Path Problem?

Previously we used **decision version** of this problem to analyse its complexity.

Decision Version (Weighted)

Given a graph G, and an integer k, does the graph G have a longest path of *total weight* **at least** k ?

We are more concerned about optimization version for this week.

Optimization Version

Given a weighted graph G, find a simple path in this graph which has the maximum weight.

But Longest Path problem is NP-Complete

- From our previous discussion, we can safely state that Longest Path problem is **NP-Complete** which makes it both **NP** and **NP-Hard**.
- So, unless **P=NP**, there is no polynomial time algorithm which gives exact solution of Longest Path problem.
- We have to take resort to **exponential time** if we are to solve this problem exactly.
- Which brings us to this week's content: **Exact Exponential Algorithms**.

But Longest Path problem is NP-Complete

- From our previous discussion, we can safely state that Longest Path problem is **NP-Complete** which makes it both **NP** and **NP-Hard**.
- So, unless **P=NP**, there is no polynomial time algorithm which gives exact solution of Longest Path problem.
- We have to take resort to **exponential time** if we are to solve this problem exactly.
- Which brings us to this week's content: **Exact Exponential Algorithms**.

But Longest Path problem is NP-Complete

- From our previous discussion, we can safely state that Longest Path problem is **NP-Complete** which makes it both **NP** and **NP-Hard**.
- So, unless **P=NP**, there is no polynomial time algorithm which gives exact solution of Longest Path problem.
- We have to take resort to **exponential time** if we are to solve this problem exactly.
- Which brings us to this week's content: **Exact Exponential Algorithms**.

But Longest Path problem is NP-Complete

- From our previous discussion, we can safely state that Longest Path problem is **NP-Complete** which makes it both **NP** and **NP-Hard**.
- So, unless **P=NP**, there is no polynomial time algorithm which gives exact solution of Longest Path problem.
- We have to take resort to **exponential time** if we are to solve this problem exactly.
- Which brings us to this week's content: **Exact Exponential Algorithms**.

Our take on Exact Exponential Algorithms

- We will first try to solve longest path problem **naively**. Naive algorithm is super exponential (we will see this later), but a very good starting point for designing better algorithms.
- Then we will focus on **improving** the time complexity. As our solution will be exact, the time needed will still be exponential.
- And finally, we will bring a special class of graph which needs **polynomial** time to find out the longest path.

Our take on Exact Exponential Algorithms

- We will first try to solve longest path problem **naively**. Naive algorithm is super exponential (we will see this later), but a very good starting point for designing better algorithms.
- Then we will focus on **improving** the time complexity. As our solution will be exact, the time needed will still be exponential.
- And finally, we will bring a special class of graph which needs **polynomial** time to find out the longest path.

Our take on Exact Exponential Algorithms

- We will first try to solve longest path problem **naively**. Naive algorithm is super exponential (we will see this later), but a very good starting point for designing better algorithms.
- Then we will focus on **improving** the time complexity. As our solution will be exact, the time needed will still be exponential.
- And finally, we will bring a special class of graph which needs **polynomial** time to find out the longest path.

Why and where naive approach is needed?

Why?

Brute force algorithms are slow generally and exponential in our case. But if we start our analysis with brute force approach, we can figure out why it is slow and try to make it better.

Where?

Naive approach works perfectly fine for smaller input instances. But as the graph size gets bigger, it becomes almost impossible to get optimal solution using brute force approach.

Why and where naive approach is needed?

Why?

Brute force algorithms are slow generally and exponential in our case. But if we start our analysis with brute force approach, we can figure out why it is slow and try to make it better.

Where?

Naive approach works perfectly fine for smaller input instances. But as the graph size gets bigger, it becomes almost impossible to get optimal solution using brute force approach.

Brute Force Algorithm

The Brute Force approach:

- Look at all subsets of vertices.
    - Where the subset contains at least 2 vertices.
- Take all possible paths in a subset.
- Pick the path which has the maximum weight/length.

Brute Force Algorithm

The Brute Force approach:

- Look at all subsets of vertices.
    - Where the subset contains at least 2 vertices.
- Take all possible paths in a subset.
- Pick the path which has the maximum weight/length.

Brute Force Algorithm

The Brute Force approach:

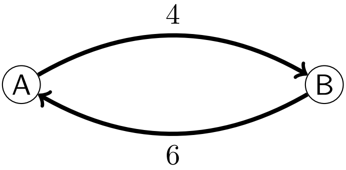
- Look at all subsets of vertices.
  - Where the subset contains at least 2 vertices.
- Take all possible paths in a subset.
- Pick the path which has the maximum weight/length.

Brute Force Algorithm

The Brute Force approach:

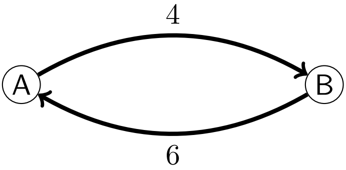
- Look at all subsets of vertices.
    - Where the subset contains at least 2 vertices.
- Take all possible paths in a subset.
- Pick the path which has the maximum weight/length.

Brute Force Algorithm Example



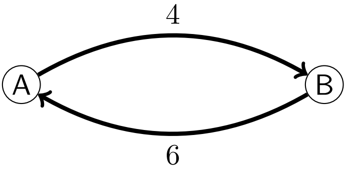
- Here, the set of vertices: $V = \{A, B\}$
- The subsets of V: $\{\}, \{A\}, \{B\}, \{A, B\}$
- Subsets to consider: $S = \left\{\{A, B\}\right\}$

Brute Force Algorithm Example



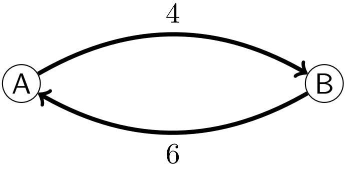
- Here, the set of vertices: $V = \{A, B\}$
- The subsets of V: $\{\}, \{A\}, \{B\}, \{A, B\}$
- Subsets to consider: $S = \{\{A, B\}\}$

Brute Force Algorithm Example



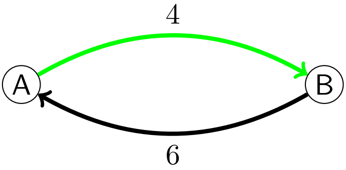
- Here, the set of vertices: $V = \{A, B\}$
- The subsets of V: $\{\}, \{A\}, \{B\}, \{A, B\}$
- Subsets to consider: $S = \{\{A, B\}\}$

Brute Force Algorithm Example



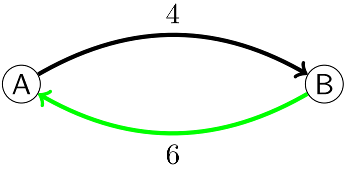
- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:
  - $P_1 : A - B$, weight $= 4$
  - $P_2 : B - A$, weight $= 6$
- Longest Path $= P_2 : B - A$ of weight 6.

Brute Force Algorithm Example



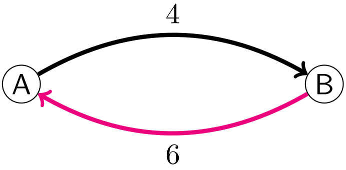
- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:
  - $P_1 : A - B$, weight $= 4$
  - $P_2 : B - A$, weight $= 6$
- Longest Path $= P_2 : B - A$ of weight 6.

Brute Force Algorithm Example



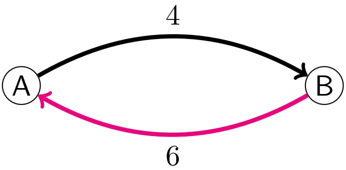
- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:
  - $P_1 : A – B$, weight $= 4$
  - $P_2 : B – A$, weight $= 6$
- Longest Path $= P_2 : B – A$ of weight 6.

Brute Force Algorithm Example



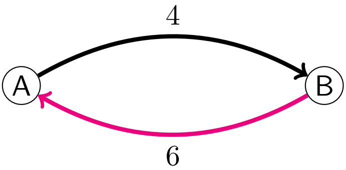
- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:
  - $P_1 : A – B$, weight $= 4$
  - $P_2 : B – A$, weight $= 6$
- Longest Path $= P_2 : B – A$ of weight 6.

Brute Force Algorithm Example



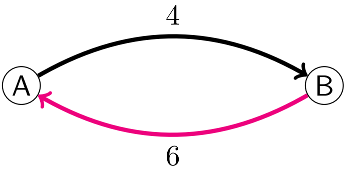
- For a graph of 2 vertices:
  - 1 subset of vertices of cardinality 2.
  - Total permutations of the vertices in a subset: $2!$
  - Total paths in the graph $= \binom{2}{2} \times 2! = 2$.

Brute Force Algorithm Example



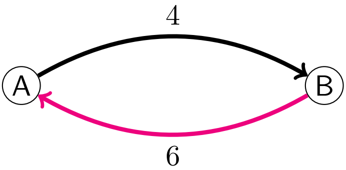
- For a graph of 2 vertices:
  - 1 subset of vertices of cardinality 2.
  - Total permutations of the vertices in a subset: $2!$
  - Total paths in the graph $= \binom{2}{2} \times 2! = 2$.

Brute Force Algorithm Example



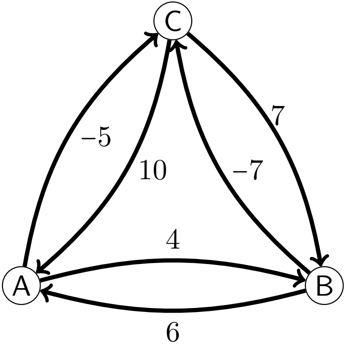
- For a graph of 2 vertices:
  - 1 subset of vertices of cardinality 2.
  - Total permutations of the vertices in a subset: $2!$
  - Total paths in the graph $= \binom{2}{2} \times 2! = 2$.

Brute Force Algorithm Example



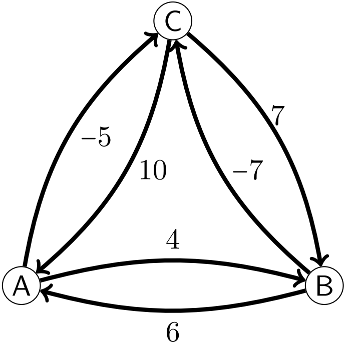
- For a graph of 2 vertices:
  - 1 subset of vertices of cardinality 2.
  - Total permutations of the vertices in a subset: $2!$
  - Total paths in the graph $= \binom{2}{2} \times 2! = 2$.

Brute Force Algorithm Example



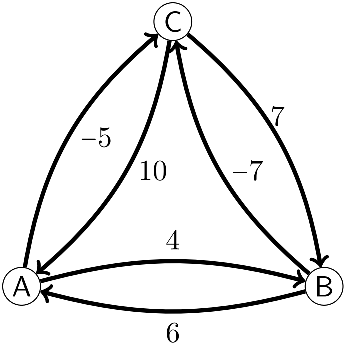
- Here, the set of vertices: $V = \{A, B, C\}$
- The subsets of V: $\{\}, \{A\}, \{B\}, \{C\}$, $\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}$
- Subsets to consider: $S = \{\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$

Brute Force Algorithm Example



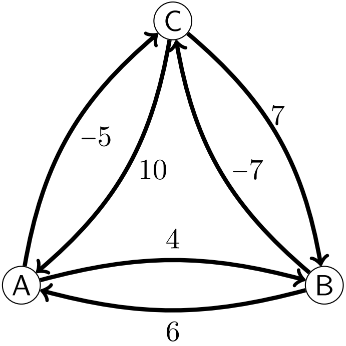
- Here, the set of vertices: $V = \{A, B, C\}$
- The subsets of V: $\{\}, \{A\}, \{B\}, \{C\}$, $\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}$
- Subsets to consider: $S = \{\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}$

Brute Force Algorithm Example



- Here, the set of vertices: $V = \{A, B, C\}$
- The subsets of V: $\{\}, \{A\}, \{B\}, \{C\}$, $\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}$
- Subsets to consider: $S = \big\{\{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\big\}$

Brute Force Algorithm Example

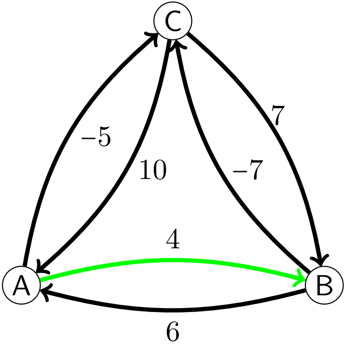


- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:

- $P_{1_1} : A - B$, weight $= 4$
- $P_{1_2} : B - A$, weight $= 6$

- Current longest path $= P_{1_2} : B - A$ of weight 6.

Brute Force Algorithm Example

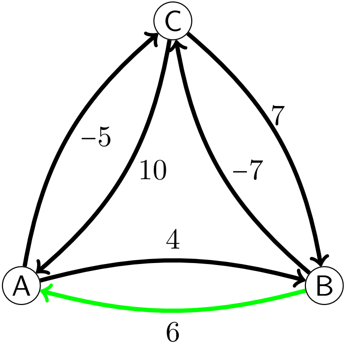


- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:

- $P_{1_1} : A - B$, weight $= 4$
- $P_{1_2} : B - A$, weight $= 6$

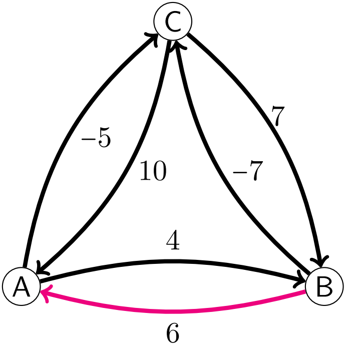
- Current longest path $= P_{1_2} : B - A$ of weight 6.

Brute Force Algorithm Example



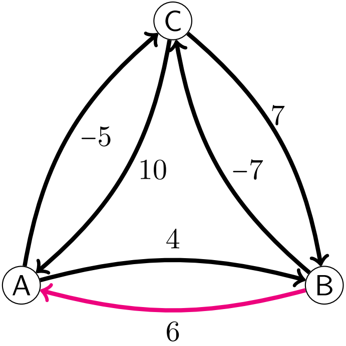
- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:
- $P_{1_1} : A - B$, weight $= 4$
- $P_{1_2} : B - A$, weight $= 6$
- Current longest path $= P_{1_2} : B - A$ of weight 6.

Brute Force Algorithm Example



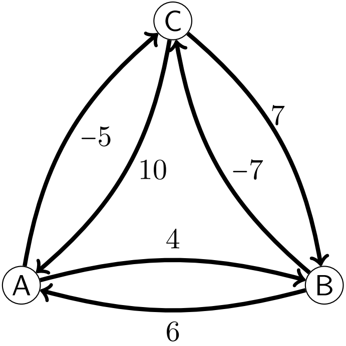
- Subset to consider: $S_1 = \{A, B\}$
- Paths from this subset:
- $P_{1_1} : A - B$, weight $= 4$
- $P_{1_2} : B - A$, weight $= 6$
- Current longest path $= P_{1_2} : B - A$ of weight 6.

Brute Force Algorithm Example



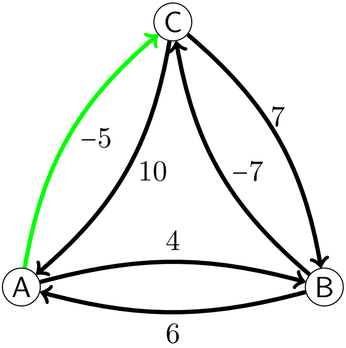
- Previous longest path $= P_{1_2} : B - A$ of weight 6.
- Subset to consider: $S_2 = \{A, C\}$
- Paths from this subset:
  - $P_{2_1} : A - C$, weight $= -5$
  - $P_{2_2} : C - A$, weight $= 10$
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example



- Previous longest path $= P_{1_2} : B – A$ of weight 6.
- Subset to consider: $S_2 = \{A, C\}$
- Paths from this subset:
  - $P_{2_1} : A – C$, weight $= -5$
  - $P_{2_2} : C – A$, weight $= 10$
- Current longest path $= P_{2_2} : C – A$ of weight 10.

Brute Force Algorithm Example

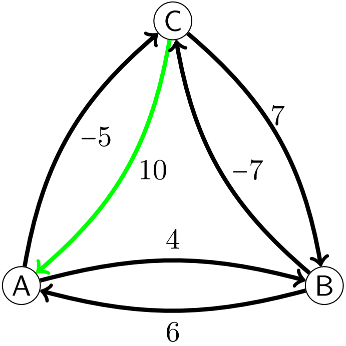


- Previous longest path $= P_{1_2} : B - A$ of weight 6.
- Subset to consider: $S_2 = \{A, C\}$
- Paths from this subset:

- $P_{2_1} : A - C$, weight $= -5$
- $P_{2_2} : C - A$, weight $= 10$

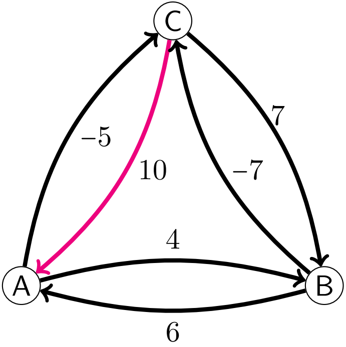
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example



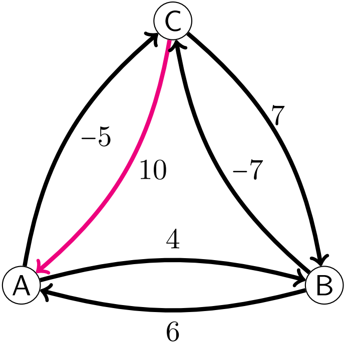
- Previous longest path $= P_{1_2} : B – A$ of weight 6.
- Subset to consider: $S_2 = \{A, C\}$
- Paths from this subset:
- $P_{2_1} : A – C$, weight $= –5$
- $P_{2_2} : C – A$, weight $= 10$
- Current longest path $= P_{2_2} : C – A$ of weight 10.

Brute Force Algorithm Example



- Previous longest path $= P_{1_2} : B - A$ of weight 6.
- Subset to consider: $S_2 = \{A, C\}$
- Paths from this subset:
- $P_{2_1} : A - C$, weight $= -5$
- $P_{2_2} : C - A$, weight $= 10$
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example

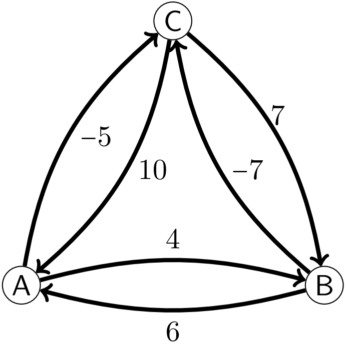


- Previous longest path $= P_{2_2} : C - A$ of weight 10.
- Subset to consider: $S_3 = \{B, C\}$
- Paths from this subset:

- $P_{3_1} : B - C$, weight $= -7$
- $P_{3_2} : C - B$, weight $= 7$

- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example

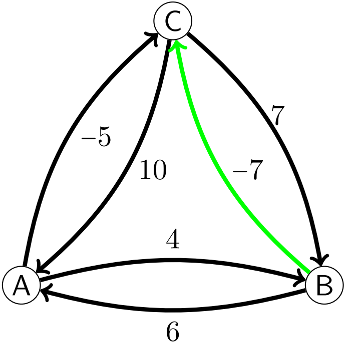


- Previous longest path $= P_{2_2} : C - A$ of weight 10.
- Subset to consider: $S_3 = \{B, C\}$
- Paths from this subset:

- $P_{3_1} : B - C$, weight $= -7$
- $P_{3_2} : C - B$, weight $= 7$

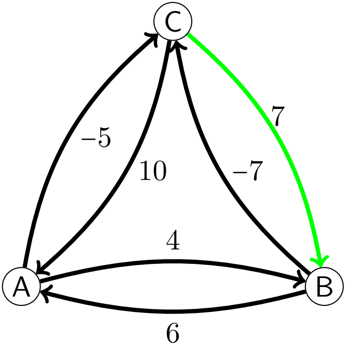
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example



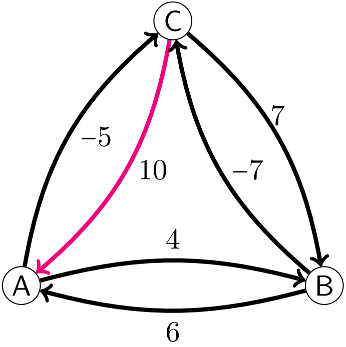
- Previous longest path $= P_{2_2} : C - A$ of weight 10.
- Subset to consider: $S_3 = \{B, C\}$
- Paths from this subset:
- $P_{3_1} : B - C$, weight $= -7$
- $P_{3_2} : C - B$, weight $= 7$
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example



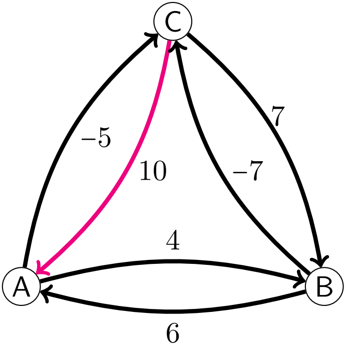
- Previous longest path $= P_{2_2} : C - A$ of weight 10.
- Subset to consider: $S_3 = \{B, C\}$
- Paths from this subset:
  - $P_{3_1} : B - C$, weight $= -7$
  - $P_{3_2} : C - B$, weight $= 7$
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example



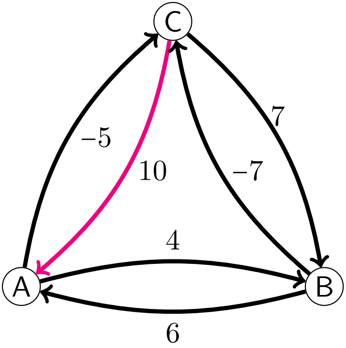
- Previous longest path $= P_{2_2} : C - A$ of weight 10.
- Subset to consider: $S_3 = \{B, C\}$
- Paths from this subset:
- $P_{3_1} : B - C$, weight $= -7$
- $P_{3_2} : C - B$, weight $= 7$
- Current longest path $= P_{2_2} : C - A$ of weight 10.

Brute Force Algorithm Example



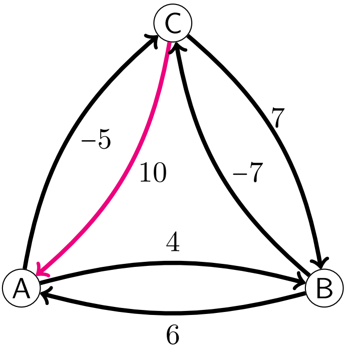
- We have already considered 3 subsets of V, each containing 2 vertices.
- The total number of unique paths obtained from these subsets is $= \binom{3}{2} \times 2! = 6$.

Brute Force Algorithm Example



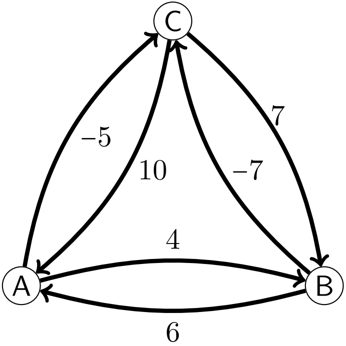
- We have already considered 3 subsets of V, each containing 2 vertices.
- The total number of unique paths obtained from these subsets is $= \binom{3}{2} \times 2! = 6$.

Brute Force Algorithm Example



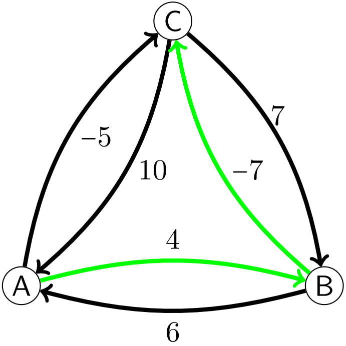
- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:
  - $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
  - $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
  - $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
  - $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
  - $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
  - $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example



- Previous longest path $= P_{2_2} : C – A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:
  - $P_{4_1} : A – B – C$, weight $= 4 + (-7) = -3$
  - $P_{4_2} : A – C – B$, weight $= (-5) + 7 = 2$
  - $P_{4_3} : B – A – C$, weight $= 6 + (-5) = 1$
  - $P_{4_4} : B – C – A$, weight $= (-7) + 10 = 3$
  - $P_{4_5} : C – A – B$, weight $= 10 + 4 = 14$
  - $P_{4_6} : C – B – A$, weight $= 7 + 6 = 13$
- Current longest path $= P_{4_5} : C – A – B$ of weight 14.

Brute Force Algorithm Example

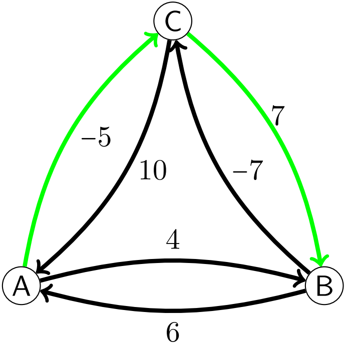


- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:

- $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
- $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
- $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
- $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
- $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
- $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$

- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example

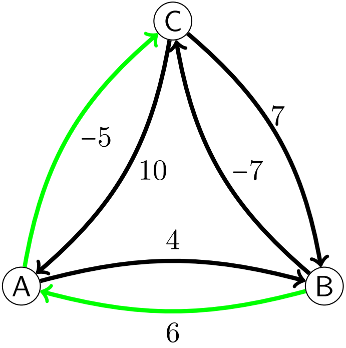


- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:

- $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
- $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
- $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
- $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
- $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
- $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$

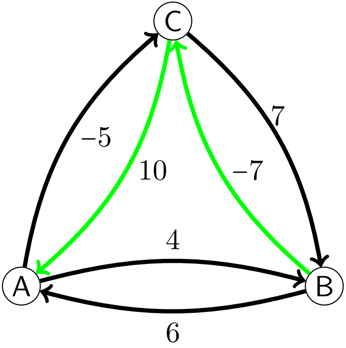
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example



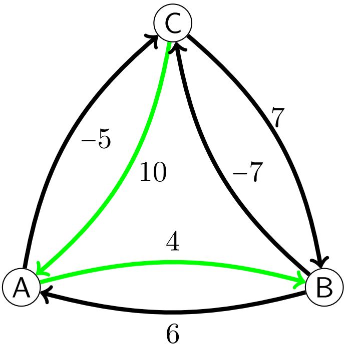
- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:
- $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
- $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
- $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
- $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
- $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
- $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example



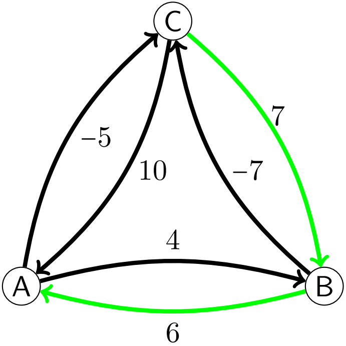
- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:
- $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
- $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
- $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
- $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
- $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
- $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example



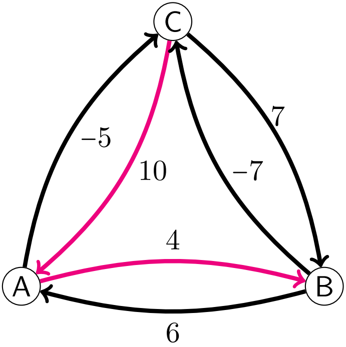
- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:
  - $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
  - $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
  - $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
  - $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
  - $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
  - $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example



- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:
- $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
- $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
- $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
- $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
- $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
- $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example

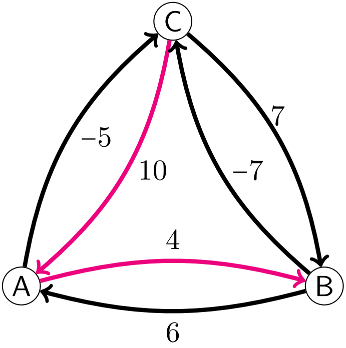


- Previous longest path $= P_{2_2} : C - A$ of weight 6.
- Subset to consider: $S_4 = \{A, B, C\}$
- Paths from this subset:

- $P_{4_1} : A - B - C$, weight $= 4 + (-7) = -3$
- $P_{4_2} : A - C - B$, weight $= (-5) + 7 = 2$
- $P_{4_3} : B - A - C$, weight $= 6 + (-5) = 1$
- $P_{4_4} : B - C - A$, weight $= (-7) + 10 = 3$
- $P_{4_5} : C - A - B$, weight $= 10 + 4 = 14$
- $P_{4_6} : C - B - A$, weight $= 7 + 6 = 13$

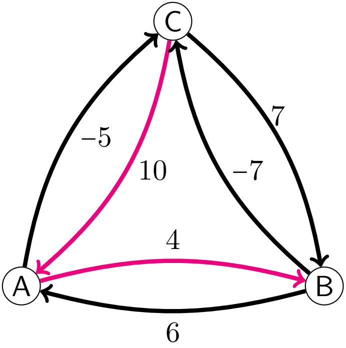
- Current longest path $= P_{4_5} : C - A - B$ of weight 14.

Brute Force Algorithm Example



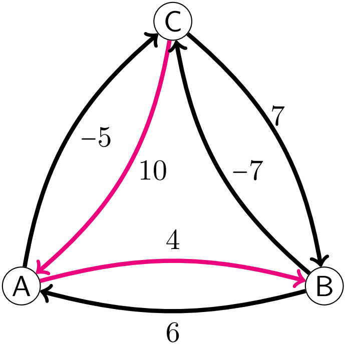
- For all possible subsets having 2 vertices, we got $\binom{3}{2} \times 2! = 6$ unique paths.
- Then, for the only possible subset having 3 vertices, we have got $\binom{3}{3} \times 3! = 6$ unique paths.
- So, from the complete graph of 3 vertices, we have got $(\binom{3}{2} \times 2! + \binom{3}{3} \times 3!) = 12$ unique paths in total.

Brute Force Algorithm Example



- For all possible subsets having 2 vertices, we got $\binom{3}{2} \times 2! = 6$ unique paths.
- Then, for the only possible subset having 3 vertices, we have got $\binom{3}{3} \times 3! = 6$ unique paths.
- So, from the complete graph of 3 vertices, we have got $(\binom{3}{2} \times 2! + \binom{3}{3} \times 3!) = 12$ unique paths in total.

Brute Force Algorithm Example



- For all possible subsets having 2 vertices, we got $\binom{3}{2} \times 2! = 6$ unique paths.
- Then, for the only possible subset having 3 vertices, we have got $\binom{3}{3} \times 3! = 6$ unique paths.
- So, from the complete graph of 3 vertices, we have got $(\binom{3}{2} \times 2! + \binom{3}{3} \times 3!) = 12$ unique paths in total.

Brute Force Algorithm Complexity

However this approach takes a toll on running time.

- In a graph on n vertices, there are $O(2^n)$ possible subsets.
- Each subset of p vertices(where $2 \leq p \leq n$) has $p!$ unique paths. So, $p! = O(n!)$.
- Computing the cost of a path takes linear time.
- **Brute-Force Algorithm running time:** number of subsets × number of possible tours × cost of computing one tour $= O(2^n) \times O(n!) \times O(n) = O(2^n n^n)$

Brute Force Algorithm Complexity

However this approach takes a toll on running time.

- In a graph on n vertices, there are $O(2^n)$ possible subsets.
- Each subset of p vertices(where $2 \leq p \leq n$) has $p!$ unique paths. So, $p! = O(n!)$.
- Computing the cost of a path takes linear time.
- **Brute-Force Algorithm running time:** number of subsets $\times$ number of possible tours $\times$ cost of computing one tour $= O(2^n) \times O(n!) \times O(n) = O(2^n n^n)$

Brute Force Algorithm Complexity

However this approach takes a toll on running time.

- In a graph on n vertices, there are $O(2^n)$ possible subsets.
- Each subset of p vertices(where $2 \leq p \leq n$) has $p!$ unique paths. So, $p! = O(n!)$.
- Computing the cost of a path takes linear time.
- **Brute-Force Algorithm running time:** number of subsets $\times$ number of possible tours $\times$ cost of computing one tour $= O(2^n) \times O(n!) \times O(n) = O(2^n n^n)$

Brute Force Algorithm Complexity

However this approach takes a toll on running time.

- In a graph on n vertices, there are $O(2^n)$ possible subsets.
- Each subset of p vertices(where $2 \leq p \leq n$) has $p!$ unique paths. So, $p! = O(n!)$.
- Computing the cost of a path takes linear time.
- **Brute-Force Algorithm running time:** number of subsets $\times$ number of possible tours $\times$ cost of computing one tour $= O(2^n) \times O(n!) \times O(n) = O(2^n n^n)$

Brute Force Algorithm Complexity

However this approach takes a toll on running time.

- In a graph on n vertices, there are $O(2^n)$ possible subsets.
- Each subset of p vertices(where $2 \leq p \leq n$) has $p!$ unique paths. So, $p! = O(n!)$.
- Computing the cost of a path takes linear time.
- **Brute-Force Algorithm running time:** number of subsets $\times$ number of possible tours $\times$ cost of computing one tour $= O(2^n) \times O(n!) \times O(n) = O(2^n n^n)$

Dynamic Programming Approach for Longest Path

A Better Exact Algorithm is found using Dynamic Programming Approach.

- Let's look at the insights into the construction of Dynamic Programming approach for Longest Path Problem.
- We will use DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem.

Dynamic Programming Approach for Longest Path

A Better Exact Algorithm is found using Dynamic Programming Approach.

- Let's look at the insights into the construction of Dynamic Programming approach for Longest Path Problem.
- We will use DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem.

Dynamic Programming Approach for Longest Path

A Better Exact Algorithm is found using Dynamic Programming Approach.

- Let's look at the insights into the construction of Dynamic Programming approach for Longest Path Problem.
- We will use DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem.

DP: Initializing

- We are given a graph $G(V, E)$, with $V = c_1, c_2, ..., c_n$ where $n = |V|$.
- We are also given weights $w(c_i, c_j)$ between two vertices c_i and c_j.
- If $(c_i, c_j) \notin E$, we set $w(c_i, c_j) = -\infty$.
- We want a permutation of V such that the total weight of the path is maximized.
- Formally, we want a permutation π of $1, 2, ..., n$ and $1 < k \leq n$ to maximize the objective function:

$$LP(\pi, k) = \sum_{i=1}^{k-1} w(c_{\pi_i}, c_{\pi_{i+1}}) \quad (1)$$

DP: Initializing

- We are given a graph $G(V, E)$, with $V = c_1, c_2, ..., c_n$ where $n = |V|$.
- We are also given weights $w(c_i, c_j)$ between two vertices c_i and c_j.
- If $(c_i, c_j) \notin E$, we set $w(c_i, c_j) = -\infty$.
- We want a permutation of V such that the total weight of the path is maximized.
- Formally, we want a permutation π of $1, 2, ..., n$ and $1 < k \leq n$ to maximize the objective function:

$$LP(\pi, k) = \sum_{i=1}^{k-1} w(c_{\pi_i}, c_{\pi_{i+1}}) \tag{1}$$

DP: Initializing

- We are given a graph $G(V, E)$, with $V = c_1, c_2, ..., c_n$ where $n = |V|$.
- We are also given weights $w(c_i, c_j)$ between two vertices c_i and c_j.
- If $(c_i, c_j) \notin E$, we set $w(c_i, c_j) = -\infty$.
- We want a permutation of V such that the total weight of the path is maximized.
- Formally, we want a permutation π of $1, 2, ..., n$ and $1 < k \leq n$ to maximize the objective function:

$$LP(\pi, k) = \sum_{i=1}^{k-1} w(c_{\pi_i}, c_{\pi_{i+1}}) \quad (1)$$

DP: Initializing

- We are given a graph $G(V, E)$, with $V = c_1, c_2, ..., c_n$ where $n = |V|$.
- We are also given weights $w(c_i, c_j)$ between two vertices c_i and c_j.
- If $(c_i, c_j) \notin E$, we set $w(c_i, c_j) = -\infty$.
- We want a permutation of V such that the total weight of the path is maximized.
- Formally, we want a permutation π of $1, 2, ..., n$ and $1 < k \leq n$ to maximize the objective function:

$$LP(\pi, k) = \sum_{i=1}^{k-1} w(c_{\pi_i}, c_{\pi_{i+1}}) \tag{1}$$

DP: Initializing

- We are given a graph $G(V, E)$, with $V = c_1, c_2, ..., c_n$ where $n = |V|$.
- We are also given weights $w(c_i, c_j)$ between two vertices c_i and c_j.
- If $(c_i, c_j) \notin E$, we set $w(c_i, c_j) = -\infty$.
- We want a permutation of V such that the total weight of the path is maximized.
- Formally, we want a permutation π of $1, 2, ..., n$ and $1 < k \leq n$ to maximize the objective function:

$$LP(\pi, k) = \sum_{i=1}^{k-1} w(c_{\pi_i}, c_{\pi_{i+1}}) \tag{1}$$

DP: Defining Sub-Problems

- Any Dynamic Programming approach requires sub-problems.
- We will define one sub-problem per **each subset** of vertices and **starting** vertex of the longest path *till now*.

Required Sub-Problem

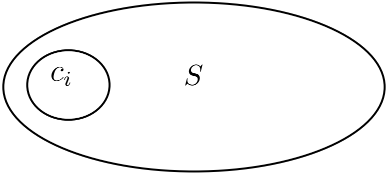
For every non-empty subset $S \subseteq V$ and c_i, let $OPT[c_i, S]$ be the longest path which starts at c_i ($c_i \in S$) and contains the total maximum weight by using vertices of S.

DP: Defining Sub-Problems

- Any Dynamic Programming approach requires sub-problems.
- We will define one sub-problem per **each subset** of vertices and **starting** vertex of the longest path *till now*.

Required Sub-Problem

For every non-empty subset $S \subseteq V$ and c_i, let $OPT[c_i, S]$ be the longest path which starts at c_i ($c_i \in S$) and contains the total maximum weight by using vertices of S.



DP: Defining Recurrence Relationship

- Let's setup the recurrence relationship.
- $OPT[c_i, S]$ contains the longest path starting from $c_i, c_i \in S$ and uses vertices of S (not necessarily all vertices of S).
- We will condition on the **neighbour** of c_i to setup the required recurrence relationship.

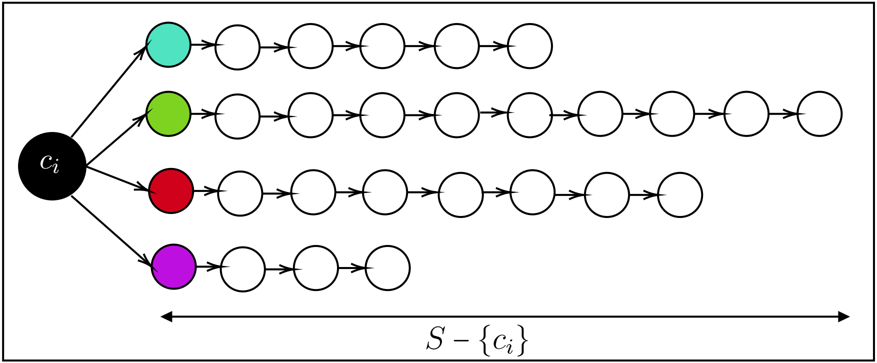
DP: Defining Recurrence Relationship

- Let's setup the recurrence relationship.
- $OPT[c_i, S]$ contains the longest path starting from $c_i, c_i \in S$ and uses vertices of S (not necessarily all vertices of S).
- We will condition on the **neighbour** of c_i to setup the required recurrence relationship.

DP: Defining Recurrence Relationship

- Let's setup the recurrence relationship.
- $OPT[c_i, S]$ contains the longest path starting from $c_i, c_i \in S$ and uses vertices of S (not necessarily all vertices of S).
- We will condition on the **neighbour** of c_i to setup the required recurrence relationship.

Visualizing Recurrence: Conditioning on Neighbors



S

DP: Defining Recurrence

Recurrence Relationship

$$OPT[c_i, S] = \max_{\forall c_j \in S - \{c_i\}} \{0, w(c_i, c_j) + OPT[c_j, S - \{c_i\}]\}$$
$$where\ c_j \in S - \{c_i\}$$

- Base case: $OPT[c_i, S] = 0$, where $S = \{c_i\}$ i.e. $OPT[c_i, \{c_i\}] = 0$
- Finally, the Longest Path is found by using:

$$LongestPath(G) = \max_{c_i \in V} \{OPT[c_i, V]\} \quad (2)$$

DP: Defining Recurrence

Recurrence Relationship

$$OPT[c_i, S] = \max_{\forall c_j \in S - \{c_i\}} \{0, w(c_i, c_j) + OPT[c_j, S - \{c_i\}]\}$$
$$where\ c_j \in S - \{c_i\}$$

- Base case: $OPT[c_i, S] = 0$, where $S = \{c_i\}$ i.e. $OPT[c_i, \{c_i\}] = 0$
- Finally, the Longest Path is found by using:

$$LongestPath(G) = \max_{c_i \in V} \{OPT[c_i, V]\} \tag{2}$$

DP: Defining Recurrence

Recurrence Relationship

$$OPT[c_i, S] = \max_{\forall c_j \in S - \{c_i\}} \{0, w(c_i, c_j) + OPT[c_j, S - \{c_i\}]\}$$
$$where\ c_j \in S - \{c_i\}$$

- Base case: $OPT[c_i, S] = 0$, where $S = \{c_i\}$ i.e. $OPT[c_i, \{c_i\}] = 0$
- Finally, the Longest Path is found by using:

$$LongestPath(G) = \max_{c_i \in V}\{OPT[c_i, V]\} \quad (2)$$

DP: Finalizing Recurrence

Recurrence Relationship (Detailed)

$$OPT[c_i, S] = \begin{cases} 0 & , S = \{c_i\} \\ \max\limits_{\forall c_j \in S - \{c_i\}} \{0, (w(c_i, c_j) + OPT[c_j, S - \{c_i\}])\} & , \text{else} \end{cases}$$

Algorithm 1 Dynamic Programming for Longest Path

Inputs: A Directed Graph $G(V, E)$
Output: Total Maximum Weight of Longest Path of Graph $G(V, E)$

```
foreach v ∈ V do
    OPT[v, {v}] = 0 //Base Case
end
for i ← 1 to |V| do
    //Normal case, Recurse on each subset
    foreach S ⊆ V do
        //Explicit 0 handles both negative weights and no neighbors case
```

$$OPT[c_i, S] \leftarrow \max_{\forall c_j \in S - \{c_i\}} \{0, w(c_i, c_j) + OPT[c_j, S - \{c_i\}]\}$$

```
    end
end
```

return $max_{c_i \in V}\{OPT[c_i, V]\}$

Time Complexity of DP Algorithm

- We need total number of sub-problems and work done per sub-problem.

Time Complexity of DP Algorithm

- We need total number of sub-problems and work done per sub-problem.
- Per vertex k, we need to compute for each subset $\binom{n}{k}$
- Thus, total number of sub-problems of size k is $k\binom{n}{k}$

Time Complexity of DP Algorithm

- We need total number of sub-problems and work done per sub-problem.
- Per vertex k, we need to compute for each subset $\binom{n}{k}$
- Thus, total number of sub-problems of size k is $k\binom{n}{k}$
- To solve a sub-problem, we just need to observe vertices for the longest path.
- Thus, sub-problem of size k is solvable in $O(k)$ time.

Time Complexity of DP Algorithm

- We need total number of sub-problems and work done per sub-problem.
- Per vertex k, we need to compute for each subset $\binom{n}{k}$
- Thus, total number of sub-problems of size k is $k\binom{n}{k}$
- To solve a sub-problem, we just need to observe vertices for the longest path.
- Thus, sub-problem of size k is solvable in $O(k)$ time.
- Therefore, total running time of DP algorithm is: $O(\sum_{k=1}^{n} k^2\binom{n}{k})$

Time Complexity of DP algorithm

- To find $O(\sum_{k=1}^{n} k^2 \binom{n}{k}) = O(\sum_{k=0}^{n} k^2 \binom{n}{k})$, we need geometric series.

Time Complexity of DP algorithm

- To find $O(\sum_{k=1}^{n} k^2 \binom{n}{k}) = O(\sum_{k=0}^{n} k^2 \binom{n}{k})$, we need geometric series.
- We know, $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$

Time Complexity of DP algorithm

- To find $O(\sum_{k=1}^{n} k^2 \binom{n}{k}) = O(\sum_{k=0}^{n} k^2 \binom{n}{k})$, we need geometric series.
- We know, $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$
- Differentiating once, $n(1+x)^{n-1} = \sum_{k=0}^{n} k \binom{n}{k} x^{k-1}$
- Setting $x = 1$, $n\, 2^{n-1} = \sum_{k=0}^{n} k \binom{n}{k}$ .... (I)

Time Complexity of DP algorithm

- To find $O(\sum_{k=1}^{n} k^2 \binom{n}{k}) = O(\sum_{k=0}^{n} k^2 \binom{n}{k})$, we need geometric series.
- We know, $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$
- Differentiating once, $n(1+x)^{n-1} = \sum_{k=0}^{n} k \binom{n}{k} x^{k-1}$
- Setting $x = 1$, $n\,2^{n-1} = \sum_{k=0}^{n} k \binom{n}{k}$ .... (I)
- Differentiating again, $n(n-1)(1+x)^{n-2} = \sum_{k=0}^{n} k(k-1) \binom{n}{k} x^{k-2}$
- Setting $x = 1$, $n\,(n-1)\,2^{n-2} = \sum_{k=0}^{n} k(k-1) \binom{n}{k}$ .... (II)

Time Complexity of DP algorithm

- To find $O(\sum_{k=1}^{n} k^2 \binom{n}{k}) = O(\sum_{k=0}^{n} k^2 \binom{n}{k})$, we need geometric series.
- We know, $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$
- Differentiating once, $n(1+x)^{n-1} = \sum_{k=0}^{n} k \binom{n}{k} x^{k-1}$
- Setting $x = 1$, $n\,2^{n-1} = \sum_{k=0}^{n} k \binom{n}{k}$ .... (I)
- Differentiating again, $n(n-1)(1+x)^{n-2} = \sum_{k=0}^{n} k(k-1) \binom{n}{k} x^{k-2}$
- Setting $x = 1$, $n\,(n-1)\,2^{n-2} = \sum_{k=0}^{n} k(k-1) \binom{n}{k}$ .... (II)
- Adding (I) and (II), we get
  $O(\sum_{k=0}^{n} k^2 \binom{n}{k}) = O(n(n+1)2^{n-2}) = O(n^2\, 2^n) = O^*(2^n)$

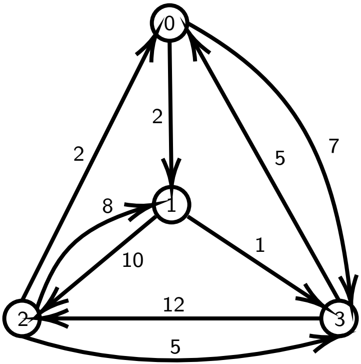
Space Complexity of DP Algorithm

- OPT table needs to have one entry for each subset $S \subseteq V$ and each vertex $c_i \ \forall_{i \in |V|}$.

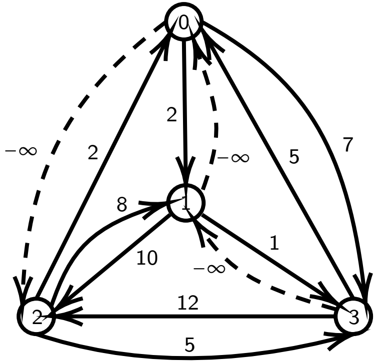
Space Complexity of DP Algorithm

- OPT table needs to have one entry for each subset $S \subseteq V$ and each vertex $c_i \ \forall_{i \in |V|}$.
- We can have $O(2^n)$ possible subsets and $O(n)$ possible vertices.
- Hence, space complexity of DP algorithm is $\Omega(n\,2^n)$.

Dynamic Programming Simulation



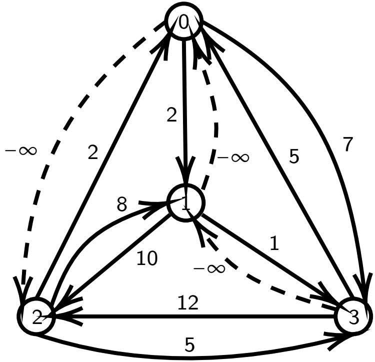
Dynamic Programming Simulation



nodes	0	1	2	3
0	0	2	$-\infty$	7
1	$-\infty$	0	10	1
2	4	8	0	5
3	5	$-\infty$	12	0

Table: Pairwise distance table

Dynamic Programming Simulation



nodes	0	1	2	3
0	0	2	$-\infty$	7
1	$-\infty$	0	10	1
2	4	8	0	5
3	5	$-\infty$	12	0

Table: Pairwise distance table

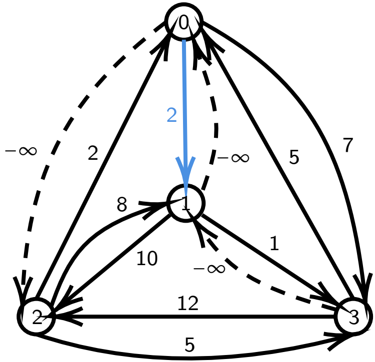
$$subsets : \phi,\ \{0\},\ \{1\},\ \{2\},\ \{3\}$$

$$\{0,1\},\ \{0,2\},\ \{0,3\},\ \{1,2\},\ \{1,3\},\ \{2,3\}$$

$$\{0,1,2\},\ \{0,1,3\},\ \{0,2,3\},\ \{1,2,3\}$$

$$\{0,1,2,3\}$$

Dynamic Programming Simulation



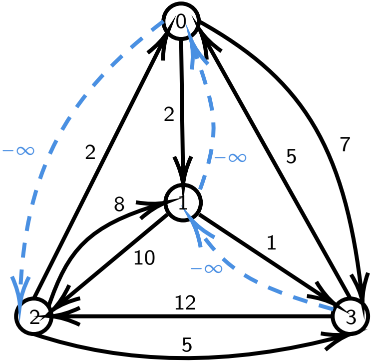
nodes	0	1	2	3
0	0	2	$-\infty$	7
1	$-\infty$	0	10	1
2	4	8	0	5
3	5	$-\infty$	12	0

Table: Pairwise distance table

$$OPT[c_i, S] = max(0, OPT[c_j, S - \{c_i\}] + d(c_i, c_j)\ \forall c_j \in S - \{c_i\})$$

$$OPT[0, \{1\}] = 2$$

Dynamic Programming Simulation



nodes	0	1	2	3
0	0	2	$-\infty$	7
1	$-\infty$	0	10	1
2	4	8	0	5
3	5	$-\infty$	12	0

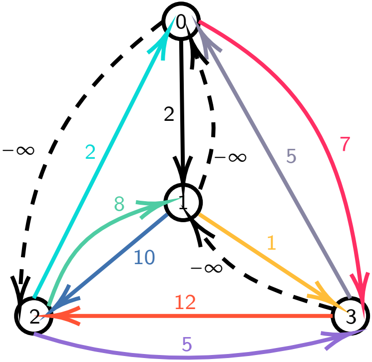
Table: Pairwise distance table

$$OPT[c_i, S] = max(0, OPT[c_j, S - \{c_i\}] + d(c_i, c_j) \ \forall c_j \in S - \{c_i\})$$

$$OPT[0, \{1\}] = 2$$

$$OPT[0, \{2\}] = OPT[1, \{0\}] = OPT[3, \{1\}] = 0$$

Dynamic Programming Simulation



$$OPT[0,\{1\}] = 2$$

$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$

$$OPT[0,\{3\}] = 7$$

$$OPT[2,\{0\}] = 2$$

$$OPT[3,\{0\}] = 5$$

$$OPT[1,\{2\}] = 10$$

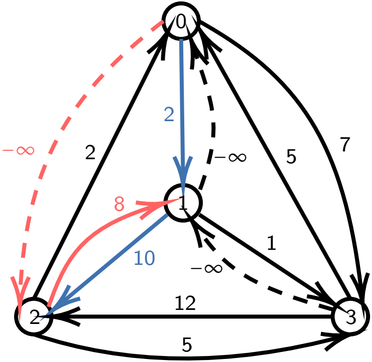
$$OPT[1,\{3\}] = 1$$

$$OPT[2,\{1\}] = 8$$

$$OPT[2,\{3\}] = 5$$

$$OPT[3,\{2\}] = 12$$

Dynamic Programming Simulation



$$OPT[0,\{1\}] = 2$$

$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$

$$OPT[0,\{3\}] = 7$$

$$OPT[2,\{0\}] = 2$$

$$OPT[3,\{0\}] = OPT[2,\{3\}] = 5$$

$$OPT[1,\{2\}] = 10$$

$$OPT[1,\{3\}] = 1$$

$$OPT[2,\{1\}] = 8$$

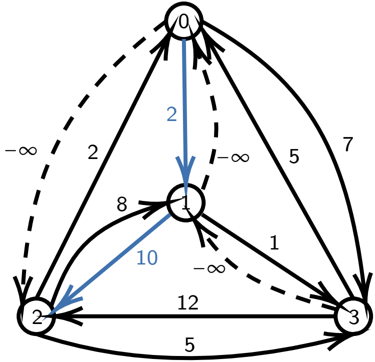
$$OPT[3,\{2\}] = 12$$

Cost Calculation

$$OPT[0,\{1,2\}] = max(0, OPT[2,\{1\}] + d(0,2), OPT[1,\{2\}] + d(0,1))$$

$$OPT[0,\{1,2\}] = max(0, 8 - \infty, 10 + 2)$$

Dynamic Programming Simulation

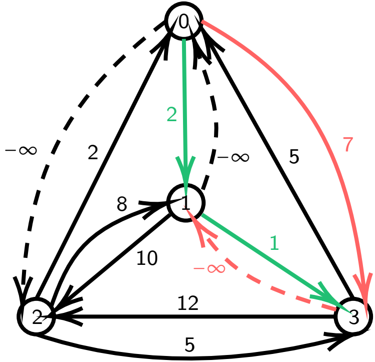


$$OPT[0,\{1\}] = 2$$
$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$
$$OPT[0,\{3\}] = 7$$
$$OPT[2,\{0\}] = 2$$
$$OPT[3,\{0\}] = OPT[2,\{3\}] = 5$$
$$OPT[1,\{2\}] = 10$$
$$OPT[1,\{3\}] = 1$$
$$OPT[2,\{1\}] = 8$$
$$OPT[3,\{2\}] = 12$$

$$Cost\ Calculation$$

$$OPT[0,\{1,2\}] = max(0, OPT[2,\{1\}] + d(0,2), OPT[1,\{2\}] + d(0,1))$$
$$OPT[0,\{1,2\}] = max(0, 8 - \infty, 10 + 2)$$
$$OPT[0,\{1,2\}] = 12$$
$$Path : 0 \rightarrow 1 \rightarrow 2$$

Dynamic Programming Simulation

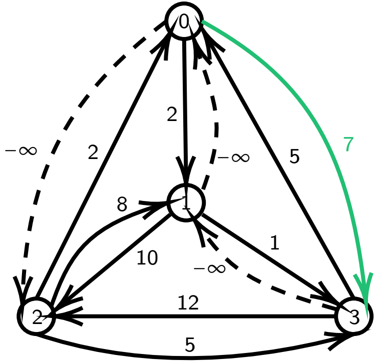


$$OPT[0, \{1\}] = 2$$
$$OPT[0, \{2\}] = OPT[1, \{0\}] = OPT[3, \{1\}] = 0$$
$$OPT[0, \{3\}] = 7$$
$$OPT[2, \{0\}] = 2$$
$$OPT[3, \{0\}] = OPT[2, \{3\}] = 5$$
$$OPT[1, \{2\}] = 10$$
$$OPT[1, \{3\}] = 1$$
$$OPT[2, \{1\}] = 8$$
$$OPT[3, \{2\}] = 12$$

Cost Calculation

$$OPT[0, \{1, 3\}] = max(0, OPT[3, \{1\}] + d(0, 3), OPT[1, \{3\}] + d(0, 1))$$
$$OPT[0, \{1, 3\}] = max(0, 0 + 7, 1 + 2)$$

Dynamic Programming Simulation



$$OPT[0,\{1\}] = 2$$

$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$

$$OPT[0,\{3\}] = 7$$

$$OPT[2,\{0\}] = 2$$

$$OPT[3,\{0\}] = OPT[2,\{3\}] = 5$$

$$OPT[1,\{2\}] = 10$$

$$OPT[1,\{3\}] = 1$$

$$OPT[2,\{1\}] = 8$$

$$OPT[3,\{2\}] = 12$$

$$Cost\ Calculation$$

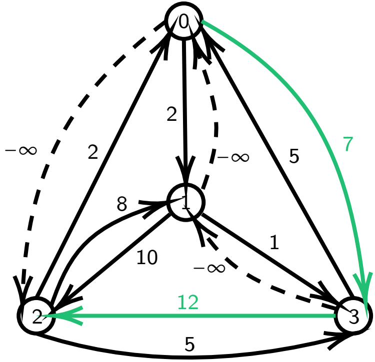
$$OPT[0,\{1,3\}] = max(0, OPT[3,\{1\}] + d(0,3), OPT[1,\{3\}] + d(0,1))$$

$$OPT[0,\{1,3\}] = max(0, 0+7, 1+2)$$

$$OPT[0,\{1,3\}] = 7$$

$$Path : 0 \rightarrow 3$$

Dynamic Programming Simulation



$$OPT[0,\{1\}] = 2$$

$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$

$$OPT[0,\{3\}] = 7$$

$$OPT[2,\{0\}] = 2$$

$$OPT[3,\{0\}] = OPT[2,\{3\}] = 5$$

$$OPT[1,\{2\}] = 10$$

$$OPT[1,\{3\}] = 1$$

$$OPT[2,\{1\}] = 8$$

$$OPT[3,\{2\}] = 12$$

$$Cost\ Calculation$$

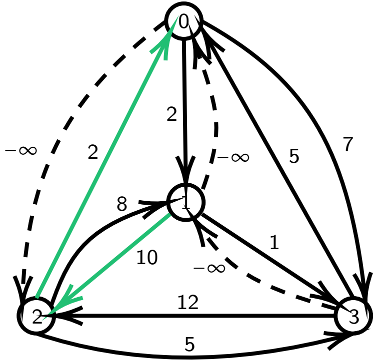
$$OPT[0,\{2,3\}] = max(0, OPT[3,\{2\}] + d(0,3), OPT[2,\{3\}] + d(0,2))$$

$$OPT[0,\{2,3\}] = max(0, 12 + 7, 5 - \infty)$$

$$OPT[0,\{2,3\}] = 19$$

$$Path : 0 \to 3 \to 2$$

Dynamic Programming Simulation



$$OPT[0,\{1\}] = 2$$

$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$

$$OPT[0,\{3\}] = 7$$

$$OPT[2,\{0\}] = 2$$

$$OPT[3,\{0\}] = OPT[2,\{3\}] = 5$$

$$OPT[1,\{2\}] = 10$$

$$OPT[1,\{3\}] = 1$$

$$OPT[2,\{1\}] = 8$$

$$OPT[3,\{2\}] = 12$$

$$Cost\ Calculation$$

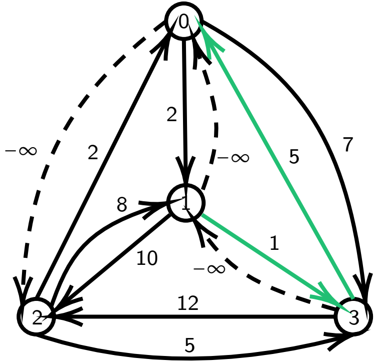
$$OPT[1,\{0,2\}] = max(0, OPT[0,\{2\}] + d(1,0), OPT[2,\{0\}] + d(1,2))$$

$$OPT[1,\{0,2\}] = max(0, 0 - \infty, 4 + 10)$$

$$OPT[1,\{0,2\}] = 14$$

$$Path : 1 \rightarrow 2 \rightarrow 0$$

Dynamic Programming Simulation



$$OPT[0,\{1\}] = 2$$
$$OPT[0,\{2\}] = OPT[1,\{0\}] = OPT[3,\{1\}] = 0$$
$$OPT[0,\{3\}] = 7$$
$$OPT[2,\{0\}] = 2$$
$$OPT[3,\{0\}] = OPT[2,\{3\}] = 5$$
$$OPT[1,\{2\}] = 10$$
$$OPT[1,\{3\}] = 1$$
$$OPT[2,\{1\}] = 8$$
$$OPT[3,\{2\}] = 12$$

Cost Calculation

$$OPT[1,\{0,3\}] = max(0, OPT[0,\{3\}] + d(1,0), OPT[3,\{0\}] + d(1,3))$$
$$OPT[1,\{0,3\}] = max(0, 7 - \infty, 5 + 1)$$
$$OPT[1,\{0,3\}] = 6$$
$$Path : 1 \rightarrow 3 \rightarrow 0$$

Dynamic Programming Simulation

$$\begin{gathered}
Cost\ Calculation \\
OPT[1,\{2,3\}] = max(0, OPT[2,\{3\}] + d(1,2), OPT[3,\{2\}] + d(1,3)) \\
OPT[1,\{2,3\}] = max(0, 5 + 10, 12 + 1) \\
OPT[1,\{2,3\}] = 15 \\
Path : 1 \rightarrow 2 \rightarrow 3
\end{gathered}$$

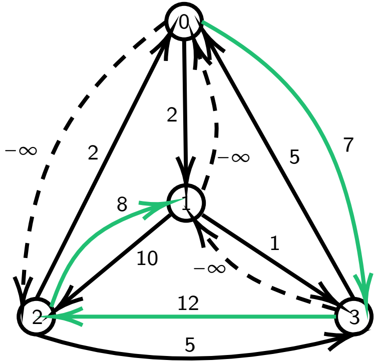
$$\begin{gathered}
Cost\ Calculation \\
OPT[2,\{0,1\}] = max(0, OPT[0,\{1\}] + d(2,0), OPT[1,\{0\}] + d(2,1)) \\
OPT[2,\{0,1\}] = max(0, 2 + 4, 0 + 8) \\
OPT[2,\{0,1\}] = 8 \\
Path : 2 \rightarrow 1
\end{gathered}$$

$$\begin{gathered}
Cost\ Calculation \\
OPT[2,\{0,3\}] = max(0, OPT[0,\{3\}] + d(2,0), OPT[3,\{0\}] + d(2,3)) \\
OPT[2,\{0,3\}] = max(0, 7 + 4, 5 + 5) \\
OPT[2,\{0,3\}] = 11 \\
Path : 2 \rightarrow 0 \rightarrow 3
\end{gathered}$$

Dynamic Programming Simulation

$$Cost\ Calculation$$
$$OPT[2,\{1,3\}] = max(0, OPT[1,\{3\}] + d(2,1), OPT[3,\{1\}] + d(2,3))$$
$$OPT[2,\{1,3\}] = max(0, 1+8, 0+5)$$
$$OPT[2,\{1,3\}] = 9$$
$$Path: 2 \to 1 \to 3$$
$$Cost\ Calculation$$
$$OPT[3,\{0,1\}] = max(0, OPT[0,\{1\}] + d(3,0), OPT[1,\{0\}] + d(3,1))$$
$$OPT[3,\{0,1\}] = max(0, 2+5, 0-\infty)$$
$$OPT[3,\{0,1\}] = 7$$
$$Path: 3 \to 0 \to 1$$
$$Cost\ Calculation$$
$$OPT[3,\{0,2\}] = max(0, OPT[0,\{2\}] + d(3,0), OPT[2,\{0\}] + d(3,2))$$
$$OPT[3,\{0,2\}] = max(0, 0+5, 4+12)$$
$$OPT[3,\{0,2\}] = 16$$
$$Path: 3 \to 2 \to 0$$
$$Cost\ Calculation$$
$$OPT[3,\{1,2\}] = max(0, OPT[1,\{2\}] + d(3,1), OPT[2,\{1\}] + d(3,2))$$
$$OPT[3,\{1,2\}] = max(0, 10-\infty, 8+12)$$
$$OPT[3,\{1,2\}] = 20$$
$$Path: 3 \to 2 \to 1$$

Dynamic Programming Simulation



$OPT[0, \{1,2\}] = 12$

$OPT[0, \{1,3\}] = 7$

$OPT[0, \{2,3\}] = 19$

$OPT[1, \{0,2\}] = 14$

$OPT[1, \{0,3\}] = 6$

$OPT[1, \{2,3\}] = 15$

$OPT[2, \{0,1\}] = 8$

$OPT[2, \{0,3\}] = 11$

$OPT[2, \{1,3\}] = 9$

$OPT[3, \{0,1\}] = 7$

$OPT[3, \{0,2\}] = 16$

$OPT[3, \{1,2\}] = 20$

$Cost\ Calculation$

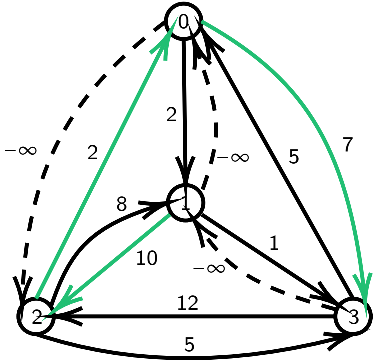
$$OPT[0, \{1,2,3\}] = max(0, OPT[1, \{2,3\}] + d(0,1), OPT[2, \{1,3\}] + d(0,2), OPT[3, \{1,2\}] + d(0,3))$$

$$OPT[0, \{1,2,3\}] = max(0, 15 + 2, 9 - \infty, 20 + 7)$$

$$OPT[0, \{1,2,3\}] = 27$$

$$Path : 0 \rightarrow 3 \rightarrow 2 \rightarrow 1$$

Dynamic Programming Simulation



$OPT[0, \{1, 2\}] = 12$

$OPT[0, \{1, 3\}] = 7$

$OPT[0, \{2, 3\}] = 19$

$OPT[1, \{0, 2\}] = 14$

$OPT[1, \{0, 3\}] = 6$

$OPT[1, \{2, 3\}] = 15$

$OPT[2, \{0, 1\}] = 8$

$OPT[2, \{0, 3\}] = 11$

$OPT[2, \{1, 3\}] = 9$

$OPT[3, \{0, 1\}] = 7$

$OPT[3, \{0, 2\}] = 16$

$OPT[3, \{1, 2\}] = 20$

$Cost\ Calculation$

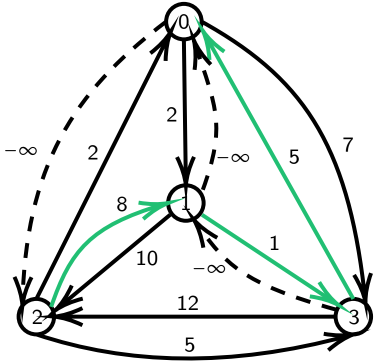
$$OPT[1, \{0, 2, 3\}] = max(0, OPT[0, \{2, 3\}] + d(1, 0), OPT[2, \{0, 3\}] + d(1, 2), OPT[3, \{0, 2\}] + d(1, 3))$$

$$OPT[1, \{0, 2, 3\}] = max(0, 19 - \infty, 11 + 10, 16 + 1)$$

$$OPT[1, \{0, 2, 3\}] = 21$$

$$Path : 1 \to 2 \to 0 \to 3$$

Dynamic Programming Simulation



$OPT[0,\{1,2\}] = 12$

$OPT[0,\{1,3\}] = 7$

$OPT[0,\{2,3\}] = 19$

$OPT[1,\{0,2\}] = 14$

$OPT[1,\{0,3\}] = 6$

$OPT[1,\{2,3\}] = 15$

$OPT[2,\{0,1\}] = 8$

$OPT[2,\{0,3\}] = 11$

$OPT[2,\{1,3\}] = 9$

$OPT[3,\{0,1\}] = 7$

$OPT[3,\{0,2\}] = 16$

$OPT[3,\{1,2\}] = 20$

$Cost\ Calculation$

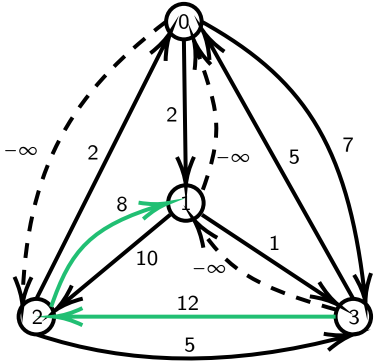
$$OPT[2,\{0,1,3\}] = max(0, OPT[1,\{0,3\}] + d(2,1), OPT[0,\{1,3\}] + d(2,0), OPT[3,\{1,0\}] + d(2,3))$$

$$OPT[2,\{0,1,3\}] = max(0, 7+4, 6+8, 7+5)$$

$$OPT[2,\{0,1,3\}] = 14$$

$$Path: 2 \to 1 \to 3 \to 0$$

Dynamic Programming Simulation



$OPT[0, \{1, 2\}] = 12$

$OPT[0, \{1, 3\}] = 7$

$OPT[0, \{2, 3\}] = 19$

$OPT[1, \{0, 2\}] = 14$

$OPT[1, \{0, 3\}] = 6$

$OPT[1, \{2, 3\}] = 15$

$OPT[2, \{0, 1\}] = 8$

$OPT[2, \{0, 3\}] = 11$

$OPT[2, \{1, 3\}] = 9$

$OPT[3, \{0, 1\}] = 7$

$OPT[3, \{0, 2\}] = 16$

$OPT[3, \{1, 2\}] = 20$

$Cost\ Calculation$

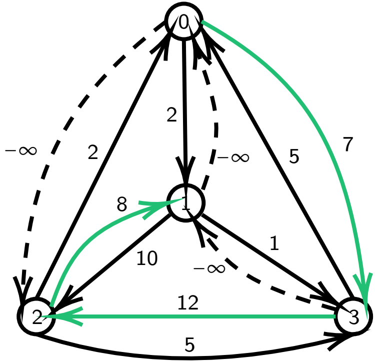
$$OPT[3, \{0, 1, 2\}] = max(0, OPT[1, \{0, 2\}] + d(3, 1), OPT[0, \{1, 2\}] + d(3, 0), OPT[2, \{1, 0\}] + d(3, 2))$$

$$OPT[3, \{0, 1, 2\}] = max(0, 12 + 5, 14 - \infty, 8 + 12)$$

$$OPT[3, \{0, 1, 2\}] = 20$$

$$Path : 3 \rightarrow 2 \rightarrow 1$$

Dynamic Programming Simulation



So, path: $0 \to 3 \to 2 \to 1$ has the maximum cost of 27. Thus, this is the longest path of this graph.

$$Cost\ Calculation$$

$$OPT[0, \{1,2,3\}] = max(0, OPT[1, \{2,3\}] + d(0,1), OPT[2, \{1,3\}] + d(0,2), OPT[3, \{1,2\}] + d(0,3))$$

$$OPT[0, \{1,2,3\}] = max(0, 15 + 2, 9 - \infty, 20 + 7)$$

$$OPT[0, \{1,2,3\}] = 27$$

$$Path : 0 \to 3 \to 2 \to 1$$

Polynomial time variations

Special graphs	Complexity	Comments
Tree	Linear	[2]
Cacti Graph	$O(n^2)$	[3]
Bipartite Permutation Graph	Linear	[4]
Directed Acyclic Graph	Linear	Dynamic approach
Interval Graph	$O(n^4)$	Dynamic approach [5]
Circular Arc Graph	$O(n^4)$	Dynamic approach [6]
Co-compatibility Graph	$O(n^7)$	From Hasse diagram [7]

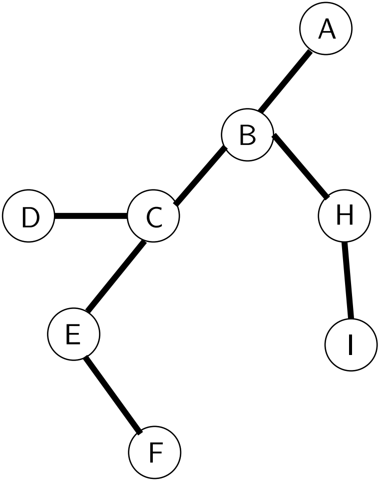
Longest Path in tree

- A tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph.
- The idea is if we start BFS from any node x and find a node with the longest distance from x, it must be an endpoint of the longest path. It can be proved using contradiction. (Bulterman et al. [2], Uehara et al. [3]).
- We can use two BFS. First BFS to find an endpoint of the longest path and second BFS from this endpoint to find the actual longest path.

Longest Path in tree

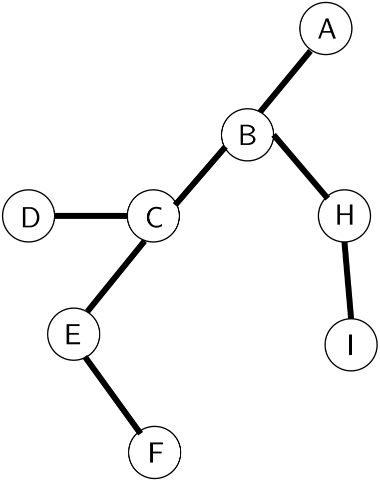
- A tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph.
- The idea is if we start BFS from any node x and find a node with the longest distance from x, it must be an endpoint of the longest path. It can be proved using contradiction. (Bulterman et al. [2], Uehara et al. [3]).
- We can use two BFS. First BFS to find an endpoint of the longest path and second BFS from this endpoint to find the actual longest path.

Polynomial Algorithm



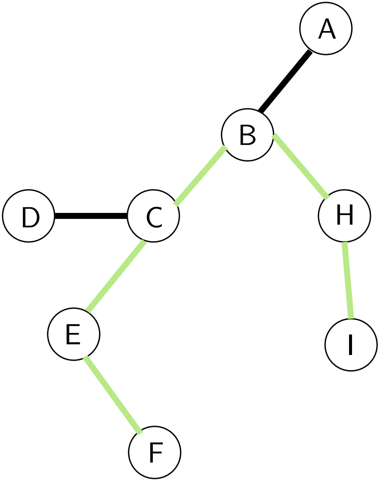
- This is a simple tree having 8 nodes.

Polynomial Algorithm



- This is a simple tree having 8 nodes.
- What is the longest path here?

Polynomial Algorithm



- The longest path is clearly F->E->C->B->H->I having length 5.
- Now lets jump into the algorithm.

Polynomial Algorithm

Algorithm 2 BFS method

Input: *node u*
Output: *Returns farthest node and its distance from node u*
Function `BFS`(*node u*)**:**

    $dist[V] \longleftarrow -1$; ▷ mark all distance with -1
    $queue.push(u)$;
    $dist[u] \longleftarrow 0$; ▷ distance of u from u will be 0
    while *queue is not empty* **do**
        $node\ t = queue.popFront()$;
        foreach $node\ v \in adj[t]$ **do**
            if $dist[v] = -1$ **then**
                $queue.push(v)$;
                $dist[v] = dist[t] + 1$;
            end
        end
    end
    return $node, distance \in max(dist[V])$
End Function

Polynomial Algorithm

Algorithm 3 LongestPath Method

Output: *Returns longest path of a given tree*

Function `LongestPathLength()`**:**

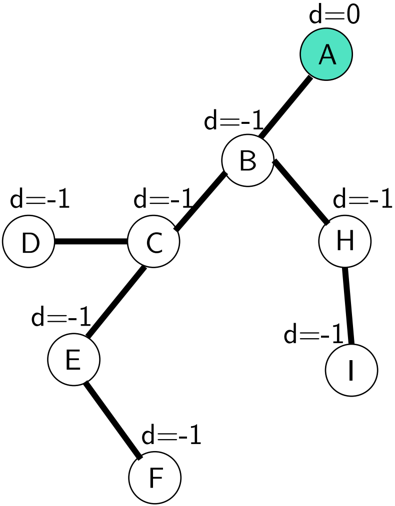
$node\ s \longleftarrow BFS(random\ node)$; ▷ first bfs to find one end point of longest path

$node\ t \longleftarrow BFS(s.node)$; ▷ second bfs to find actual longest path

return $node\ t.distance$

End Function

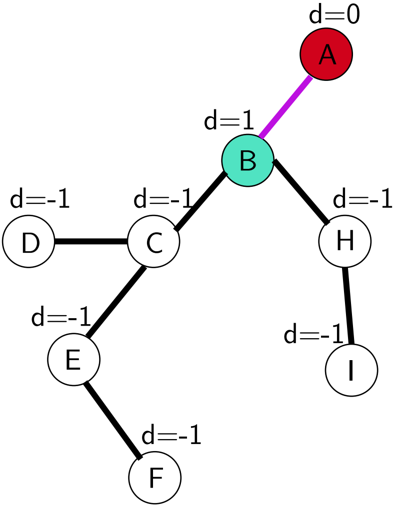
Polynomial Algorithm



Running $BFS(NodeA)$

$dist[V] \longleftarrow -1;$
$queue.push(u);$
$dist[u] \longleftarrow 0;$

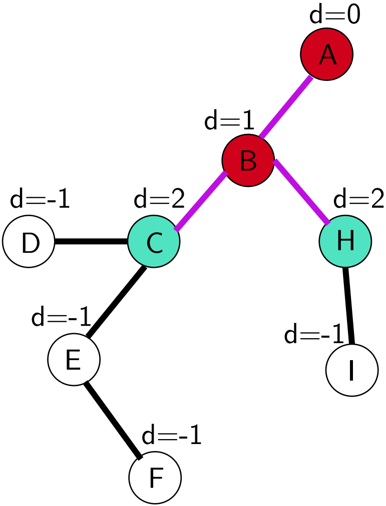
Polynomial Algorithm



Running $BFS(NodeA)$

while $queue\ is\ not\ empty$ **do**
    $node\ t = queue.popFront()$;
    foreach $node\ v \in adj[t]$ **do**
        if $dist[v] = -1$ **then**
            $queue.push(v)$;
            $dist[v] = dist[t] + 1$;
        end
    end
end

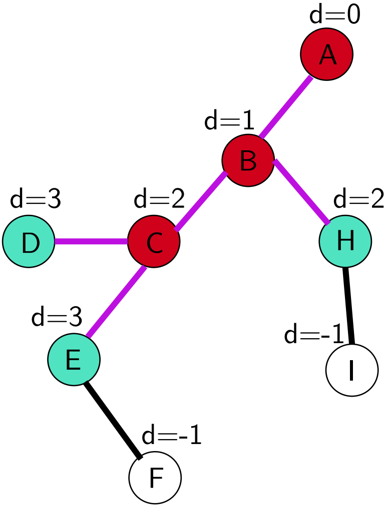
Polynomial Algorithm



Running $BFS(NodeA)$

while $queue\ is\ not\ empty$ **do**
  $node\ t = queue.popFront()$;
  foreach $node\ v \in adj[t]$ **do**
    if $dist[v] = -1$ **then**
      $queue.push(v)$;
      $dist[v] = dist[t] + 1$;
    end
  end
end

Polynomial Algorithm

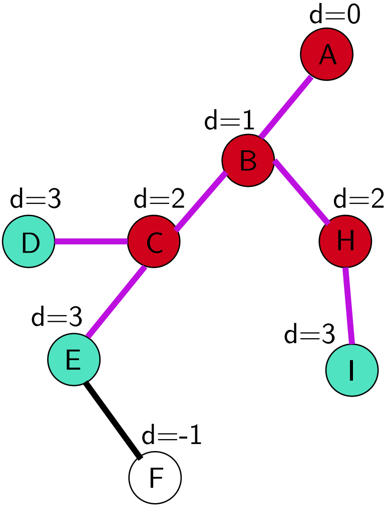


Running $BFS(NodeA)$

---

while $queue\ is\ not\ empty$ **do**
    $node\ t = queue.popFront();$
    foreach $node\ v \in adj[t]$ **do**
        if $dist[v] = -1$ **then**
            $queue.push(v);$
            $dist[v] = dist[t] + 1;$
        end
    end
end

Polynomial Algorithm

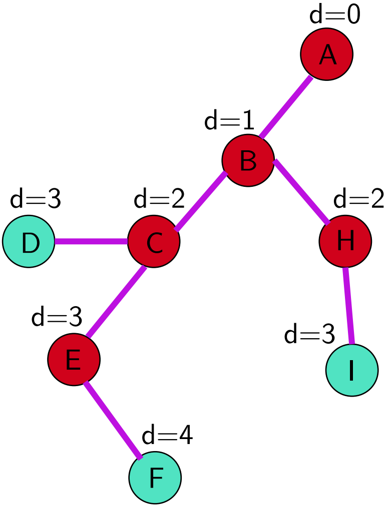


Running $BFS(NodeA)$

---

while *queue is not empty* **do**
  $node\ t = queue.popFront()$;
  foreach $node\ v \in adj[t]$ **do**
    if $dist[v] = -1$ **then**
      $queue.push(v)$;
      $dist[v] = dist[t] + 1$;
    end
  end
end

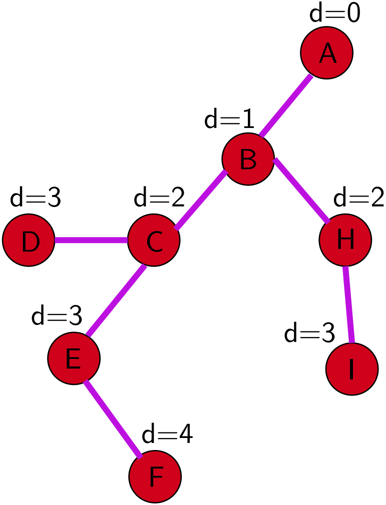
Polynomial Algorithm



Running $BFS(NodeA)$

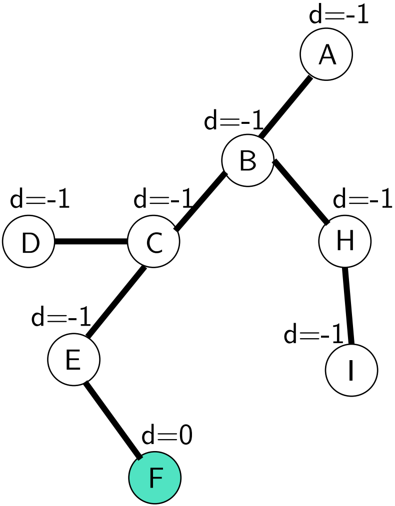
while *queue is not empty* **do**
  $node\ t = queue.popFront()$;
  foreach $node\ v \in adj[t]$ **do**
    if $dist[v] = -1$ **then**
      $queue.push(v)$;
      $dist[v] = dist[t] + 1$;
    end
  end
end

Polynomial Algorithm



- Now we need the node with highest distance.
- Node F has the highest distance with $d = 4$.
- Now we will run BFS with F as the starting node.

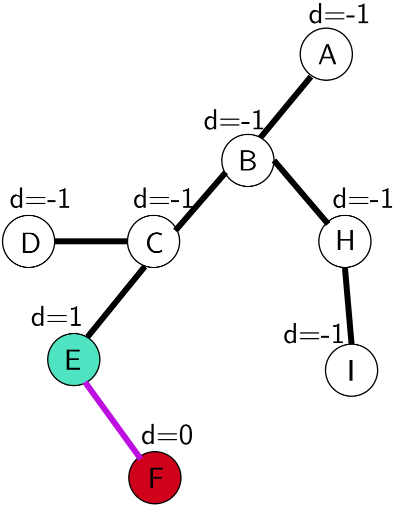
Polynomial Algorithm



Running $BFS(NodeF)$

$dist[V] \longleftarrow -1;$
$queue.push(u);$
$dist[u] \longleftarrow 0;$

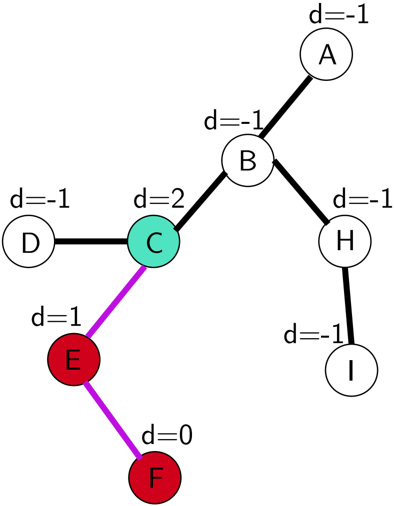
Polynomial Algorithm



Running $BFS(NodeF)$

while *queue is not empty* **do**
    $node\ t = queue.popFront()$;
    foreach $node\ v \in adj[t]$ **do**
        if $dist[v] = -1$ **then**
            $queue.push(v)$;
            $dist[v] = dist[t] + 1$;
        end
    end
end

Polynomial Algorithm

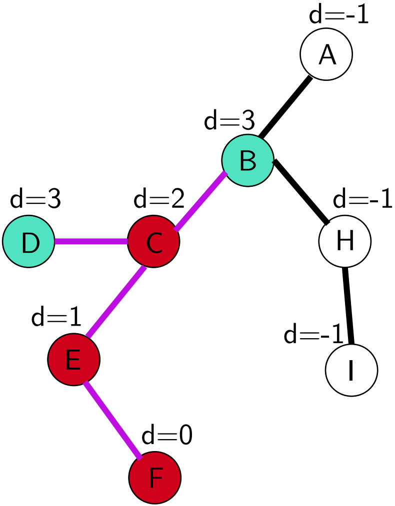


Running $BFS(NodeF)$

---

while *queue is not empty* **do**
    $node\ t = queue.popFront()$;
    foreach $node\ v \in adj[t]$ **do**
        if $dist[v] = -1$ **then**
            $queue.push(v)$;
            $dist[v] = dist[t] + 1$;
        end
    end
end

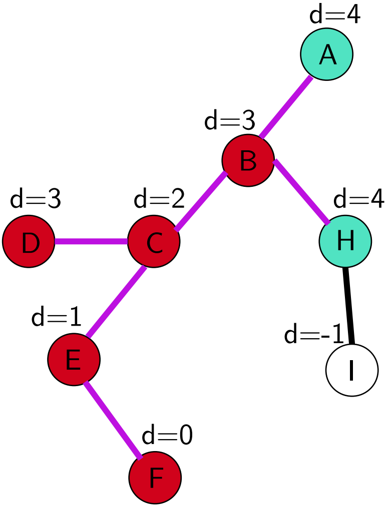
Polynomial Algorithm



Running $BFS(NodeF)$

while *queue is not empty* **do**
    node t = $queue.popFront()$;
    foreach *node* $v \in adj[t]$ **do**
        if $dist[v] = -1$ **then**
            $queue.push(v)$;
            $dist[v] = dist[t] + 1$;
        end
    end
end

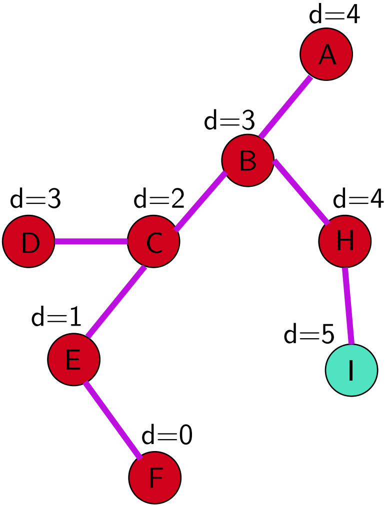
Polynomial Algorithm



Running $BFS(NodeF)$

while *queue is not empty* **do**
  $node\ t = queue.popFront()$;
  foreach $node\ v \in adj[t]$ **do**
    if $dist[v] = -1$ **then**
      $queue.push(v)$;
      $dist[v] = dist[t] + 1$;
    end
  end
end

Polynomial Algorithm

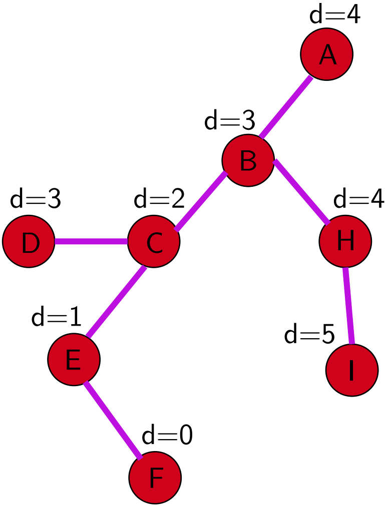


Running $BFS(NodeF)$

---

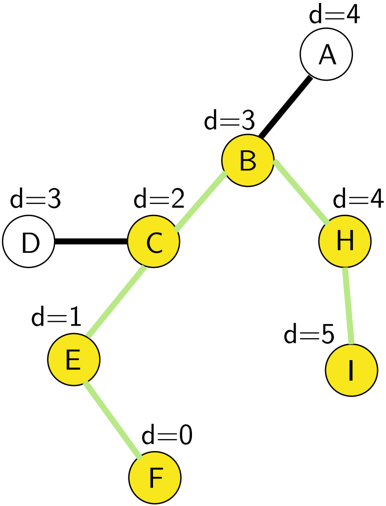
while $queue\ is\ not\ empty$ **do**
  $node\ t = queue.popFront();$
  foreach $node\ v \in adj[t]$ **do**
    if $dist[v] = -1$ **then**
      $queue.push(v);$
      $dist[v] = dist[t] + 1;$
    end
  end
end

Polynomial Algorithm



- Node I has the highest distance with $d = 5$.
- So Node F and I are the two endpoints of the longest path.

Polynomial Algorithm



- So backtracking from Node I, we get the path to F as I->H ->B->C->E->F
- This is the same path we saw before starting our algorithm.

Complexity analysis

- **Time Complexity:** This algorithm uses two BFS.One breadth first search takes $O(|V| + |E|)$ time where $|V|$ is the number of vertices and $|E|$ is the number of edges. $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$.
- **Space Complexity:** This algorithm uses a queue and for storing the path we will need another array of size of the number of vertices.So space complexity can be expressed as $O(|V|)$ where $|V|$ is the number of vertices.

References I

Michael Held and Richard M. Karp.
A dynamic programming approach to sequencing problems.
Journal of the Society for Industrial and Applied Mathematics, 10(1):196–210, 1962.

R.W. Bulterman, F.W. van der Sommen, G. Zwaan, T. Verhoeff, A.J.M. van Gasteren, and W.H.J. Feijen.
On computing a longest path in a tree.
Information Processing Letters, 81(2):93 – 96, 2002.

Ryuhei Uehara and Yushi Uno.
Efficient algorithms for the longest path problem.
In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation*, pages 871–883, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

References II

Ryuhei Uehara and Gabriel Valiente.
Linear structure of bipartite permutation graphs and the longest path problem.
Information Processing Letters, 103(2):71 – 77, 2007.

Kyriaki Ioannidou, George Mertzios, and Stavros Nikolopoulos.
The longest path problem is polynomial on interval graphs.
pages 403–414, 08 2009.

George B. Mertzios and Ivona Bezáková.
Computing and counting longest paths on circular-arc graphs in polynomial time.
Discrete Applied Mathematics, 164:383 – 399, 2014.
LAGOS'11: Sixth Latin American Algorithms, Graphs, and Optimization Symposium, Bariloche, Argentina — 2011.

References III

Kyriaki Ioannidou and Stavros D. Nikolopoulos.
The longest path problem is polynomial on cocomparability graphs.
Algorithmica, 65(1):177–205, Jan 2013.