CSCI 4470 Algorithms

Part VI Graph Algorithms Notes

- 20 Elementary Graph Algorithms
- 21 Minimum Spanning Trees
- **22 Single-Source Shortest Paths**
- 23 All-Pairs Shortest Paths
- 24 Maximum Flow
- 25 Matchings in Bipartite Graphs

Chapter 22: Single-Source Shortest Paths

Before Single-Source Shortest Paths

Graph Types and Algorithms:

1. **Unweighted Graphs:** BFS is applicable.

- If all weights are 1, BFS is efficient.

2. **Weighted Graphs:** Requires algorithms like Dijkstra or Bellman-Ford.

Examples and Variations:

1. Single-Destination Shortest Path:

1. **Single-Destination Shortest Path:** Reverse edges and solve as a single-source problem.

- i. **Application**: Useful when multiple sources share a common destination.
- ii. **Algorithmic Approach**: Reverse edges, solve as SSSP, and consider edge cases like negative cycles.

2. Single-Pair Shortest Path:

2. **Single-Pair Shortest Path:** Can be solved using single-source algorithms.

- Find the shortest path between a specific pair of vertices (u, v).

- **Approach**: Can be solved using single-source shortest path algorithms by considering u as the source.

3. All-Pair Shortest Path:

3. **All-Pair Shortest Path:** Run single-source algorithm for each vertex.

Key Terms:

- s**:** The source vertex from which all shortest paths are calculated.
- (s, u)**:** The shortest path from the source vertex s to another vertex u.
- (s, v)**:** The shortest path from the source vertex s to another vertex v.

Single-Source Shortest Path (SSSP)

Single-Source Shortest Path (SSSP): Focusing on weighted graphs, where the shortest path is not solely determined by the number of edges.

Goal: Find the shortest paths from a specific source vertex s to all other vertices in a directed graph $G = (V, E)$ with edge weights $w(u, v)$.

Given $G = (V, E)$ with weights $w(u, v)$ and a vertex s, find the shortest paths from s to all $v \in V$.

1. Subset $G' = (V', E')$: V' includes all vertices reachable from s.

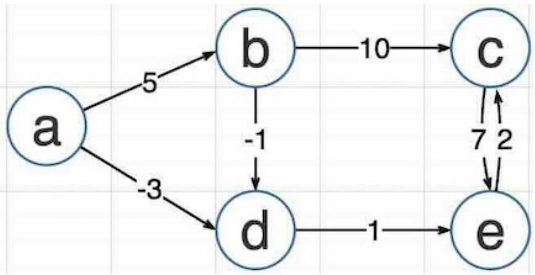
2. Shortest Paths: For each $v \in V'$, find the unique, simplest, and minimum weight path from s to v in G.

- A minimum weight path is referred to as a shortest path.

SSSP Example, Given $G(V, E)$ with $V = \{a, b, c, d, e\}$ and directed edges $E = \{(a, b, 5), (a, d, -3), (b, c, 10), (b, d, -1), (c, e, 7), (d, e, 1), (e, c, 2)\}$:

Vertices and Directed Edges:

- V: Set of vertices a, b, c, d, e.
- E: Set of directed edges with weights.



Single-Source Shortest Path (from a):

- s**:** Source vertex is a.
- (s, u)**:** Shortest path from a to u.
  - (s, u)**:** For $u = b$, path is $(a, b, 5)$.
- (s, v)**:** Shortest path from a to v.

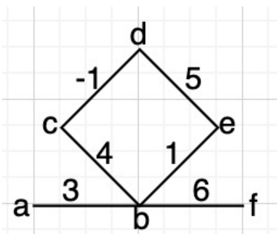
- (s, v): For $v = e$, path is $(a, d, -3) \to (d, e, 1)$, total weight -2.

Negative Weight Edges and Cycles:

- The graph G can have negative weight edges
- If graph G has negative weight edges but no negative weight cycles, the shortest path is well defined
- If graph G has any negative weight cycles, the shortest path is no longer well defined

example, the graph G, vertex={a, b, c, d, e, f}, edges={(a, b, 3), (b, c, 4), (b, e, 1)(c, d, -1), (d, e, 5), (b, f, 6)}

1. **Negative Weights:** Algorithms like Bellman-Ford can handle these, but negative cycles create problems.
2. **Cycles:** They make the shortest path undefined.

- 

- Can not have cycle in the shortest path, because make the shortest path undefined.
- Cycles also increase distance of the path.
- i.e., The shortest path of `d(a,f)= 3 + 6` , if there is a cycle as the diagram shown.
  the shortest path of `d(a,f)= 3 + 6 + 9` , where $9 = 4 + 5 + 1 + (-1)$

Negative Cycles:

The Simple Path in a graph is one that doesn't revisit any vertices. In the context of shortest paths, it ensures no cycles are included, making the path truly the shortest.

Dashed Algorithm: This algorithm calculates the shortest distance between a single source and all vertices.

- No Negative weights and cycles in the algorithm
- **Bellman-Ford Algorithm:** This algorithm can detect negative cycles and is used when graphs have negative weights.

Notations

Proof of Single-Source Shortest Path: Let $p =< v_0, v_1, ..., v_k >$ be any path in G:

- Denoted $v_0 \overset{p}{\rightsquigarrow} v_k$
- $w(p)$ is the weight of p, then $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

Let u and v be any two vertices in G:

- If there exists a path from u to v in G $\delta(u, v) = \min\{w(p)| \, u \overset{p}{\rightsquigarrow} v\}$
- Otherwise, $\delta(u, v) = \infty$

The graph G' has $w(p) = \delta(s, v)$ for every path p from s to v in V'

- G' will be a tree

Optimal Substructure

1. **Optimal Substructure:** Shortest paths contain shortest paths to intermediate vertices.

Lemma: Given a weighted directed graph $G = (V, E)$.
Let $p =< v_0, v_2, ..., v_k >$ be a shortest path from vertex v_0 to v_k and for any i, j such that $1 \leq i \leq j \leq k$.
Let v_i and v_j be the intermediate vertices. Let p_{ij} be the subpath of p from v_i to v_i. Then p_{ij} is a shortest path from v_i to v_j.

Proof: Decompose p into p_{0i}, p_{ij}, p_{jk}

$$p_{0\to k} = v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

Assume: p_{0k} is the shortest path from v_0 to v_k, p_{ij} is not a shortest path from v_i to v_j and let p_{ij} be a shorter path.

Derive a contradiction.

$$p'_{0\to k} = v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

$$w(p_{0k}) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$$

$$w(p'_{0k}) = w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$$

$$\Rightarrow w(p_{0k}) > w(p'_{0k})$$

2. **Examples:** Demonstrated with specific weights and paths in the graph.

PROPERTIES OF SHORTEST PATHS AND RELAXATION

Triangle Inequality:

- For edge (u, v), $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper Bound Property:

- $v.d$ (shortest path estimate) always upper bounds $\delta(s, v)$, remains unchanged once it equals $\delta(s, v)$.

No-path Property:

- If no path from s to v, then $v.d$ and $\delta(s, v) = \infty$.

Convergence Property:

- If $s \rightsquigarrow u \to v$ is a shortest path and $u.d = \delta(s, u)$ before relaxing (u, v), then $v.d = \delta(s, v)$ after.

Path Relaxation Property:

- If a path $p = \langle v_0, ..., v_k \rangle$ is shortest from v_0 to v_k and edges of p are relaxed in order, then $v_k.d = \delta(s, v_k)$.

Predecessor Subgraph Property:

- Once $v.d = \delta(s, v)$ for all v, a shortest-paths tree rooted at s forms.

> These properties are essential for understanding shortest path algorithms like Dijkstra's and Bellman-Ford, and are derived from the relaxation process.

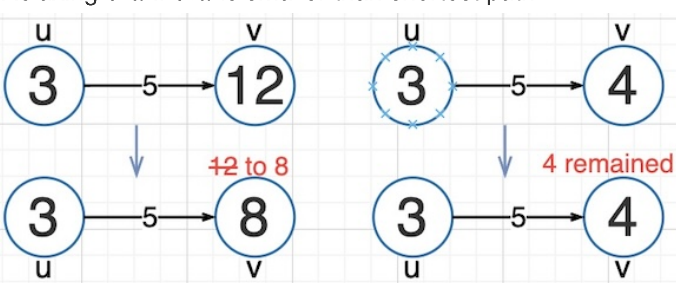
Algorithms

- **Bellman-Ford**, **SSSP in DAGs**, and **Dijkstra**, ***All these algorithms share two subroutines and rely on several properties.***

```
Initialize-Single-Source(G,s)
//initialize .d and .π
for each vertex v in V
     v.d = ∞
     v.π = NIL
s.d = 0
```

```
Relax(u,V,W)
// update v.d and v.π if shorter path exists
if v.d > u.d + w(u,v)
     v.d = u.d + w(u,v)
     v.π = u
```

- From u to v the shortest path is $3 + 5$
- Relaxing $v.d$ if $v.d$ is smaller than shortest path



The Bellman-Ford

1. Bellman-Ford Algorithm:

- Handles graphs with negative edge weights, detects negative cycles.
- **Complexity**: $O(V \cdot E)$ expensive due to repeated edge relaxations. Reports error if G
- Contains any negative weight cycles

Correctness of Bellman-Ford:

1. $v.d = \delta(s, v)$ for all $v \in V$.
2. Returns **TRUE** if no negative-weight cycles,
3. Returns **FALSE** otherwise

Key Steps:

- Initialize vertex distances to infinity, source distance to 0.
- Relax edges $V - 1$ times.
- Check for negative cycles.

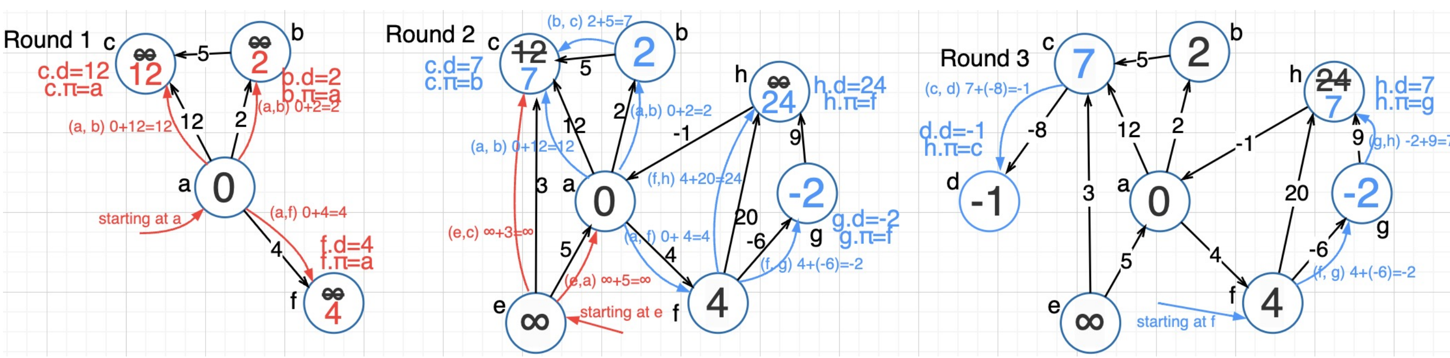
Properties:

- Shortest paths are simple paths (no positive and negative cycles).
- Path length is at most $k - 1$ for k vertices.

Bellman-Ford Example:

```
BELLMAN-FORD(G, W, s)
01. INITIALIZE-SINGLE-SOURCE (G,s)   // Line 1 takes O(V) time
02. for i = 1 to |G.V|-1
03.    for each edge (u, v) ∈ G.E
04.        RELAX(u, v, w)            // Lines 2 - 4 take O(VE) time
05. for each edge (u.v) ∈ G.E
06. if v.d > u.d = w(u.v)
07.    // G contains a negative weight cycle
08.    return FALSE                  // Lines 5 - 8 take O(E) time
09. return TRUE  // Line 9 takes O(1) time
```

> Running time: Total is $O(V \cdot E)$



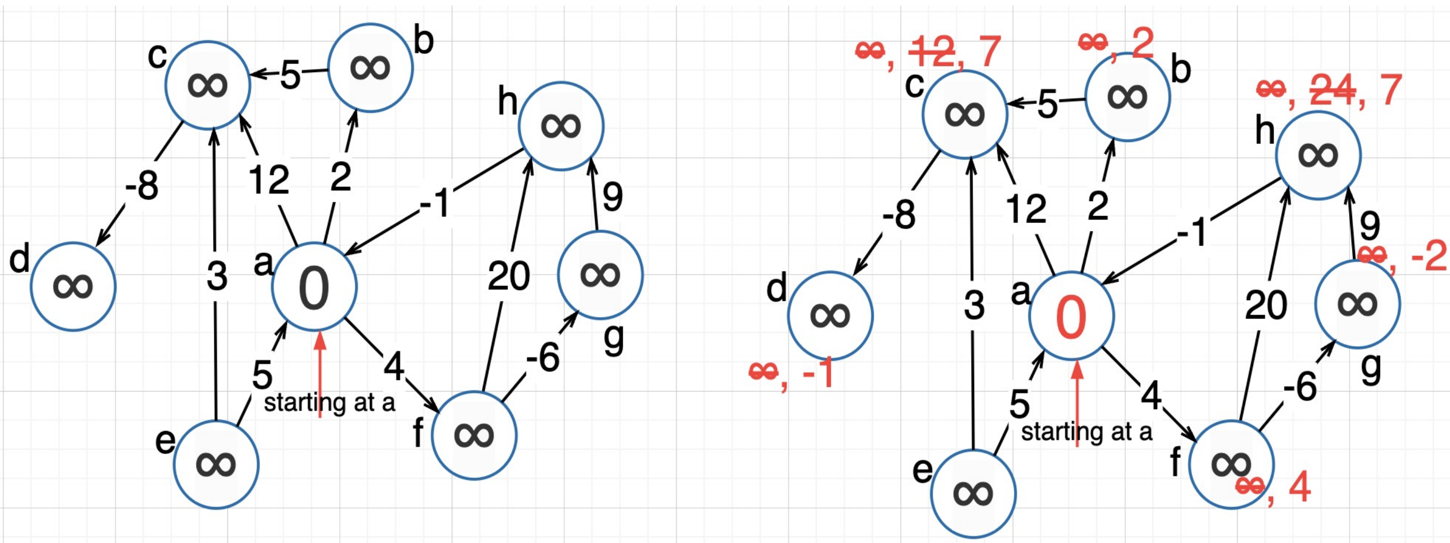
- Round 0 to 4 in the Table

	$v.d$, **round 0 - 4**				
v	**0**	**1**	**2**	**3**	**4**
a	∞	0	0	0	
b	∞	2	2	2	
c	∞	12	7	7	
d	∞	∞	∞	-1	
e	∞	∞	∞	∞	
f	∞	4	4	4	
g	∞	∞	-2	-2	

	$v.d$, **round 0 - 4**			
h	∞	∞	24	7

- Round 4: No change (Can Stop)

Method2: For every iteration, to check all edges



Iterations	(a,b)	(a,c)	(a,f)	(b,c)	(c,d)	(e,c)	(e,a)	(f,g)	(f,h)	(g,h)	(h,a)
1st iteration	√	√	√	√	√	√	√	√	√	√	√
2nd iteration	√	√	√	√	√	√	√	√	√	√	√

- Only need 2 iterations can find out the shortest path, |V|- 1 sometimes not necessary.
- $\{(a=0),(b=2),(c=7),(d=-1),(e=\infty),(f=4),(g=-2),(h=7)\}$

SSSP in DAGs

2. Directed Acyclic Graph (DAG) Algorithm:

- Efficiently finds shortest paths in DAGs ONLY, No negative weight cycles
- Negative edge weights allowed
- **Complexity**: $O(V+E)$, faster due to topological sorting.

```
Dag-Shortest-Paths(G, w, s)
Topologically sort the vertices of G
Initialize-Single-Source(G,s)
for each vertex u, taken in topologically sorted order
   for each v ∈ G.Adj[u]
       Relax(u,v, w)
```

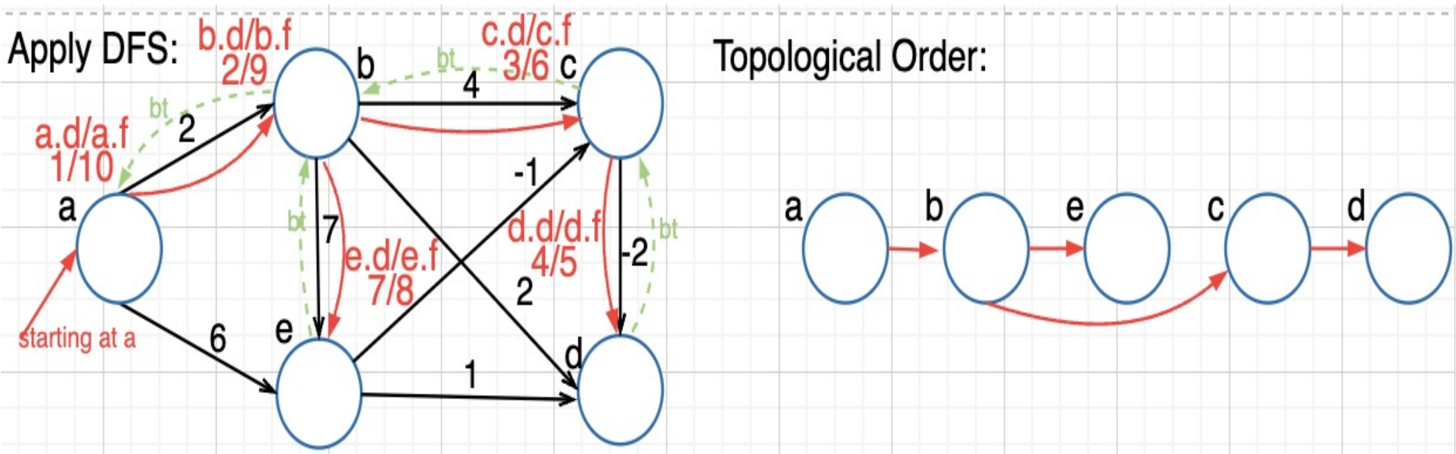
Key Steps:

- Perform topological sort.
- Relax edges in topological order.

DAG-Shortest-Paths(G, w, s) Example:

Given that $G(V, E)$, vertex$=\{a, b, c, d, e\}$, and the edges with weights$=$ $\{(a, b, 2), (a, e, 6), (b, c, 4), (b, d, 2), (b, e, 7), (e, c, -1), (e, d, 1), (c, d, -2)\}$

1. Perform `DFS(G)` to find the topological order (use reverse alpha order)



2. Then Initialize Nodes, and **Relax the Nodes in topological sorting order**

v	$v.d$	$v.f$	-		0	1	$u.\pi$
a	1	10	-	a	0	0	NiL
b	2	9	-	b	∞	(a, b) $0+2=2$	a
c	3	6	-	c	∞	~~(b, c)~~ $\cancel{2+4=6}$ (e, c) $6+(-1)=5$	$\cancel{b}, e$
d	4	5	-	d	∞	~~(c, d)~~ $\cancel{6+(-2)=4}$ (c, d) $5+(-2)=3$	$\cancel{e}, c$
e	7	8	-	e	∞	(a, e) $0+6=6$	a

- Starting at a, so relaxing from a, b, and $a, e\ldots$
- then b, e, c, d by the topological sorting order, it means the edges ONLY goes from left to right
- Each vertex only relaxes the edges from itself only 1 time, we can see that the topological sorting order is the shortest path in the example a, b, e, c, d is the $v.d = \delta(s, v_k)$
  - The relaxing in the `Bellman-Ford()` is $V - 1$ times which is $O(V \cdot E)$

Using DAG-Shortest-Paths Algorithm for PERT Charts Example:

Problem: Rather than looking for the shortest path, we want to know the longest path

Two strategies:

- Replace all weights $w(u, v)$ with $-w(u, v)$

- Initialize weights to - ∞ and relax to increase $v.d$ instead of decreasing

Dijkstra Algorithm

3. Dijkstra's Algorithm:

Greedy Algorithm: Similar to Prim's for minimum spanning trees.

Key Steps:

1. Initialize distances; source to 0, others to infinity.
2. Use a min-heap for vertices and distances.
3. Extract min, relax edges until the queue is empty.

Properties:

- Guarantees shortest paths in graphs with non-negative weights.
- Not suitable for negative weight edges.

Conditions: Weighted, directed graph G with no negative edges, best represented using adjacency lists.
Purpose: Finds shortest paths from a source to all vertices efficiently using a greedy approach.
Min-Heap: Utilizes a min-heap for efficient vertex selection with the smallest distance。
Complexity: $O((V+E)\log V)$ or $O(E\log V)$, efficient for graphs with non-negative weights.

- Involves:
  - Extracting min value from a priority queue (min-heap), $O(V\log V)$.
  - Relaxing edges and updating edge values in the priority queue, $O(E\log V)$.

```
DIJKSTRA(G, w, s)
01.  INITIALIZE-SINGLE-SOURCE(G, s)
// Initialize distances from source to all vertices // O(V)
02.  S = ∅
// S, a set to store vertices whose final shortest-path weights from the source are already determined   // O(1)
03.  Q = ∅      // Priority queue Q to store all vertices of the graph  // O(1)
04.  for each vertex u ∈ G.V             // O(V)
05.      INSERT(Q, u)
// Insert each vertex into the priority queue Q, // O(log V) for each vertex, total O(V log V)
06.  while Q ≠ ∅    // The linear O(V)
07.      u = EXTRACT-MIN(Q)
// Extract vertex u with the smallest distance value from Q, // The Logarithmic O(log V) for each vertex, total O(V log V)
08.      S = S ∪ {u}
// Add u to the set S of vertices with finalized shortest-path weights  // Constant O(1)
09.      for each vertex v in G.Adj[u]
// For each neighbor v of u, // The linear O(E), as each edge will be considered once
10.          RELAX(u, v, w)
// Relax the edge (u, v) to potentially find a shorter path from source to v through u    // O(1)
11.          if the call of RELAX decreased v.d
12.              DECREASE-KEY(Q, v, v.d)
// If RELAX was successful in decreasing the distance value of v, update v's position in the priority queue,      // O(log V) for each decrease, total O(E log V)
```

Explanation:

1. **Initialize Distances**: O(V) - Initializing distances and predecessors for all vertices.
2. **Initialize Set S**: O(1) - Creating an empty set.
3. **Initialize Priority Queue Q**: O(1) - Creating an empty priority queue.
4. **For Each Vertex**: O(V) - Iterating through each vertex.
5. **Insert into Priority Queue**: O(log V) per insertion, total O(V log V) - Inserting each vertex into the priority queue.
6. **While Loop**: O(V) - In the worst case, every vertex is dequeued once.
7. **Extract Min**: O(log V) per extraction, total O(V log V) - Extracting the vertex with the minimum distance value.
8. **Add to Set S**: O(1) - Adding a vertex to the set.
9. **For Each Neighbor**: O(E) - In total, all adjacent vertices are considered.
10. **Relax Edge**: O(1) - Relaxing an edge takes constant time.
11. **If Statement**: O(1) - Checking a condition takes constant time.
12. **Decrease Key**: O(log V) per decrease, total O(E log V) - Updating a vertex's position in the priority queue.

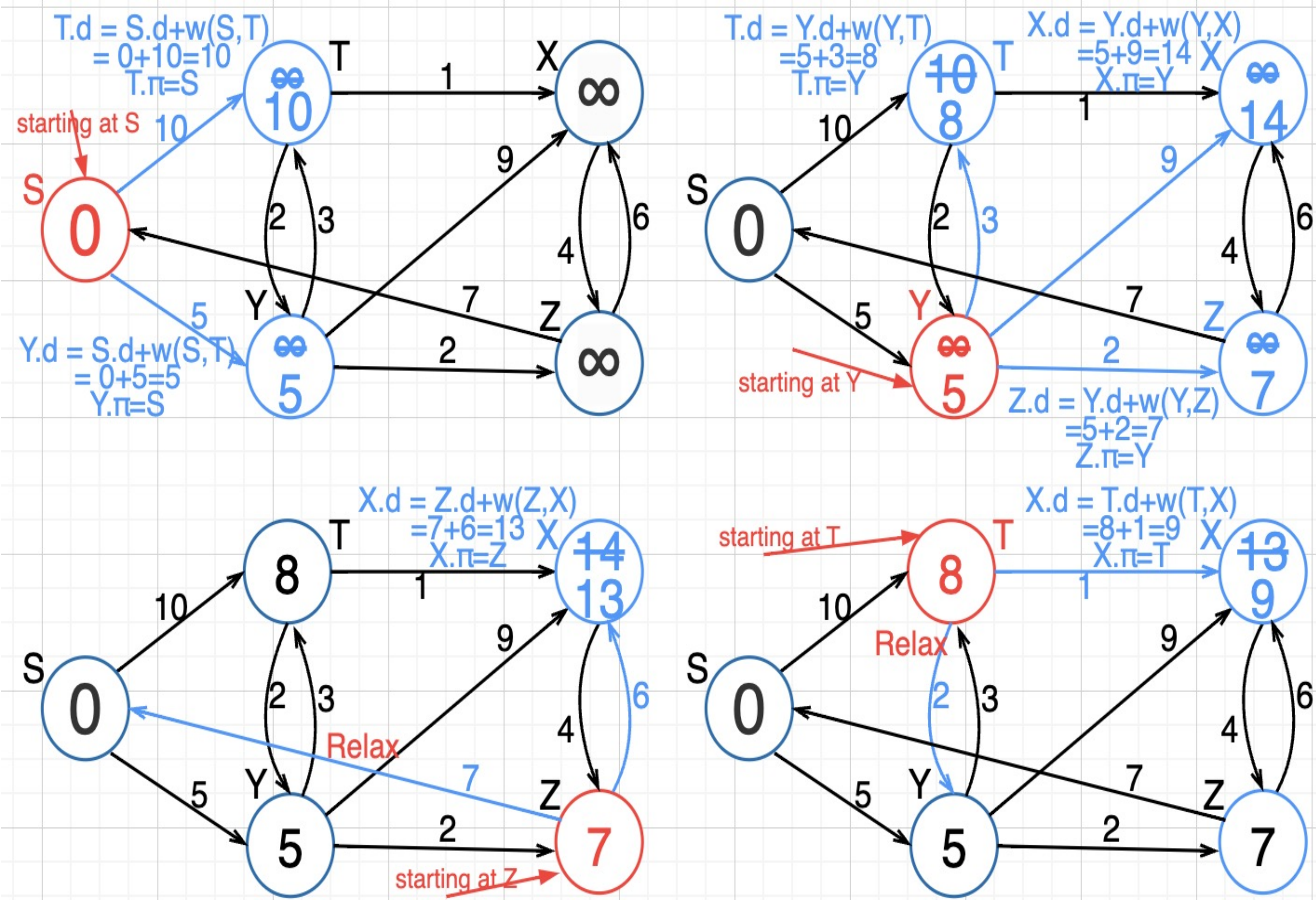
Example: Dijkstra's Algorithm

Given Graph $G(V, E)$: Vertices: $\{S, Y, Z, T, X\}$, Edges with weights: $(S, T, 10), (S, Y, 5), (T, Y, 2), (Y, T, 3),$ $(Y, Z, 2), (Y, X, 9), (Z, X, 6), (Z, S, 7), (T, X, 1), (X, Z, 4)$

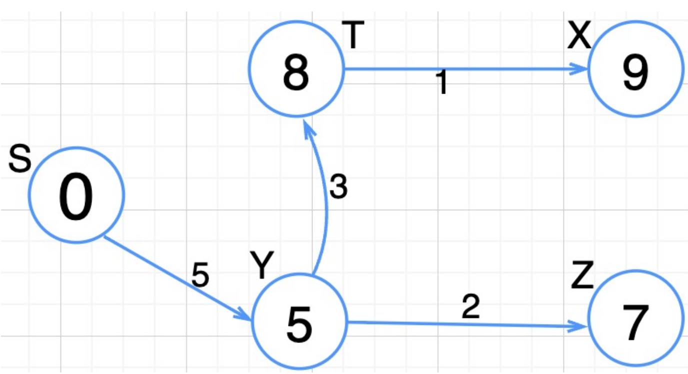
Dijkstra Algorithm: Tracing, What is the shortest path from S to X?? To find the shortest path from S to X using Dijkstra's Algorithm

1. Initialization:

- Starting from vertex S, , Set $S.d = 0$ (distance from source to source is 0).
- For all other vertices V, set $V.d = \infty$ and $V.\pi = \text{NIL}$.
- $S.\pi$ is not defined as S is the starting point.
- The vertex S keep s tracking of all vertices that its visited.



- The shortest path from S to X, Removed all paths that are not belonged to the shortest path in the graph



Step	Vertex	d	π	Visited	Min d	Operation	Note
Init	S	0	NIL	No	0	Initialize	Start at S
Init	Y	∞	NIL	No			
Init	Z	∞	NIL	No			
Init	T	∞	NIL	No			
Init	X	∞	NIL	No			
1	S	0	NIL	**Yes**	0	Visit	Update Y, T

Step	Vertex	d	π	Visited	Min d	Operation	Note
1	Y	5	S	No		Relax	
1	T	10	S	No		Relax	
2	Y	5	S	**Yes**	5	Visit	Update T, Z, X
2	T	8	Y	No		Relax	
2	Z	7	Y	No		Relax	
2	X	14	Y	No		Relax	
3	Z	7	Y	**Yes**	7	Visit	No update
4	T	8	Y	**Yes**	8	Visit	Update X
4	X	9	T	No		Relax	
5	X	9	T	**Yes**	9	Visit	Path to X finalized

Theorem Proof:

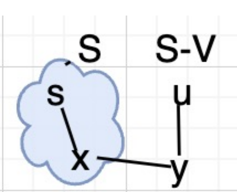
Dijkstra's algorithm, run on a weighted directed graph G=(V, E) with nonnegative edge weights, terminates with $v.d = \delta(s, v)$ for all v in V.
By Induction prove that $v.d = \delta(s, v)$ for all v in S is the shortest path in the graph G.

- The goal is to prove that the calculated distances converge to the shortest paths upon algorithm termination.
- The proof is based on induction, with two main parts:

1. **Base Case:** Initially, the set S (vertices with finalized shortest-path weights) is empty, and the base case holds trivially. When the source vertex is added to S, its distance is correctly initialized to 0.
2. **Inductive Step:** Assume that for all vertices in S, the calculated distance is equal to the shortest path distance. When a new vertex u is dequeued from the heap, the goal is to show that the distance of u has converged to the shortest path distance.

S: Set of nodes with finalized shortest paths from the start.
For proof, Assume that vertex `s` and `x` are in the Set S

S-V: Set of nodes with tentative shortest paths, yet to be finalized.
For proof, Assume that vertex `y` and `u` is in the set $S - V$

- 

In the Induction Proof: $s.d = \delta(s, s) = 0$,

1. Base Case: $S = \emptyset$, and $S = \{S\}$
2. $v.d = \delta(s, v)$ for all the vertices in S

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
```

in the code `u = EXTRACT-MIN(Q)` , when Dequeue the Vertex `u` from the `min-heap`

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

$u.d \le y.d$, $\delta(s,u) \le u.d$ (upper bound properties)
$\delta(s,y) \le \delta(s,u)$, $y.d = \delta(s,y)$
$\delta(s,y) \le \delta(s,u) \le u.d \le y.d = \delta(s,y)$
$\delta(s,y) = \delta(s,u) = y.d = u.d$, $u.d = \delta(s,u)$

Linear Programming in Graphs

Objective and Form:

- Optimize a linear objective function subject to linear constraints.
- $Ax \le b$, $x \ge 0$, where c, x are coefficient vectors, A is a coefficient matrix, and b is a constant vector.
- Form: min/max $\sum_{i=1}^{n} C_i X_i$

Key Concepts:

- **Objective Function**: Function to minimize/maximize.
- **Constraints**: Linear inequalities defining feasible region.
- **Feasible Region**: All points satisfying constraints.
- **Optimal Solution**: Point minimizing objective function in feasible region.

Constraint Graph Construction:

- Vertices for variables, edges for constraints.
- Extra vertex v_0 with 0-weight edges to others.

Example Constraint Graph:

- Vertices: $v_0, v_1, v_2, \ldots$ for $x_1, x_2, \ldots$
- Edges: Directed, weighted, representing constraints.

Solving the Problem:

1. **Create Vertices**: For each X_i, create V_i.
2. **Add Source Vertex**: Add V_0.
3. **Connect Source to All**: Connect V_0 to all V_i with 0-weight
4. **Add Edges for Constraints**: For each $X_j - X_i \le b_k$, add edge $V_i \to V_j$ with weight b_k.

Direction of Edges in Constraint Graph

To determine the direction of edges in the constraint graph for inequalities $X_j - X_i \le b_k$:

1. **Rearrange Inequality**: If necessary, rearrange to $X_i \le X_j + b_k$.

2. **Direct Edge**: Draw an edge from X_i to X_j with weight b_k.

Solving the Problem:

- Convert to shortest path problem, run Bellman-Ford from v_0.
- Shortest paths give variable values.

Key Points:

- **Negative Weight Cycles**: No solution if present.
- **Shortest Paths**: Represent solution.
- **Bellman-Ford Algorithm**: Handles negative weights.

Linear Programming in Graphs_

Objective:

- Optimize linear function $\sum C_i X_i$ within constraints.

Constraints:

- Represented by $Ax \leq b$, $x \geq 0$.

Key Concepts:

- **Feasible Region**: Set of points satisfying constraints.
- **Optimal Solution**: Point in feasible region that optimizes the function.

Graph Construction:

- Create vertices for variables and edges for constraints.
- Use a source vertex v_0 connected to all others with zero weight.

Solving:

- Transform to shortest path problem, use Bellman-Ford algorithm from v_0.
- Solutions reflected in shortest paths from v_0.

Key Points:

- **Negative Weight Cycles**: Indicate no solution.
- **Bellman-Ford**: Suitable for negative weights.

Example Constraint Graph:

- Vertices: $v_0, v_1, v_2, \ldots$ representing $x_1, x_2, \ldots$
- Edges: Represent constraints, directed and weighted.

1. Convert Inequalities to Graph:

- Turn linear inequalities $X_j - X_i \leq b_k$ into a directed graph: variables $X_1, \ldots, X_n$ as vertices $V_1, \ldots, V_n$, add V_0.

2. Zero-Weight Edges:

- Link V_0 to all vertices with zero-weight edges.

3. Find Shortest Paths:

- Obtain $X_1, \ldots, X_n$ values by finding shortest paths from V_0 to all vertices.

The Given System of Inequalities for all constraints $X_j - X_i \leq b_k$

$$\text{subject to } \begin{cases} X_1 - X_2 \leq 0 \\ X_1 - X_5 \leq -1 \\ X_2 - X_5 \leq 1 \\ -X_1 + X_3 \leq 5 \\ -X_1 + X_4 \leq 4 \\ -X_3 + X_4 \leq -1 \\ -X_3 + X_5 \leq -3 \\ -X_4 + X_5 \leq -3 \end{cases}$$

Step 1. **Create Vertices**: For each variable X_i, create a vertex V_i in the graph.

- **Vertices**: $V_0, V_1, V_2, V_3, V_4, V_5$

Step 2. **Add a Source Vertex**: Add an additional vertex V_0 which will act as a source vertex.

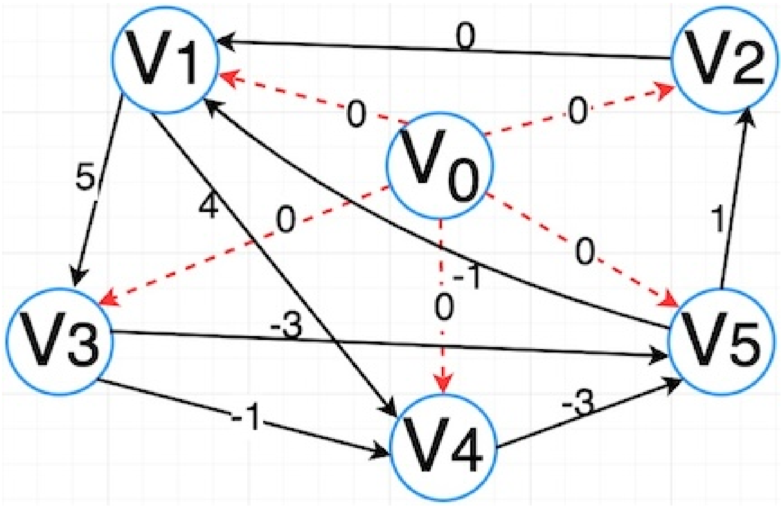
- A new Source Vertex V_0

Step 3. **Connect Source to All Vertices**: Connect V_0 to all other vertices V_i with edges of weight 0.

- **Connect V_0 to All Vertices**: Add edges with weight 0 from V_0 to V_1, V_2, V_3, V_4, V_5.

Step 4. **Add Edges for Constraints**: For each constraint $X_j - X_i \leq b_k$, add a directed edge from vertex V_i to vertex V_j with weight b_k.

- **Add Edges for Constraints**:
- $X_1 - X_2 \leq 0$: Add edge from V_2 to V_1 with weight 0.
- $X_1 - X_5 \leq -1$: Add edge from V_5 to V_1 with weight -1.
- $X_2 - X_5 \leq 1$: Add edge from V_5 to V_2 with weight 1.
- $-X_1 + X_3 \leq 5$: Add edge from V_1 to V_3 with weight 5.
- $-X_1 + X_4 \leq 4$: Add edge from V_1 to V_4 with weight 4.
- $-X_3 + X_4 \leq -1$: Add edge from V_4 to V_3 with weight -1.
- $-X_3 + X_5 \leq -3$: Add edge from V_5 to V_3 with weight -3.
- $-X_4 + X_5 \leq -3$: Add edge from V_5 to V_4 with weight -3.



Solving the Problem:

- Convert to a shortest path problem.
- Run Bellman-Ford algorithm starting from v_0.
- Shortest paths from v_0 give the values of variables.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix} \tag{1}$$

Result: $x = \{-5, -3, 0, -1, -4\}$