Graph Theory and Its Applications

Dr. G.H.J. Lanel

Lecture 9

Outline

Outline

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation
  - Flight itineraries
  - Circuit wiring
  - Network routing

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation,
  - Flight itineraries,
  - Circuit wiring,
  - Network routing

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation,
  - Flight itineraries,
  - Circuit wiring,
  - Network routing.

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation,
  - Flight itineraries,
  - Circuit wiring,
  - Network routing.

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation,
  - Flight itineraries,
  - Circuit wiring,
  - Network routing.

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation,
  - Flight itineraries,
  - Circuit wiring,
  - Network routing.

- Find the shortest path from point A to point B.
- Shortest in time, distance, cost, etc.
- Numerous applications:
  - Map navigation,
  - Flight itineraries,
  - Circuit wiring,
  - Network routing.

Outline

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph
  - Goal: to find a path with shortest length

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph
  - Goal: to find a path with shortest length

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph
  - Goal: to find a path with shortest length

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph.
  - Goal: to find a path with shortest length.

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph.
  - Goal: to find a path with shortest length.

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph.
  - Goal: to find a path with shortest length.

- Weighted graphs:
  - Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge.
  - Cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.
  - Goal: to find a smallest cost path.
- Unweighted graphs:
  - Input is an unweighted graph.
  - Goal: to find a path with shortest length.

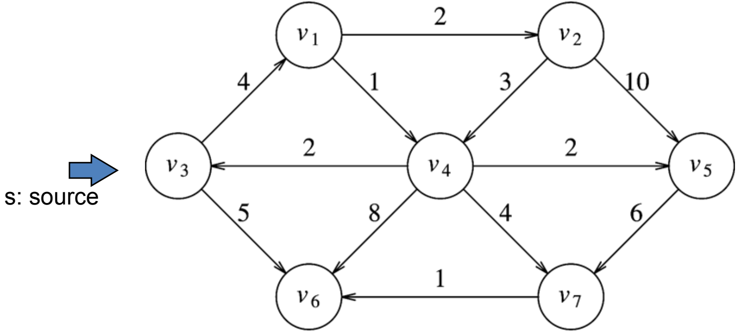
Outline

Single-source shortest path problem

Given a weighted graph $G = (V, E)$, and a source vertex s, find the minimum weighted path from s to every other vertex in G.

Single-source shortest path problem

Given a weighted graph $G = (V, E)$, and a source vertex s, find the minimum weighted path from s to every other vertex in G.



Point to point shortest path problem

- Given $G = (V, E)$ and two vertices A and B, find a shortest path from A (source) to B (destination).
- Solution:
  1. Run the code for Single Source Shortest Path using source as A.
  2. Stop algorithm when B is reached.

Point to point shortest path problem

- Given $G = (V, E)$ and two vertices A and B, find a shortest path from A (source) to B (destination).
- Solution:
  1. Run the code for Single Source Shortest Path using source as A.
  2. Stop algorithm when B is reached.

Point to point shortest path problem

- Given $G = (V, E)$ and two vertices A and B, find a shortest path from A (source) to B (destination).
- Solution:
  1. Run the code for Single Source Shortest Path using source as A.
  2. Stop algorithm when B is reached.

Point to point shortest path problem

- Given $G = (V, E)$ and two vertices A and B, find a shortest path from A (source) to B (destination).
- Solution:
  1. Run the code for Single Source Shortest Path using source as A.
  2. Stop algorithm when B is reached.

All pairs shortest path problem

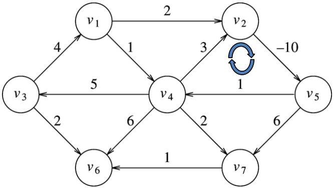
- Given $G = (V, E)$, find a shortest path between all pairs of vertices.
- Solutions: Solve Single Source Shortest Path for each vertex as source

All pairs shortest path problem

- Given $G = (V, E)$, find a shortest path between all pairs of vertices.
- Solutions: Solve Single Source Shortest Path for each vertex as source

Negative weights shortest path problem

- Graphs can have negative weights.

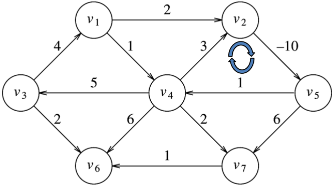


- E.g.,
  - Shortest positive-weight path is a net gain.
  - Path may include individual losses.
- Problem: Negative weight cycles

  Allow arbitrarily-low path costs
- Solution: Detect presence of negative-weight cycles

Negative weights shortest path problem

- Graphs can have negative weights.



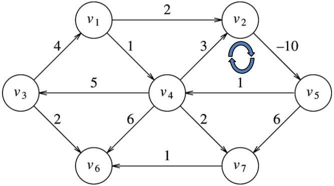
- E.g.,
  - Shortest positive-weight path is a net gain.
  - Path may include individual losses.
- Problem: Negative weight cycles

  Allow arbitrarily-low path costs

- Solution: Detect presence of negative-weight cycles

Negative weights shortest path problem

- Graphs can have negative weights.



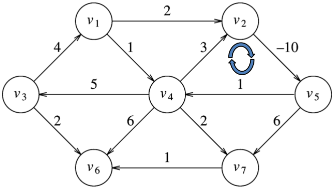
- E.g.,
  - Shortest positive-weight path is a net gain.
  - Path may include individual losses.
- Problem: Negative weight cycles

  Allow arbitrarily-low path costs

- Solution: Detect presence of negative-weight cycles

Negative weights shortest path problem

- Graphs can have negative weights.

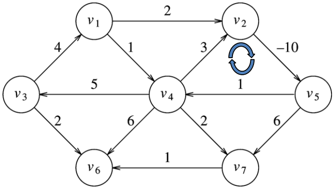


- E.g.,
  - Shortest positive-weight path is a net gain.
  - Path may include individual losses.
- Problem: Negative weight cycles

  Allow arbitrarily-low path costs
- Solution: Detect presence of negative-weight cycles

Negative weights shortest path problem

- Graphs can have negative weights.

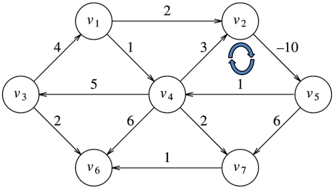


- E.g.,
  - Shortest positive-weight path is a net gain.
  - Path may include individual losses.
- Problem: Negative weight cycles

  Allow arbitrarily-low path costs
- Solution: Detect presence of negative-weight cycles

Negative weights shortest path problem

- Graphs can have negative weights.



- E.g.,
  - Shortest positive-weight path is a net gain.
  - Path may include individual losses.
- Problem: Negative weight cycles

  Allow arbitrarily-low path costs
- Solution: Detect presence of negative-weight cycles

Outline

- No weights on edges.
- Find shortest length paths.
- Same as weighted shortest path with all weights equal.
- Use BFS algorithm.

- No weights on edges.
- Find shortest length paths.
- Same as weighted shortest path with all weights equal.
- Use BFS algorithm.

- No weights on edges.
- Find shortest length paths.
- Same as weighted shortest path with all weights equal.
- Use BFS algorithm.

- No weights on edges.
- Find shortest length paths.
- Same as weighted shortest path with all weights equal.
- Use BFS algorithm.

For each vertex, keep track of

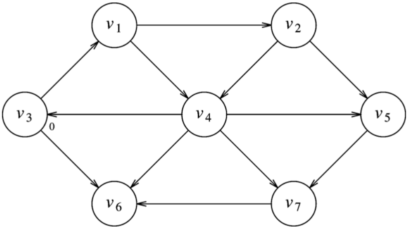
- Whether we have visited it (known).
- Its distance from the start vertex (d_v).
- Its predecessor vertex along the shortest path from the start vertex (p_v).

For each vertex, keep track of

- Whether we have visited it (known).
- Its distance from the start vertex (d_v).
- Its predecessor vertex along the shortest path from the start vertex (p_v).

For each vertex, keep track of

- Whether we have visited it (known).
- Its distance from the start vertex (d_v).
- Its predecessor vertex along the shortest path from the start vertex (p_v).

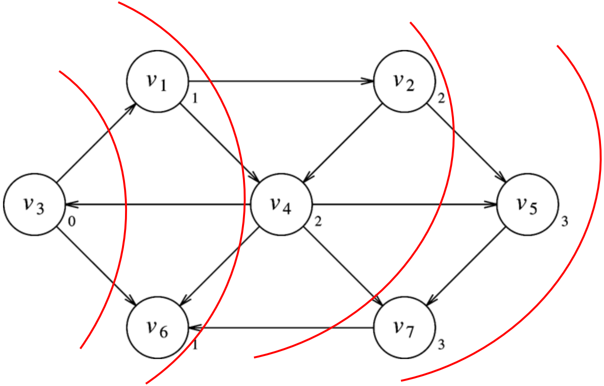


v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	known	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

source



Outline

Use Dijkstras algorithm:

- GREEDY strategy: Always pick the next closest vertex to the source.
- Use priority queue to store unvisited vertices by distance from s.
- Does not work with negative weights.

Use Dijkstras algorithm:

- GREEDY strategy: Always pick the next closest vertex to the source.
- Use priority queue to store unvisited vertices by distance from s.
- Does not work with negative weights.

Use Dijkstras algorithm:

- GREEDY strategy: Always pick the next closest vertex to the source.
- Use priority queue to store unvisited vertices by distance from s.
- Does not work with negative weights.

- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1.** Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2.** Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3.** If $u = t$, then stop.
- **Step 4.** For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5.** Change T to $T - \{u\}$ and go to step 2.

- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1.** Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2.** Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3.** If $u = t$, then stop.
- **Step 4.** For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5.** Change T to $T - \{u\}$ and go to step 2.

- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1.** Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2.** Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3.** If $u = t$, then stop.
- **Step 4.** For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5.** Change T to $T - \{u\}$ and go to step 2.

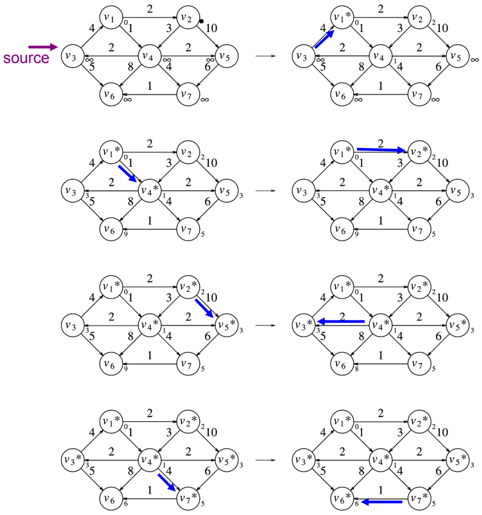
- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1**. Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2**. Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3**. If $u = t$, then stop.
- **Step 4**. For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5**. Change T to $T - \{u\}$ and go to step 2.

- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1**. Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2**. Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3**. If $u = t$, then stop.
- **Step 4**. For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5**. Change T to $T - \{u\}$ and go to step 2.

- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1**. Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2**. Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3**. If $u = t$, then stop.
- **Step 4**. For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5**. Change T to $T - \{u\}$ and go to step 2.

- **Input**: Vertices with cost.
- **Output**: The shortest path with the cost between s and v.
- **Step 1**. Set $\lambda(s) = 0$ and for all vertices $v \neq s, \lambda(v) = \infty$. Set $T = V$, the vertex set of G.
- **Step 2**. Let u be a vertex in T for which $\lambda(u)$ is minimum.
- **Step 3**. If $u = t$, then stop.
- **Step 4**. For every edge $e = uv$ incident with u, if $v \in T$ and $\lambda(v) > \lambda(u) + \omega(e)$ change the value of $\lambda(v)$ to $\lambda(u) + \omega(e)$.
- **Step 5**. Change T to $T - \{u\}$ and go to step 2.

An Example



Why Dijkstra's Algorithm Works

- A least-cost path from vertex X to vertex Y contains least-cost paths from X to every vertex on the path to Y.
- E.g., if $X \to C_1 \to C_2 \to C_3 \to Y$ is the least-cost path from X to Y, then
  - $X \to C_1 \to C_2 \to C_3$ is the least-cost path from X to C_3.
  - $X \to C_1 \to C_2$ is the least-cost path from X to C_2.
  - $X \to C_1$ is the least-cost path from X to C_1.

Why Dijkstra's Algorithm Works

- A least-cost path from vertex X to vertex Y contains least-cost paths from X to every vertex on the path to Y.
- E.g., if $X \to C_1 \to C_2 \to C_3 \to Y$ is the least-cost path from X to Y, then
  - $X \to C_1 \to C_2 \to C_3$ is the least-cost path from X to C_3.
  - $X \to C_1 \to C_2$ is the least-cost path from X to C_2.
  - $X \to C_1$ is the least-cost path from X to C_1.

Why Dijkstra's Algorithm Works

- A least-cost path from vertex X to vertex Y contains least-cost paths from X to every vertex on the path to Y.
- E.g., if $X \to C_1 \to C_2 \to C_3 \to Y$ is the least-cost path from X to Y, then
  - $X \to C_1 \to C_2 \to C_3$ is the least-cost path from X to C_3.
  - $X \to C_1 \to C_2$ is the least-cost path from X to C_2.
  - $X \to C_1$ is the least-cost path from X to C_1.

Why Dijkstra's Algorithm Works

- A least-cost path from vertex X to vertex Y contains least-cost paths from X to every vertex on the path to Y.
- E.g., if $X \to C_1 \to C_2 \to C_3 \to Y$ is the least-cost path from X to Y, then
  - $X \to C_1 \to C_2 \to C_3$ is the least-cost path from X to C_3.
  - $X \to C_1 \to C_2$ is the least-cost path from X to C_2.
  - $X \to C_1$ is the least-cost path from X to C_1.

Why Dijkstra's Algorithm Works

- A least-cost path from vertex X to vertex Y contains least-cost paths from X to every vertex on the path to Y.
- E.g., if $X \rightarrow C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow Y$ is the least-cost path from X to Y, then
  - $X \rightarrow C_1 \rightarrow C_2 \rightarrow C_3$ is the least-cost path from X to C_3.
  - $X \rightarrow C_1 \rightarrow C_2$ is the least-cost path from X to C_2.
  - $X \rightarrow C_1$ is the least-cost path from X to C_1.

What about graphs with negative edges?

- Problem: Will the Dijkstras algorithm work as is?
- Solution: Do not mark any vertex as known. Instead allow multiple updates.

What about graphs with negative edges?

- Problem: Will the Dijkstras algorithm work as is?
- Solution: Do not mark any vertex as known. Instead allow multiple updates.