Shortest Path Algorithm

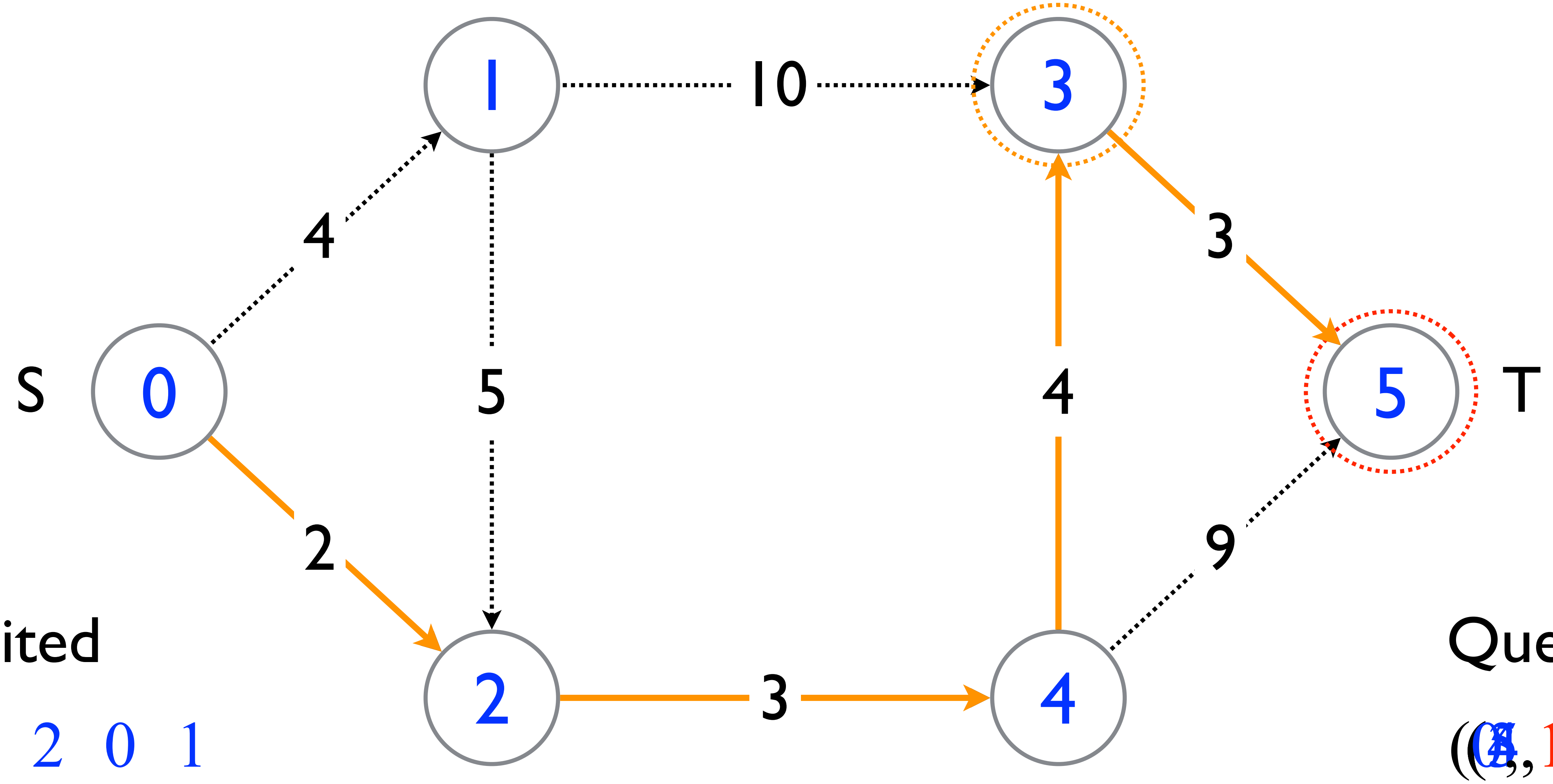
Data Structures and Algorithms

Emory University

Jinho D. Choi

EMORY
UNIVERSITY

Dijkstra's Algorithm



Visited

5 3 4 2 0 1

Queue

(0, 12)

(2, 10)

(1, 13)

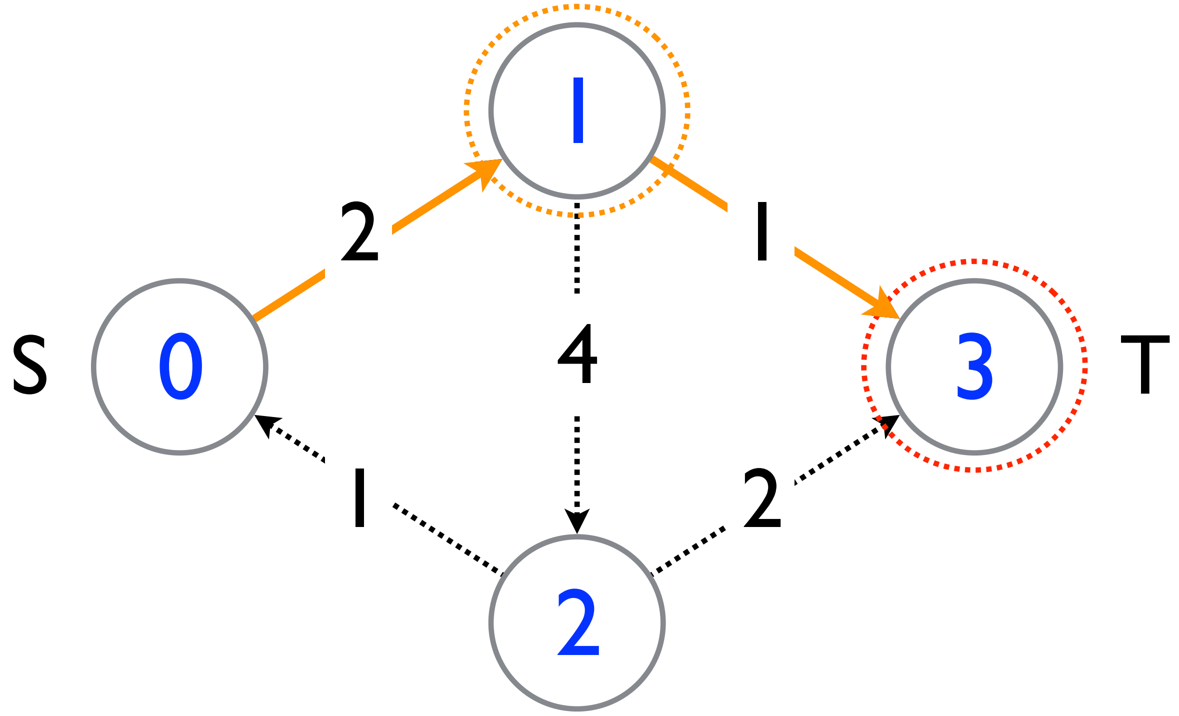
	0	1	2	3	4	5
Distance	12	13	10	3	7	0
Previous	2	3	4	5	3	∅

EMORY UNIVERSITY

Dijkstra's Algorithm

Visited

3 1 2 0



Queue

(3, 0)

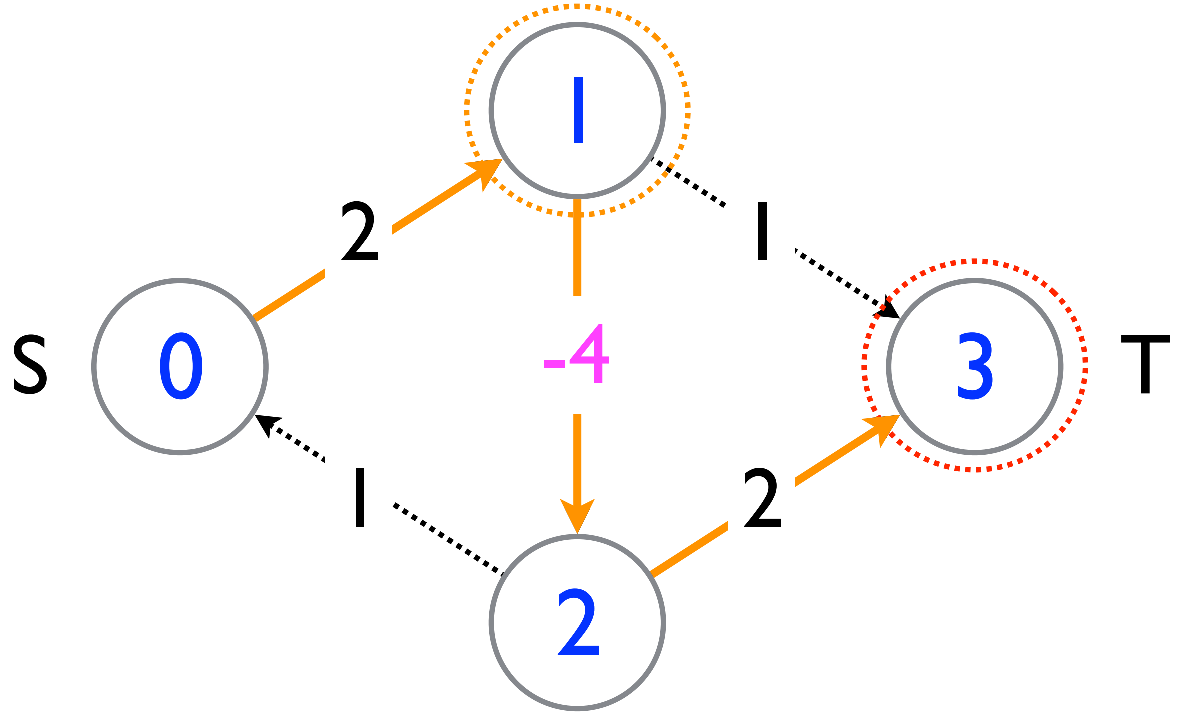
(0, 2)

	0	1	2	3
Distance	3	1	2	0
Previous	1	3	3	∅

Dijkstra's Algorithm

Visited

3 1 2 0



Queue

(1, -2) (3, 0)

(0, 3) (2, 2)

	0	1	2	3
Distance	3	-2	2	0
Previous	1	2	3	∅

Vertex Distance Pair

```
private class VertexDistancePair implements Comparable<VertexDistancePair>
{
  public int    vertex;
  public double distance;

  public VertexDistancePair(int vertex, double distance)
  {
    this.vertex = vertex;
    this.distance = distance;
  }

  @Override
  public int compareTo(VertexDistancePair p)
  {
    double diff = this.distance - p.distance;
    if (diff > 0) return  1;
    if (diff < 0) return -1;
    return 0;
  }
}
```

Dijkstra's Algorithm

```
public Integer[] getShortestPath(Graph graph, int source, int target)
{
  PriorityQueue<VertexDistancePair> queue = new PriorityQueue<>();
  Integer[] previous = new Integer[graph.size()];
  double[] distances = new double[graph.size()];
  Set<Integer> visited = new HashSet<>();

  init(distances, previous, target);
  queue.add(new VertexDistancePair(target, distances[target]));

  while (!queue.isEmpty())
  {
    // To be filled.
  }

  return previous;
}
```

Dijkstra's Algorithm

```
private void init(double[] distances, Integer[] previous, int target)
{
  for (int i=0; i<distances.length; i++)
  {
    if (i == target)
      distances[i] = 0;
    else
    {
      distances[i] = Double.MAX_VALUE;
      previous[i]  = null;
    }
  }
}
```

Complexity?

```
while (!queue.isEmpty())
{
  VertexDistancePair u = queue.poll();
  visited.add(u.vertex);

  for (Edge edge : graph.getIncomingEdges(u.vertex))
  {
    int v = edge.getSource();

    if (!visited.contains(v))
    {
      double dist = distances[u.vertex] + edge.getWeight();

      if (dist < distances[v])
      {
        distances[v] = dist;
        previous [v] = u.vertex;
        queue.add(new VertexDistancePair(v, dist));
      }
    }
  }
}
```

Dijkstra's vs A*

```
private void init(double[] distances, Integer[] previous, int target)
{
  for (int i=0; i<distances.length; i++)
  {
    if (i == target)
      distances[i] = 0 + heuristic(i, target);
    else
    {
      distances[i] = Double.MAX_VALUE;
      previous[i]  = null;
    }
  }
}
```

```
while (!queue.isEmpty())
{
  VertexDistancePair u = queue.poll();
  visited.add(u.vertex);

  for (Edge edge : graph.getIncomingEdges(u.vertex))
  {
    int v = edge.getSource();

    if (!visited.contains(v))
    {
      double dist = distances[u.vertex] + edge.getWeight();

      if (dist < distances[v])
      {
        distances[v] = dist + heuristic(v, target);
        previous [v] = u.vertex;
        queue.add(new VertexDistancePair(v, dist));
      }
    }
  }
}
```

A* Abstract Class

```
public abstract class AStar
{
  public Integer[] getShortestPath(Graph graph, int source, int target)
  {
    PriorityQueue<VertexDistancePair> queue = new PriorityQueue<>();
    Integer[] previous = new Integer[graph.size()];
    double[] distances = new double[graph.size()];
    Set<Integer> visited = new HashSet<>();

      ...

    return previous;
  }
}

protected abstract double heuristic(int source, int target);
```

Dijkstra’s Algorithm

```
public class Dijkstra extends AStar
{
  @Override
  protected double heuristic(int source, int target)
  {
    return 0;
  }
}
```

EMORY
UNIVERSITY