CS302 - Data Structures *using C++*

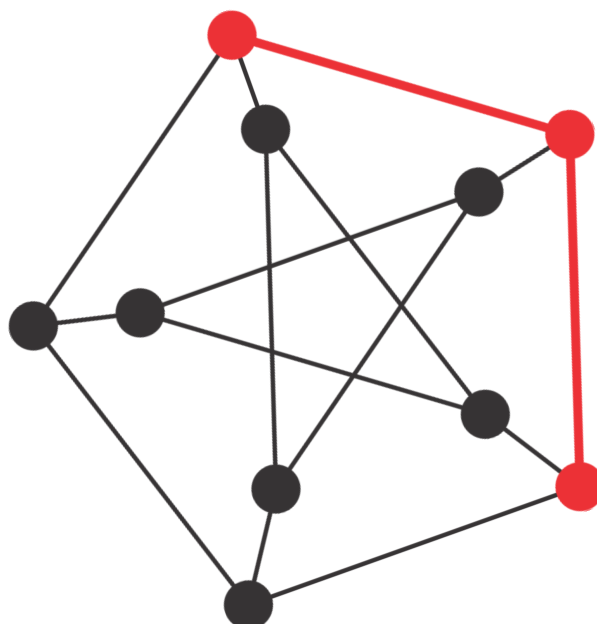
Topic: Dijkstra's Algorithm for Shortest Paths

Kostas Alexis

AUTONOMOUS ROBOTS LAB

Shortest Paths on Graphs: Dijkstra

- **A Formal Definition:** Dijkstra's algorithm is an algorithm for finding a *graph geodesic*, i.e., the shortest path between two graph vertices in a graph. It functions by constructing a shortest-path tree from the initial vertex to every other vertex in the graph.



Graph geodesic: a shortest path between two graph vertices (u,v) of a graph

Shortest Paths on Graphs: Dijkstra

- **A Formal Definition:** Dijkstra's algorithm is an algorithm for finding a graph geodesic, i.e., the shortest path between two graph vertices in a graph. It functions by constructing a shortest-path tree from the initial vertex to every other vertex in the graph.
- The worst-case running time for the Djikstra algorithm on a graph with n nodes and m edges is $O(n^2)$ because it allows for directed cycles.

Shortest Paths on Graphs: Dijkstra

- **A Formal Definition:** Dijkstra's algorithm is an algorithm for finding a graph geodesic, i.e., the shortest path between two graph vertices in a graph. It functions by constructing a shortest-path tree from the initial vertex to every other vertex in the graph.
- The worst-case running time for the Djikstra algorithm on a graph with n nodes and m edges is $O(n^2)$ because it allows for directed cycles.
- It even finds the shortest paths from a source node s to all other nodes in the graph.

Shortest Paths on Graphs: Dijkstra

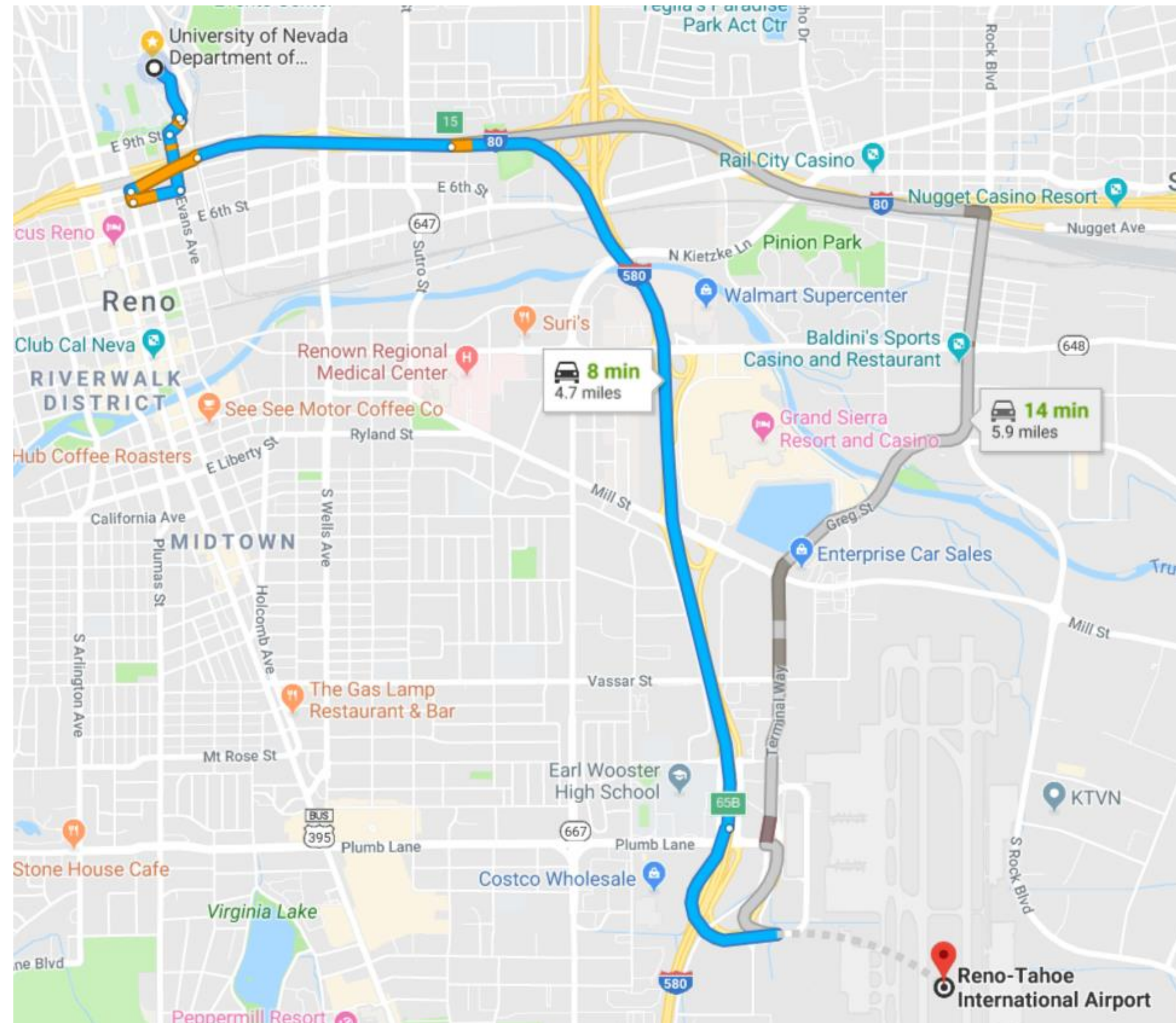
- **A Formal Definition:** Dijkstra's algorithm is an algorithm for finding a graph geodesic, i.e., the shortest path between two graph vertices in a graph. It functions by constructing a shortest-path tree from the initial vertex to every other vertex in the graph.
- The worst-case running time for the Djikstra algorithm on a graph with n nodes and m edges is $O(n^2)$ because it allows for directed cycles.
- It even finds the shortest paths from a source node s to all other nodes in the graph.
- This is basically $O(n^2)$ for node selection and $O(m)$ for distance updates.

Shortest Paths on Graphs: Dijkstra

- **A Formal Definition:** Dijkstra's algorithm is an algorithm for finding a graph geodesic, i.e., the shortest path between two graph vertices in a graph. It functions by constructing a shortest-path tree from the initial vertex to every other vertex in the graph.
- The worst-case running time for the Djikstra algorithm on a graph with n nodes and m edges is $O(n^2)$ because it allows for directed cycles.
- It even finds the shortest paths from a source node s to all other nodes in the graph.
- This is basically $O(n^2)$ for node selection and $O(m)$ for distance updates.
- While $O(n^2)$ is the best possible complexity for dense graphs, the complexity can be improved significantly for sparse graphs.

AUTONOMOUS ROBOTS LAB

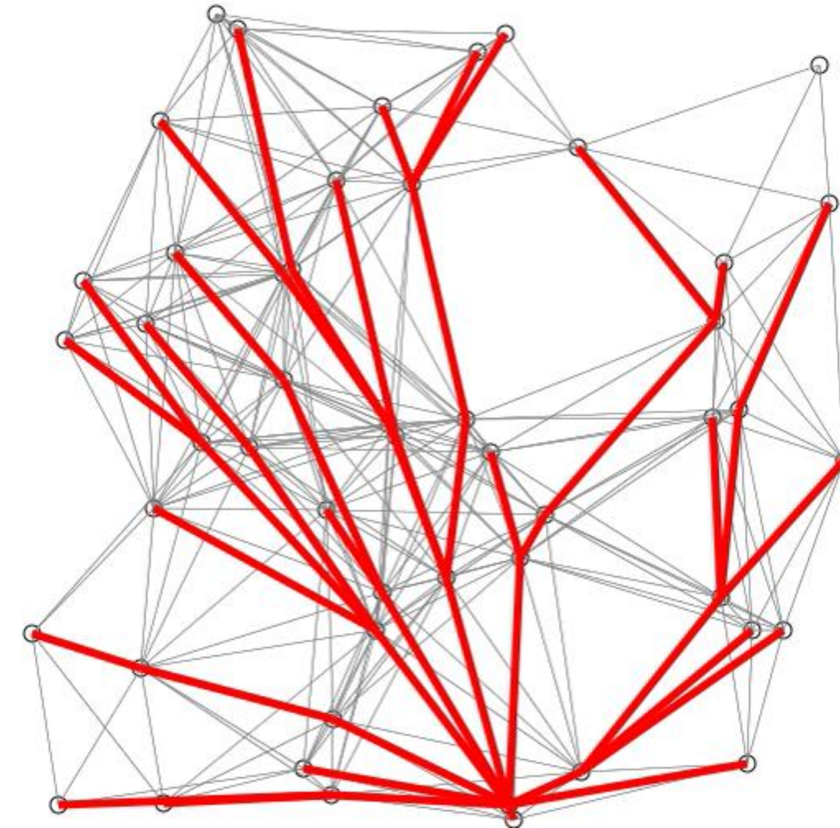
How do we get to the airport?



AUTONOMOUS ROBOTS LAB

Single-Source Shortest Path Problem

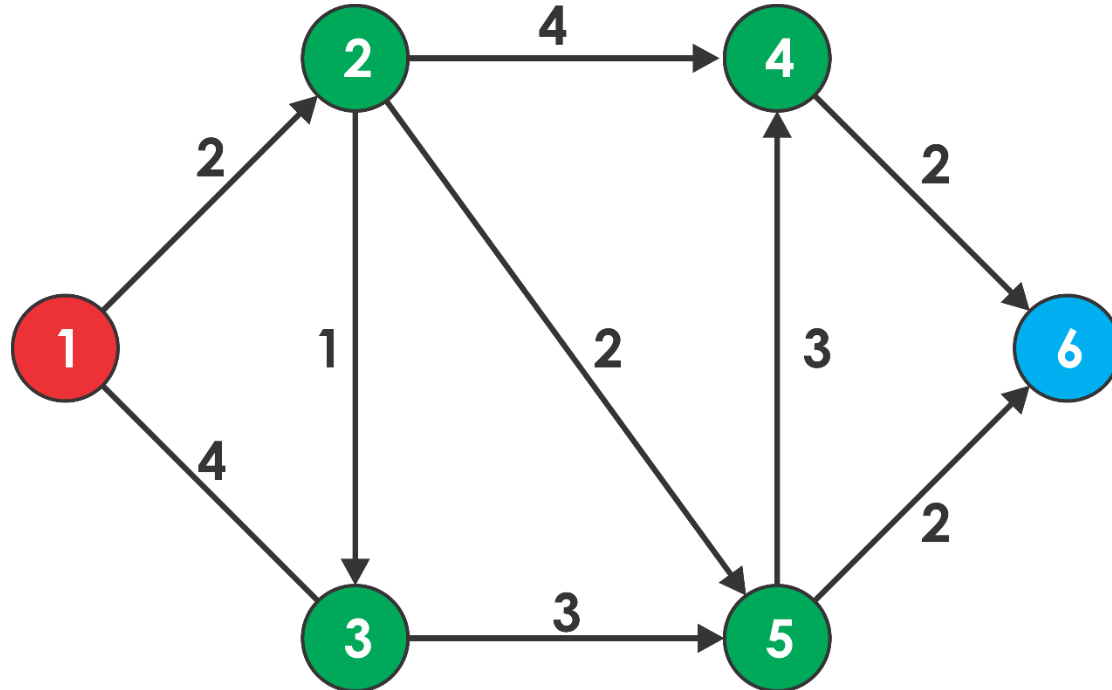
- The problem of finding shortest paths from a source vertex v to all other vertices in the graph.
- Weighted graph $G = (V, E)$
- Source vertex $s \in V$ to all vertices $v \in V$



By Shiyu Ji - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=54537901

Dijkstra's Algorithm

- Solution to the single-source shortest path problem in graph theory
  - Both directed and undirected graphs
  - All edges must have nonnegative weights
  - Graph must be connected



AUTONOMOUS ROBOTS LAB

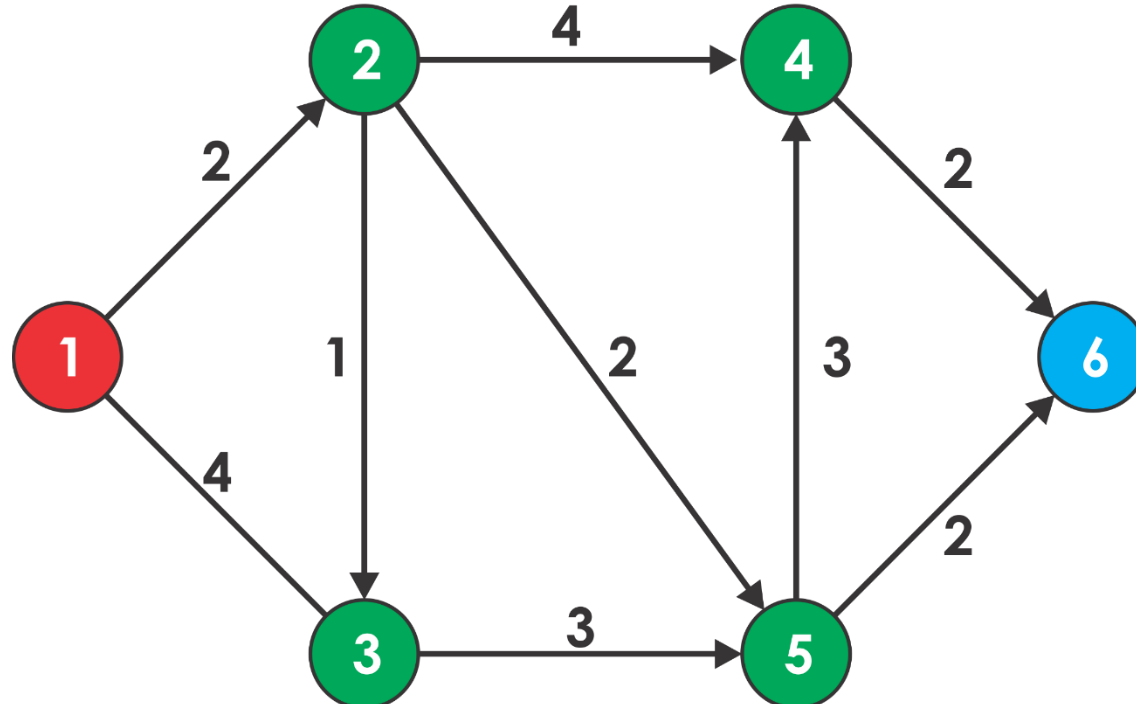
Dijkstra's Algorithm Pseudocode

Pseudocode	Comment
$dist[s] = 0$	// distance to source vertex is zero
for all $v \in V - \{s\}$	
do $dist[v] = \infty$	// set all other vertices to infinity
$S = \emptyset$	// S, the set of visited vertices is initially empty
$Q = V$	// Q, the queue initially contains all vertices
while $Q \neq \emptyset$	// while the queue is not empty
do $u = \boldsymbol{mindistance}(Q, dist)$	// select the element of Q with the min distance
$S = S \cup \{u\}$	// add u to list of visited vertices
for all $v \in \boldsymbol{neighbors}(u)$	
do if $dist(v) > dist(u) + w(u,v)$	// if new shortest path found
then $d(v) = d(u) + w(u,v)$	// set new value of shortest path
return $dist$	

Output of Dijkstra’s Algorithm

- Original algorithm outputs the value of the shortest path not the path itself
- With slight modification we can obtain the path also

Value δ(1,6) = 6
Path: {1,2,5,6}



Why it works

- **Lemma 1: Optimal Substructure**
  - The subpath of any shortest path is itself a shortest path
- **Lemma 2: Triangle Inequality**
  - If $\delta(u,v)$ is the shortest path length between any u and v, then $\delta(u,v) \leq \delta(u,v)+\delta(x,v)$

AUTONOMOUS ROBOTS LAB

Why do we use Dijkstra’s Algorithm

- Algorithms for calculating the shortest path from source to sink about as computationally expensive as calculating shortest paths from source to any vertex.

Dijkstra's Algorithm Implementation

- C++ Implementation of Dijkstra's algorithm

```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
     // Initialize min value
     int min = INT_MAX, min_index;

     for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
                        min = dist[v], min_index = v;

     return min_index;
}
```

Find vertex with minimum distance value from the set of vertices not in the path tree

Dijkstra’s Algorithm Implementation

- C++ Implementation of Dijkstra’s algorithm

```
// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
printf("Vertex Distance from Source\n");
for (int i = 0; i < V; i++)
    printf("%d tt %d\n", i, dist[i]);
}
```

Dijkstra’s Algorithm Implementation

- C++ Implementation of Dijkstra’s algorithm

```
// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
     int dist[V];          // The output array. dist[i] will hold the shortest
                           // distance from src to i

     bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
                           // path tree or shortest distar

     // Initialize all distances as INFINITE and stpSet[] as false
     for (int i = 0; i < V; i++)
          dist[i] = INT_MAX, sptSet[i] = false;

     // Distance of source vertex from itself is always 0
     dist[src] = 0;

     // Find shortest path for all vertices
     for (int count = 0; count < V-1; count++)
     {
     // Pick the minimum distance vertex from the set of vertices not
     // yet processed. u is always equal to src in the first iteration.
     int u = minDistance(dist, sptSet);
```

output array

true if vertex in the path

Initialize

Pick minimum dist vertex

Dijkstra’s Algorithm Implementation

- C++ Implementation of Dijkstra’s algorithm

```
    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the picked vertex.
    for (int v = 0; v < V; v++)

        // Update dist[v] only if is not in sptSet, there is an edge from
        // u to v, and total weight of path from src to v through u is
        // smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]+graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}
```

mark picked as visited

distances updated for vertices not visited, u-v edge exists and weight decreases

AUTONOMOUS ROBOTS LAB

https://ide.geeksforgeeks.org/jM5ZlfsAEv

Dijkstra's Algorithm Implementation

- C++ Implementation of Dijkstra's algorithm

```
// driver program to test above function
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                        {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                        {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                        {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                        {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                        {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                        {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                        };

    dijkstra(graph, 0);

    return 0;
}
```

Bellman-Ford Algorithm

- Works for negative weights (unlike Dijkstra's Algorithm)
  - Detects a negative cycle if any exists
  - Finds shortest simple path if no negative cycle exists

  - If graph $G = (V, E)$ contains negative-weight cycle, then some shortest paths may not exist

Bellman-Ford Algorithm Pseudocode

```
for all v ∈ G                                        // G the graph G=(V,E)
        distance[v] = ∞
        previous[v] = ∅
distance[S] = 0
for all v ∈ G
        for all (u,v) ∈ G                            // (u,v) an edge
                tempDistance = distance(u) + w(u,v)
                if tempDistance < distance(v)
                        distance(v) = tempDistance
                        previous(v) = u
for all (u,v) ∈ G
        if distance(u) + w(u,v) < distance(v)
                error: negative cycle exists
return distance[], previous[]
```

Thank you