METROPOLITAN TIRANA UNIVERSITY

Course: Data Structures and Algorithms

Graph Algorithms
From Paths to Optimization

Evis Plaku

Why study graph algorithms?

METROPOLITAN TIRANA UNIVERSITY

I know a shortcut

…BFS taking every possible route

A* Algorithm

watching everyone else explore useless paths

Shortest path algorithms

Introduction to path finding problems

METROPOLITAN TIRANA UNIVERSITY

Finding the shortest or optimal path in a graph is essential in many applications

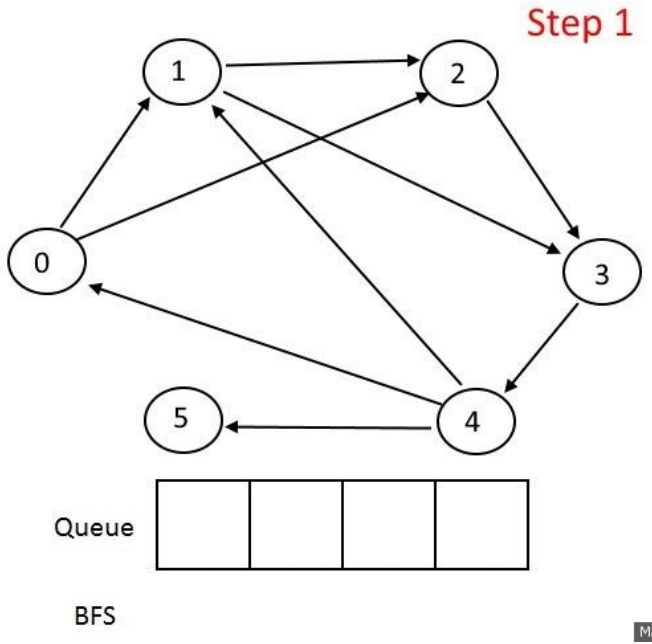
- **Graph representation**: nodes and edges represent locations and connections in real-world scenarios
- **Shortest path**: goal is to find the quickest route from a starting point to a destination
- Used in **navigation systems**, **network routing**, **robotics**, and **AI pathfinding**

Unweighted graphs: BFS for shortest path

METROPOLITAN TIRANA UNIVERSITY

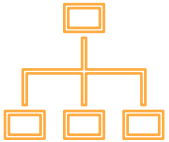
BFS explores nodes level by level, guaranteeing the shortest path in unweighted graphs

- Explores all neighbors at current depth before moving to the next level
- **No weights**: all edges are treated equally, making BFS ideal for unweighted graphs
- BFS ensures the **shortest path** by visiting nodes in increasing distance from the source



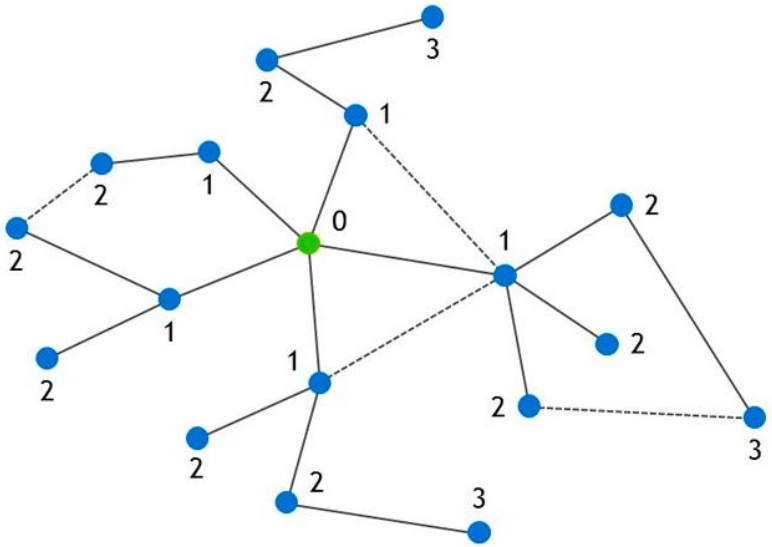
Single source shortest path problem

METROPOLITAN TIRANA UNIVERSITY



Find the shortest path from one starting node to all other nodes in a graph

- **Problem:** given a source node, find the shortest path to all other nodes
- Applies to both **directed** and **undirected graphs**, weighted or unweighted.
- Critical in **network routing**, **navigation systems**, and **AI game pathfinding**



Dijsktra’s algorithm: intuition

METROPOLITAN TIRANA UNIVERSITY

Dijkstra’s algorithm finds the shortest path by greedily selecting the closest node at each step

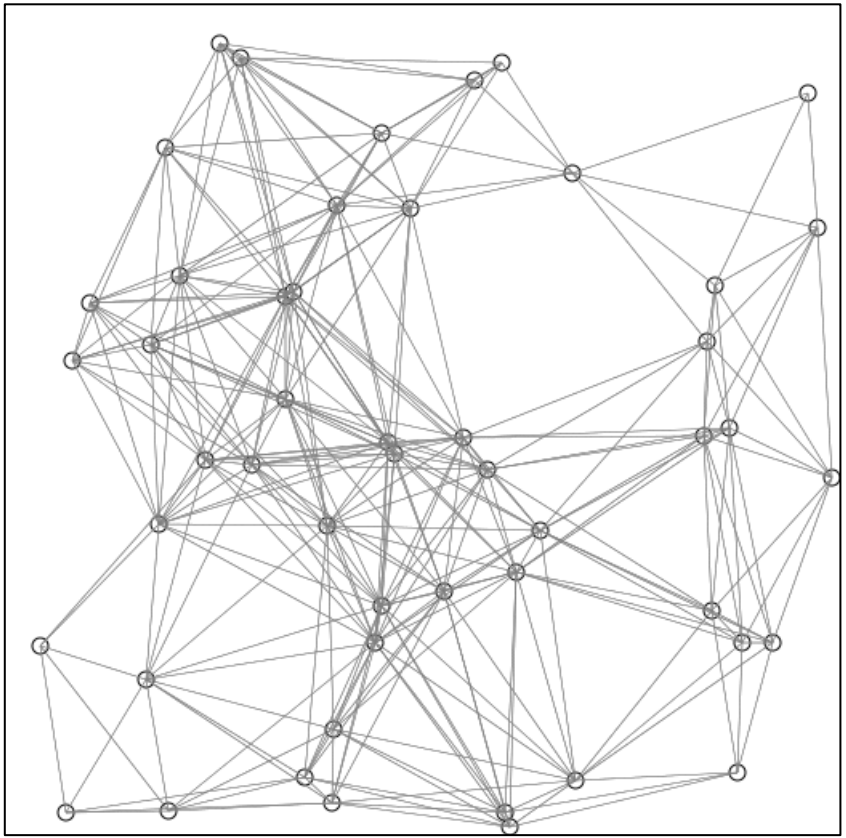
- **Greedy:** select the **nearest unvisited node** and update shortest distances iteratively
- **Priority queue:** manages nodes to ensure the closest node is processed first
- Dijkstra works when all edge weights are **non-negative** to ensure correctness

Edsger **Dijkstra** (1930 - 2002)

Perfecting oneself is as much unlearning as it is learning

Dijsktra’s algorithm: intuition

METROPOLITAN TIRANA UNIVERSITY



Dijkstra’s algorithm explores nodes in order of closest distance, updating paths until all nodes are visited

Dijkstra is optimal for graphs with **non-negative weights**

Dijsktra's algorithm

METROPOLITAN TIRANA UNIVERSITY

```
1   function Dijkstra(Graph, source):
2
3       for each vertex v in Graph.Vertices:
4           dist[v] ← INFINITY
5           prev[v] ← UNDEFINED
6           add v to Q
7       dist[source] ← 0
8
9       while Q is not empty:
10          u ← vertex in Q with minimum dist[u]
11          remove u from Q
12
13          for each neighbor v of u still in Q:
14              alt ← dist[u] + Graph.Edges(u,
v)
15              if alt < dist[v]:
16                  dist[v] ← alt
17                  prev[v] ← u
18
19      return dist[], prev[]
```

- **Input:** a graph $G = (V, E)$ consisting of a set of nodes and a set of *weighted edges*
- **Set up a map of tentative distances**
- Every node is assumed unreachable at first (distance: ∞)
- Only the source has a known distance (0)
- As the algorithm runs, we will update $dist[v]$ with shorter paths and **track** $prev[v]$ to reconstruct the optimal route later

Dijsktra's algorithm

METROPOLITAN TIRANA UNIVERSITY

```
1  function Dijkstra(Graph, source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] ← INFINITY
5          prev[v] ← UNDEFINED
6          add v to Q
7      dist[source] ← 0
8
9      while Q is not empty:
10         u ← vertex in Q with minimum dist[u]
11         remove u from Q
12
13         for each neighbor v of u still in Q:
14             alt ← dist[u] + Graph.Edges(u,
v)
15             if alt < dist[v]:
16                 dist[v] ← alt
17                 prev[v] ← u
18
19     return dist[], prev[]
```

- **Select closest node**: pick the unvisited node with the smallest known distance
- **Mark as visited:** remove it from the queue; its shortest path is now finalized
- Continue until all nodes have been visited and processed

Dijsktra's algorithm

METROPOLITAN TIRANA UNIVERSITY

```
1  function Dijkstra(Graph, source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] ← INFINITY
5          prev[v] ← UNDEFINED
6          add v to Q
7      dist[source] ← 0
8
9      while Q is not empty:
10         u ← vertex in Q with minimum dist[u]
11         remove u from Q
12
13         for each neighbor v of u still in Q:
14             alt ← dist[u] + Graph.Edges(u,
v)
15             if alt < dist[v]:
16                 dist[v] ← alt
17                 prev[v] ← u
18
19     return dist[], prev[]
```

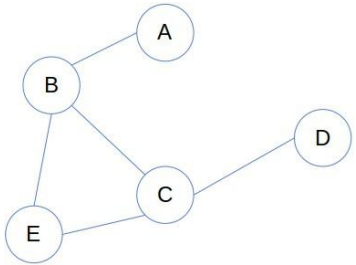
- **Explore neighbors:** for each unvisited neighbor v of the current node u, calculate a new possible distance
- **Update if shorter:** if the new distance alt is shorter than known distance $dist[v]$ update $dist[v]$
- **Track path:** set $prev[v] \mapsto u$ to remember that the shortest path to v goes through u
- **Return results:** output the shortest distance and paths from source to all other nodes

Dijsktra vs BFS: when to use each

METROPOLITAN TIRANA UNIVERSITY

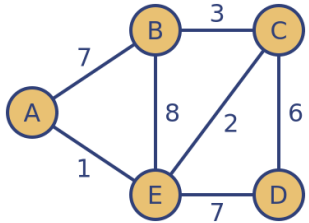
BFS

- Best used for unweighted graphs
- Shortest path = fewest edges
- Simpler and faster



Dijkstra

- Weighted graphs with non-negative edge
- Shortest path = minimal total weight
- More flexible, handles real-world distances

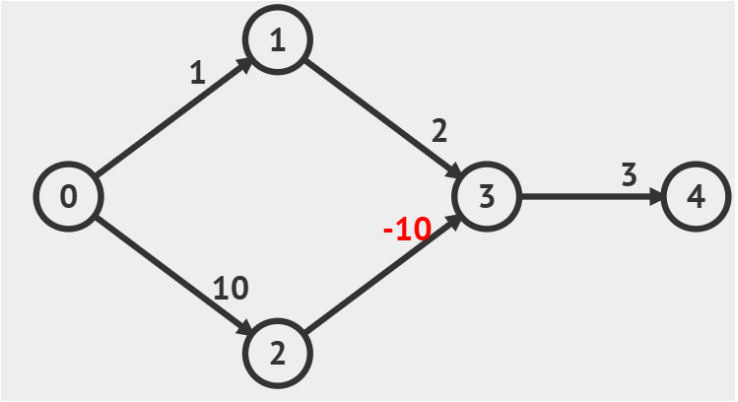


Negative weights: why Dijkstra fails?

METROPOLITAN TIRANA UNIVERSITY

Dijkstra assumes once a node's shortest path is found, it won't be improved: this breaks with negative edges

- **Greedy assumption:** Dijkstra finalizes a node too early, assuming no shorter path will appear later
- **Negative edge impact:** a cheaper path might appear **after** visiting a node, but Dijkstra won't revisit it
- Incorrect results: this can lead to **wrong shortest paths** in graphs with negative edge weights



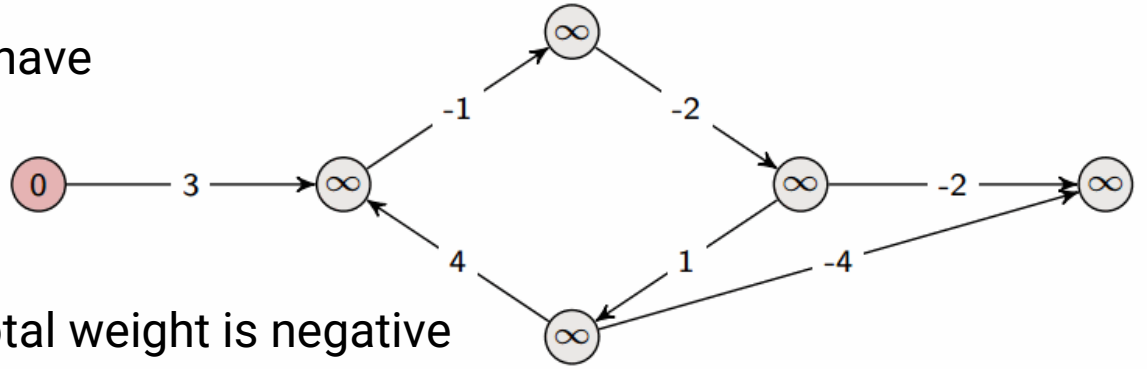
Negative weights examples: profit in financial trading, rewards in games, savings in routing, or task penalties

Bellman – Ford algorithm

METROPOLITAN TIRANA UNIVERSITY

Bellman-Ford finds shortest paths, handles negative weights, and detects negative cycles

- Works even when edges have negative values



- Identifies cycles where total weight is negative

- **Slower than Dijkstra**: but necessary for graphs with negative edges

Bellman – Ford algorithm intuition

METROPOLITAN TIRANA UNIVERSITY

```
function BellmanFord(vertices, edges, source):
    // Initialize distance and predecessor arrays
    distance := [∞] * n
    predecessor := [null] * n
    distance[source] := 0

    // Step 1: Relax edges (|V| - 1) times
    for i from 1 to |V| - 1:
        for each edge (u, v, w) in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 2: Check for negative-weight cycles
    for each edge (u, v, w) in edges:
        if distance[u] + w < distance[v]:
            // Negative cycle detected, find the cycle
            visited := [false] * n
            visited[v] := true
            while not visited[u]:
                visited[u] := true
                u := predecessor[u]
            cycle := [u]
            v := predecessor[u]
            while v != u:
                cycle := [v] + cycle
                v := predecessor[v]
            error "Negative-weight cycle detected", cycle

    return distance, predecessor
```

- **Input:** list of vertices and edges in the graph and the starting vertex

Initialization

- **Distance:** set the initial distance to all vertices as infinity (∞) except the source (0)
- **Predecessor:** set all predecessors to null as no paths are established yet
- **Source:** the distance from the source to itself if (0)

Bellman – Ford algorithm intuition

METROPOLITAN TIRANA UNIVERSITY

```
function BellmanFord(vertices, edges, source):
    // Initialize distance and predecessor arrays
    distance := [∞] * n
    predecessor := [null] * n
    distance[source] := 0

    // Step 1: Relax edges (|V| - 1) times
    for i from 1 to |V| - 1:
        for each edge (u, v, w) in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 2: Check for negative-weight cycles
    for each edge (u, v, w) in edges:
        if distance[u] + w < distance[v]:
            // Negative cycle detected, find the cycle
            visited := [false] * n
            visited[v] := true
            while not visited[u]:
                visited[u] := true
                u := predecessor[u]
            cycle := [u]
            v := predecessor[u]
            while v != u:
                cycle := [v] + cycle
                v := predecessor[v]
            error "Negative-weight cycle detected", cycle

    return distance, predecessor
```

Repeatedly update the shortest path estimates by "relaxing" all edges

- **Relax an edge:** for each edge (u, v, w) if the path from u to v is shorter than current known path, update both the distance and predecessor of v

- **Why $|V| - 1$ times**? In a graph with $|V|$ vertices, the longest path can have at most $|V| - 1$ edges

- Repeating this process ensures shortest paths are found

Bellman – Ford algorithm intuition

METROPOLITAN TIRANA UNIVERSITY

```
function BellmanFord(vertices, edges, source):
    // Initialize distance and predecessor arrays
    distance := [∞] * n
    predecessor := [null] * n
    distance[source] := 0

    // Step 1: Relax edges (|V| - 1) times
    for i from 1 to |V| - 1:
        for each edge (u, v, w) in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 2: Check for negative-weight cycles
    for each edge (u, v, w) in edges:
        if distance[u] + w < distance[v]:
            // Negative cycle detected, find the cycle
            visited := [false] * n
            visited[v] := true
            while not visited[u]:
                visited[u] := true
                u := predecessor[u]
            cycle := [u]
            v := predecessor[u]
            while v != u:
                cycle := [v] + cycle
                v := predecessor[v]
            error "Negative-weight cycle detected", cycle

    return distance, predecessor
```

Repeatedly update the shortest path estimates by "relaxing" all edges

- **Relax an edge:** for each edge (u, v, w) if the path from u to v is shorter than current known path, update both the distance and predecessor of v
- **Why $|V| - 1$ times**? In a graph with $|V|$ vertices, the longest path can have at most $|V| - 1$ edges
- Repeating this process ensures shortest paths are found

Bellman – Ford algorithm intuition

METROPOLITAN TIRANA UNIVERSITY

```
function BellmanFord(vertices, edges, source):
    // Initialize distance and predecessor arrays
    distance := [∞] * n
    predecessor := [null] * n
    distance[source] := 0

    // Step 1: Relax edges (|V| - 1) times
    for i from 1 to |V| - 1:
        for each edge (u, v, w) in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 2: Check for negative-weight cycles
    for each edge (u, v, w) in edges:
        if distance[u] + w < distance[v]:
            // Negative cycle detected, find the cycle
            visited := [false] * n
            visited[v] := true
            while not visited[u]:
                visited[u] := true
                u := predecessor[u]
            cycle := [u]
            v := predecessor[u]
            while v != u:
                cycle := [v] + cycle
                v := predecessor[v]
            error "Negative-weight cycle detected", cycle

    return distance, predecessor
```

Detect if any edge can still be relaxed, indicating a negative-weight cycle

- If any edge (u, v, w) can still be relaxed, it suggest a negative – weight cycle exists
- If such cycle exists, backtrack from v to find the vertices involved in the cycle
- The $visited$ array is used to mark all visited nodes during backtrack to avoid infinite loops and detect where the cycle starts

Bellman – Ford algorithm intuition

METROPOLITAN TIRANA UNIVERSITY

```
function BellmanFord(vertices, edges, source):
    // Initialize distance and predecessor arrays
    distance := [∞] * n
    predecessor := [null] * n
    distance[source] := 0

    // Step 1: Relax edges (|V| - 1) times
    for i from 1 to |V| - 1:
        for each edge (u, v, w) in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 2: Check for negative-weight cycles
    for each edge (u, v, w) in edges:
        if distance[u] + w < distance[v]:
            // Negative cycle detected, find the cycle
            visited := [false] * n
            visited[v] := true
            while not visited[u]:
                visited[u] := true
                u := predecessor[u]
            cycle := [u]
            v := predecessor[u]
            while v != u:
                cycle := [v] + cycle
                v := predecessor[v]
            error "Negative-weight cycle detected", cycle

    return distance, predecessor
```

Track and build the cycle

- Once a cycle is detected, trace the cycle path using the $predecessor$ links
- Keep adding nodes to $cycle$ array until you loop back to the starting node u
- Once the cycle is fully traced, report it as a negative-weight cycle and throw an error
- The algorithm returns the **shortest path distances** and the **predecessors** for each vertex or reports **negative-weight cycle** if detected

Dijkstra vs Bellman – Ford

METROPOLITAN TIRANA UNIVERSITY

Feature	Dijkstra	Bellman – Ford
Edge weights	Works only with **non-negative weights**	Handles **negative weights**
Time Complexity	O(V log V) with a priority queue	O(V * E) (slower)
Negative cycles	**Does not detect** negative cycles	D**etects** negative-weight cycles
Efficiency	More efficient for graphs with positive weights	Suitable for graphs with negative weights
Ideal Use Case	Shortest paths in **positive weight graphs**	Shortest paths with **negative weights**

Heuristic search

Informed search through heuristics

METROPOLITAN TIRANA UNIVERSITY

Heuristics guide search algorithms by estimating how close a state is to the goal

- **Uninformed search** explores blindly. It has no sense of direction or how far the goal is
- **Heuristics add intelligence** by using domain knowledge to rank possible paths
- Solutions are found faster and more efficiently

Informed search through heuristics

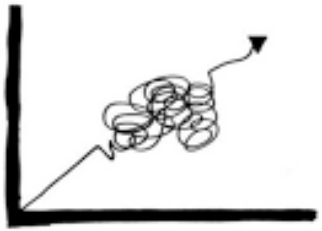
METROPOLITAN TIRANA UNIVERSITY

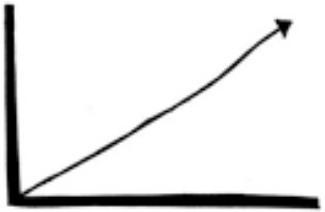
A heuristic is a function that estimates the cost from a node to the goal

- **Notation**: $h(n)$ is the estimated cost from node n to the goal

- Design: a good heuristic is **admissible** (never overestimates) and efficient to compute

- Example of admissible and efficient heuristic: **straight-line distance** (Euclidean distance) to the goal

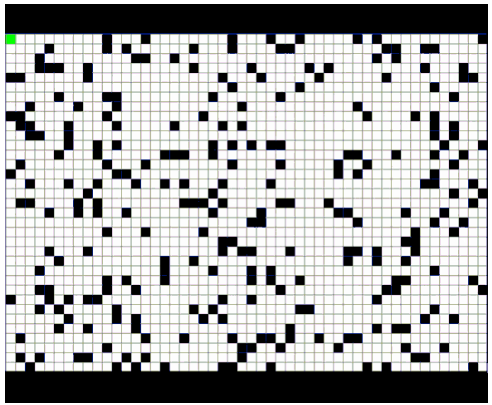


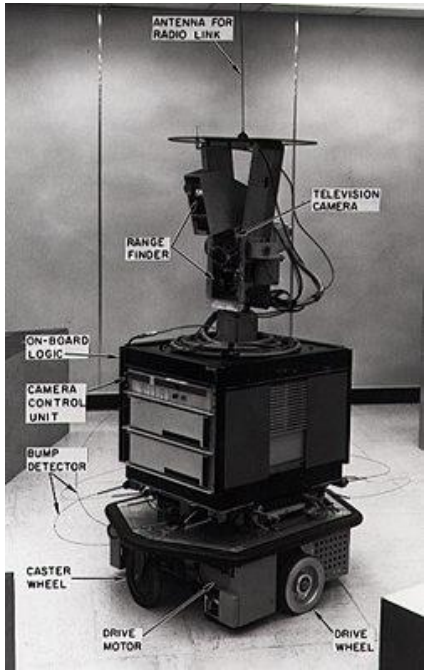


A* search: intuition

A* finds the shortest path by combining actual cost so far and estimated cost to goal

- Select nodes using the best balance of progress and promise
- **Efficient and optimal** when the heuristic is admissible and consistent





A* was invented by researchers working on Shakey the Robot's path planning

A* search key components

METROPOLITAN TIRANA UNIVERSITY

Actual cost $g(n)$

- Represents the **cost** to reach node n from the **start** node
- **Cumulative** cost accumulated by following the path from start to n
- Helps ensure the algorithm chooses the shortest path

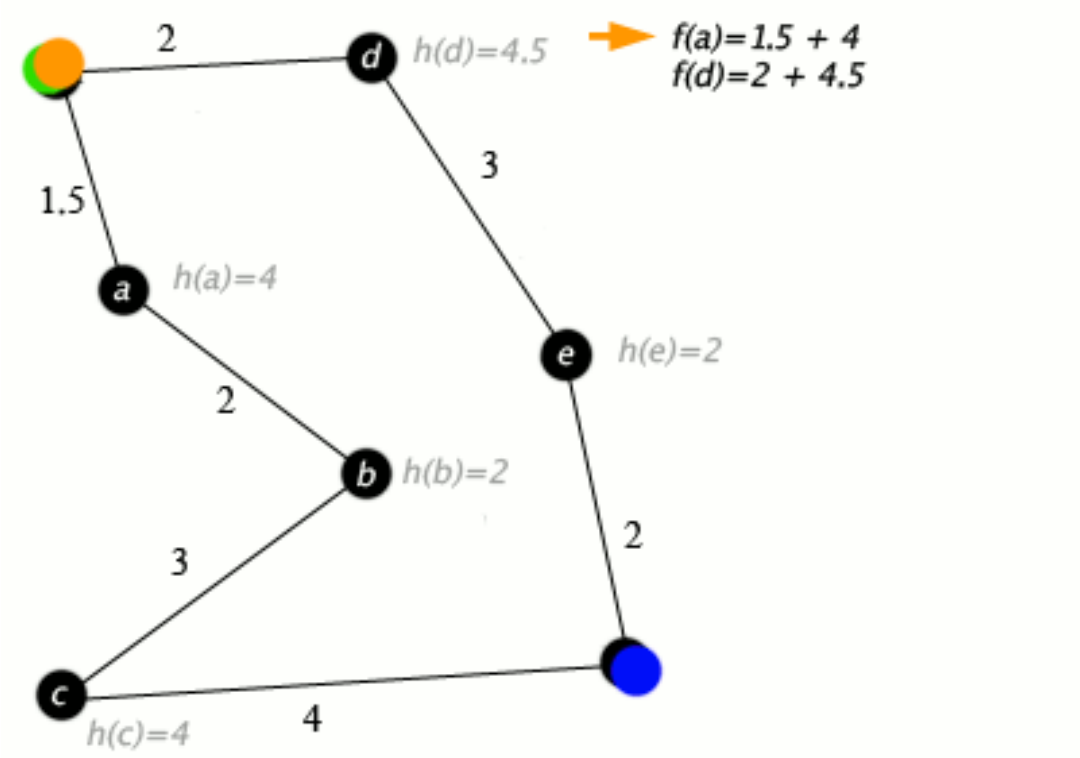
Heuristic estimate $h(n)$

- Represents the **estimated cost** from node n to the **goal**
- Uses domain knowledge (like straight-line distance) to guide search
- A good heuristic is admissible, meaning it never overestimates the true cost

$$f(n) = g(n) + h(n)$$

- Total estimated cost combines the actual cost with heuristic estimate
- Helps the algorithm balance between **exploring** already explored paths (low $g(n)$) and promising paths (low $h(n)$).
- Selects the node with the lowest $f(n)$

A* algorithm



Key: green: start; blue: goal; orange: visited

An example of an A* algorithm in action where nodes are cities connected with roads and $h(x)$ is the straight-line distance to the target point

A* algorithm

METROPOLITAN TIRANA UNIVERSITY

```
function A*(start, goal):
    open_list := priority queue with start node
    closed_list := empty set

    g[start] := 0                  # Cost from start to start
    h[start] := heuristic(start)   # Estimated cost from start to goal
    f[start] := g[start] + h[start] # Total estimated cost

    while open_list is not empty:
        current := node in open_list with the lowest f[current]
        if current == goal:
            return reconstruct_path(current) # Path from start to goal

        remove current from open_list
        add current to closed_list

        for each neighbor of current:
            if neighbor in closed_list:
                continue

            tentative_g := g[current] + distance(current, neighbor)

            if neighbor not in open_list:
                add neighbor to open_list
            else if tentative_g >= g[neighbor]:
                continue

            g[neighbor] := tentative_g
            h[neighbor] := heuristic(neighbor)
            f[neighbor] := g[neighbor] + h[neighbor]
            set neighbor's parent to current

    return failure  # If no path found
```

- The **open list** stores nodes that are yet to be evaluated (implemented as priority queue)
- **Start node** is initially added to this list because it's the first node to explore
- The **closed list** tracks nodes that have already been fully evaluated (they won't be revisited)

- $\boldsymbol{g(n)}$ represents the **actual cost** to reach node n from the start (0 for start)
- $\boldsymbol{h(n)}$ is the **heuristic estimate** of the cost to reach the goal from node n (e.g., straight line)
- $\boldsymbol{f(n)}$ is the total estimated cost, combining $g(n)$ and $h(n)$

A* algorithm

METROPOLITAN TIRANA UNIVERSITY

```
function A*(start, goal):
    open_list := priority queue with start node
    closed_list := empty set

    g[start] := 0                  # Cost from start to start
    h[start] := heuristic(start)   # Estimated cost from start to goal
    f[start] := g[start] + h[start] # Total estimated cost

    while open_list is not empty:
        current := node in open_list with the lowest f[current]
        if current == goal:
            return reconstruct_path(current) # Path from start to goal

        remove current from open_list
        add current to closed_list

        for each neighbor of current:
            if neighbor in closed_list:
                continue

            tentative_g := g[current] + distance(current, neighbor)

            if neighbor not in open_list:
                add neighbor to open_list
            else if tentative_g >= g[neighbor]:
                continue

            g[neighbor] := tentative_g
            h[neighbor] := heuristic(neighbor)
            f[neighbor] := g[neighbor] + h[neighbor]
            set neighbor's parent to current

    return failure  # If no path found
```

- The loop continues as long as there are nodes left to explore
- Select the node current from the **open list** that has the **lowest** $f(n)$ value (explore most promising node first)
- If the **current node** is the goal, **return the path** from the start to the goal
- After exploring the **current node**, remove it from the **open list** since it's now fully evaluated
- Add the **current node** to the **closed list** to indicate that it's been fully evaluated

A* algorithm

METROPOLITAN TIRANA UNIVERSITY

```
function A*(start, goal):
    open_list := priority queue with start node
    closed_list := empty set

    g[start] := 0                    # Cost from start to start
    h[start] := heuristic(start)     # Estimated cost from start to goal
    f[start] := g[start] + h[start] # Total estimated cost

    while open_list is not empty:
        current := node in open_list with the lowest f[current]
        if current == goal:
            return reconstruct_path(current) # Path from start to goal

        remove current from open_list
        add current to closed_list

        for each neighbor of current:
            if neighbor in closed_list:
                continue

            tentative_g := g[current] + distance(current, neighbor)

            if neighbor not in open_list:
                add neighbor to open_list
            else if tentative_g >= g[neighbor]:
                continue

            g[neighbor] := tentative_g
            h[neighbor] := heuristic(neighbor)
            f[neighbor] := g[neighbor] + h[neighbor]
            set neighbor's parent to current

    return failure  # If no path found
```

- For each **neighbor** of the **current node**, the algorithm considers the possibility of reaching that neighbor from the current node
- If the **neighbor** has already been fully evaluated (i.e., it is in the **closed list**), skip it and move to the next neighbor
- $tentative\_g$ calculates the new cost to reach the **neighbor** from the start node by adding the cost of reaching the current node and the cost to move from current node to neighbor
- If the neighbor is undiscovered, add it to the open list. Otherwise, skip if cost is higher

A* algorithm

METROPOLITAN TIRANA UNIVERSITY

```
function A*(start, goal):
    open_list := priority queue with start node
    closed_list := empty set

    g[start] := 0                  # Cost from start to start
    h[start] := heuristic(start)   # Estimated cost from start to goal
    f[start] := g[start] + h[start] # Total estimated cost

    while open_list is not empty:
        current := node in open_list with the lowest f[current]
        if current == goal:
            return reconstruct_path(current) # Path from start to goal

        remove current from open_list
        add current to closed_list

        for each neighbor of current:
            if neighbor in closed_list:
                continue

            tentative_g := g[current] + distance(current, neighbor)

            if neighbor not in open_list:
                add neighbor to open_list
            else if tentative_g >= g[neighbor]:
                continue

            g[neighbor] := tentative_g
            h[neighbor] := heuristic(neighbor)
            f[neighbor] := g[neighbor] + h[neighbor]
            set neighbor's parent to current

    return failure  # If no path found
```

- Update the **g value** of the **neighbor** with the newly calculated cost from the start
- Update the **h value** of the **neighbor** with its heuristic estimate to the goal
- Update the **f value** of the **neighbor** as the sum of its **g** and **h** values
- Set the **current node** as the **parent** of the **neighbor** to trace the path late

The algorithm terminates
when the goal node is reached
or the open list is empty

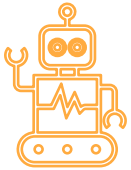
A* applications

METROPOLITAN TIRANA UNIVERSITY

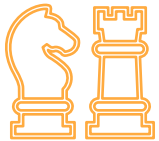
Pathfinding in Video Games

Navigation Systems

Robotics



AI in Puzzles



Autonomous Vehicles

Network Routing

Quote of the Week

METROPOLITAN TIRANA UNIVERSITY

There’s no shortcut to success …

except in graphs, where finding the shortest path is the goal!