Introduction to Artificial Intelligence

Lecture 12: Intelligent search

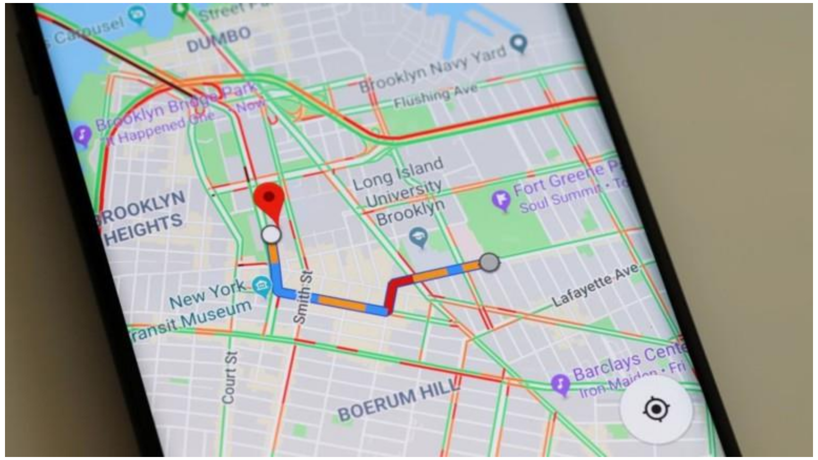
October 16, 2025

Intelligent search

- Applications
- Search algorithms

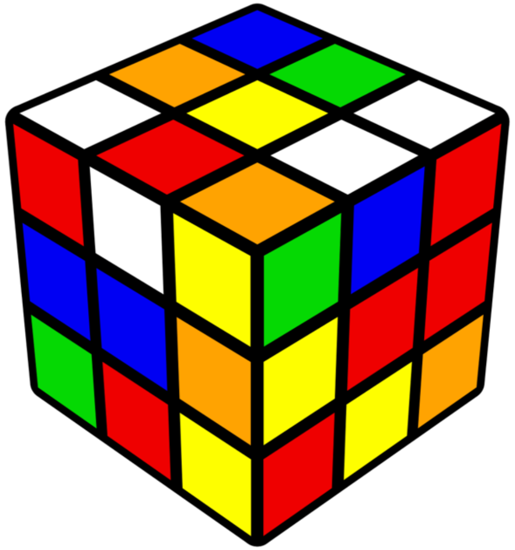
Application: route finding

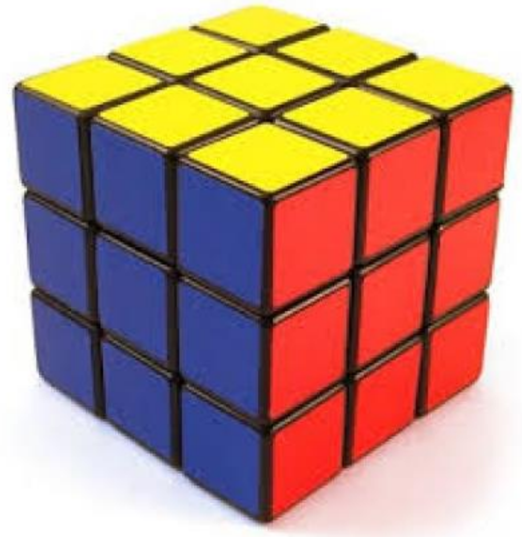
- Find optimal route among billions of road segments
- Need to search among a large number of possible routes
  - The search space becomes more complex as we go from A to a farther place B
- Robot motion planning



Application: solving puzzles

- Objective: reach a certain configuration
- Example: playing rubik's cube





Application: playing games

- AlphaGo is a computer program developed by DeepMind to play the master go game
- AlphaGo was able to defeat world champion Lee Sedol using sophisticated **tree search** combined with neural networks
  - The game tree for Go has $\sim 10^{170}$ possible positions, which are huge

Other search problems

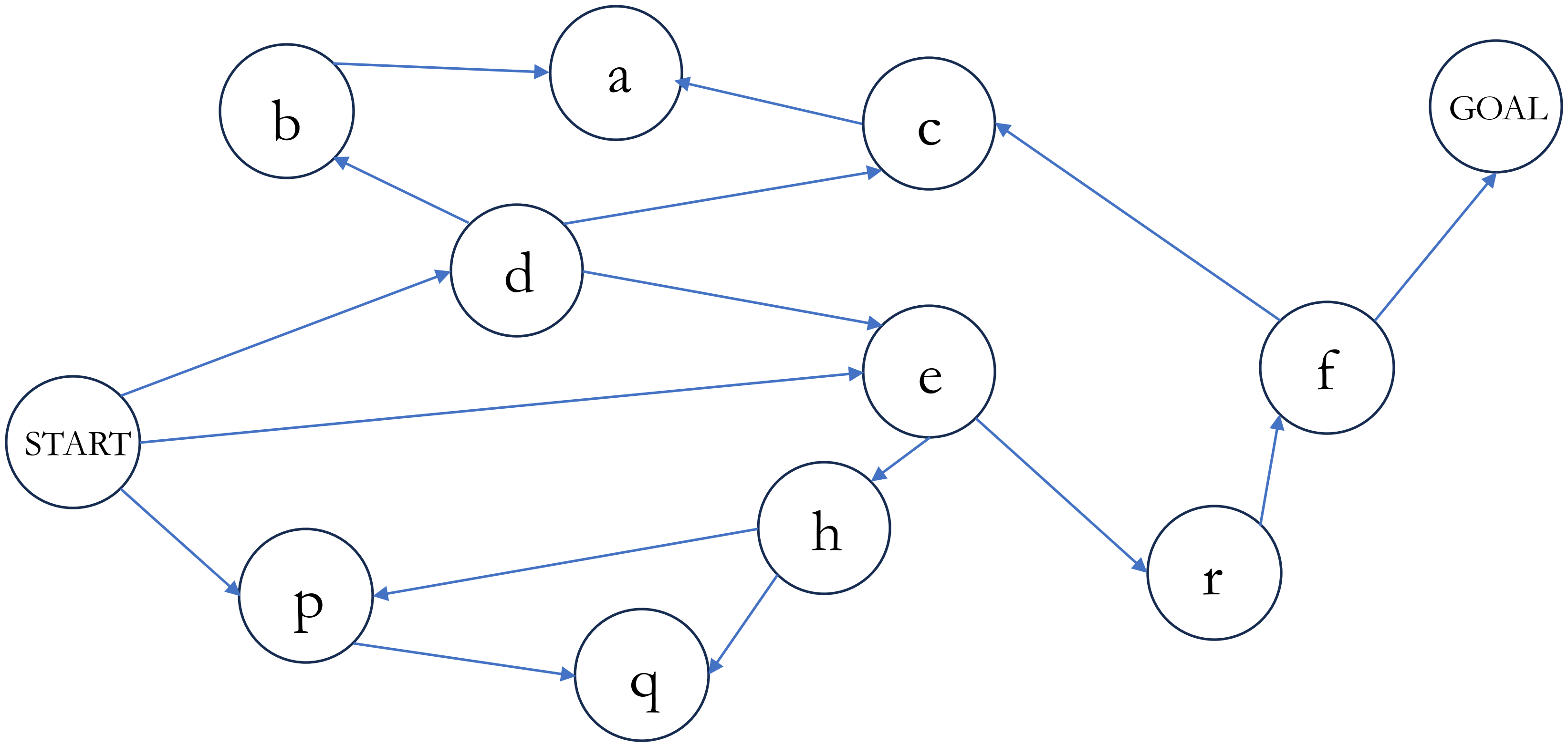
- **Route navigation:** Google maps find the optimal route
  - **Robotics:** Path planning for self-driving cars

- **AI reasoning:** Theorem proving, puzzle solving
  - **Large language models:** Chain of thought fine-tuning and prompting

The search problem

- **State:** Description of the world at a moment
  - **Initial State:** Where the agent starts in
- **Actions:** What agent can do in state s
- **Transition model:** Maps a state and action to a resulting state
- **Goal state:** Is this the goal?
  - **Action cost:** Numerical cost of applying an action

An example



- Cost = 1 for all transitions

Example problems

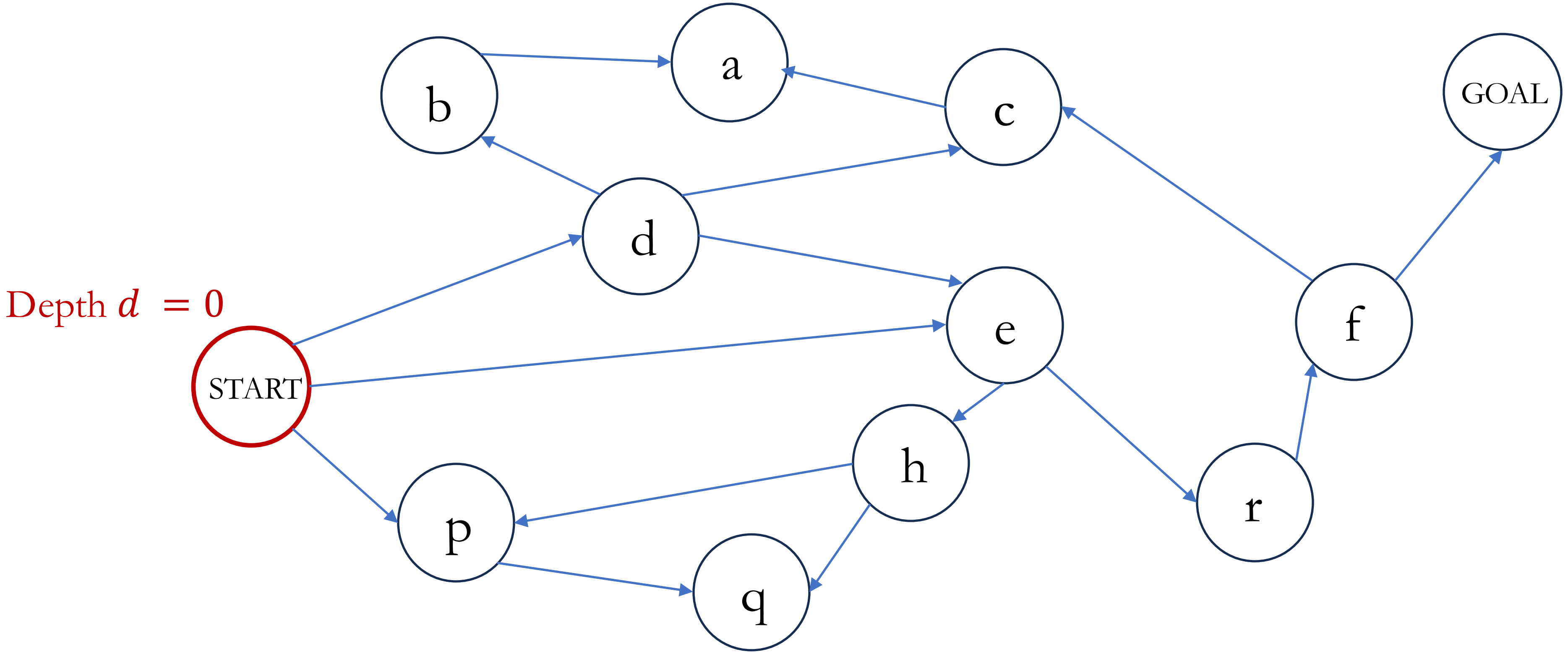
- **Goal state:** Navigate to the Student Center
  - **Initial state:** the Library
  - **State space:** All possible locations on campus
- **Actions:** Move North, South, East, West
- **Transition model:** at a current state, make a decision about the next action to take
  - Distance, obstacles, and delivery time
- **Total cost:** Distance traveled or time taken

Search algorithms

Strategy	Frontier Type
Breadth-first search	First in first out queue
Depth-first search	Last in first out stack
Uniform cost search	Priority queue

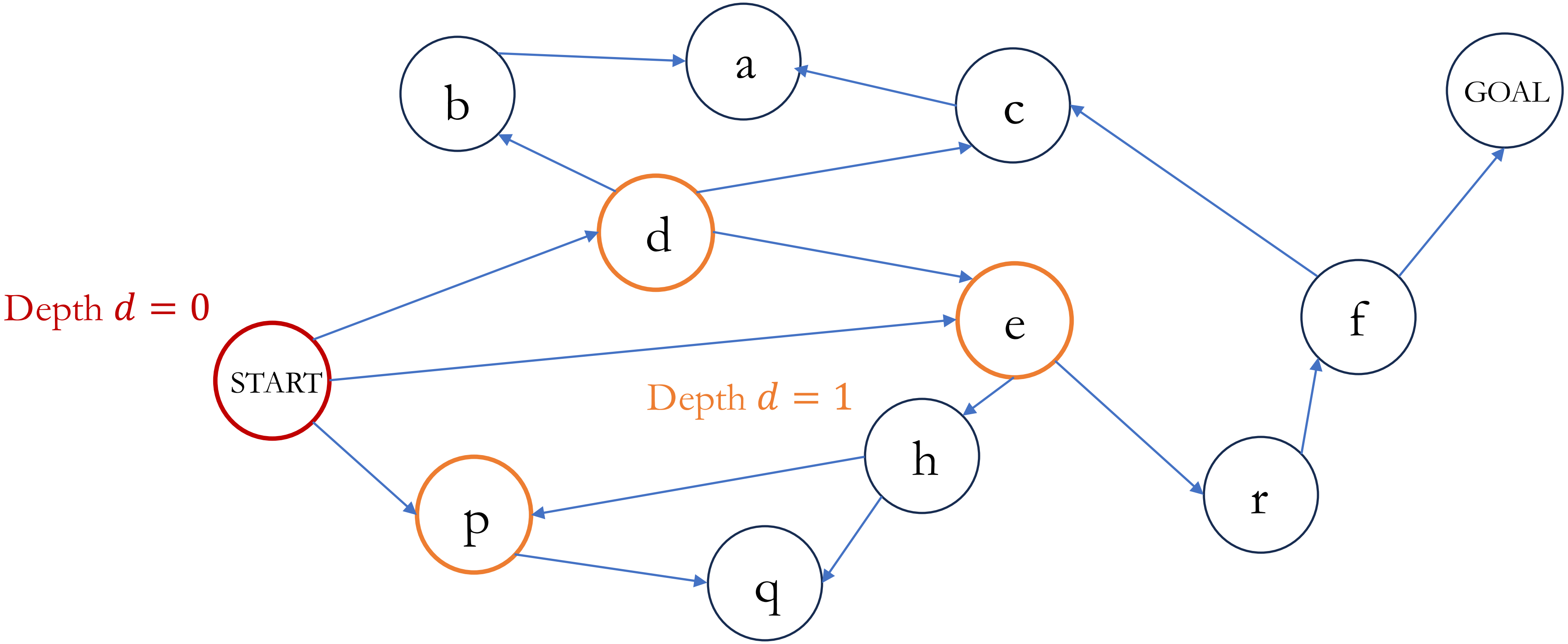
An illustration of breadth-first search

- Start: depth is zero



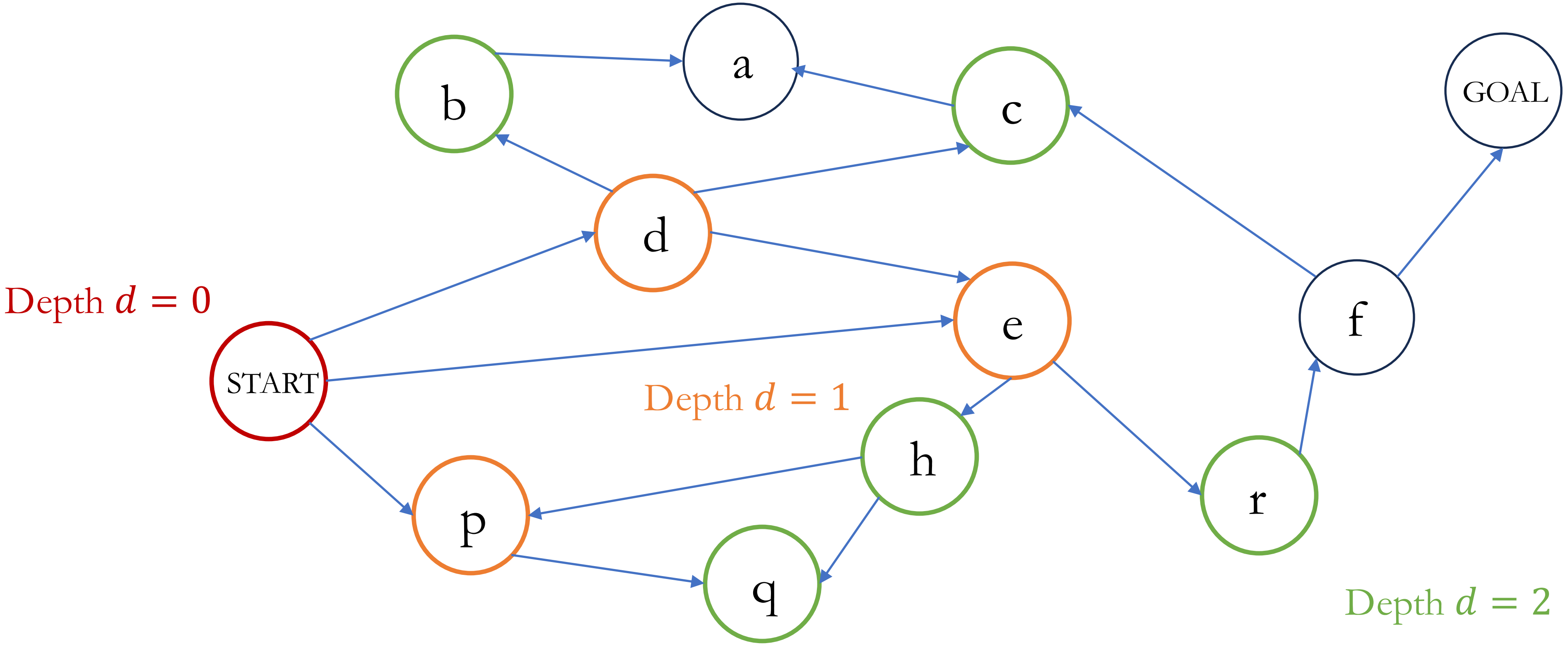
An illustration of breadth-first search

- Branch out to depth one



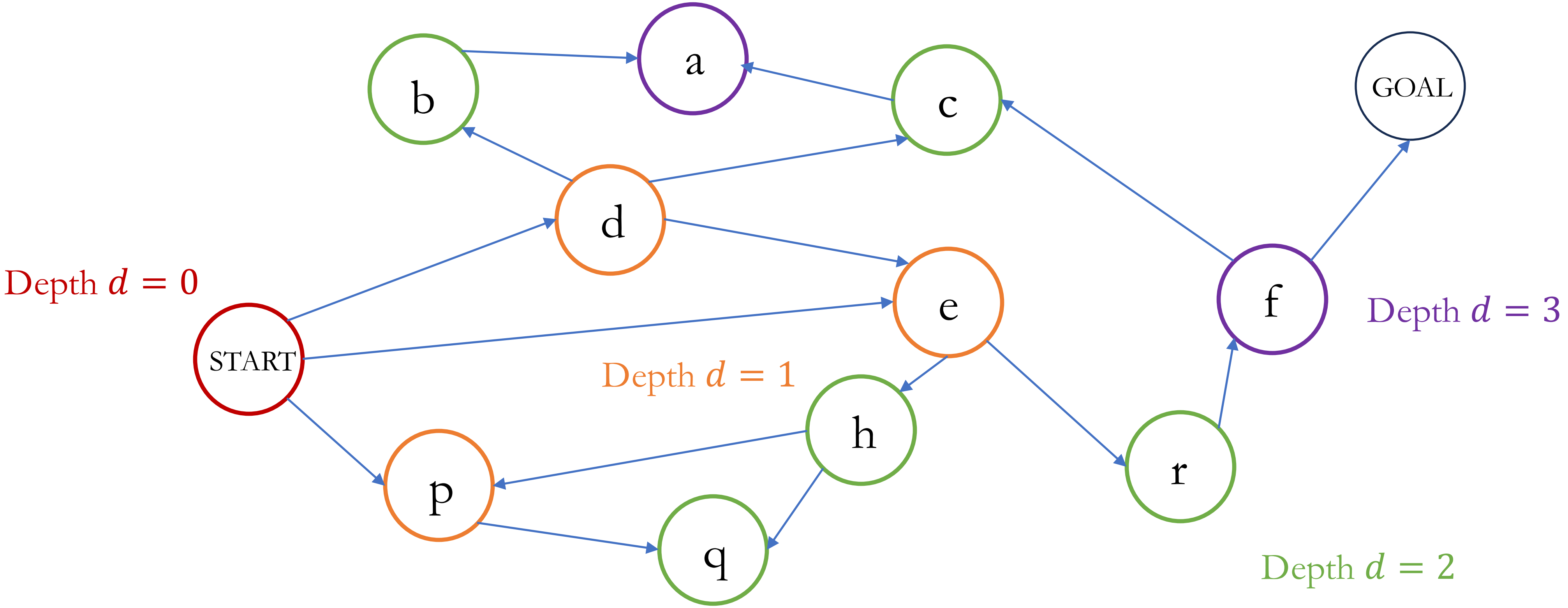
An illustration of breadth-first search

- Next explore depth two



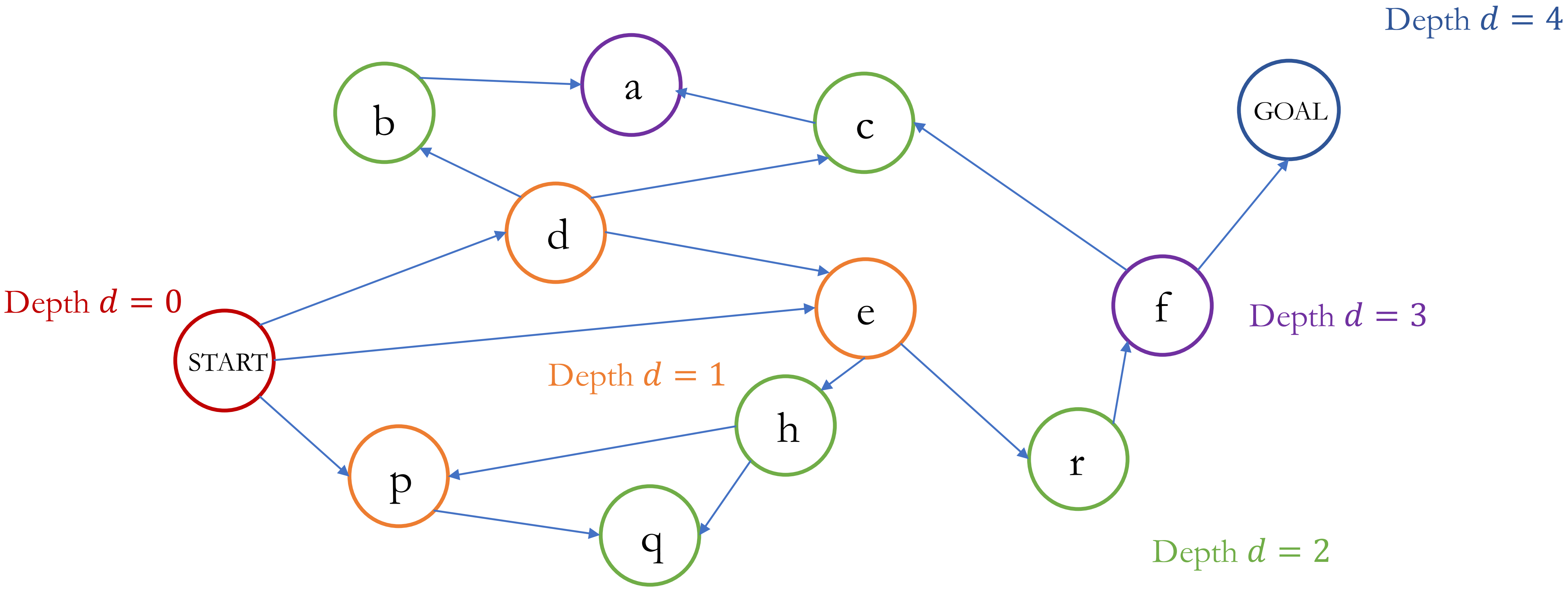
An illustration of breadth-first search

- Next: depth three

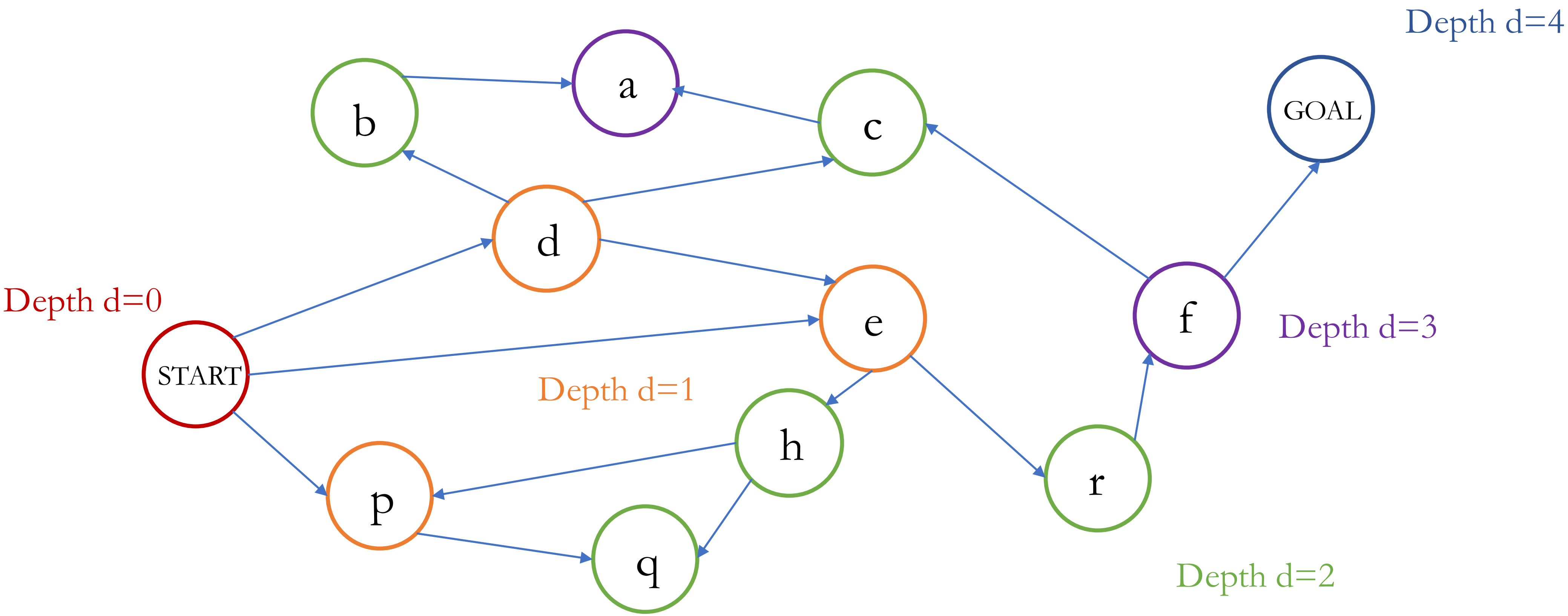


An illustration of breadth-first search

- Finally, reaching the goal



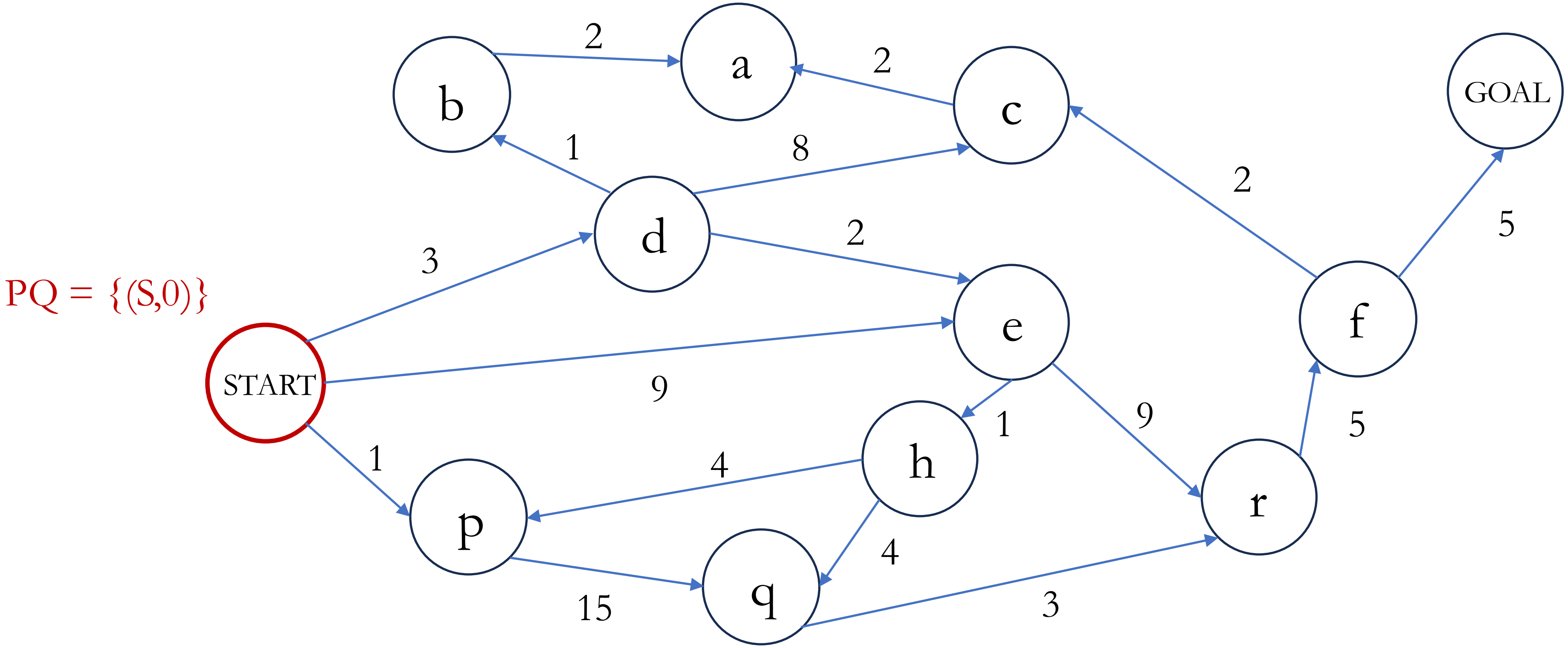
An illustration of breadth-first search



- **Guarantees shortest path** (by number of steps)
- **Memory intensive:** must store entire frontier
- **When to use:** When solution is shallow and step costs are uniform

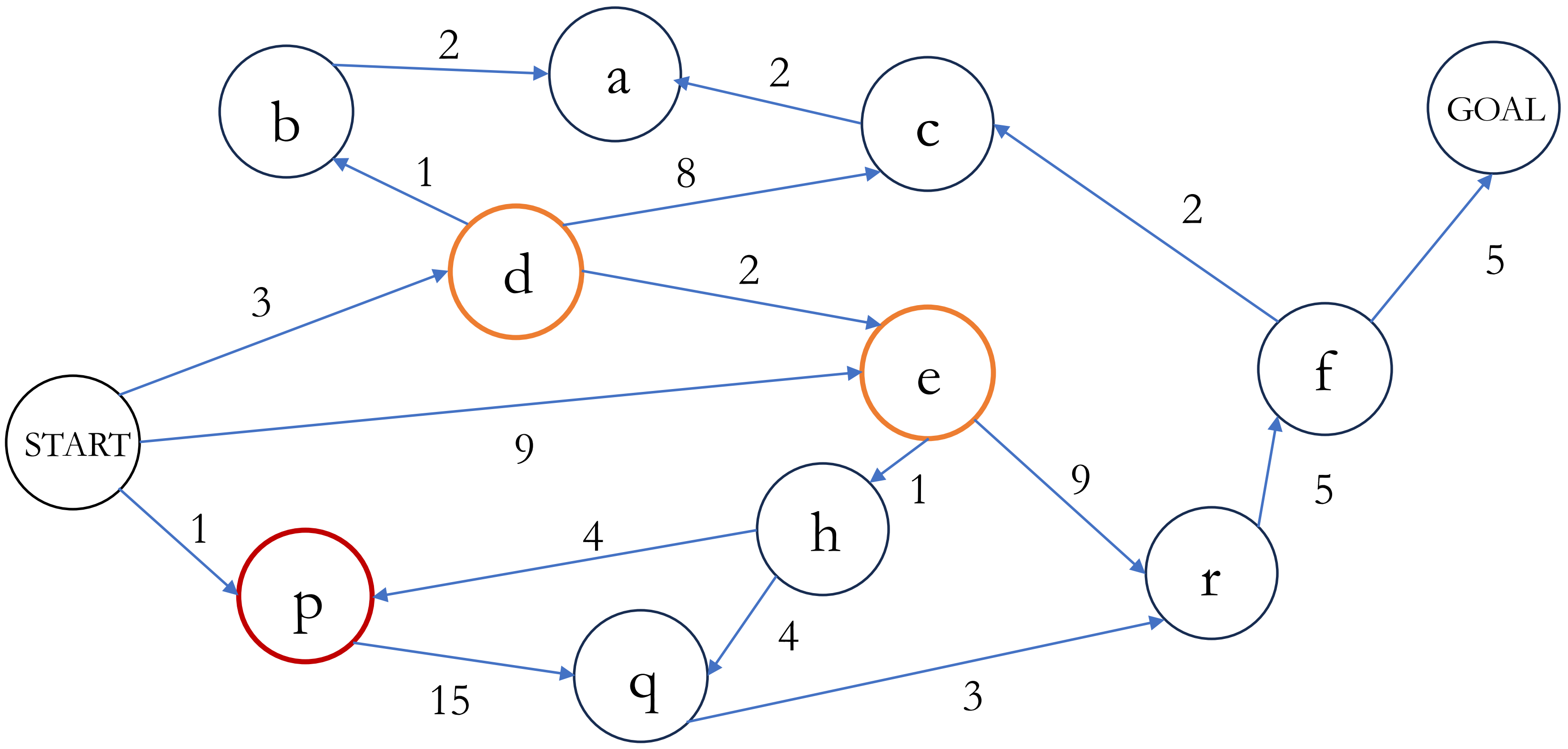
Uniform cost search

- A conceptually simple BFS approach when actions have different costs
- It uses a priority queue: pop least-cost state, add successors of that state



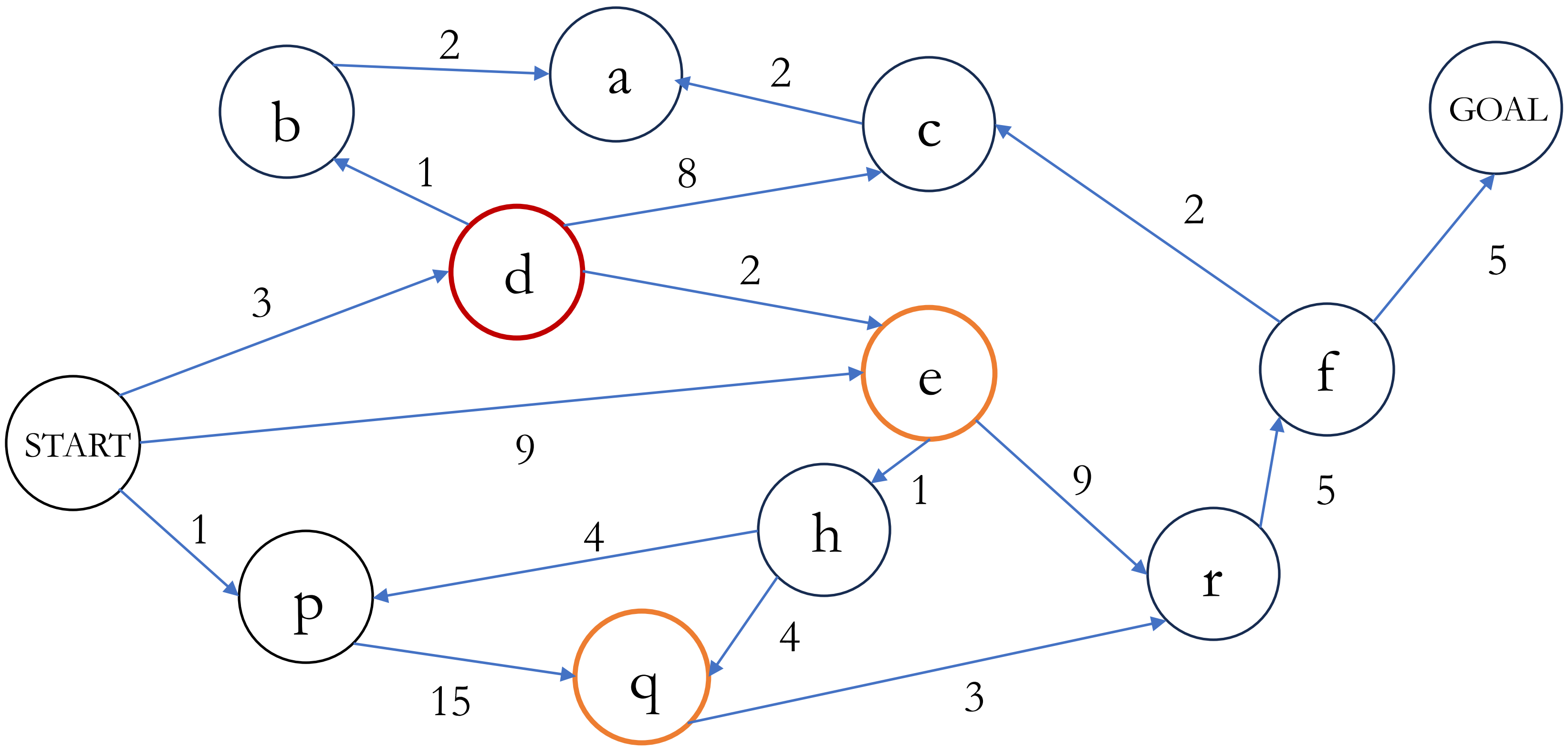
Uniform cost search

- Add three neighbors of the start state in the queue: PQ = {(p,1), (d,3), (e,9)}



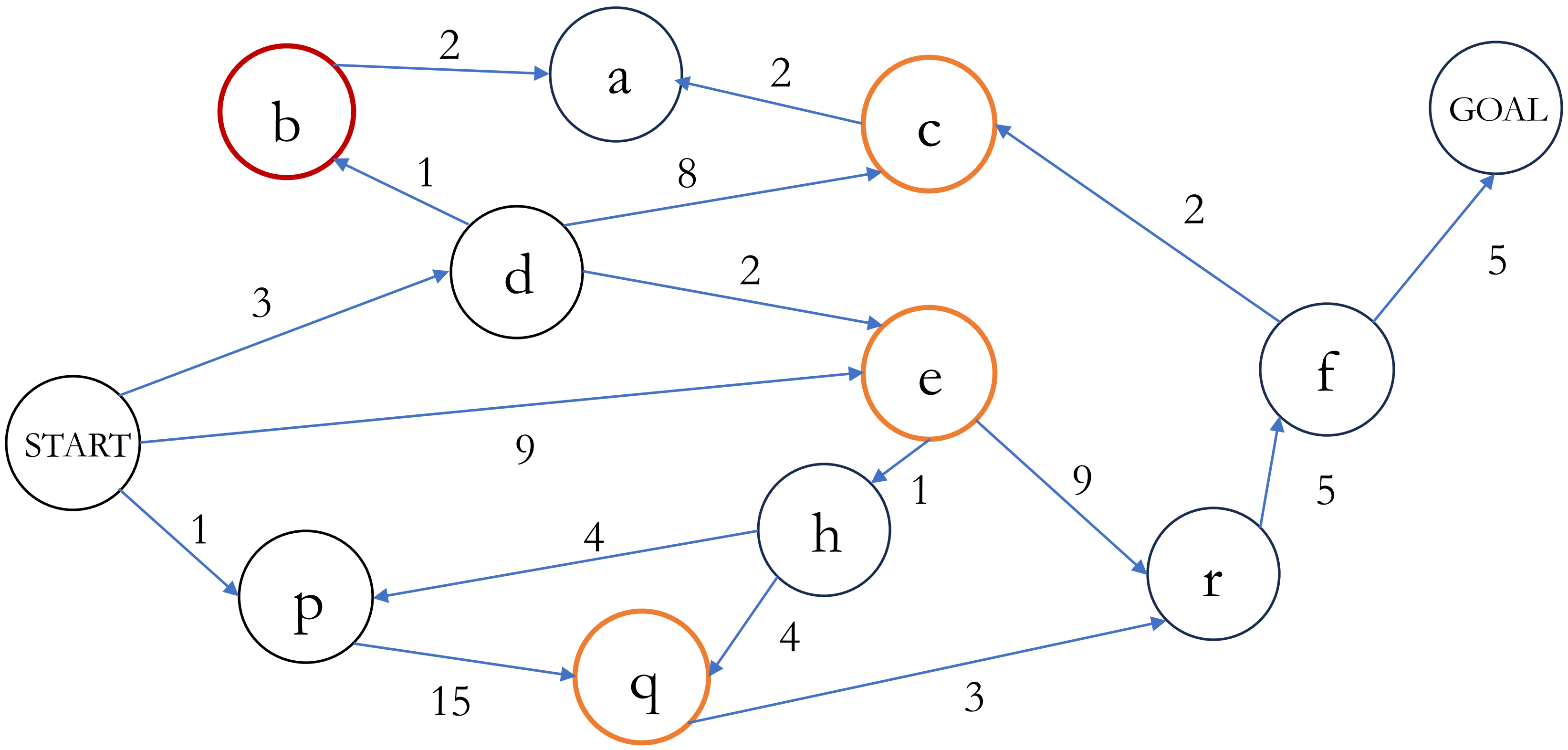
Uniform cost search

- From the minimum cost state, add the next state: PQ = {(d,3), (e,9), (q,16)}



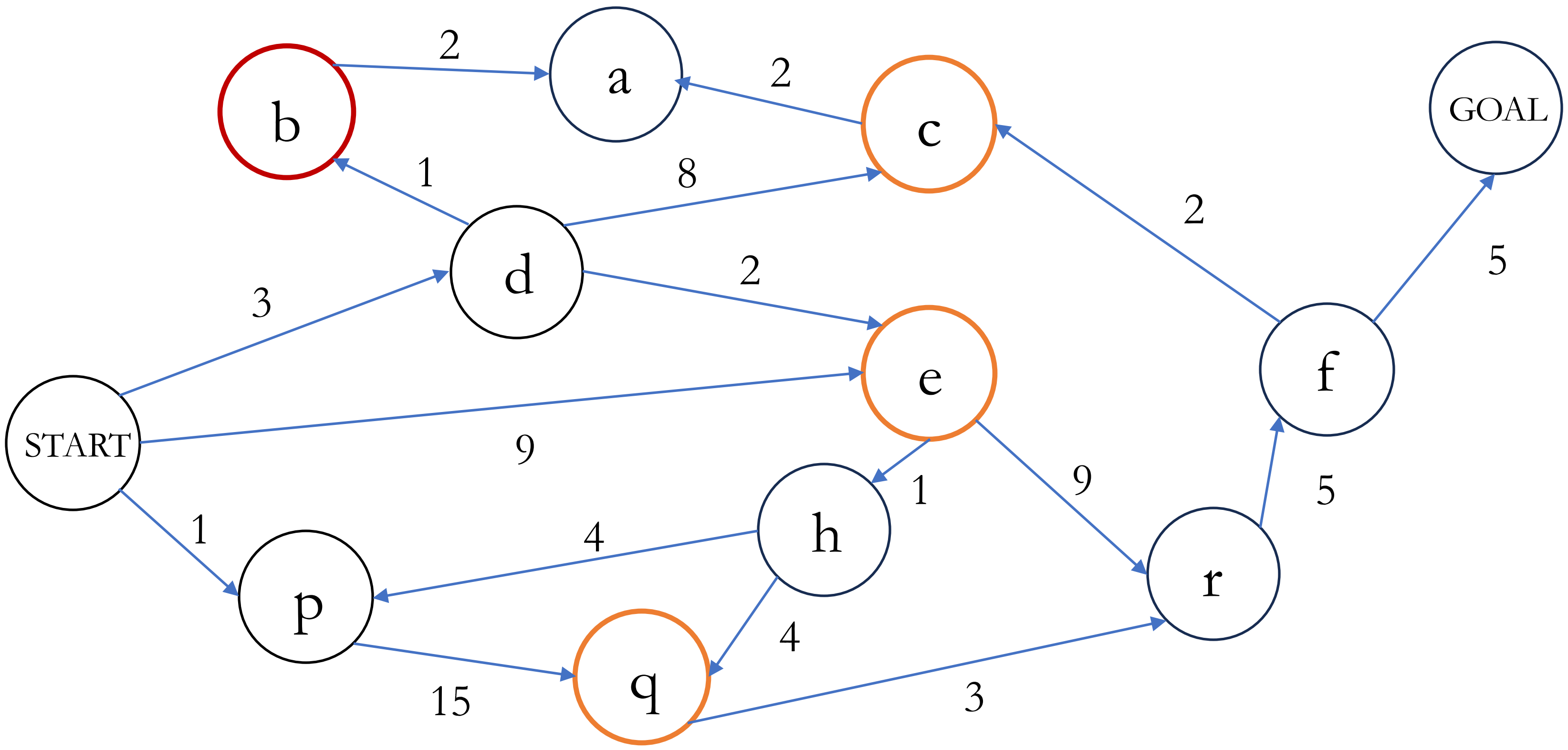
Uniform cost search

- Illustration of another step: PQ = {(b,4), (e,5), (e,9), (c, 11), (q,16)}



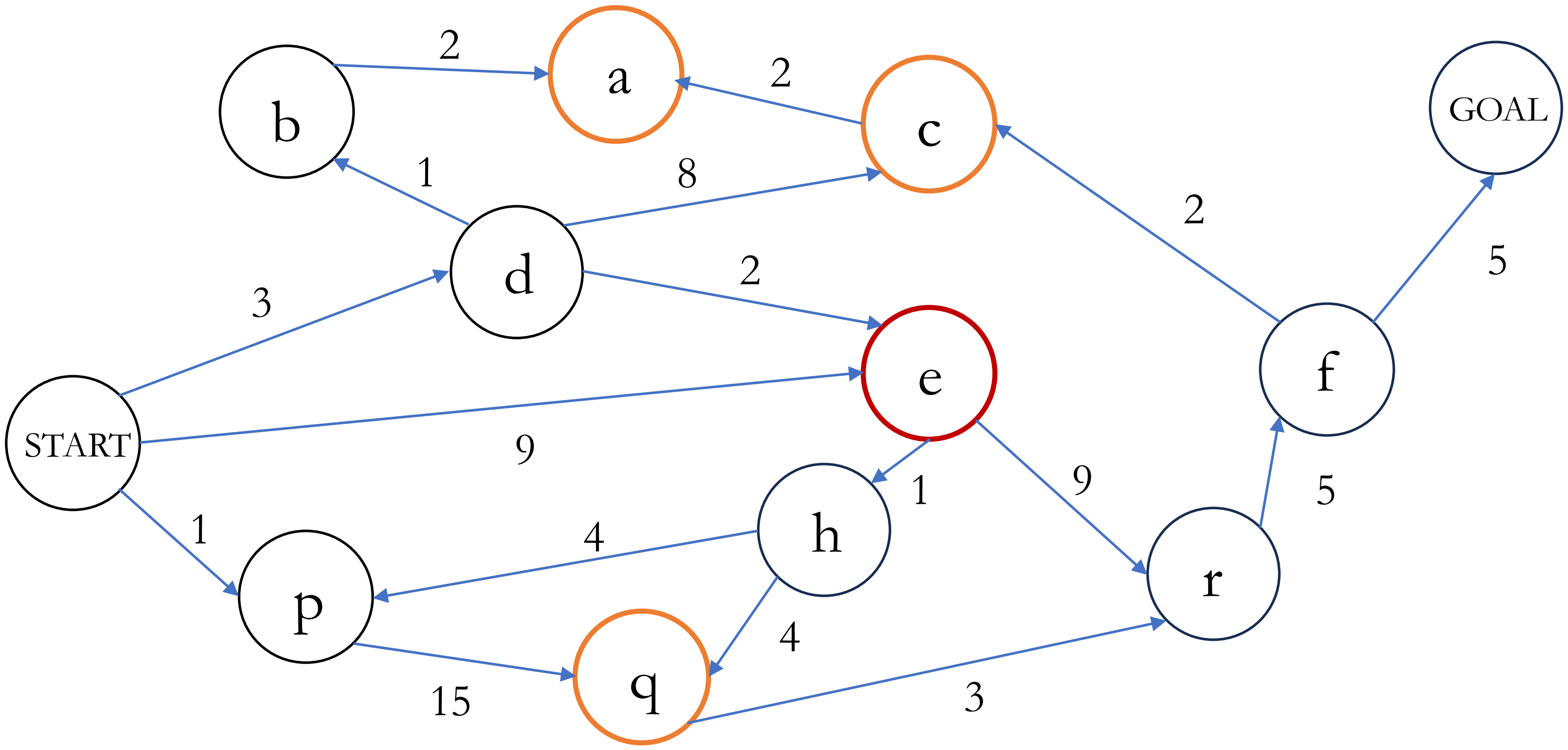
Uniform cost search

- Iterate for another step: PQ = {(b,4), (e,5), (c,11), (q,16)}
  - Note that we update e's priority here



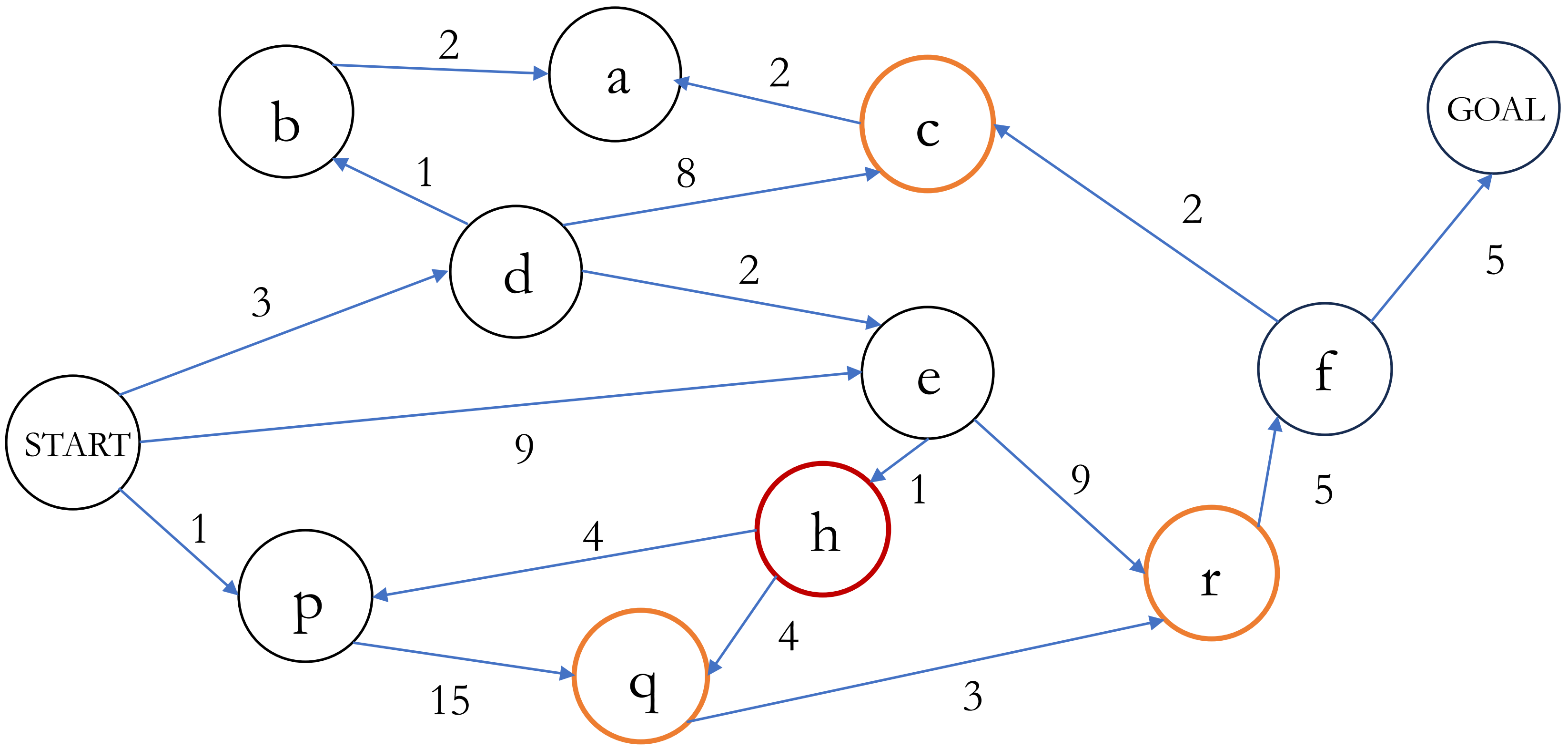
Uniform cost search

- Next, PQ = {(e,5), (a,6), (c,11), (q,16)}



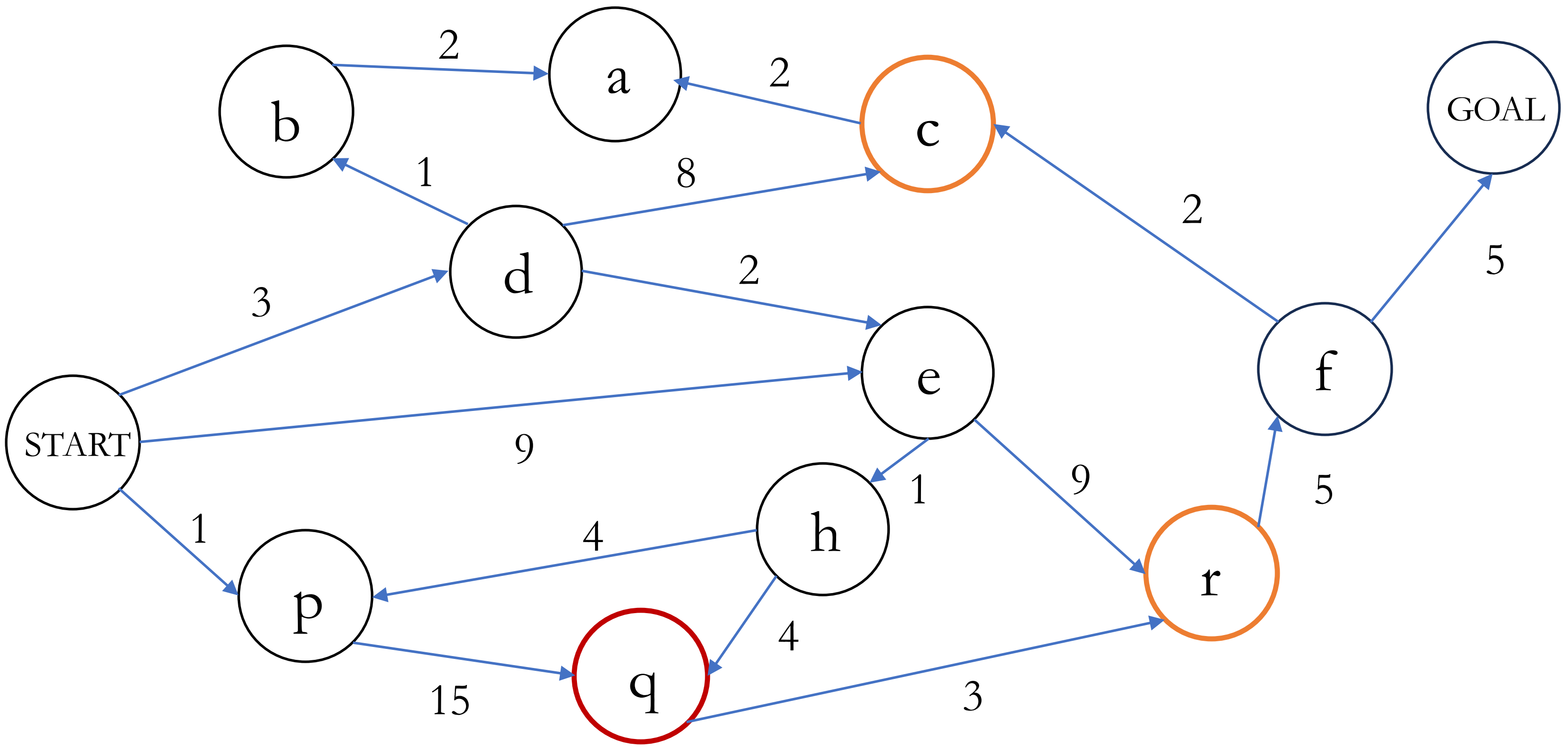
Uniform cost search

- Next, PQ = {(h,6), (c,11), (r,14), (q,16)}



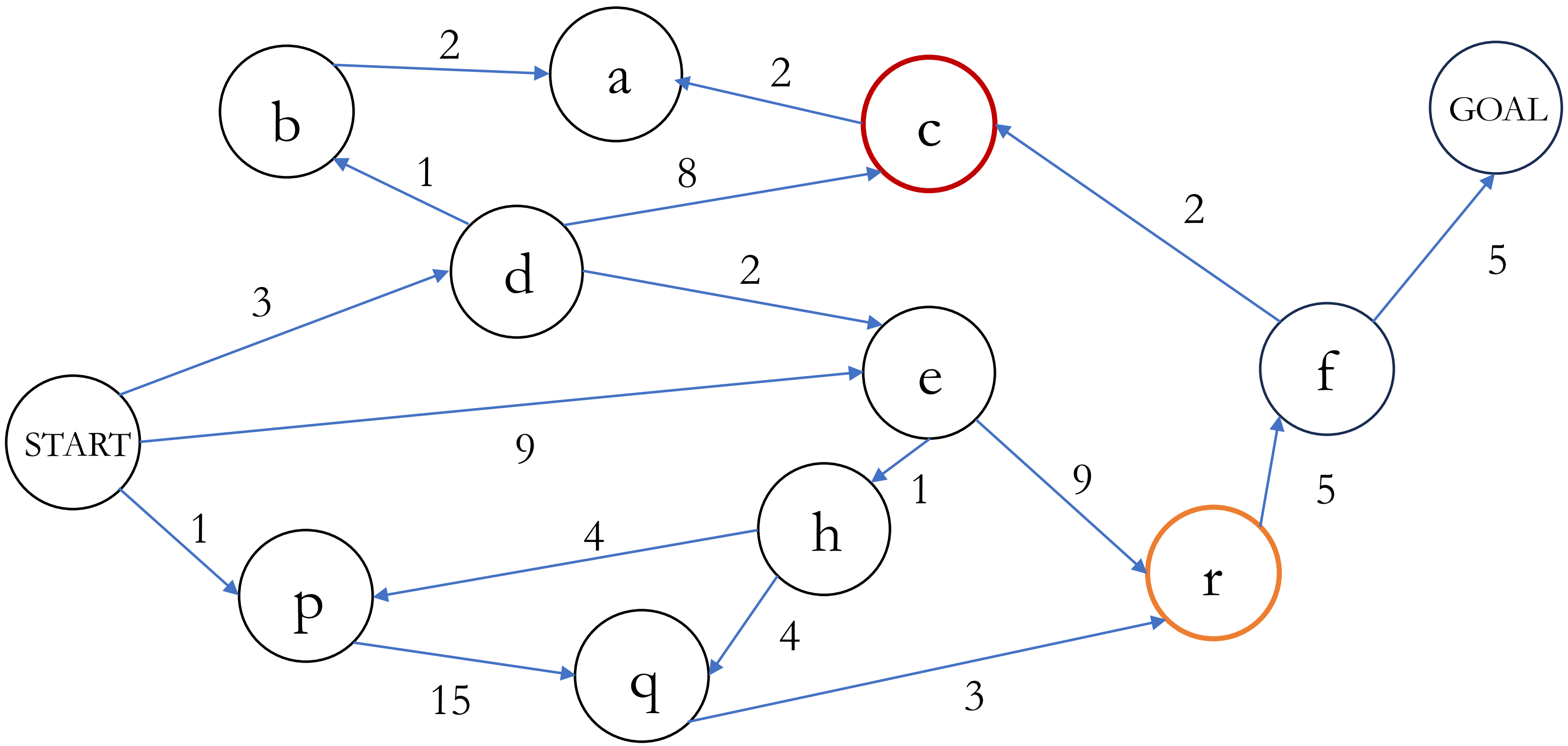
Uniform cost search

- Next, PQ = {(q,10), (c,11), (r,14)}
  - Note that we update q's priority here



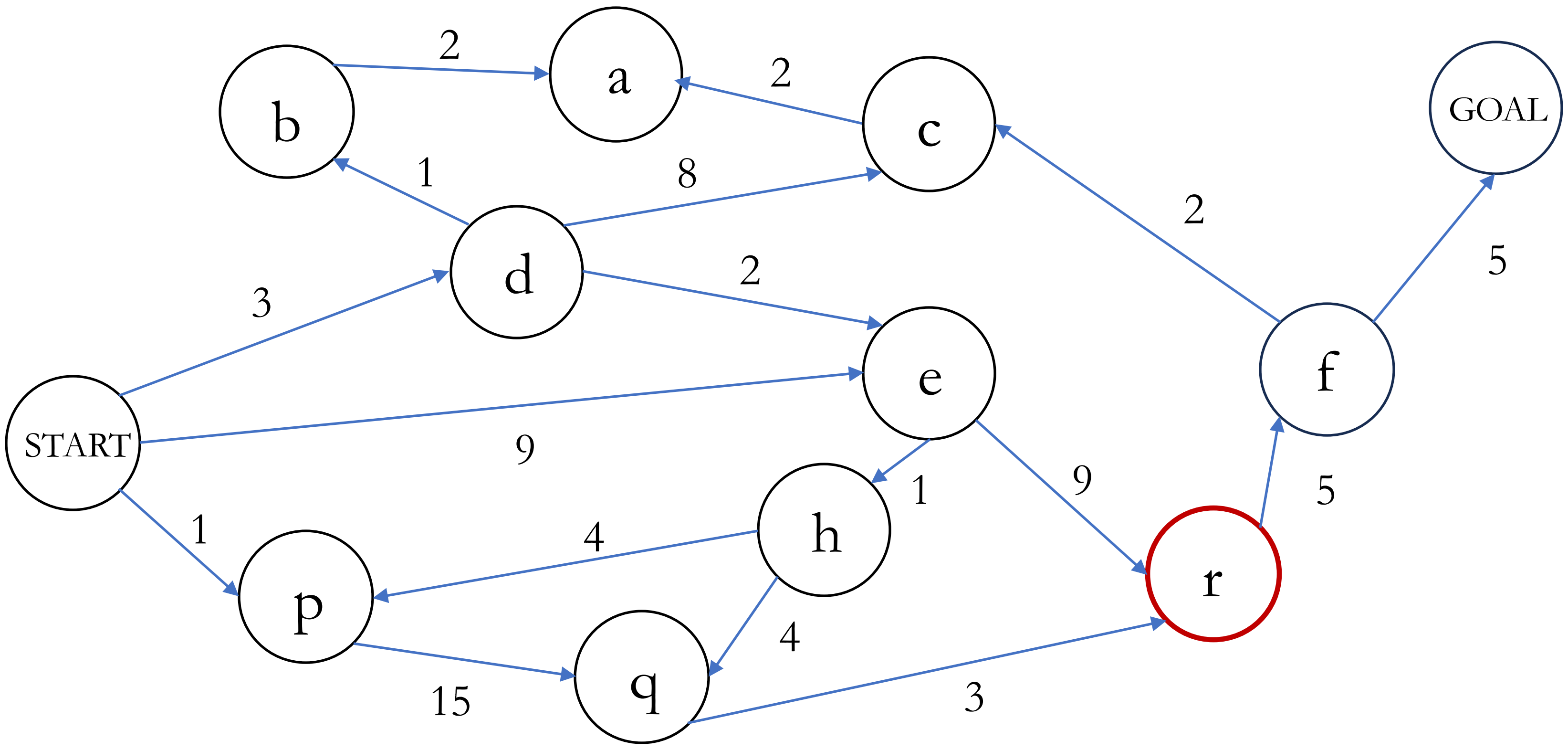
Uniform cost search

- PQ = {(c,11), (r,13)}
  - Note that we update r’s priority here



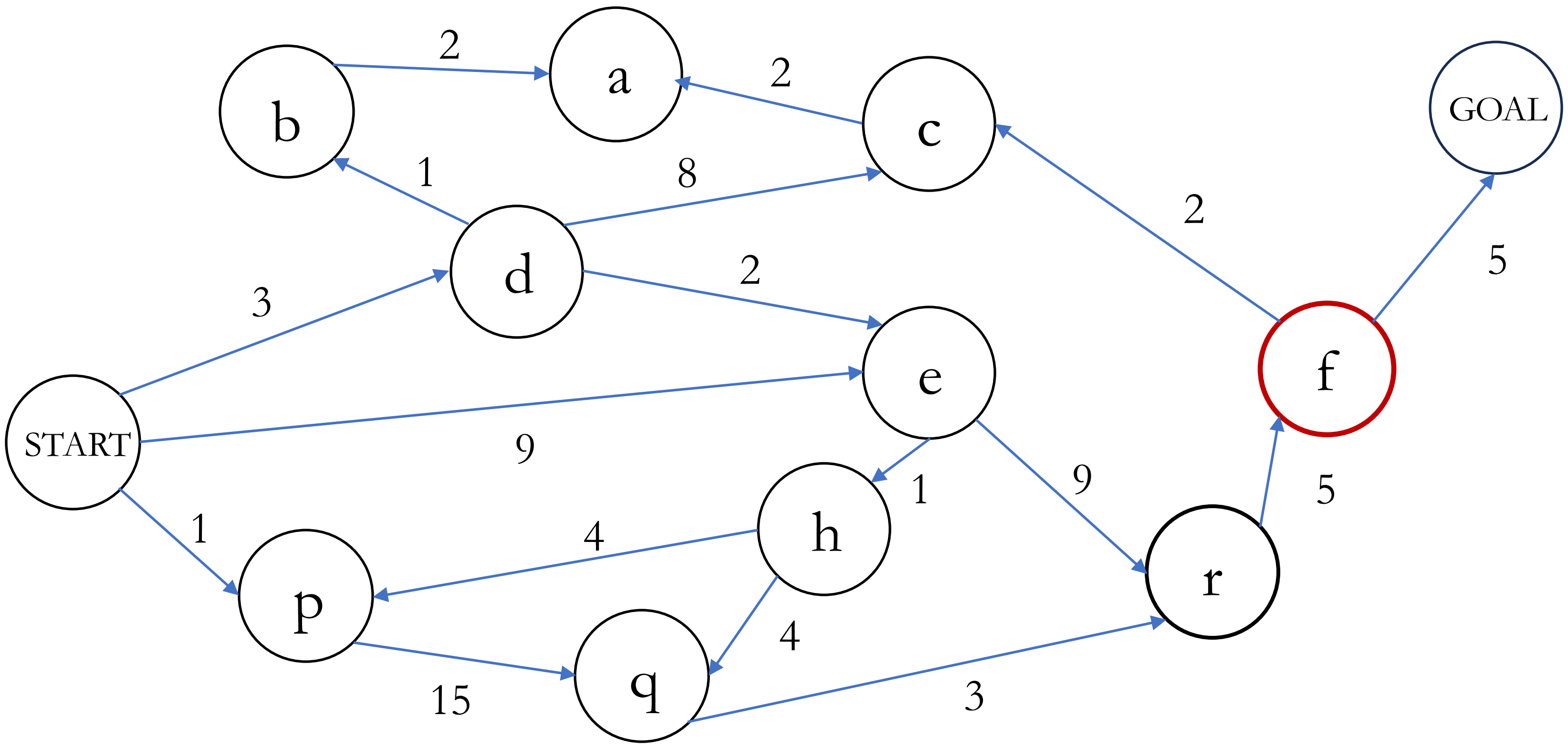
Uniform cost search

- PQ = {(r,13)}



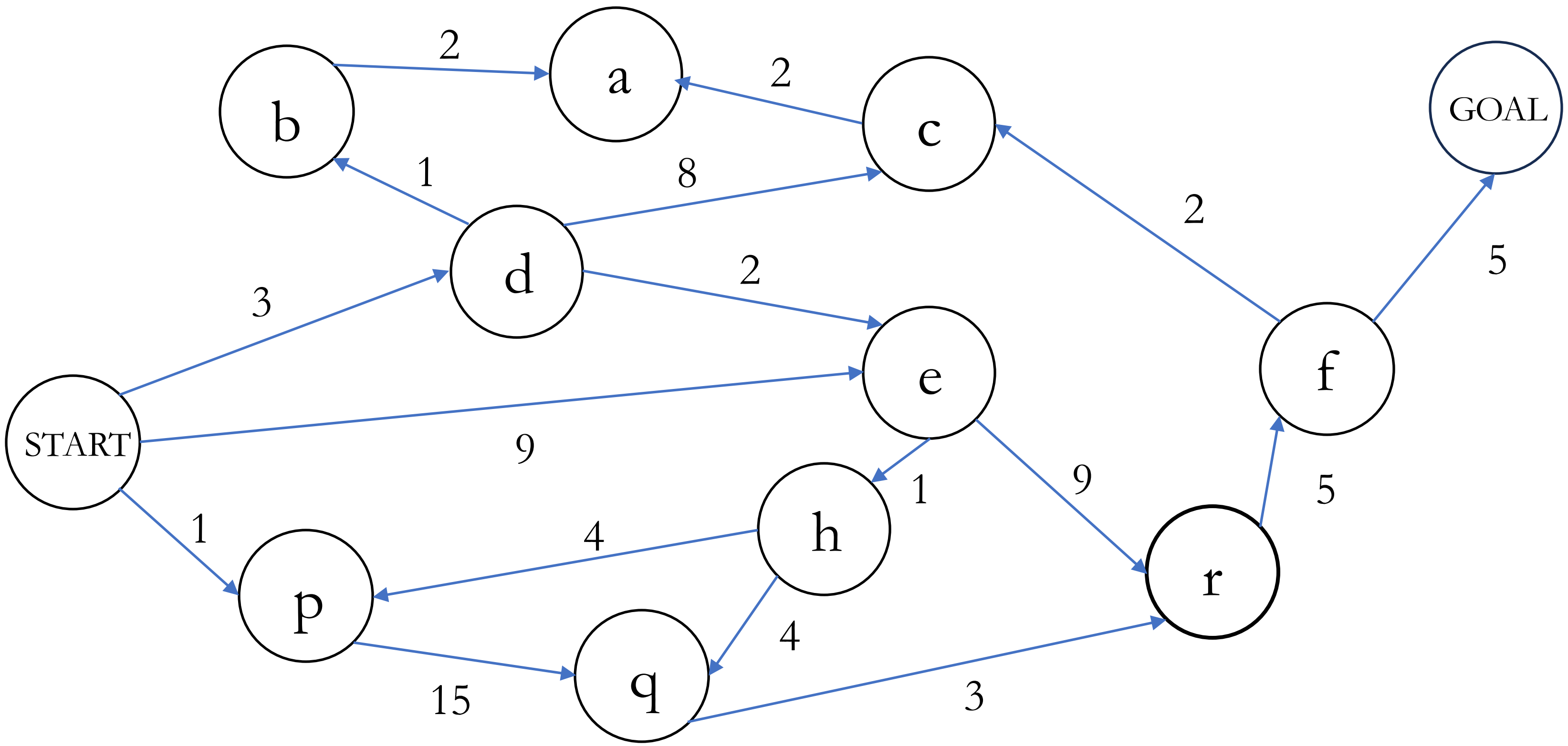
Uniform cost search

- PQ = {(f,18)}



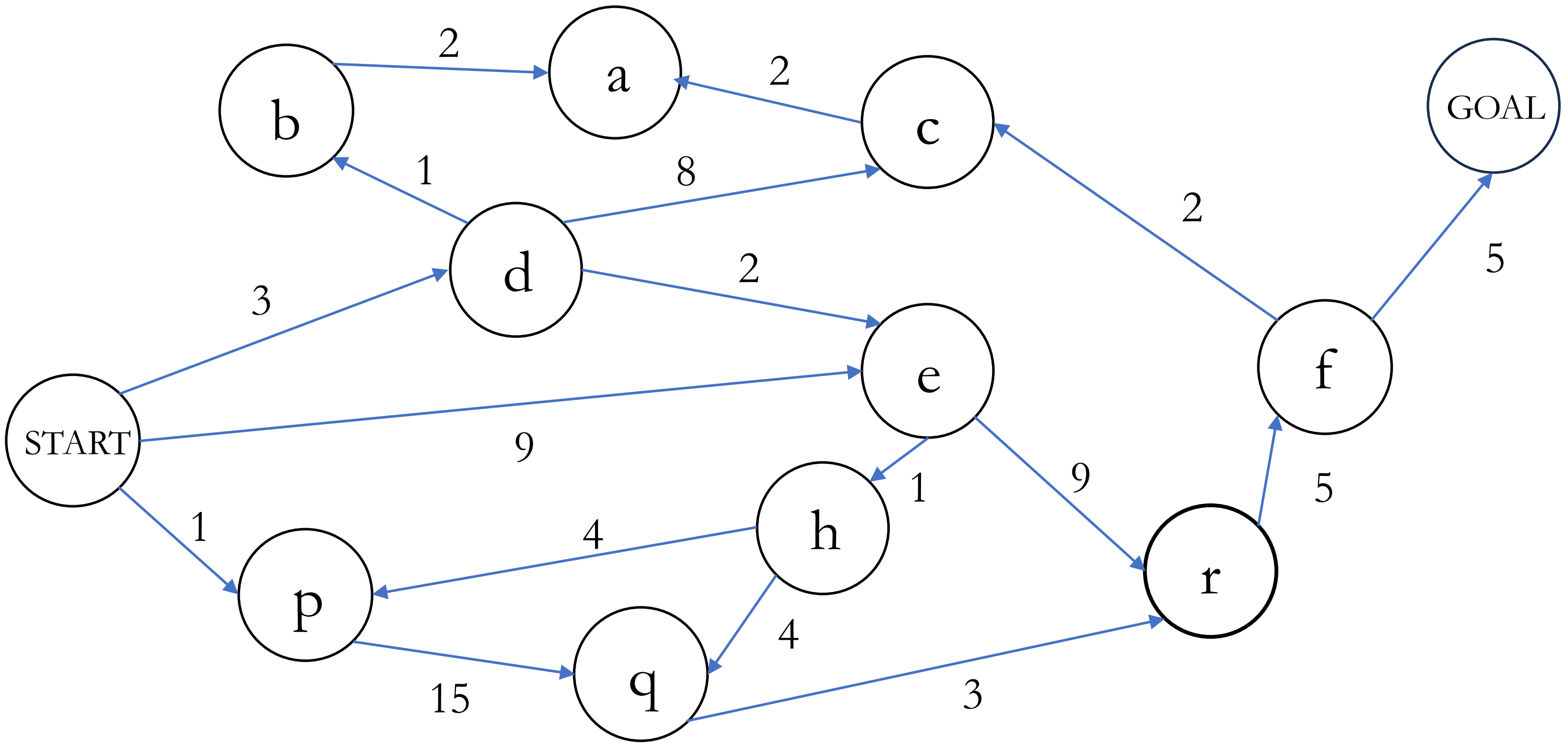
Uniform cost search

- PQ = {(G,23)}



Uniform cost search

- Final state, PQ = {(G,23)}



Quiz question: Is "terminate as soon as you discover the goal" the right stopping criterion?

Dijkstra's Algorithm

- Dijkstra invented the search algorithm in 1956
- **Dijkstra:** Finds shortest paths from source to all vertices
- **Uniform cost search:** Dijkstra’s algorithm that stops when reaching the goal

```
def dijkstra(graph, start):
    dist = {v: infinity for v in graph}
    dist[start] = 0
    pq = PriorityQueue([(0, start)])

    while pq:
        d, u = pq.pop()
        if d > dist[u]: continue  # Already found better path

        for v, weight in graph[u].neighbors:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                pq.push((dist[v], v))
```

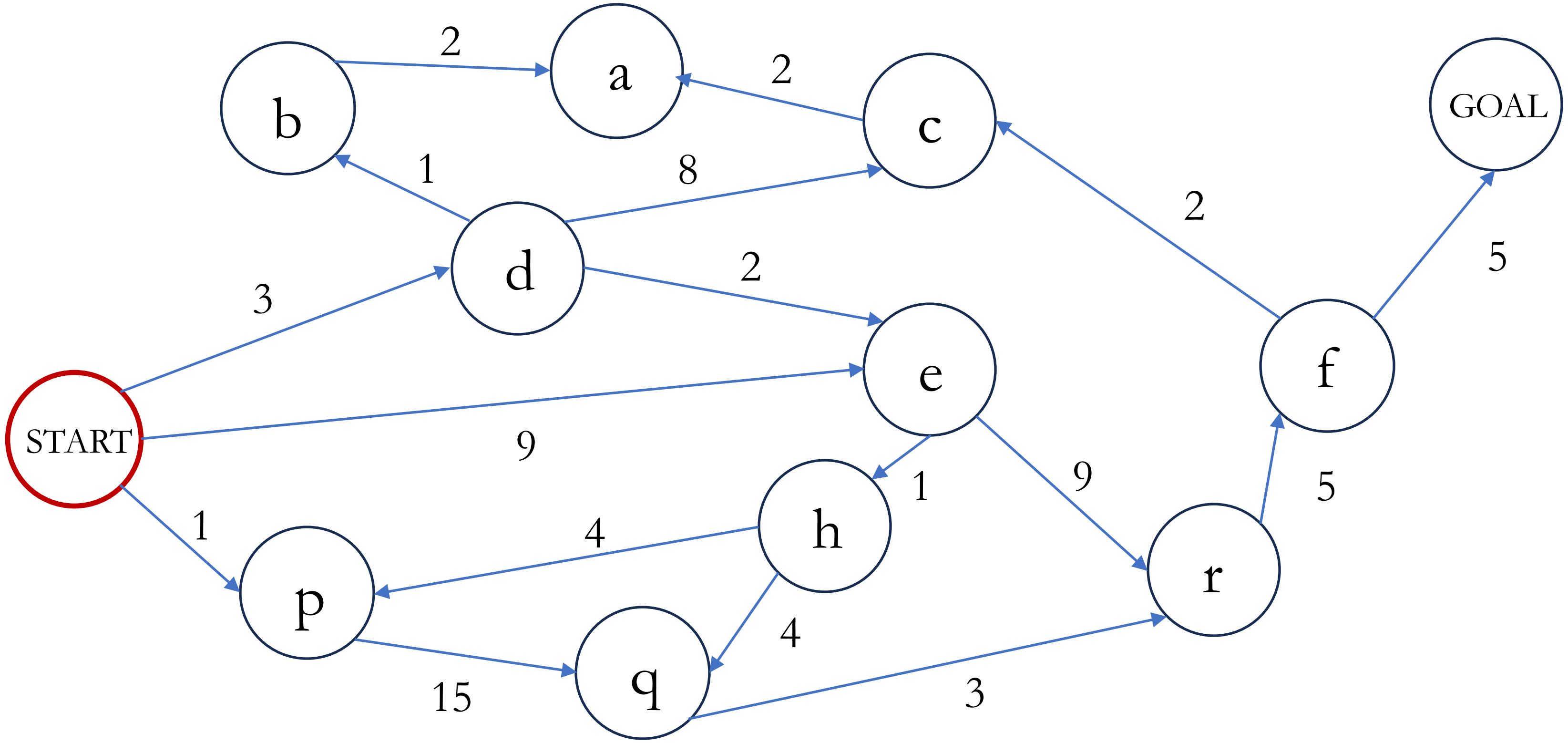
Uniform cost search

- **Advantages**
  - Complete search with systematic exploration
  - Guaranteed optimality
  - Works with varying cost functions

- **Limitations**
  - Explores in all directions
  - No guidance toward goal state
  - Can be slow for large spaces

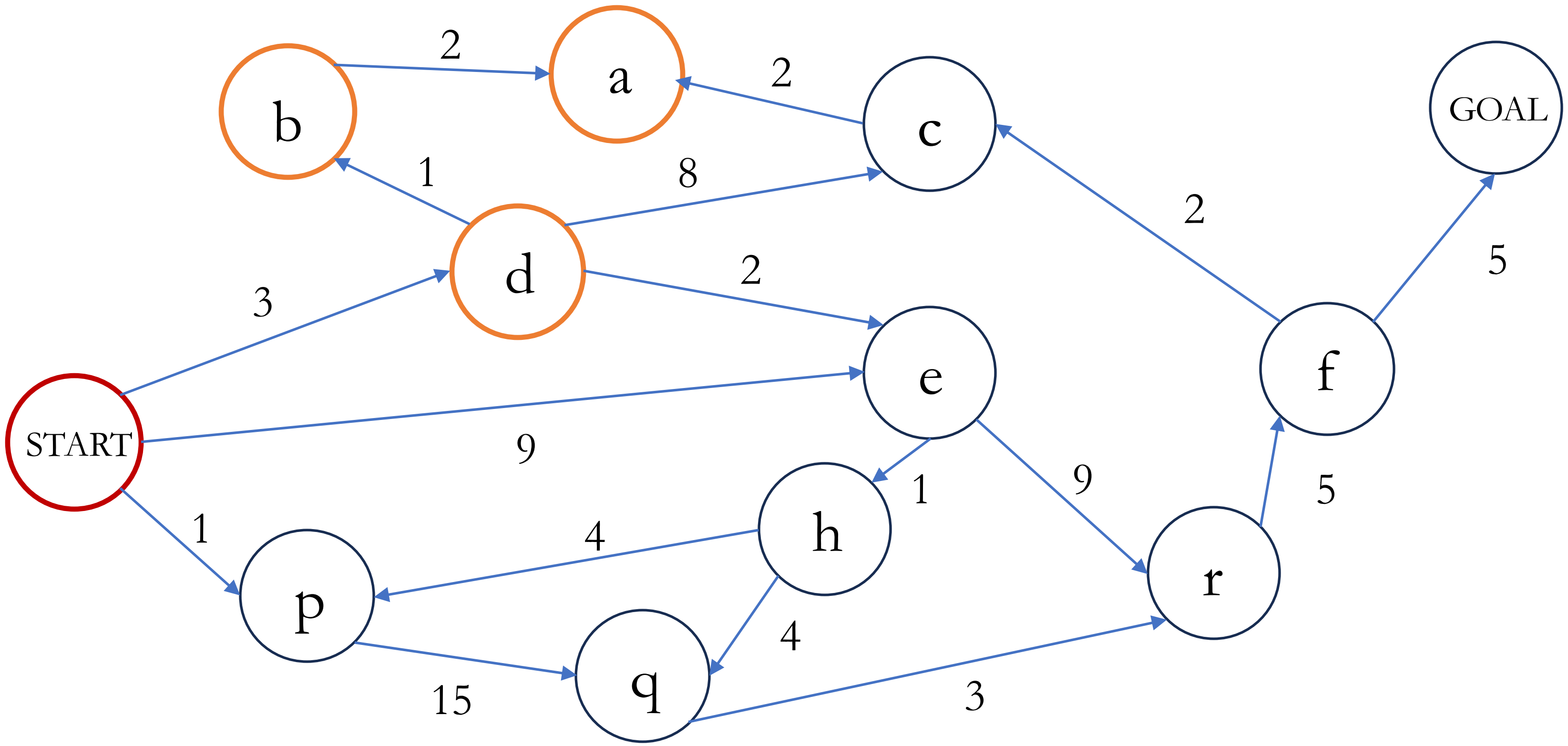
Depth-first search

- Explores as far as possible along each branch
- Move to one of the neighbors



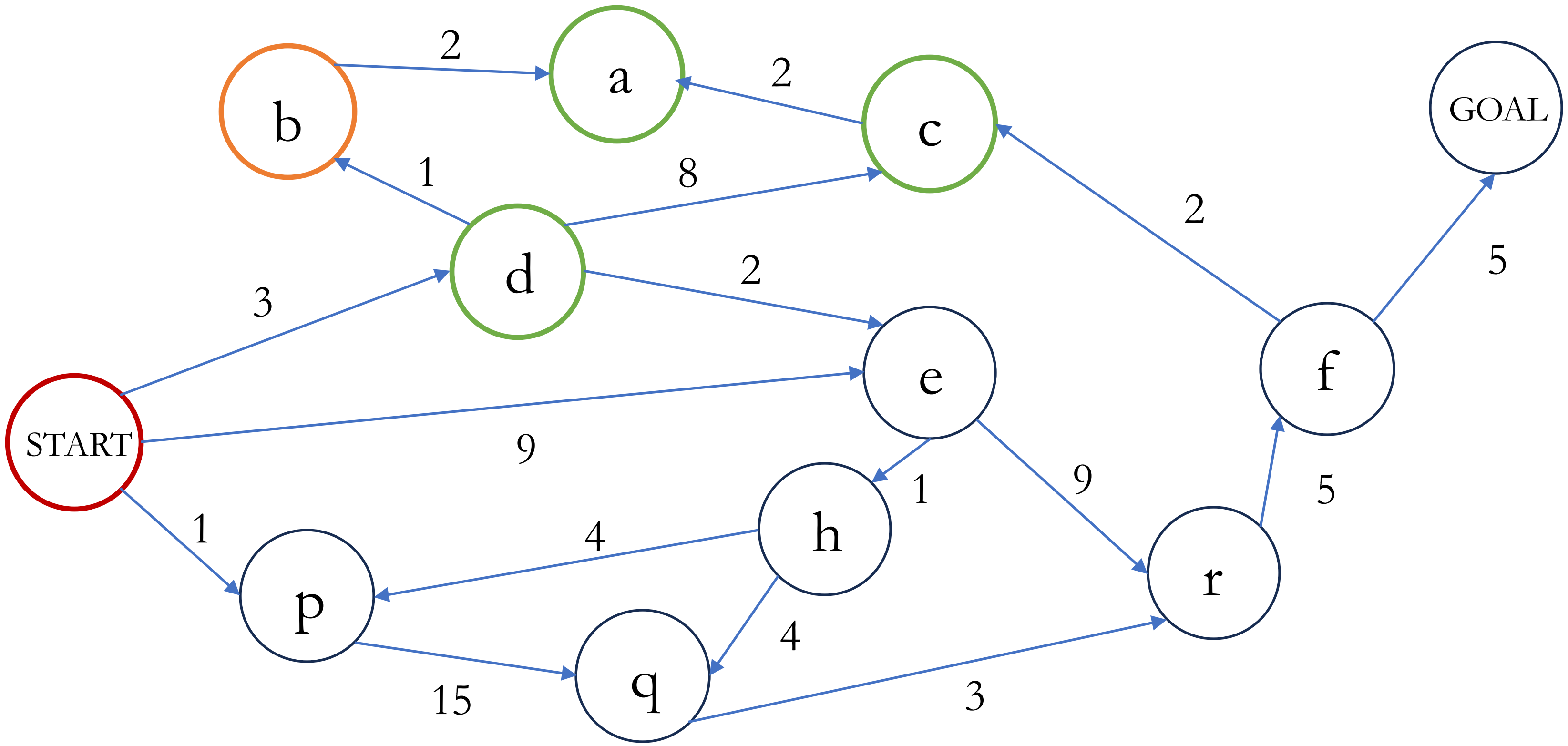
Depth-first search

- Explores as far as possible along each branch
- Keep moving further: S-> d-> b-> a



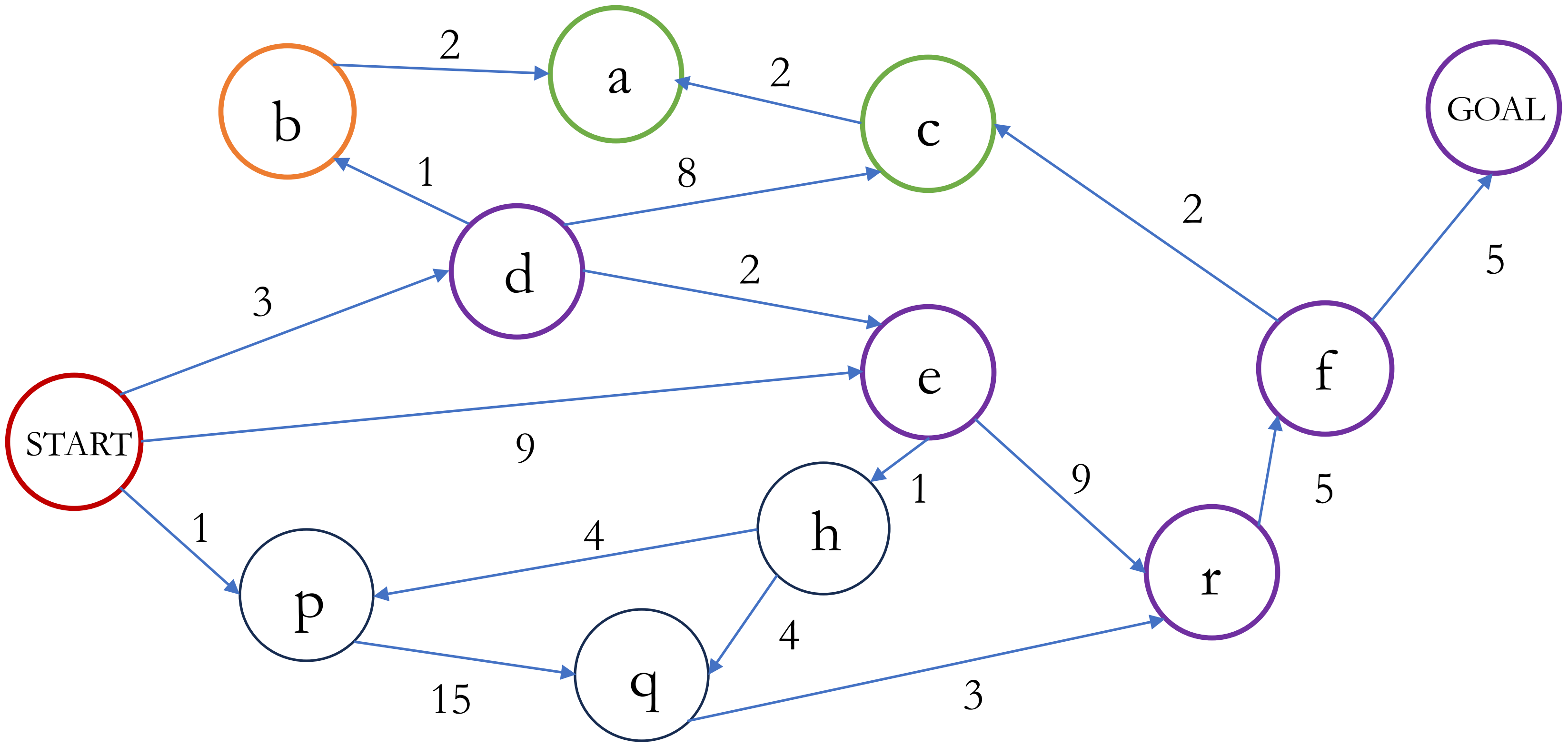
Depth-first search

- Explores as far as possible along each branch
- Backtrack: S-> d-> c



Depth-first search

- Explores as far as possible along each branch
- Backtrack again: S-> d-> e-> r-> f-> G



Comparing search algorithms

Scenario	Best Algorithm	Why?
Equal step costs, shallow solution	BFS	Guarantees shortest path by steps
Memory limited, deep tree	DFS	Linear space complexity
Varying costs, need optimal	UCS/Dijkstra	Finds least-cost path
Need all shortest paths	Dijkstra	Computes complete shortest path tree