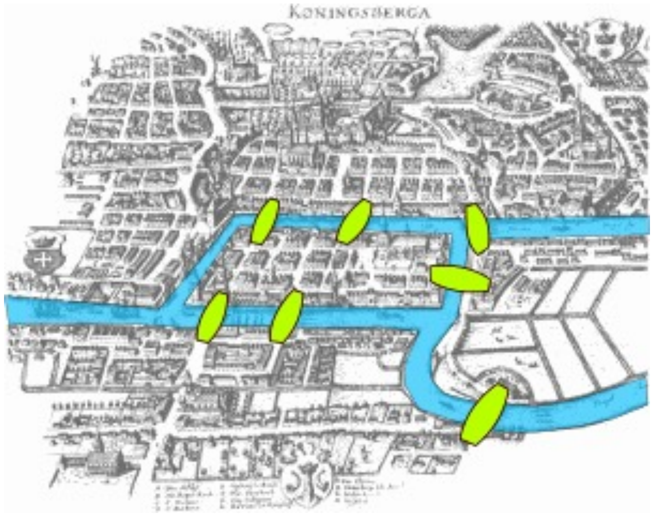


Weighted graphs, Dijkstra’s algorithm

(download slides and .py files to follow along)

Tim Kraska

MIT Department Of Electrical Engineering and Computer Science

Topics

- Last week
  - Graph models and how to implement
  - Shortest path problems on unweighted graphs
  - Depth-first search and breadth-first search
- Today
  - Shortest path on weighted graphs

Two Important Abstractions

- Last-in first-out (LIFO) sequence (often called a stack)
- First-in first-out (FIFO) sequence (often called a queue)

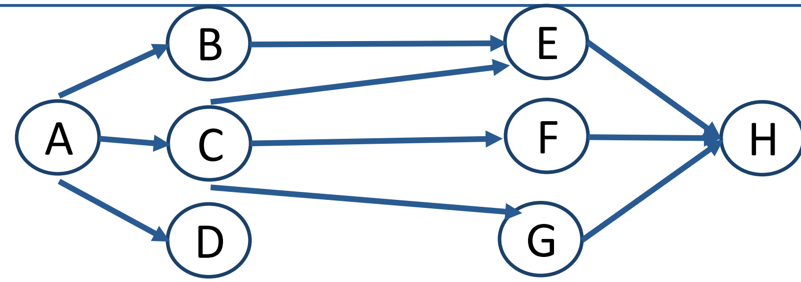
LIFO

FIFO

 Fall 2025

FIFO and BFS Shortest Path

Seeking path from A to H



Take next path from front, and delete front. Add new paths to rear.

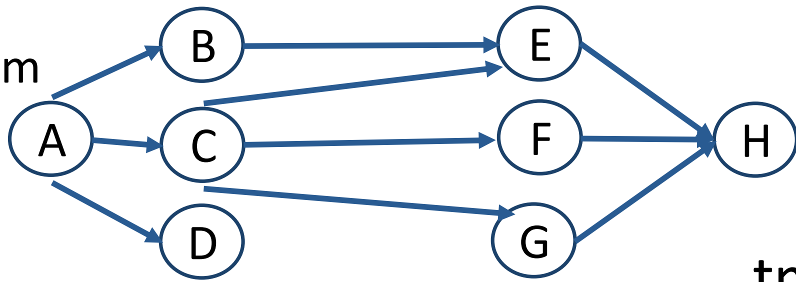
Guarantees first solution is a shortest path

queue	tmp_path	
[[A]]	A	Add paths from A
[[A,B],[A,C],[A,D]]	A→B	Add paths from B
[[A,C],[A,D],[A,B,E]]	A→C	Add paths from C, don't revisit E
[[A,D],[A,B,E],[A,C,F],[A,C,G]]	A→D	No paths from D
[[A,B,E],[A,C,F],[A,C,G]]	A→B→E	Add paths from E
[[A,C,F],[A,C,G],[A,B,E,H]]	A→C→F	

Might be other equally short paths
But don't care

LIFO and DFS Shortest Path

Seeking path from A to H



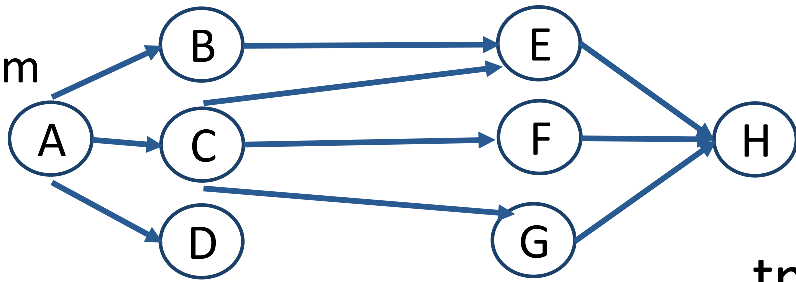
Take next path from rear, and delete rear. Add new paths to rear.

stack	tmpPath	
[[A]]	A	Add paths from A
[[A,B],[A,C],[A,D]]	A→D	No paths from D
[[A,B],[A,C]]	A→C	Add paths from C
[[A,B],[A,C,E],[A,C,F],[A,C,G]]	A→C→G	Add paths from G
[[A,B],[A,C,E],[A,C,F],[A,C,G,H]]	A→C→G→H	

Could terminate here

LIFO and DFS Shortest Path

Seeking path from A to H



Take next path from rear, and delete rear. Add new paths to rear.

stack	tmpPath	
[[A]]	A	Add paths from A
[[A,B],[A,C],[A,D]]	A→D	No paths from D
[[A,B],[A,C]]	A→C	Add paths from C
[[A,B],[A,C,E],[A,C,F],[A,C,G]]	A→C→G	Add paths from G
[[A,B],[A,C,E],[A,C,F],[A,C,G,H]]	A→C→G→H	
[[A,B],[A,C,E],[A,C,F]]	A→C→F	Add paths from F
[[A,B],[A,C,E],[A,C,F,H]]	A→C→F→H	
[[A,B],[A,C,E]]	A→C→E	Add paths from E
[[A,B],[A,C,E,H]]	A→C→E→H	
[[A,B]]	A →B	Add paths from B
[[A,B,E]]	A→B→E	Add paths from E
[[A,B,E,H]]	A→B→E→H	

BFS using a FIFO stack

```
def bfs_fifo(graph, start, goal):
    if start == goal:
        return [start]
    queue = [[start]]
    visited = {start}

    while len(queue) > 0:
        print("Current queue:", pathlist_to_string(queue))

        # simulate iterating through current frontier
        path = queue.pop(0)
        print("  Current BFS path:", path_to_string(path))

        current_node = path[-1]
        for next_node in neighbors(graph, current_node):
            if next_node in visited:
                continue
            visited.add(next_node)
            new_path = path + [next_node]
            if next_node == goal:
                return new_path

            # simulate building next frontier
            queue.append(new_path)
    return None
```

Replace discrete frontiers with "sliding" stack

FIFO

We still skip over visited notes as before

DFS using a LIFO Stack

```python
def dfs_lifo(graph, start, goal):
    stack = [[start]]

    while len(stack) > 0:
        print("Current stack:", pathlist_to_string(stack))

        # simulate running the body of a recursive dfs() call
        path = stack.pop(-1)
        print("  Current DFS path:", path_to_string(path))

        current_node = path[-1]
        if current_node == goal:
            return path

        # put children on stack in reverse order of which
        # we intend to explore them, because stack is lifo
        for next_node in reversed(neighbors(graph, current_node)):
            if next_node in path:
                continue

            # prepare to simulate running dfs() on next_node
            stack.append(path + [next_node])

    return None
```

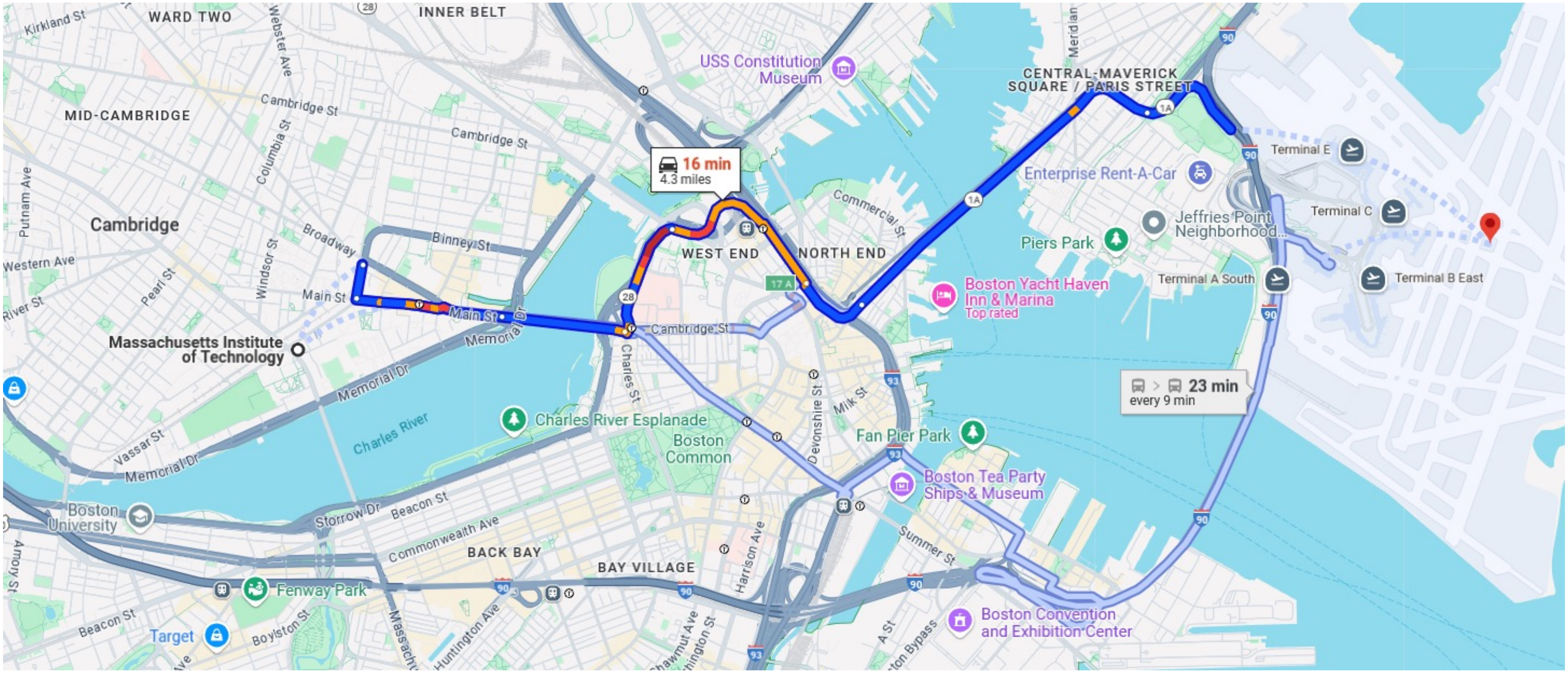
LIFO

Simulate recursion with expanding/shrinking stack

Put children on LIFO stack

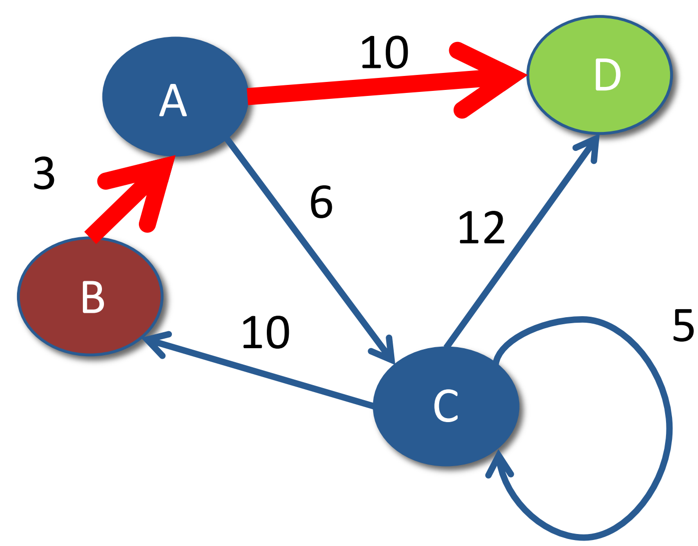
Adding weights to graphs

Plans in real life



Weighted shortest path problem

- Same graph model as before
- Each edge $A \rightarrow B$, or each action $Action(StateA) = StateB$, has an associated weight
- Cost of a path $A \rightarrow B \rightarrow \cdots \rightarrow N$ is the sum of the weights along the edges
- A shortest path between two nodes is one that minimizes the path cost



Optimality of BFS on unweighted graphs

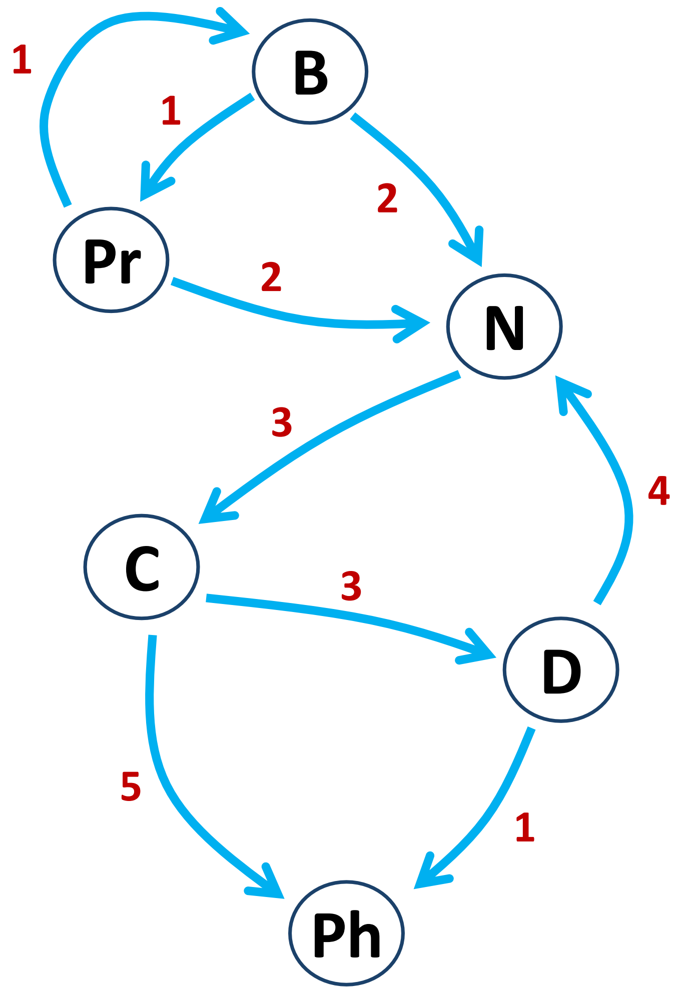
- All nodes in each frontier are discovered at their shortest distance from the start
- Proof sketch:
  - Frontier 1 has all paths of length 1
  - Some paths of length 2 end up in frontier 2; others end up back in frontiers 1 or 0
  - Suppose a path in frontier 2 is not shortest
  - Then there is some other path of length 0 or 1 to that path's end node
  - That path would have been discovered in frontier 0 or 1, and hence not discovered in frontier 2 due to visited set
  - So all paths in frontier 2 are shortest

Run BFS on weighted graphs?!

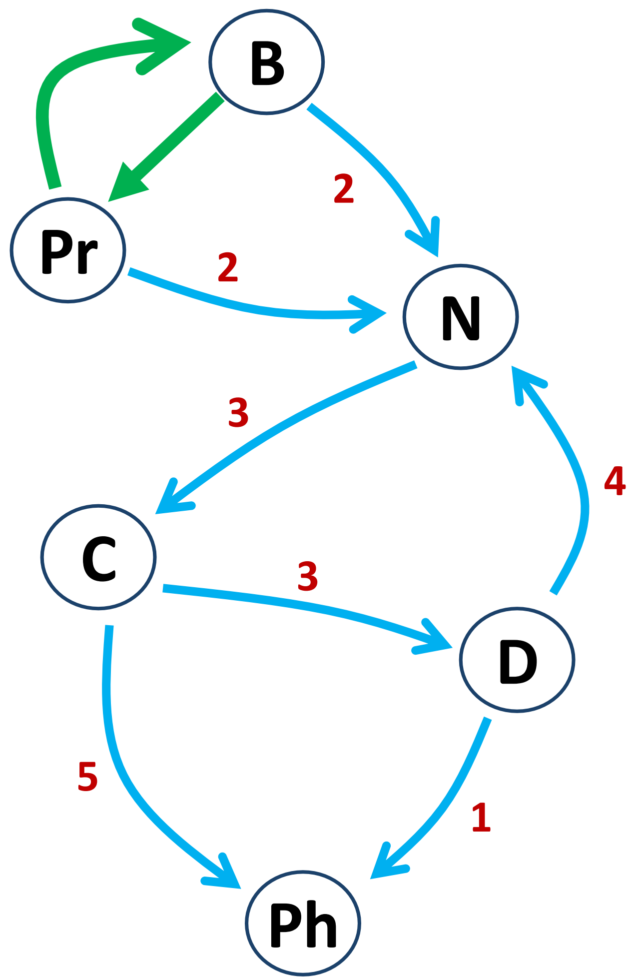
Change the problem not the algorithms:

- If edge weights are all integers, split the edges into unit lengths
- To handle non-integer weights, scale up all weights until practically to integers
  - Computers have finite precision to represent floating point, so scaling must eventually result in integers
- Not very efficient, but it works!

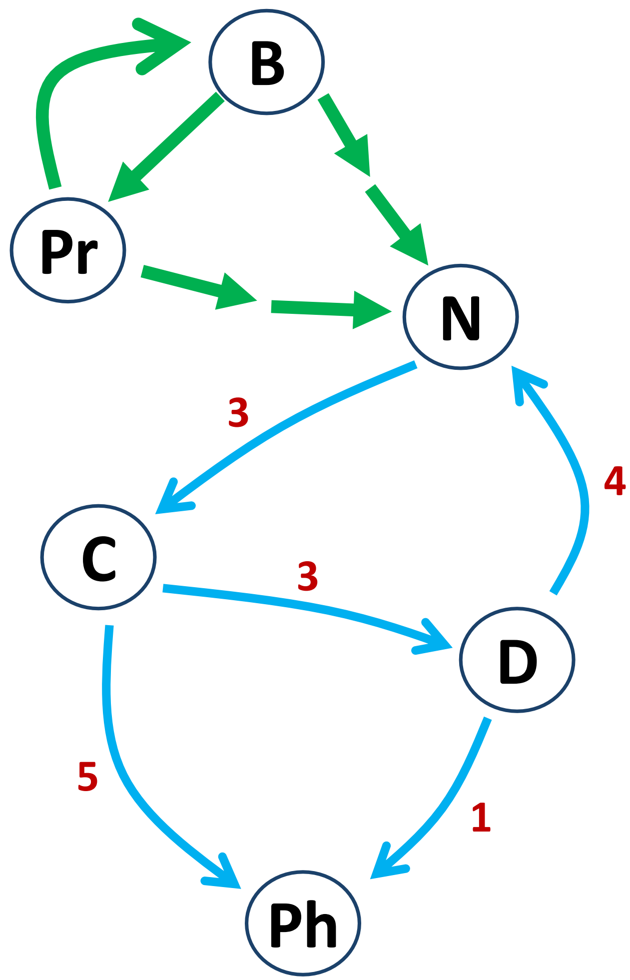
“Discretized” BFS



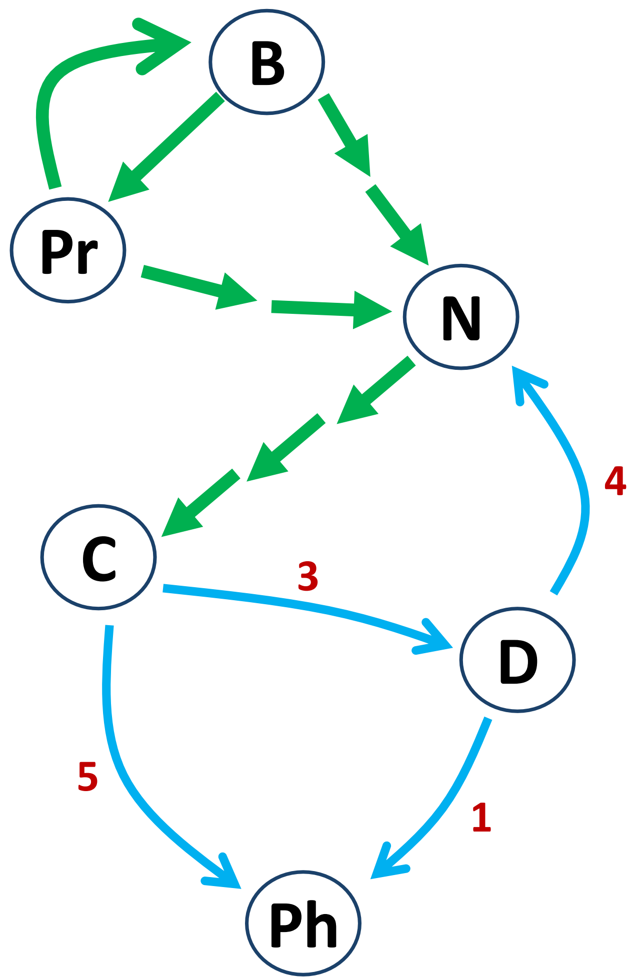
“Discretized” BFS



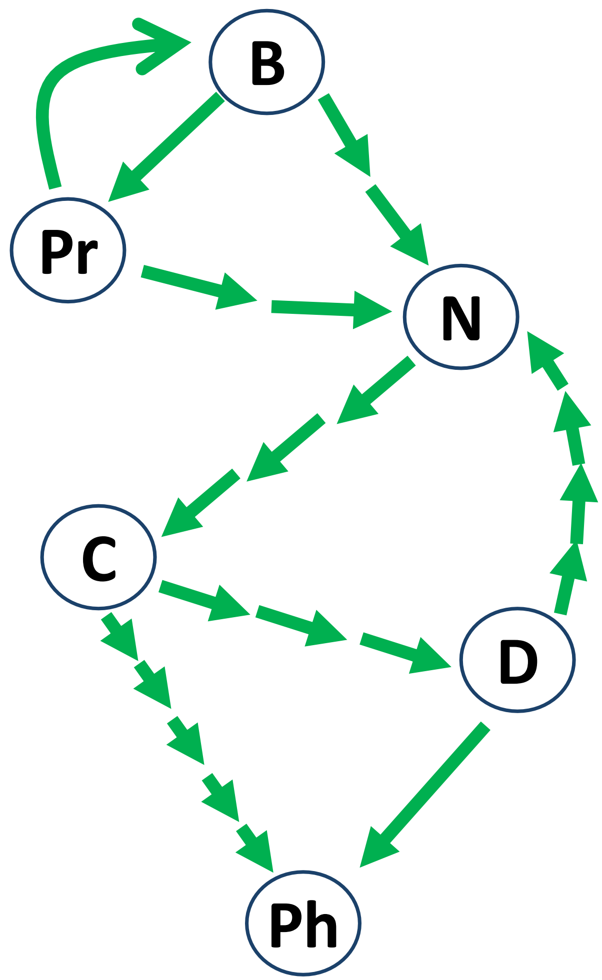
“Discretized” BFS



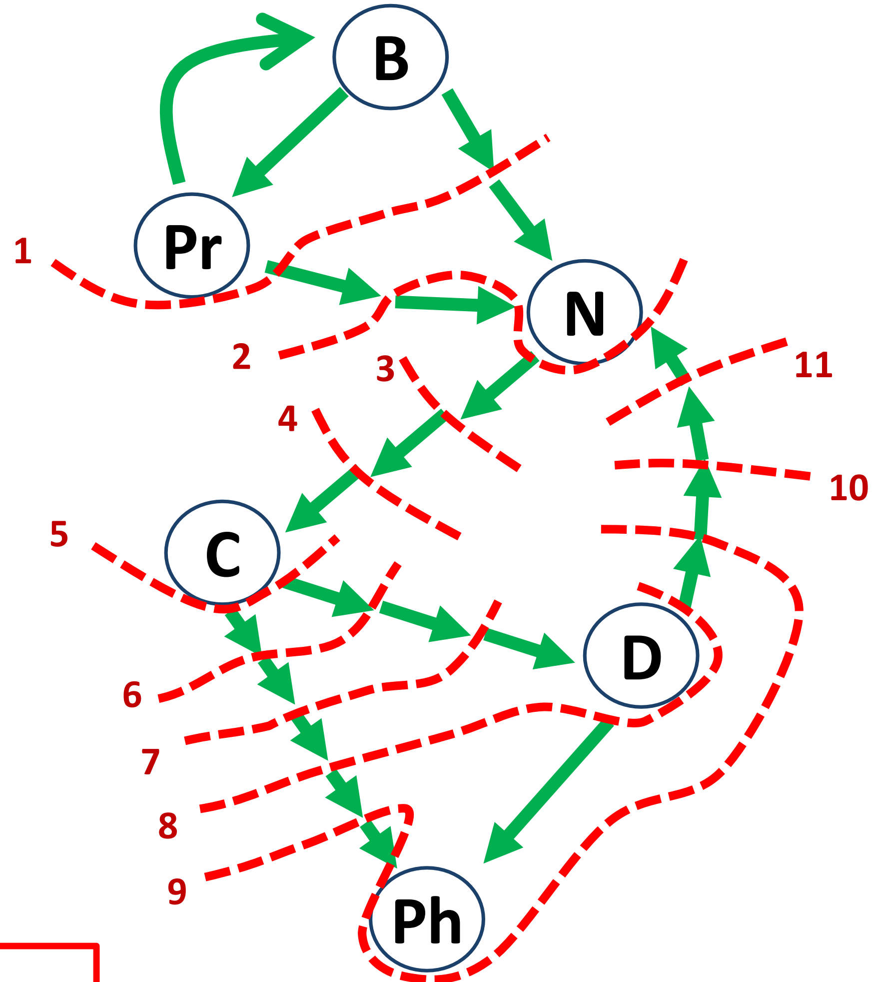
“Discretized” BFS



“Discretized” BFS



"Discretized" BFS



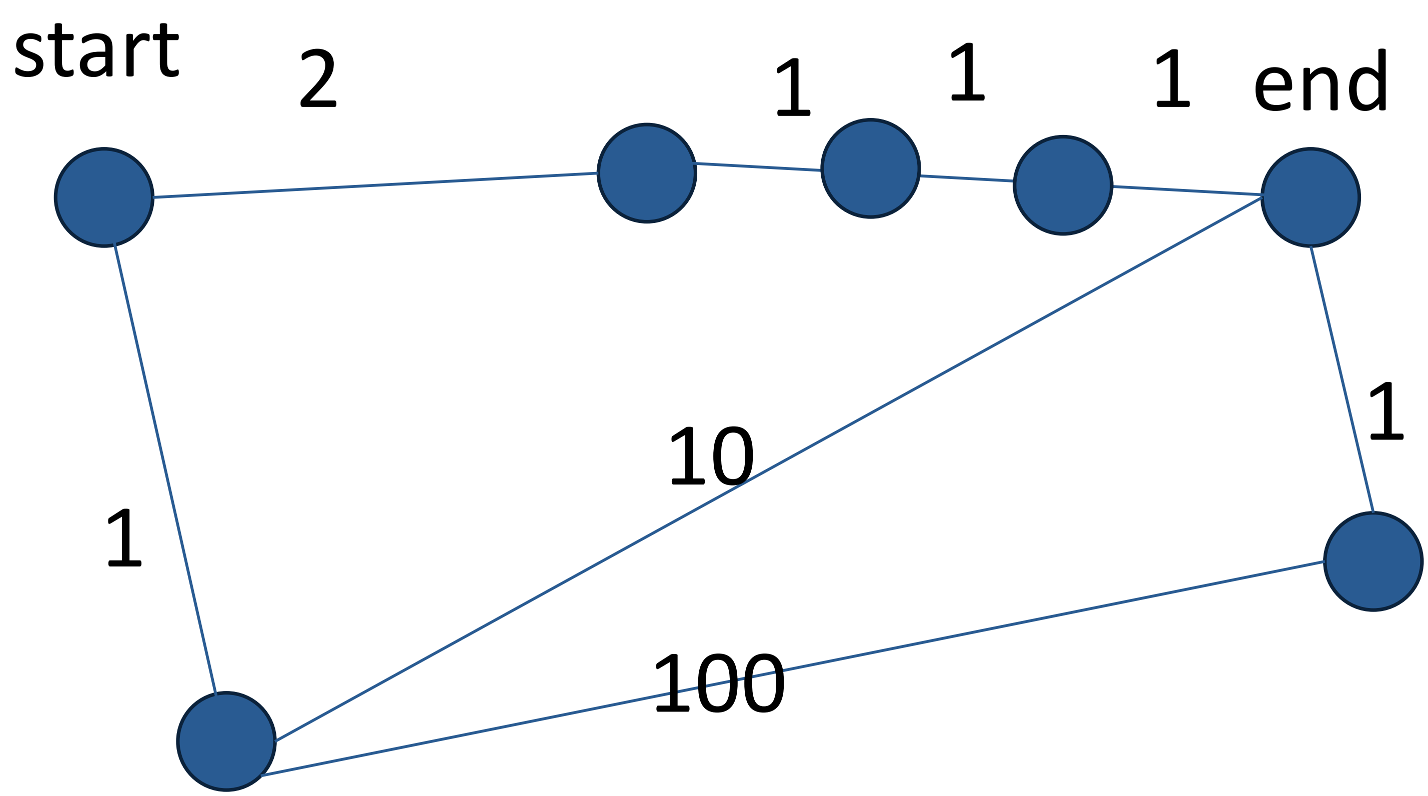
Runtime depends on magnitude of weights vs discretization unit

Simulate “discretized” BFS efficiently?

- Want to traverse an original weighted edge in one step, not many small steps
- Key concept in BFS is the frontier, how to maintain that now?
- Suppose we know state A is on the true frontier and expand it to state B
- We’ve projected a potential future frontier onto B, but by the same reasoning, other states $C, D, E, \ldots$ may also have projected future frontiers
- **The next true frontier is the minimum of all their projected frontiers**

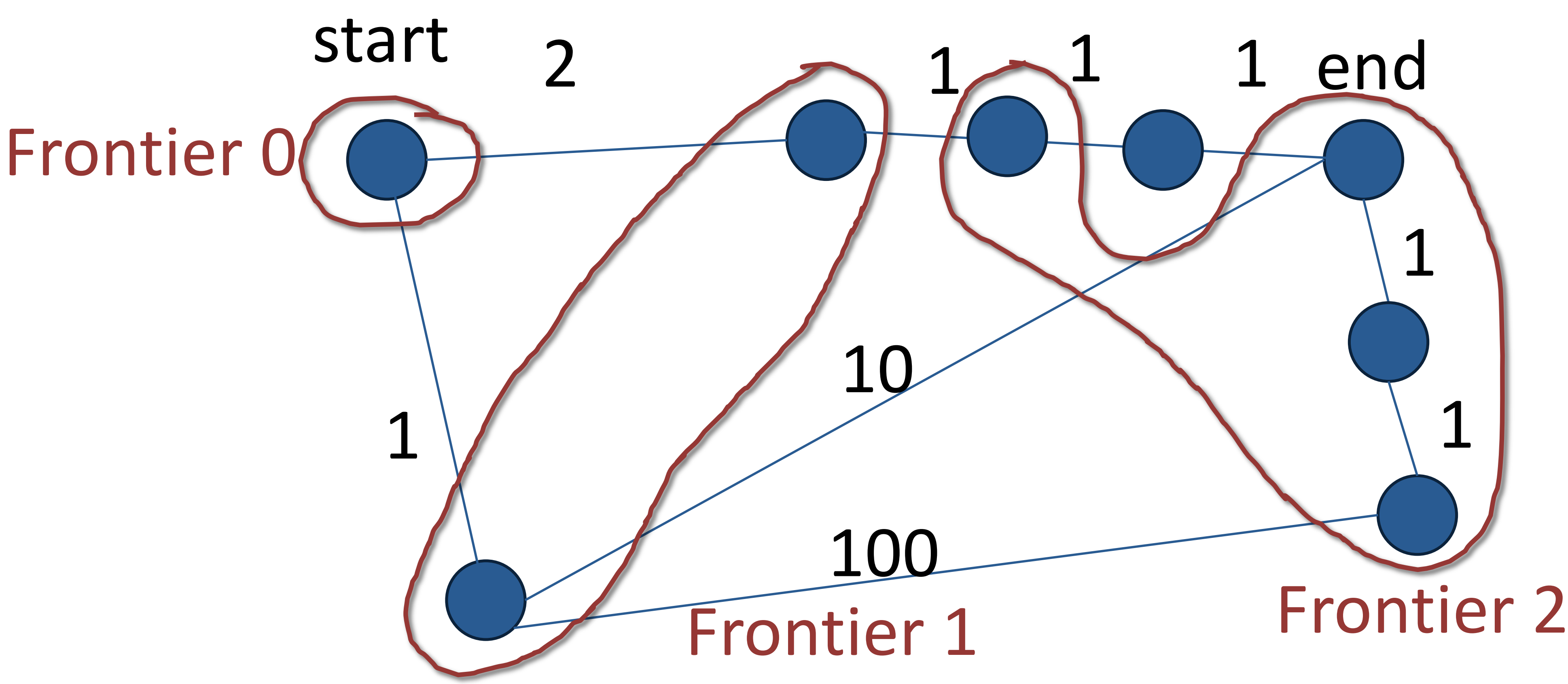
Dijkstra's algorithm

BFS on weighted graphs



BFS on weighted graphs

- Is this the shortest path?



Can we modify breadth first to work with weighted graphs?

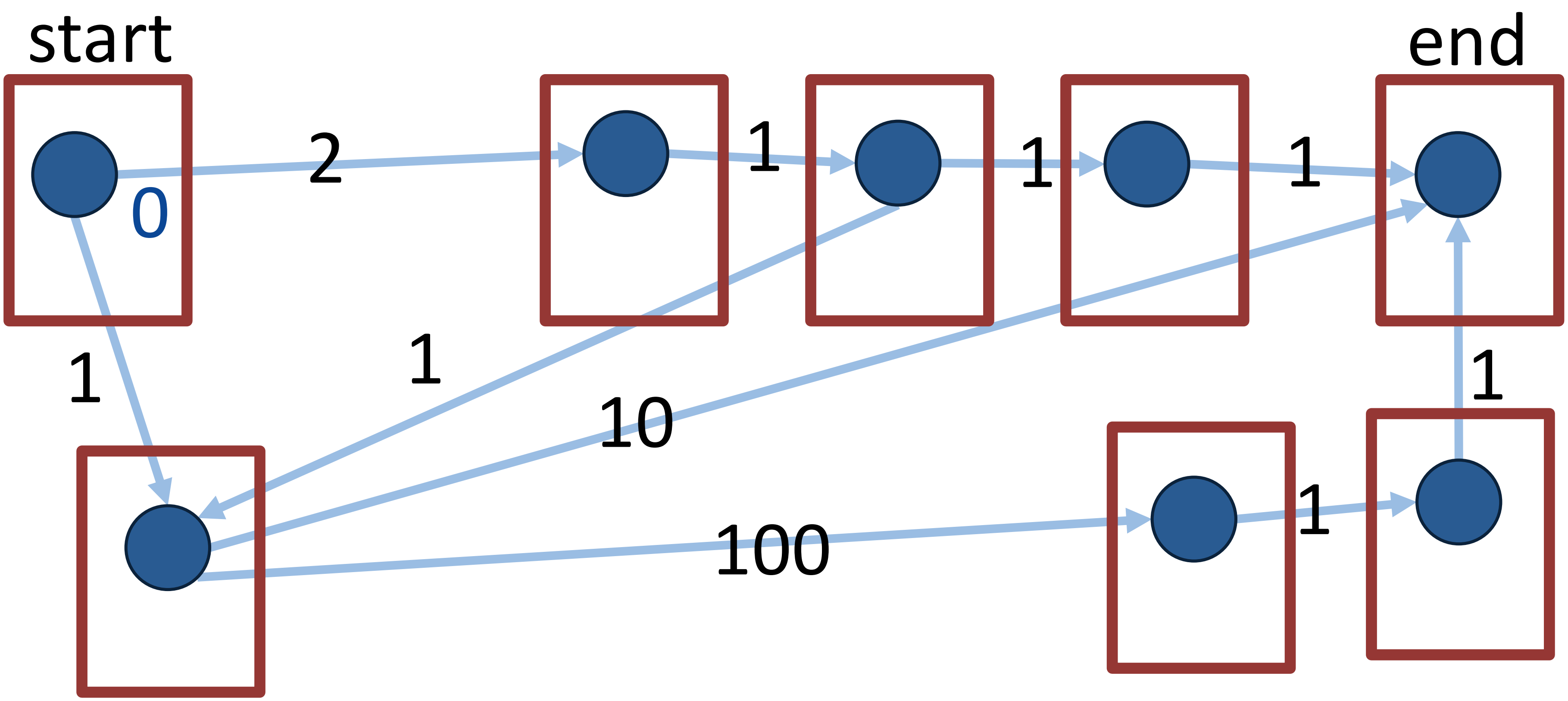
- With breadth first, we are guaranteed that we reach nodes via a minimum distance
  - Because our notion of “frontier” corresponds to distances
  - Since all edges have the same weight.
- For weighted graphs, we don’t have this guarantee

Can we modify breadth first to work with weighted graphs?

- With breadth first, we are guaranteed that we reach nodes via a minimum distance
  - Because our notion of "frontier" corresponds to distances
  - Since all edges have the same weight.
- For weighted graphs, we don't have this guarantee
- Ideas for DIJKSTRA's shortest path:
  - Explicitly maintain (weighted) distance to source node
  - Consider an edge at the frontier closest to source
    - And check if existing shorted distances need to be updated

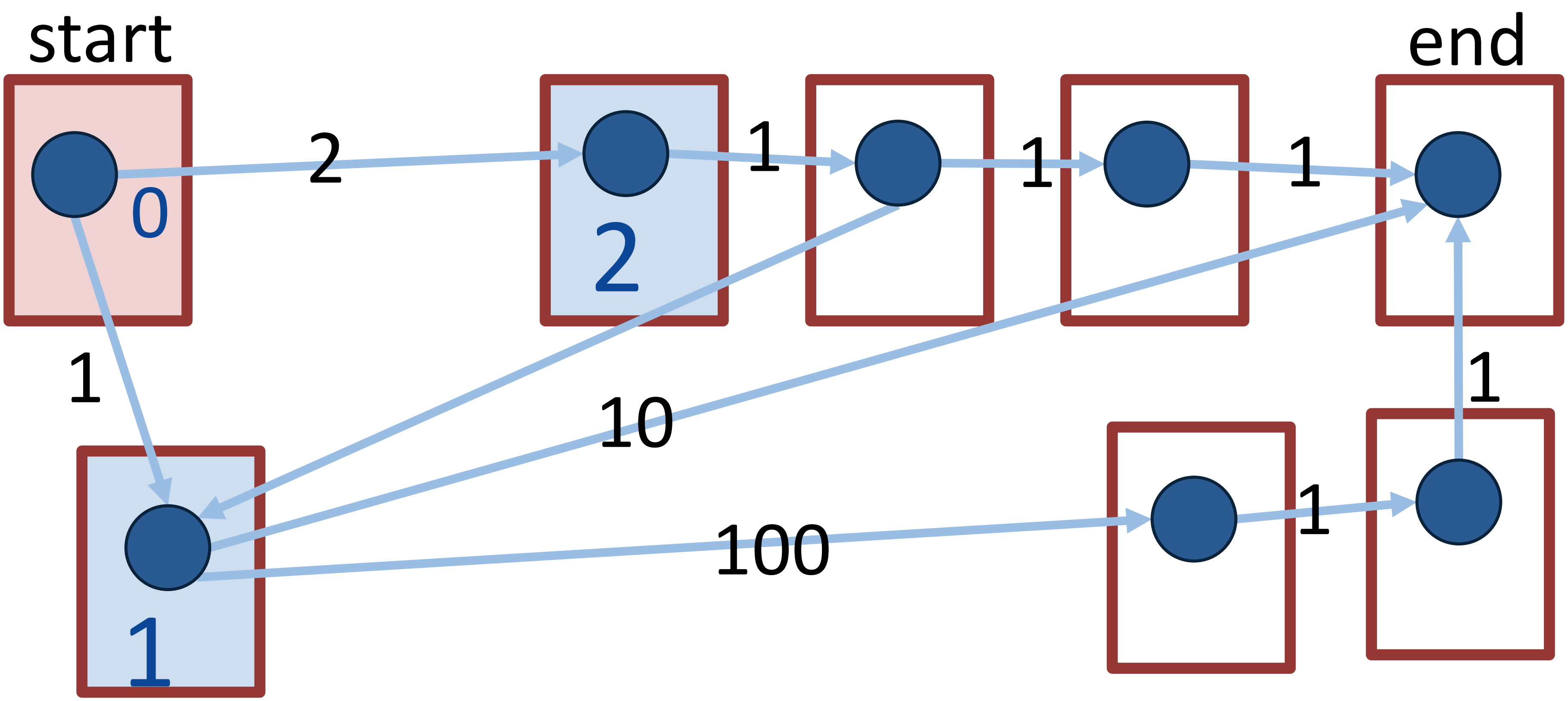
Dijkstra

- Take the path with the lowest cost (start 0)
- Expand its neighbors (filter out path we already know)



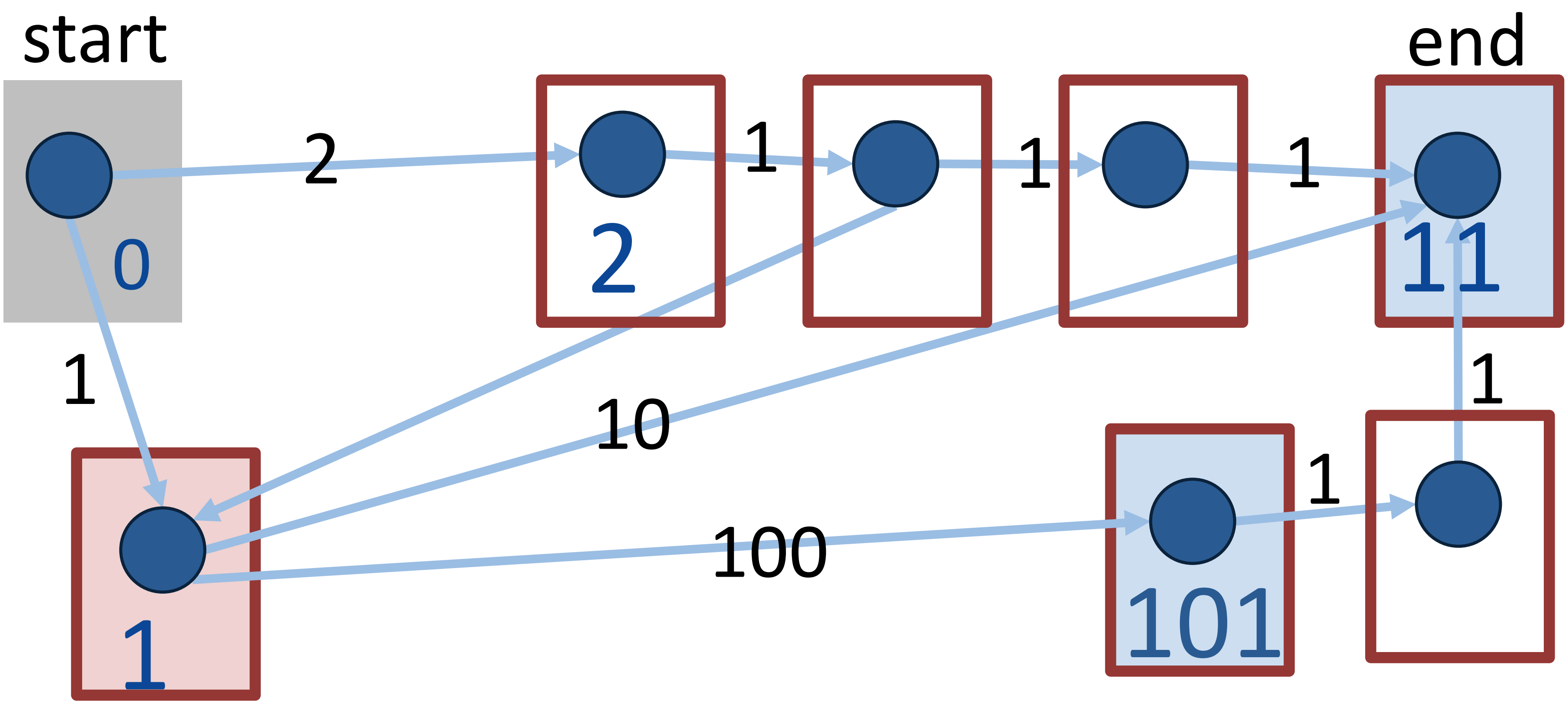
Dijkstra

- Take the path with the lowest cost (start 0)
- Expand its neighbors (filter out path we already know)



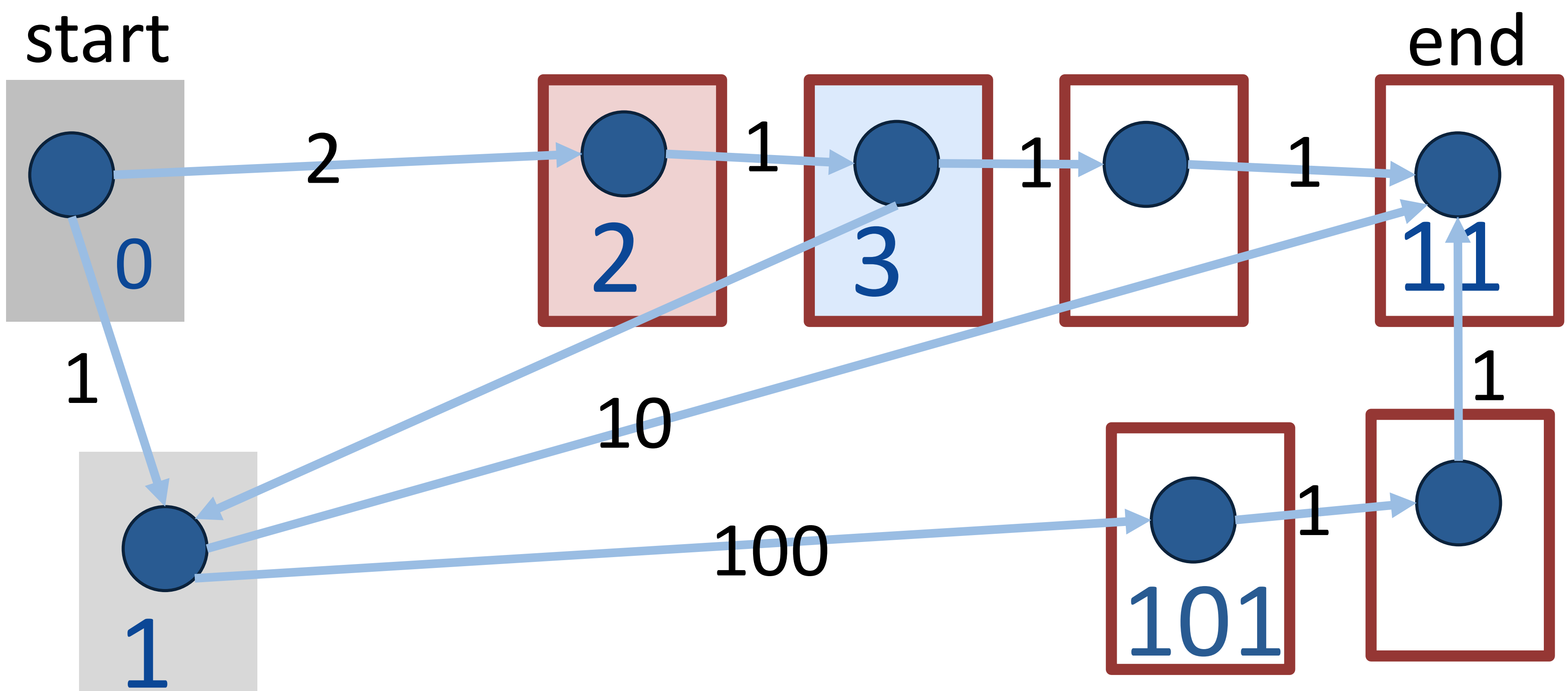
Dijkstra

- Take the path with the lowest cost (start 0)
- Expand its neighbors (filter out path we already know)



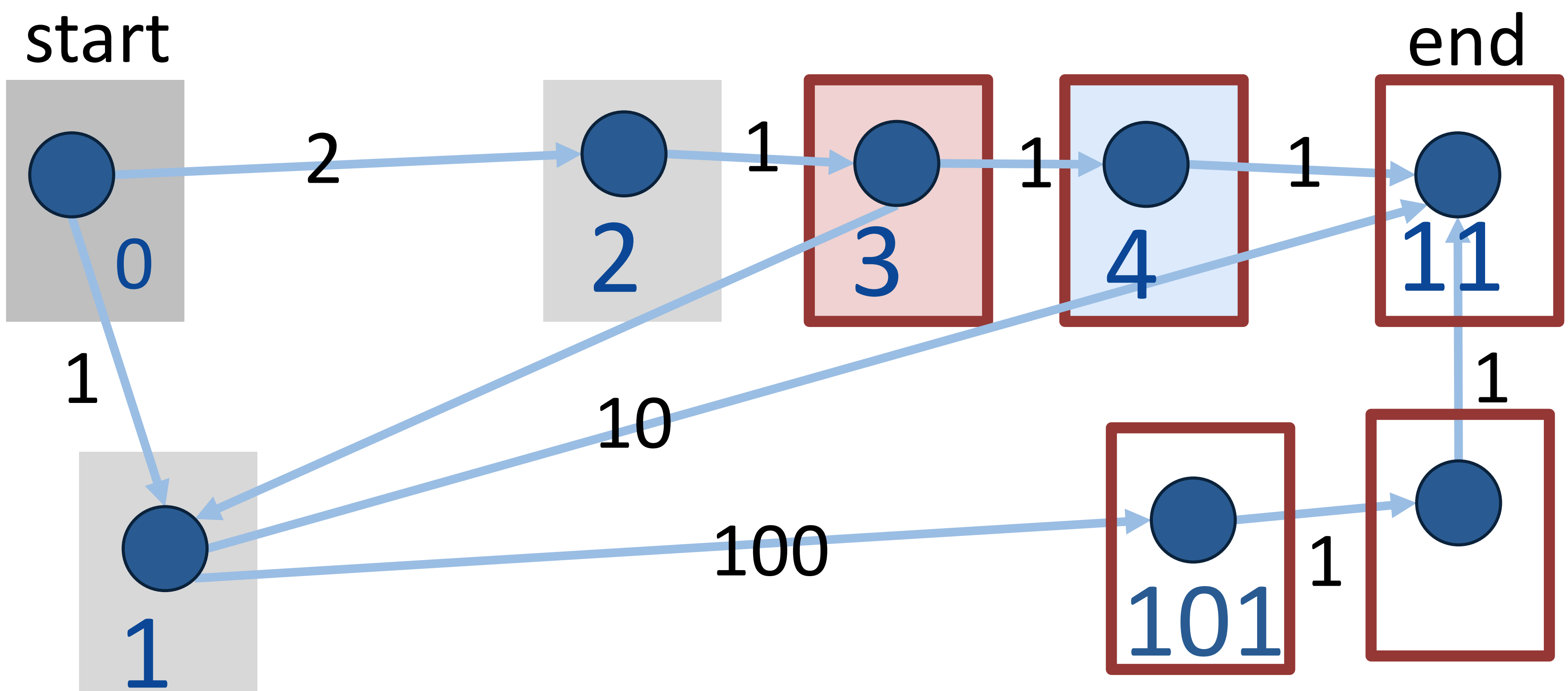
Dijkstra

- Take the path with the lowest cost (start 0)
- Expand its neighbors (filter out path we already know)



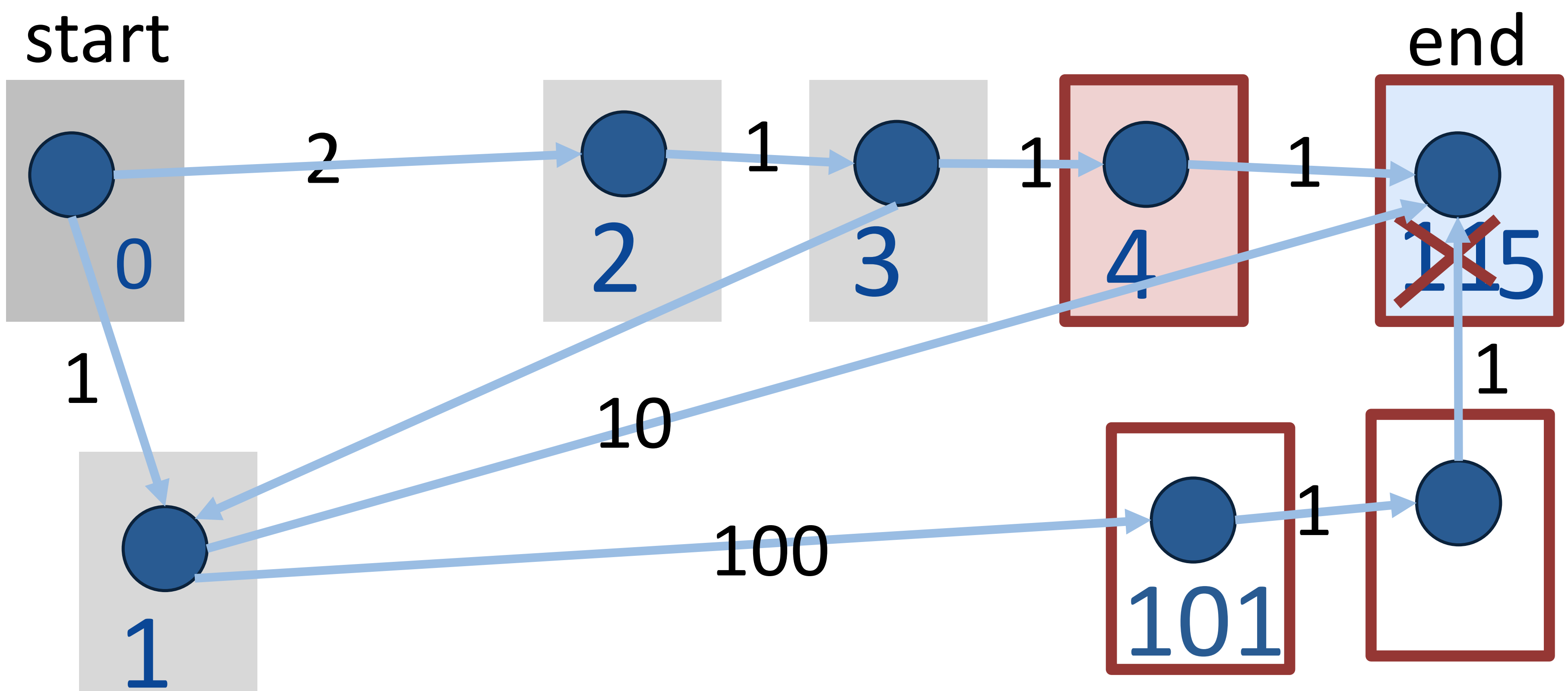
Dijkstra

- Take the path with the lowest cost (start 0)
- Expand its neighbors (filter out path we already know)



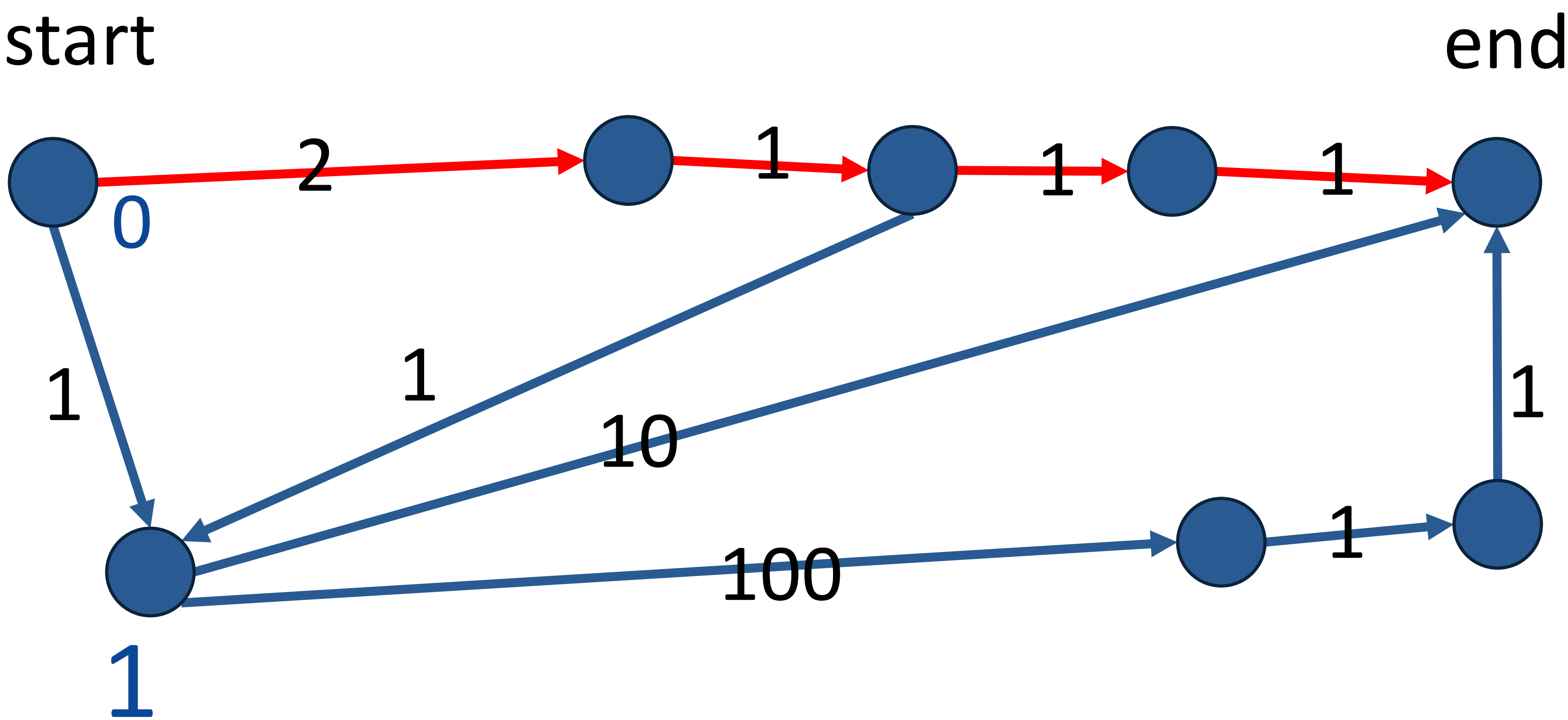
Dijkstra

- Take the path with the lowest cost (start 0)
- Expand its neighbors (filter out path we already know)



Dijkstra

- Shortest path ! Success !
- We even get the shortest distance from start to any node!



Dijkstra : Two Key Data Structures and Basic Idea

- (priority) queue: store best cost and path for discovered nodes that are
- finished: Similar to visited. Nodes that are on past frontiers

Helper methods (for the priority queue)

- update_node(queue, node, new_cost, new_path): add or update a path to the queue with the cost
- remove_min(queue): remove the path with the lowest cost from the queue

Outline of Algorithm

- For current node (initially start node), chose node with shortest distance from current node to visit first.
- Check each of its neighbors
  - Calculate distance from starting node to neighbor
- Terminate only if the target is contained in the minimum cost path
  - Note, that doesn't imply that we have to explore all possible path

Priority Queue Implementation

```python
def remove_min(queue):
    assert len(queue) > 0
    best_idx = 0
    min_so_far = queue[0]
    for idx in range(1, len(queue)):
        item = queue[idx]
        if item < min_so_far:
            best_idx = idx
            min_so_far = item
    return queue.pop(best_idx)
```

```python
def update_node(queue, node, new_cost, new_path):
    idx = find_node(queue, node)
    if idx is None:
        print(f"  Adding path to {node!r}")
        queue.append((new_cost, new_path))
        print(f"  Current queue: {queue}")
    else:
        old_cost, old_path = queue[idx]
        if new_cost < old_cost:
            print(f"  Updating path to {node!r}")
            queue[idx] = (new_cost, new_path)
            print(f"  Current queue: {queue}")
        else:
            print("  No change to queue")

def find_node(queue, node):
    for idx in range(len(queue)):
        (cost, path) = queue[idx]
        if path[-1] == node:
            return idx
    return None
```

Alternative implementation (heap)

- Searching through a list for the minimum and removing it is $O(n)$
- By using a data structure called a **heap**, can find minimum in $O(1)$ and remove in $O(\log n)$
  - Heaps are usually implemented on top of an underlying list data structure
  - The indicies have special meaning, effectively representing a kind of tree
- Python's **`heapq`** package provides a basic implementation

```
def dijkstra(graph, start, goal):
    # store best cost and path for discovered nodes that are
    # on present or future frontier
    queue = [(0, [start])]
    # separately, store nodes that are on past frontiers
    finished = set()

    while len(queue) > 0:
        print(f"Current queue: {queue}")

        # get a path off the true frontier
        cost, path = remove_min(queue)
        current_node = path[-1]
        finished.add(current_node)
        print(f"  Finished {current_node!r} with cost {cost}. Finished queue : {finished}")

        # optimality guaranteed for current node, return if goal
        if current_node == goal:
            return (cost, path)

        # update paths to neighbors on priority queue
        for edge in neighbors(graph, current_node):
            (next_node, weight) = edge
            if next_node not in finished:
                print(f"Processing {current_node!r}-->{next_node!r} with weight {weight}")
                new_cost = cost + weight
                new_path = path + [next_node]
                update_node(queue, next_node, new_cost, new_path)

        print()

    return None
```

Only terminate when priority queue is empty or if the true frontier contains the target

Take the node with the smallest cost (true frontier)

Expand the queue, except if the new path was already part of the true frontier (i.e., we know the shortest path already to that node)

If the next node wasn't part of the true frontier yet, we add it to the queue or update the path with its new cost

Example

Dijkstra's algorithm: key points

- On the queue, tag each discovered states with its projected frontier so far
  - Remember that we're actually storing paths on the queue, so we can return one that reaches a goal
- True BFS frontier lies at state/path with smallest cost
  - Pick such a state/path to expand/extend
  - **Guaranteed that that is a shortest path**
  - So never put that state back on the queue
- When extending a path to a neighbor already on the queue, update the state's path and cost if the cost is lower

Not LIFO or FIFO, but a priority queue

Effectively in the visited set

NOT considered visited

Takeaways and considerations

Best-first search

- **Dijkstra's algorithm** simulates running discretized BFS on weighted graphs
  - Requires edge weights to be non-negative (reasonably common assumption)
  - Efficient implementation uses heap-based priority queue
- **A* search (not covered)** extends Dijkstra's with a heuristic estimate of cost-to-go
  - Optimal paths guaranteed if heuristic is **admissible** and **consistent** (i.e., always an underestimate everywhere)
- General framework for shortest paths in weighted graphs is called **best-first search** or **informed search**
  - Dijkstra's or close variants lacking a heuristic are called **uniform-cost search**

Summarizing

- Graphs are powerful modeling framework
  - Capture relationships among objects
    - **Local structure** composes into networks
  - Can then infer **global properties**, and optimize them
- Many important problems can be framed as finding a **shortest path** on a graph
  - **DFS** and **BFS** are fundamental algorithms for solving it
- Dijkstra's algorithm better for finding shortest path in large graphs with (non-negative) weighted edges
  - Many variants optimized for specific applications