L2: Shortest Paths in *Weighted* Graphs

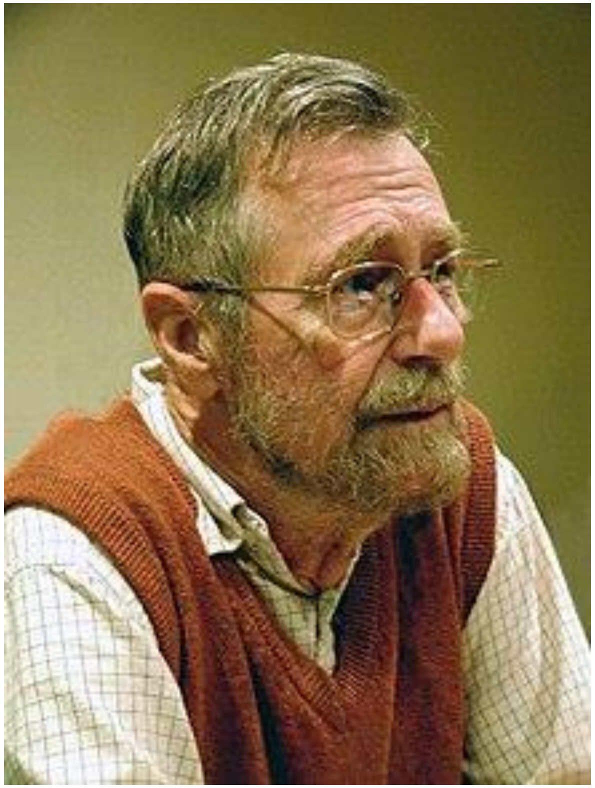
Alex Steiger

CompSci 201: Spring 2024

4/10/2024

Person in CS: Edsger Dijkstra

- Dutch computer scientist, 1930 – 2002.
- PhD in 1952, Turing award in 1972.
- Originally planned to study law, switched to physics, then to computer science.
- "After having programmed for some three years…I had to make up my mind, either to…become a…theoretical physicist, or to …become….. what? A programmer? But was that a respectable profession?…Full of misgivings I knocked on Van Wijngaarden's office door, asking him whether I could "speak to him for a moment"; when I left his office a number of hours later, I was another person. For after having listened to my problems patiently…he went on to explain quietly that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come?"



Logistics, coming up

- Today, Wednesday, April 10
  - APT Quiz 2 due
  - Covers linked list and trees
  - No regular APTs due this week, just the quiz

- Next Wednesday, 4/17
  - Midterm exam 3
    - Practice exams coming soon to Canvas
  - APT 9 extended to Thursday 4/20

Midterm Exam 3

- Logistics:
  - 60 minutes, in-person, multiple-choice + fill-in-blank
  - Can bring 1 reference/notes page

- Topics could include:
  - Trees, binary search trees, binary heaps, recursion
  - AVL trees: High-level concept of balance factor/rotations, yes, details of performing rotations, no.
  - Greedy, Huffman
  - Graphs, DFS, BFS, Dijkstra's

Today's agenda

- Finish WordLadder Problem

- Shortest paths in weighted graphs:
  - Dijkstra's algorithm

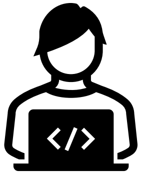
Example WordLadder Problem

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the **number of words** in the **shortest transformation sequence** from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

leetcode.com/problems/word-ladder/description/



Live coding

L24-WOTO1-WordLadder-Sp24

Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *

solutions

2

Suppose you have:

beginWord = "cat"

endWord = "dog"
wordList = ["hot","dot","dog","lot","log","cog", "cot"]

The length of the shortest word ladder is... *

4

3

Consider this makeGraph method, part of a correct solution to the wordLadder problem. Assume the oneOff method correctly returns true if two strings differ by a single character and false otherwise, and runs in O(1) time.

If N is the length of the wordList, what is the asymptotic runtime complexity of the makeGraph method as a function of N? *

```java
private Map<String, HashSet<String>> makeGraph(List<String> wordList) {
    Map<String, HashSet<String>> aList = new HashMap<>();
    for (String w: wordList) {
        aList.put(w, new HashSet<>());
        for (String other: wordList) {
            if (oneOff(w, other)) {
                aList.get(w).add(other);
            }
        }
    }
    return aList;
}
```

- O(1)
- O(N)
- O(N log(N))
- (selected) O(N^2)
- O(N^2 log(N))

4

Consider this code, part of a correct solution to the wordLadder problem. It works with an adjacency list representation aList such as would be generated by the makeGraph method.

If there are N words in total in the wordList, and each word can be transformed into at most a constant number O(1) other words by changing a single character, then what is the runtime complexity of this code? *

```java
    Queue<String> toExplore = new LinkedList<>();
    Map<String, Integer> ladderLength = new HashMap<>();
    toExplore.add(beginWord); ladderLength.put(beginWord, value:1);

    while (toExplore.size() > 0) {
        String word = toExplore.remove();
        for (String other : aList.get(word)) {
            if (!ladderLength.containsKey(other)) {
                ladderLength.put(other, ladderLength.get(word)+1);
                toExplore.add(other);
            }
        }
    }
    return ladderLength.getOrDefault(endWord, defaultValue:0);
}
```

- ○ O(1)
- ◉ O(N)
- ○ O(N log(N))
- ○ O(N^2)
- ○ O(N^2 log(N))

5

For the approach to the wordLadder problem outlined above, what dominates the runtime complexity of the algorithm? Assume that each word is at most a constant length and that each word can be transformed into at most a constant number of other words. *

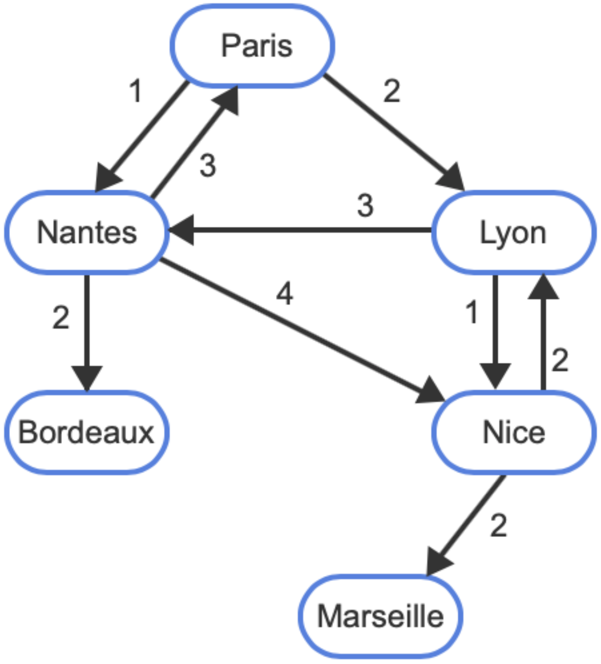
- (selected) Building the graph takes most of the time
- Running the search on the graph takes most of the time

Weighted Graphs and Dijkstra's Algorithm

Weighted Graphs

Each edge has an associated **weight** representing cost, distance, etc.

In mapping applications, maybe one road is twice as long as another.

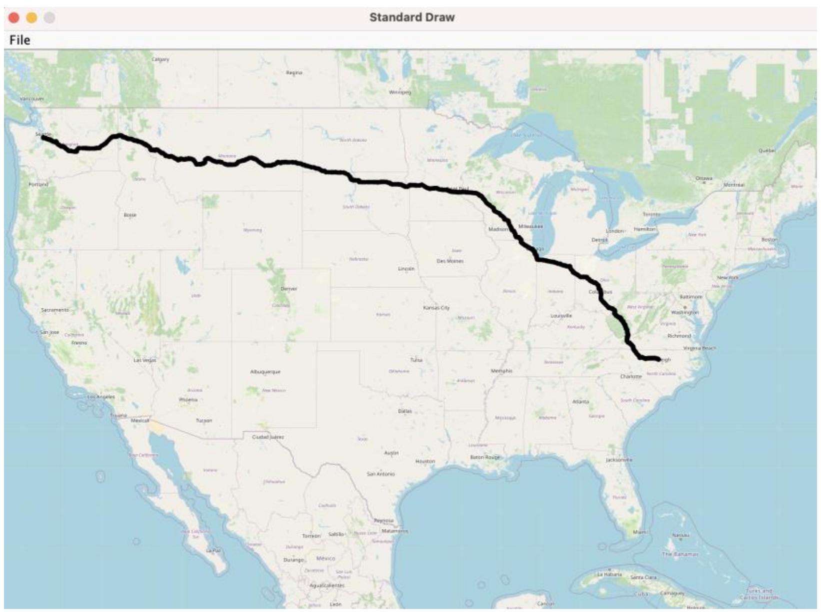


Zybook chapter 24

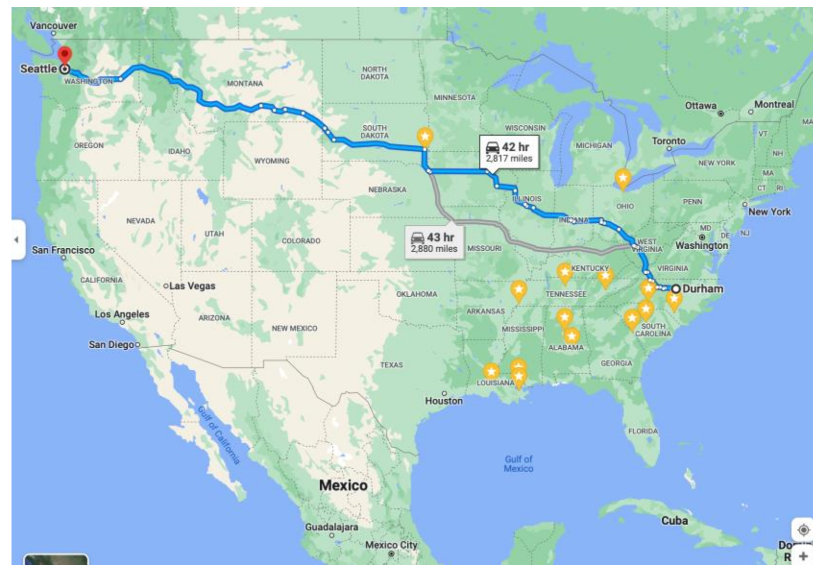
Project 6: Route

Durham, NC → Seattle WA,
~2800 miles

Project 6



Google Maps Directions



Project 6: Route Demo

Partner project, can work (and submit) with one other person. Make sure to read the directions on using Git with a partner, and to submit together on gradescope.

GraphProcessor: Implement algorithms with real-world graph data.

No analysis for this project.

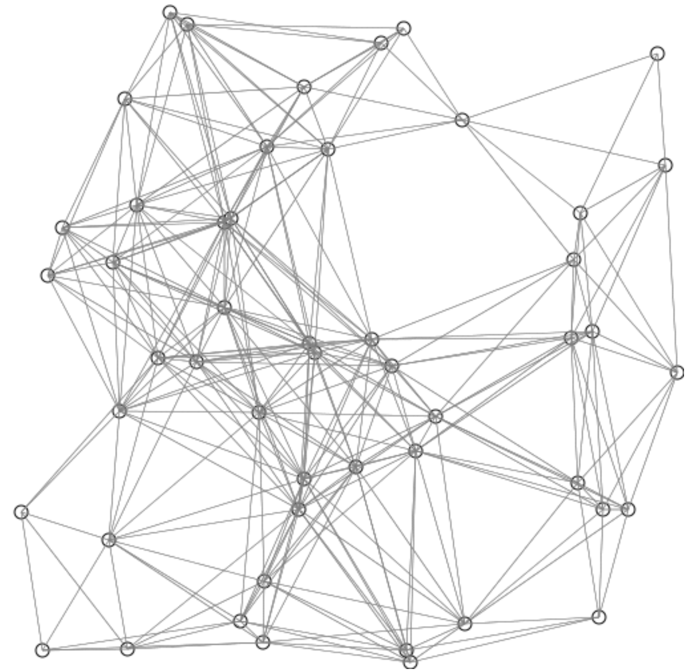
Shortest weighted paths?

- BFS gives shortest paths in *unweighted* graphs.
- Modify BFS to account for weights; called Dijkstra's algorithm.
- BFS = queue, Dijkstra's = …
  - Priority queue!

Exploring a node with Dijkstra’s Algorithm, Pseudocode

While unexplored nodes remain

- Explore current = the closest unexplored node
- For each neighbor:
  - Update shortest path to neighbor if shorter to go through current



wikipedia.org/wiki/Dijkstra%27s_algorithm

Just like BFS (explore closer nodes first) except… now we need to account for weights.

"Textbook" Dijkstra Initialization

- Initialize distances to:
  - 0 for the start node,
  - Infinity for everything else
- Add all nodes to a priority queue, using their distance as the priority

```java
public Map<Character, Integer> textbookDijkstra(char start, Map<Character, List<Character>> aList) {
    Map<Character, Integer> distance = new HashMap<>();
    for (char c : aList.keySet()) { distance.put(c, Integer.MAX_VALUE); }
    distance.put(start, value:0);
    Comparator<Character> comp = (a, b) -> distance.get(a) - distance.get(b);
    PriorityQueue<Character> toExplore = new PriorityQueue<>(comp);
    toExplore.addAll(aList.keySet());
```

“Textbook” Dijkstra Exploration

- While there are unexplored nodes:
  - Get the *closest* unexplored node to the start
  - Look at all neighbors:
    - If the path through current is shorter:
      - Update distance, update priority in priority queue

```java
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    for (char neighbor : aList.get(current)) {
        int newDist = distance.get(current) + getWeight(current, neighbor);
        if (newDist < distance.get(neighbor)) {
            distance.put(neighbor, newDist);
            //toExplore.decreasePriority(neighbor);
        }
    }
}
return distance;
```

Practical Dijkstra Initialization

Like the previous implementation, but only add vertices to the queue once they are actually reached/visited.

```
public Map<Character, Integer> stdDijkstra(char start, Map<Character, List<Character>> aList) {
    Map<Character, Integer> distance = new HashMap<>();
    distance.put(start, 0);
    Comparator<Character> comp = (a, b) -> distance.get(a) - distance.get(b);
    PriorityQueue<Character> toExplore = new PriorityQueue<>(comp);
    toExplore.add(start);
```

Don't need to add anything for all nodes yet.

Practical Dijkstra search loop

Keep searching while there are unexplored nodes.

Choose to explore from the *next closest (to* **start***) unexplored node to start* at each iteration.

```
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    int currDist = distance.get(current);
    for (char neighbor : aList.get(current)) {…
}
return distance;
```

Search all neighbors of current. If you find a ***shorter path*** to neighbor through current, update to reflect that.

Details: Checking each neighbor

All neighbors of current node

Distance to neighbor through current = distance to current + weight on edge from current to neighbor

```
for (char neighbor : aList.get(current)) {
    int newDist = currDist + getWeight(current, neighbor);
    if (!distance.containsKey(neighbor)) {
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    }
    else if (newDist < distance.get(neighbor)) {
        // implement decreasePriority by removal and re-insertion
        toExplore.remove(neighbor);
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    }
}
```

If neighbor newly discovered:

- Record new distance
- Add to priority queue

If neighbor already discovered, update:

- Remove from PQ
- Record new distance
- Add back to PQ

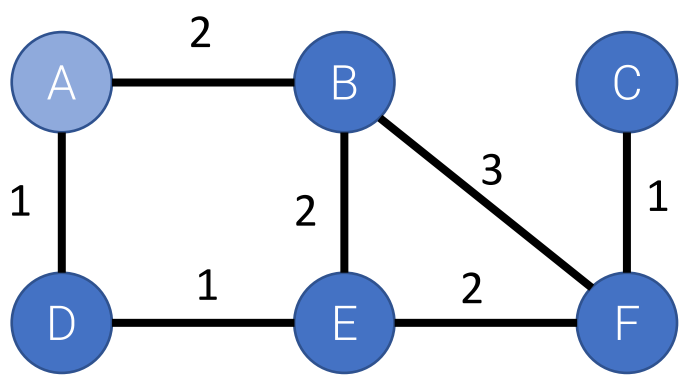
Implementing decreasePriority

- Most standard library binary heaps (including java.util) don't support an efficient update/decrease priority operation.

```
else if (newDist < distance.get(neighbor)) {
    // implement decreasePriority by removal and re-insertion
    toExplore.remove(neighbor);
    distance.put(neighbor, newDist);
    toExplore.add(neighbor);
}
```

- Our code works, but is O(N) time
  - Java's PQ takes O(N) to remove ***given*** node (O(1) for smallest)
  - Other PQ implementations support O(log N)-time decreasePriority, but they are not in Java library

Initialize search at A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

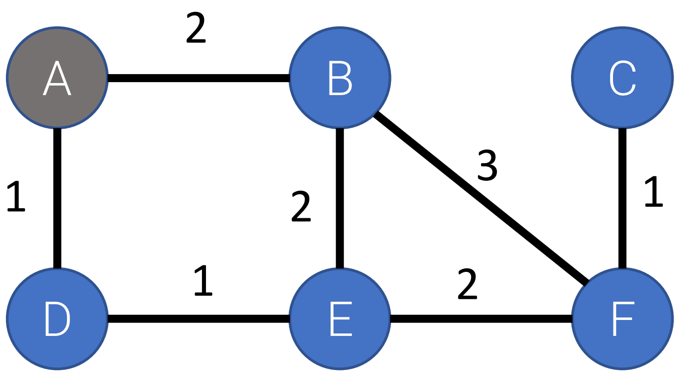
A

previous (map)

distance (map)

A = 0

Remove A from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

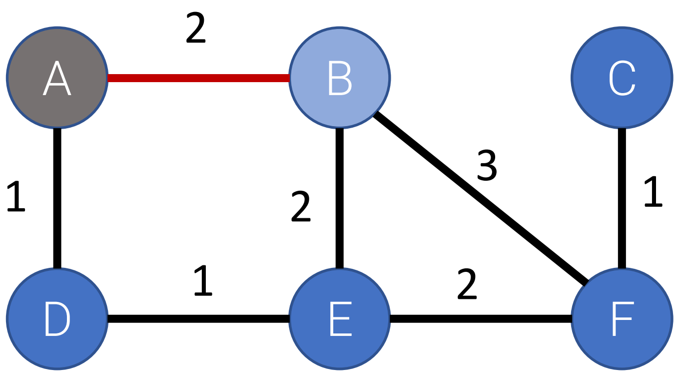
toExplore (PriorityQueue)

previous (map)

distance (map)

A = 0

Find B from A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

B

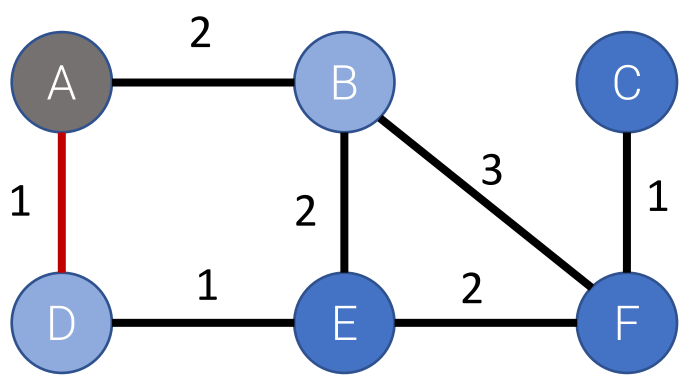
previous (map)

B <- A

distance (map)

A = 0
B = 2 (A + 2)

Find D from A



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

D
B

D comes first because lower distance/prio.

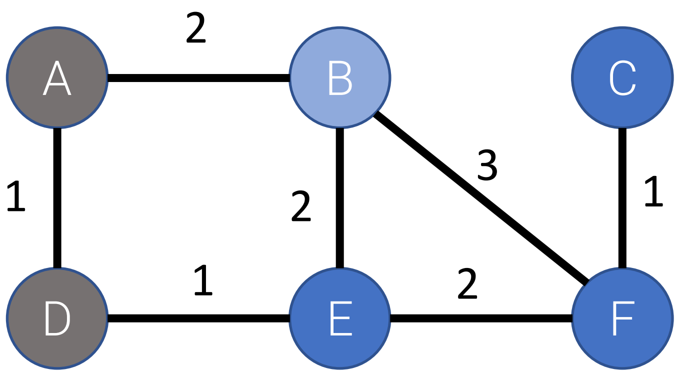
previous (map)

B <- A
D <- A

distance (map)

A = 0
B = 2
D = 1 (A + 1)

Remove D from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

B

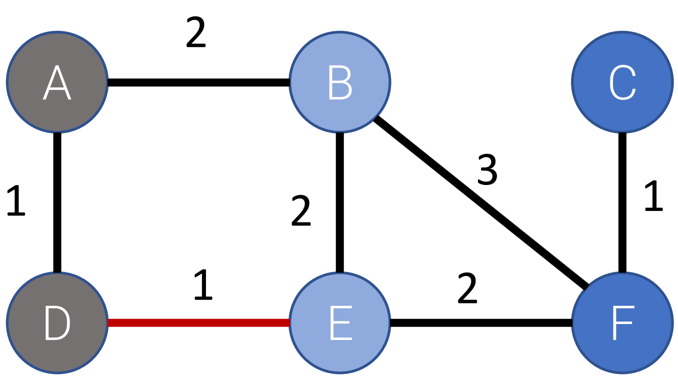
previous (map)

B <- A
D <- A

distance (map)

A = 0
B = 2
D = 1

Find E from D



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

B
E

B and E are tied in distance, suppose B comes first

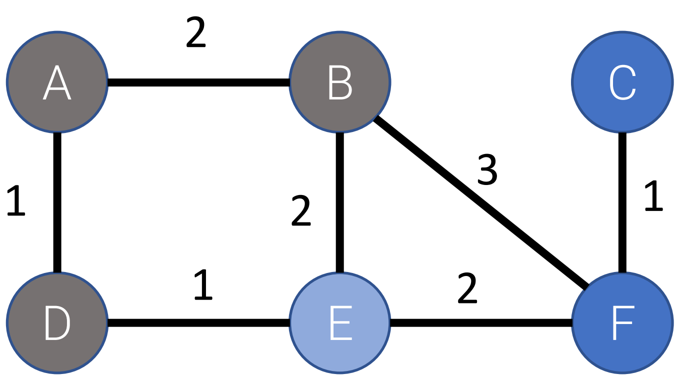
previous (map)

B <- A
D <- A
E <- D

distance (map)

A = 0
B = 2
D = 1
E = 2 (D + 1)

Remove B from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

E

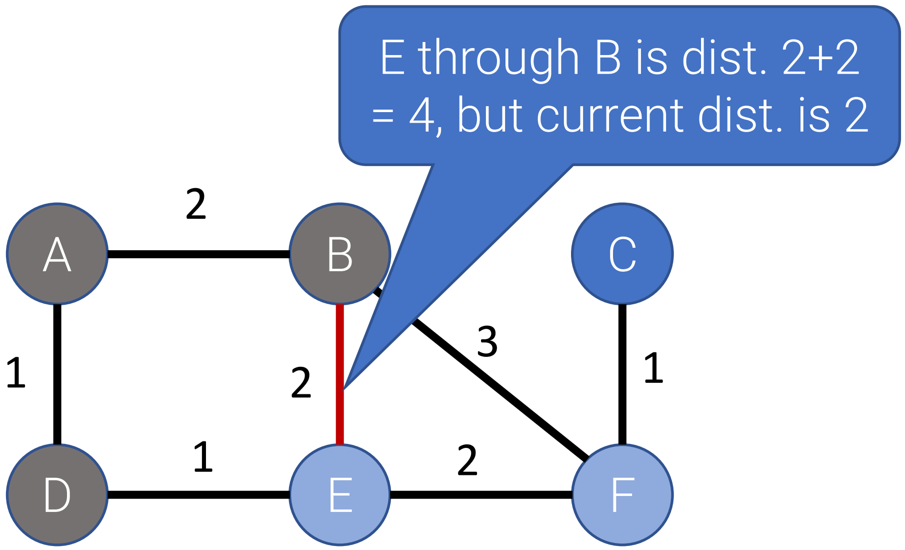
previous (map)

B <- A
D <- A
E <- D

distance (map)

A = 0
B = 2
D = 1
E = 2

Find **longer** path to E from B



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

E
F

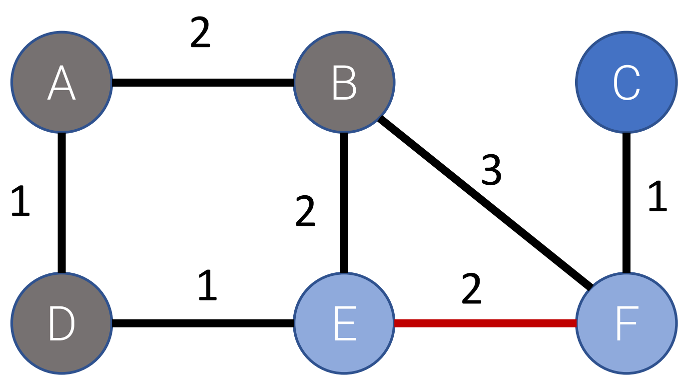
previous (map)

B <- A
D <- A
E <- D
F <- B

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 5

Find F from B



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

E
F

E has lower distance

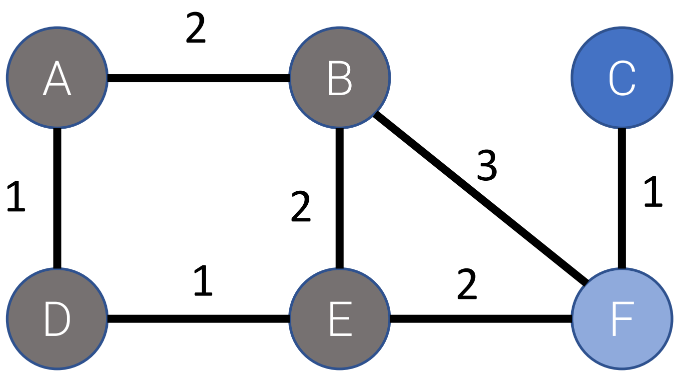
previous (map)

B <- A
D <- A
E <- D
F <- B

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 5 (B + 3)

Remove E from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

F

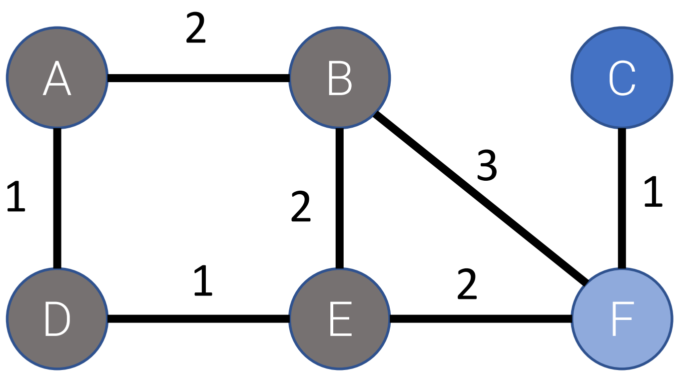
previous (map)

B <- A
D <- A
E <- D
F <- B

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 5

Find **shorter** path to F from E



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

F

previous (map)

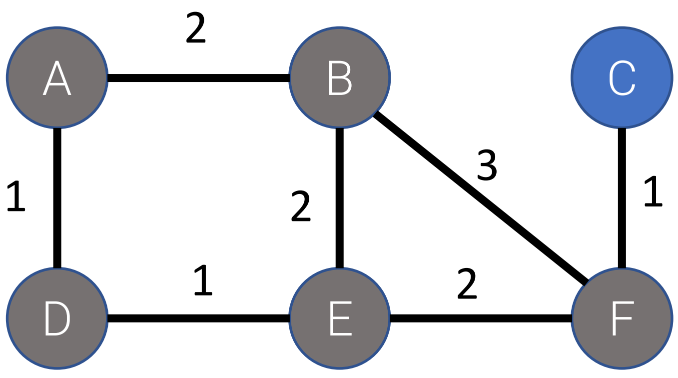
B <- A
D <- A
E <- D
F <- E (instead of B)

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 5 -> 4 (E + 2)

Decrease dist./prio. to 5 from 4

Remove F from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

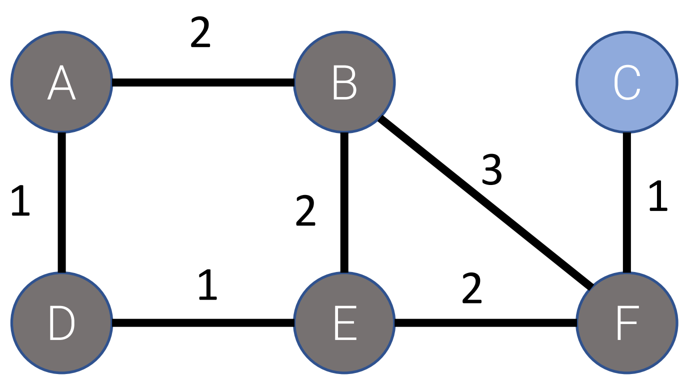
previous (map)

B <- A
D <- A
E <- D
F <- E

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 4

Find C from F



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

F
C

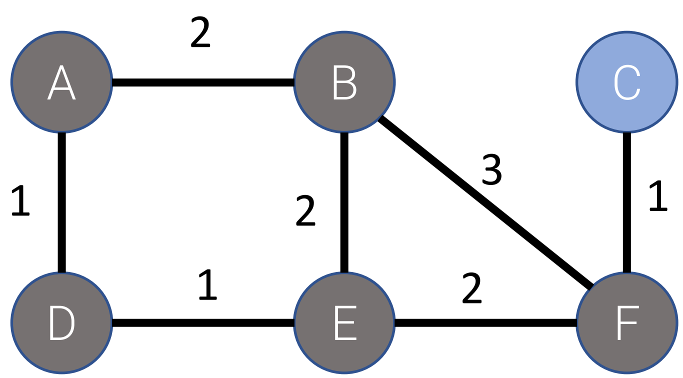
previous (map)

B <- A
D <- A
E <- D
F <- E
C <- F

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 4
C = 5 (F + 1)

Remove old F from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

C

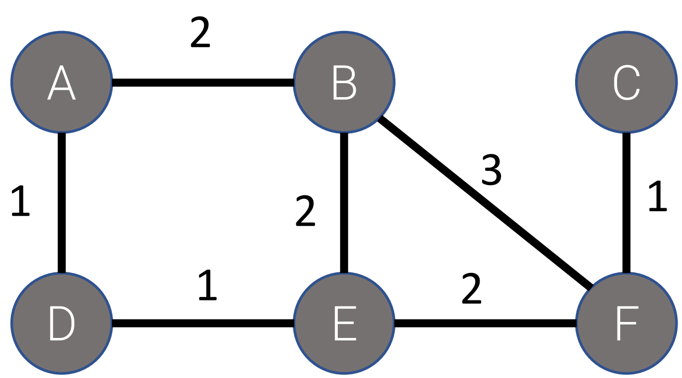
previous (map)

B <- A
D <- A
E <- D
F <- E
C <- F

distance (map)

A = 0
B = 2
D = 1
E = 2
F = 4
C = 5

Remove C from PriorityQueue



Adjacency List:
A=[B, D]
B=[A, E, F]
C=[F]
D=[A, E]
E=[B, D, F]
F=[B, C, E]

toExplore (PriorityQueue)

previous (map)

B <- A
D <- A
E <- D
F <- E
C <- F

distance (map)

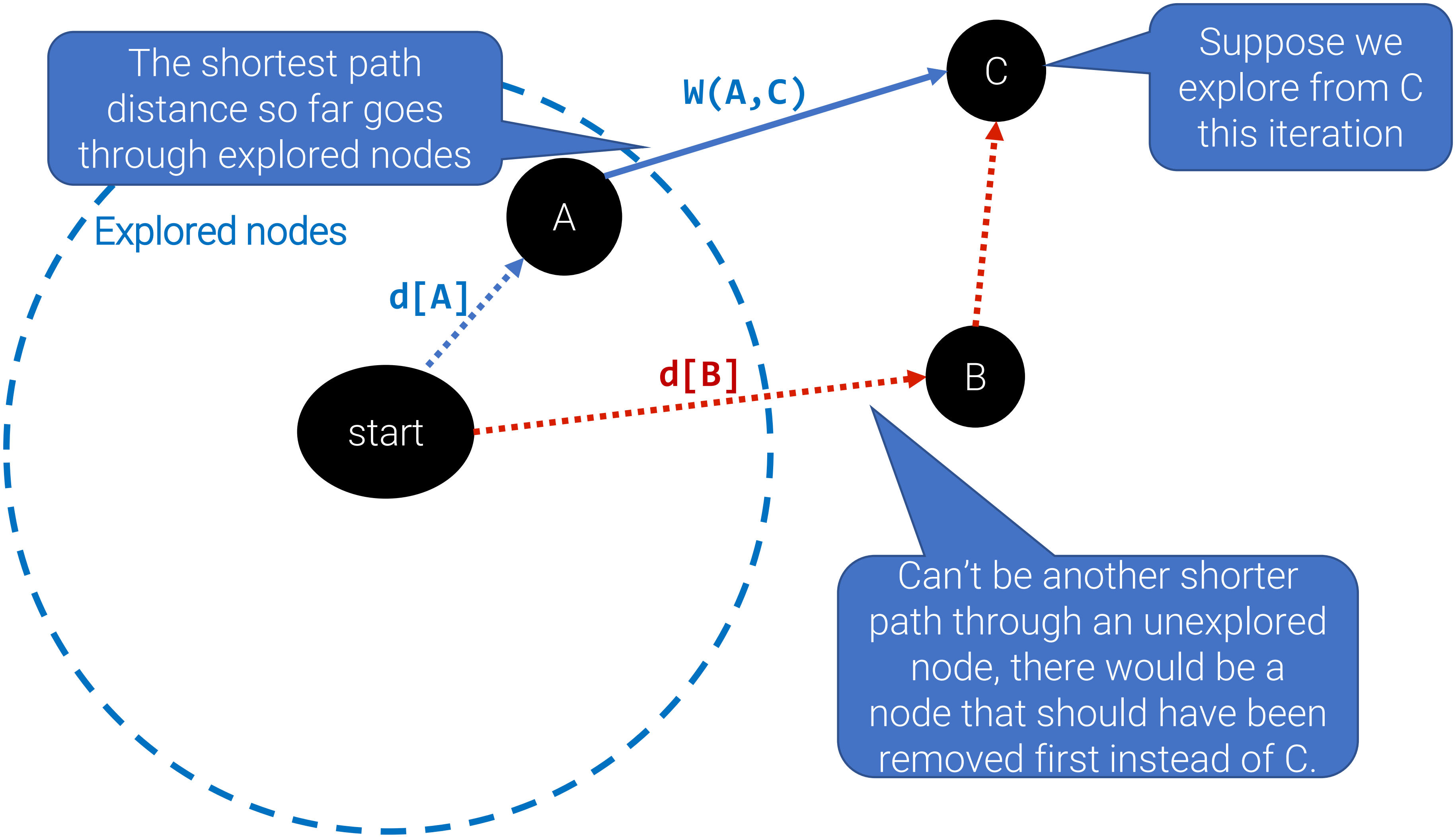
A = 0
B = 2
D = 1
E = 2
F = 4
C = 5

Is Dijkstra's algorithm guaranteed to be correct? (Informal)

- **Claim.** Distance is correct shortest path distance for all nodes ***explored*** so far, and shortest path distance ***through explored nodes*** for all others.

- Formal proof is ***by induction,*** see CompSci 230.
  - Assume the property is true up to some point in the algorithm, then…
  - Consider the next node we explore:

Is Dijkstra's algorithm guaranteed to be correct? (Informal)



Runtime Complexity of Dijkstra's Algorithm (with N nodes, M edges) *assuming O(log N) decreasePriority*

N iterations?

O(log(N)), heap

```
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    for (char neighbor : aList.get(current)) {⋯
}
return distance;
```

Iterations over neighbors

O(1) in HashMap, O(log(N)) in TreeMap

Like BFS, consider each node once and each edge twice, log(N) operations for each: **O((N+M)log(N))**