CS 2110
Lecture 14

Shortest paths
  • Dijkstra’s algorithm
  • Back pointers
Cycles

- A directed graph can be topologically ordered if and only if it has no cycles.

- A **cycle** is a path $v_0, v_1, ..., v_p$ such that $v_0 = v_p$.

- A graph is **acyclic** if it has no cycles.

- A directed acyclic graph is a **DAG**.
Is this graph a DAG?

• If a node is part of a cycle, it must have an incoming edge
• Deleting a node with indegree zero would not remove any cycles
• Keep deleting such nodes and see whether graph “disappears”

Yes! It was a DAG.

The order in which we removed nodes was a topological order!
Shortest paths

Single source shortest path algorithms
Unweighted graphs

• “Shortest” means “smallest path length” (fewest edges)

• Poll: How long are the shortest paths from s to a, b, c, & d?
  • A: 1, 2, 3, 4
  • B: 1, 2, 2, 3
  • C: 1, 1, 2, 2
  • D: 3, 4, 3, 2
Breadth-first search

• Shortest path length = BFS layer

• Recall: frontier queue is sorted by (unweighted) distance from start

• When visiting a vertex in layer $n$, all neighbors are either:
  • Undiscovered: they are in layer $n+1$
  • Discovered: they are in layer $n$ or $n-1$

<table>
<thead>
<tr>
<th>Frontier queue</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex state</td>
<td></td>
</tr>
<tr>
<td>Vertex</td>
<td>s</td>
</tr>
<tr>
<td>Layer</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm: BFS layer

- Keep track of each node’s layer
  - Only update when node is first discovered
  - Equal to: (layer of predecessor) + 1

```java
start.discovered = true;
start.layer = 0;
frontier.add(start);
while (!frontier.isEmpty()) {
    Vertex v = frontier.remove();
    for (Vertex neighbor : v.successors) {
        if (!neighbor.discovered) {
            neighbor.discovered = true;
            neighbor.layer = v.layer + 1;
            frontier.add(neighbor);
        }
    }
}
```
What about the path?

• Whenever a new vertex is discovered, need to record the path taken to get there

• But most of these paths are \textit{redundant}
  • Shortest path from start to neighbor = shortest path from start to v + edge from v to neighbor

• Back pointers
  • Only need to store edge from discovery predecessor
  • By following edges backwards, can reconstruct path

• Note: shortest path may not be unique; this will only yield one of them
  • Depends on neighbor iteration order
Back pointers

<table>
<thead>
<tr>
<th>Vertex</th>
<th>s</th>
<th>state</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Pred</td>
<td>null</td>
<td>s</td>
<td>s</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Code: unweighted shortest paths

```java
start.layer = 0;
frontier.add(start);
while (!frontier.isEmpty()) {
    Vertex v = frontier.remove();
    for (Vertex neighbor : v.successors) {
        if (neighbor.pred == null && neighbor != start) {
            neighbor.pred = v;
            neighbor.layer = v.layer + 1;
            frontier.add(neighbor);
        }
    }
}
}
```

```java
List<Vertex> pathTo(Vertex end) {
    List<Vertex> path = new LinkedList<>();
    Vertex v = end;
    while (v != null) {
        path.add(0, v);
        v = v.pred;
    }
    return path;
}
```

Don’t need to track “discovered” separately, since layer and pred are only assigned for discovered vertices.
Example: train routes
Weighted shortest path

- “Shortest” = “smallest sum of weights”

- Changes
  - FIFO queue is not sorted by weight
  - When a neighbor is first discovered, the discovery path might not be the shortest path

- Poll: What is the weight of the shortest path from s to c?
  - A: 4
  - B: 5
  - C: 6
  - D: 7
History

- Described by Edsger Dijkstra in 1956 (26 years old)
- No theory of efficiency (big-Oh notation) to compare against alternatives
  - “my solution is preferred to another one … “the amount of work to be done seems considerably less.”
If a shorter path is found to a neighbor that was already discovered, update that neighbor’s dist, pred.

Problem: what if neighbor was already visited & settled?

- Would invalidate its neighbors’ shortest paths, and so on...
- This sounds messy
When do we know that a path is the shortest?

- Shortest path to any undiscovered vertex must be longer than the \textit{smallest} candidate path to a vertex in the frontier.
- Therefore, the \textit{smallest candidate path in the frontier} must be the \textit{shortest possible path} to that vertex.
  - Any new path discovered in the future must go through a vertex in the frontier and would therefore be longer.
Invariant and theorem

• “Settled set” – vertices whose neighbors have all been discovered
  • Path distance is shortest possible
  • Neighbors are either in the frontier or are settled themselves

• For each frontier vertex, we know the shortest path so far, and it only goes through settled vertices
  • Paths are only updated when v is being visited, about to be settled

• Theorem:
  • If f is the vertex in the frontier with the smallest candidate path, then that path is the shortest possible path from start to f
    • Consequence: f is ready to be settled
Checkpoint

• If $f$ is the vertex in the frontier with the smallest candidate path, then that path is the shortest possible path from start to $f$

• Poll: What does the theorem guarantee about this frontier?
  - Cortland: 20 mi
  - Enfield: 10 mi
  - Dryden: 11 mi
  - Trumansburg: 15 mi
Why the theorem?

- Looking for patterns (invariants, theorems) can suggest good implementations
- For rich data structures like graphs, thorough testing is difficult, line coverage is not a good indicator
  - The data is richer than the code
Algorithm

• Like BFS, but:
  • Track dist (weight) instead of layer
  • Use priority queue instead of FIFO queue for frontier
    • Always remove node with smallest candidate path
• (Don’t need to explicitly track settled set)

```java
start.discovered = true;
start.dist = 0;
frontier.add(start, start.dist);
while (!frontier.isEmpty()) {
  Vertex v = frontier.removeMin();
  for (Edge e : v.outgoing) {
    Vertex neighbor = e.to;
    double dist = v.dist + e.weight;
    if (!neighbor.discovered || dist < neighbor.dist) {
      neighbor.discovered = true;
      neighbor.dist = dist;
      frontier.addOrUpdate(neighbor, dist);
    }
  }
}
```
Example

```java
start.discovered = true;
start.dist = 0;
frontier.add(start, start.dist);
while (!frontier.isEmpty()) {
    Vertex v = frontier.removeMin();
    for (Edge e : v.outgoing) {
        Vertex neighbor = e.to;
        double dist = v.dist + e.weight;
        if (!neighbor.discovered || dist < neighbor.dist) {
            neighbor.discovered = true;
            neighbor.dist = dist;
            frontier.addOrUpdate(neighbor, dist);
        }
    }
}
```
Priority queue ADT

• Operations
  • add(elem: E, priority)
  • removeMin(): E

• Optional (but needed for shortest paths)
  • updatePriority(elem: E, priority)

• Can implement with linear search, but inefficient
• Next lecture: well-suited data structure (heap)
Complexity analysis

while (!frontier.isEmpty()) {
    Vertex v = frontier.removeMin();
    for (Edge e : v.outgoing) {
        Vertex neighbor = e.to;
        double dist = v.dist + e.weight;
        if (dist < neighbor.dist) {
            neighbor.dist = dist;
            frontier.addOrUpdate(neighbor, dist);
        }
    }
}
Extensions

• Dijkstra’s algorithm finds the shortest paths from a single source to all reachable destinations

• What about just finding the shortest path to a specific target?
  • Example: navigating from point A to point B

• Can just run Dijkstra’s and stop when target is settled

• Can settle target sooner if you know a lower bound distance between any two points
  • E.g. straight line distance on map
  • A* algorithm
Heaps

- Priority queues
- Heapsort
Wrapup: shortest paths
Complexity analysis

```java
while (!frontier.isEmpty()) {
    Vertex v = frontier.removeMin();
    for (Edge e : v.outgoing) {
        Vertex neighbor = e.to;
        double dist = v.dist + e.weight;
        if (dist < neighbor.dist) {
            neighbor.dist = dist;
            frontier.addOrUpdate(neighbor, dist);
        }
    }
}

Big-Oh: \( O(R_{\text{worst}} V + A_{\text{worst}} E) \)
With heap: \( R_{\text{worst}} A_{\text{worst}} \subseteq O(\log V) \), so complexity of Dijkstra’s algorithm is: \( O((V + E) \log V) \)
```

<table>
<thead>
<tr>
<th>Self cost</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( V )</td>
</tr>
<tr>
<td>( R ) (cost of removeMin())</td>
<td>( &lt; V R_{\text{worst}} )</td>
</tr>
<tr>
<td>outdegree(v)</td>
<td>( E )</td>
</tr>
<tr>
<td>1</td>
<td>( E )</td>
</tr>
<tr>
<td>1</td>
<td>( E )</td>
</tr>
<tr>
<td>1</td>
<td>( E )</td>
</tr>
<tr>
<td>( A ) (cost of add/update)</td>
<td>( &lt; E A_{\text{worst}} )</td>
</tr>
</tbody>
</table>

\[ (1 + R_{\text{worst}})V + (5 + A_{\text{worst}})E \]
Extensions

• Dijkstra’s algorithm finds the shortest paths from a single source to all reachable destinations
• What about all sources?
• What about just finding the shortest path to a specific target?
  • Example: navigating from point A to point B
• Can just run Dijkstra’s and stop when target is settled
• Can settle target sooner if you know a lower bound distance between any two points
  • E.g. straight line distance on map
  • A* algorithm
Priority queues
Common pattern: give me the “next” thing

Different choices for “next”:

- Queue (FIFO): who has been waiting the longest?
- Stack (LIFO): who was added most recently?
- **Priority queue: who is most important?**

Applications:

- Shortest paths
- Task deadlines (what is due soonest, regardless of when it was assigned)
Desired functionality

- `peek()`: Return the most important element
- `poll()`: Remove and return the most important element (aka “pop”)
- `add()`: Add a new element

Want these operations to be *fast* (low time complexity)

- Ideally, `peek()` should be $O(1)$ (always know what the best value is)
- `poll()` and `add()` must preserve whatever invariant makes `peek()` fast without being slow themselves
Candidate implementations ($N$ elements)

- Unsorted list
  - peek: $O(N)$, poll: $O(N)$, add: $O(1)$

- Sorted list
  - peek: $O(1)$, poll: $O(1)$, add: $O(N)$

- Balanced binary search tree (BST)
  - peek: $O(\log N)$, poll: $O(\log N)$, add: $O(\log N)$
  - Balancing is complicated
Do we need/want to keep elements sorted?

Often, processing one element (poll) will cause many new elements to be added to the queue (add).

- E.g. exploring a cave: take the right fork, but at the end of that tunnel, three new tunnels open up

Keeping all these TODOs sorted is wasteful – we’ll keep having to move things around when new tasks come in, and all we care about is which one is next

Strategy: relax invariant
Binary heaps (data structure)

Do not confuse with “heap memory” – different use of the word “heap”:
- “The stack”: Local variables live in activation records on the call stack
- “The heap”: Objects are allocated in memory apart from the call stack
A Heap...

Is a binary tree satisfying 2 properties:

1. **Heap-ordered** (order invariant). Every node is “more important” than its children
   - **Min-heap**: every node is $\leq$ its children (smallest on top) “earliest deadline,” “shortest distance”
   - **Max-heap**: every node is $\geq$ its children (biggest on top) “largest reward”
Heap-order (max-heap)

Every element is <= its parent

Note: Bigger elements can be deeper in the tree!
A Heap...

Is a binary tree satisfying 2 properties:

1. **Heap-ordered** (order invariant). Every node is “more important” than its children

2. **Completeness** (shape invariant). Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.
Completeness

Every level (except last) completely filled.

Nodes on bottom level are as far left as possible.
Completeness

Not a heap because:

- missing a node on level 2
- bottom level nodes are not as far left as possible
Perfect binary tree

(Not a heap!)

Perfect binary tree with $2^k-1$ nodes has $k$ levels

Add one more node: $2^k$ nodes has $k+1$ levels

$n$ nodes has $\lceil \log(n + 1) \rceil \leq O(\log n)$ levels
Which of the following are valid max-heaps?

(a) none of them
Poll 1

Which of the following are valid heaps?

(a) No
(b) No
(c) No
(d) Yes
Back to priority queues

Heaps can implement priority queues

• Efficiency we will achieve (storing $N$ elements):
  • add(): $O(\log N)$
  • poll(): $O(\log N)$
  • peek(): $O(1)$

• No linear time operations: better than lists
• peek() is constant time: better than balanced trees
Heap algorithms
Exercise: Adding an element

Must preserve:
1. Shape invariant
2. Order invariant

Goal: minimize changes
What is 50's left child?

A. 2
B. 19
C. 22
D. 38
E. Nothing
Heap: add(e)

1. Put in the new element in a new node (leftmost empty leaf)
Heap: add(e)

1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

Time is $O(\log n)$
Heap: peek()

1. Return root value

Time is $O(1)$
Heap: `poll()` (aka `remove()`)

1. Save root element in a local variable
Heap: `poll()` (aka `remove()`)

Must preserve:
1. Shape invariant
2. Order invariant

Goal: minimize changes

1. Save root element in a local variable
2. Assign last value to root, delete last node.
Exercise: restore the order invariant

Must preserve:
1. Shape invariant
2. Order invariant

Goal: minimize changes
Checkpoint: Bubble down

For a max-heap, when “bubbling down”, which child should you swap with?

A. Left
B. Right
C. Whichever is larger
D. Whichever is smaller
E. Doesn’t matter
For a max-heap, when "bubbling down", which child should you swap with?

- Left: 0%
- Right: 0%
- Whichever is larger: 0%
- Whichever is smaller: 0%
- Doesn’t matter: 0%
1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

Heap: `poll()` (aka `remove()`)

Time is $O(\log n)$
Heap implementation

Max heap
Tree implementation

```java
public class HeapNode<E> {
    private E value;
    private HeapNode left;
    private HeapNode right;
    ...
}
```

But since tree is complete, even more space-efficient implementation is possible…
public class Heap<E> {
    /** represents a complete binary tree in `heap[0..size)` */
    private E[] heap;
    private int size;
    ...
}
Numbering tree nodes

Number node starting at root row by row, left to right

Same order as level-order traversal

Children of node $k$ are nodes $2k+1$ and $2k+2$
Parent of node $k$ is node $(k-1)/2$
Tree nodes as array elements

size = 12
Represent tree with array

- Store node number $i$ in index $i$ of array $b$
- **Children** of $b[k]$ are $b[2k + 1]$ and $b[2k + 2]$
- **Parent** of $b[k]$ is $b[(k-1)/2]$
Exercise: map tree to array

What is the array representation of this tree?

A. \{1, 2, 3, 4, 5, 9\}
B. \{3, 1, 2, 4, 5, 9\}
C. \{9, 4, 3, 1, 5, 2\}
D. \{9, 4, 5, 3, 1, 2\}
### Heap Representation

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>1, 2, 3, 4, 5, 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>B:</td>
<td>3, 1, 2, 4, 5, 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>C:</td>
<td>9, 4, 3, 1, 5, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>D:</td>
<td>9, 4, 5, 3, 1, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
</tr>
</tbody>
</table>
Demo code
Here’s a heap, stored in an array:

\[
9 \ 5 \ 2 \ 1 \ 2 \ 2
\]

What is the state of the array after execution of add(6)? Assume the existing array is large enough to store the additional element.

A. \([9 \ 5 \ 2 \ 1 \ 2 \ 2 \ 6]\)  
B. \([9 \ 5 \ 6 \ 1 \ 2 \ 2 \ 2]\)  
C. \([9 \ 6 \ 5 \ 1 \ 2 \ 2 \ 2]\)  
D. \([9 \ 6 \ 5 \ 2 \ 1 \ 2 \ 2]\)
Here's a heap, stored in an array:

\[
\begin{bmatrix}
9 & 5 & 2 & 1 & 2 & 2
\end{bmatrix}
\]

\[\Rightarrow\]

\[
\begin{bmatrix}
9 & 5 & 6 & 1 & 2 & 2 & 2
\end{bmatrix}
\]

Write the array after execution of add(6)
Specifying priorities

• Use element ordering: Comparable or Comparator
  • Example: Assignments ordered by their due date
  • Used by java.util.PriorityQueue<E> (min-heap)

• Separate priority values: heap stores (element, priority) pairs
Removing values, changing priorities

• $O(N)$ to search whole tree for value

• Optimization: maintain an “index”: for each value, remember which array index it is stored at
  • Map<V, Integer>
  • If Map’s get(), put() are $O(1)$, will not change heap’s complexity analysis

• With index, remove(), update() are $O(\log N)$
  • Remove: replace with rightmost leaf, bubble down (like poll())
  • Update: bubble up or down if new priority is bigger/smaller
Heapsort
Heapsort

Goal: sort array \textit{in place}

Approach:
1. turn array into a heap
2. poll repeatedly
Heapsort

// Make b[0..n-1] into a max-heap (in place)

Goal: sort array in place
Approach:
1. turn array into a heap
2. poll repeatedly
Heapsort

// Make b[0..n-1] into a max-heap (in place)
// inv: b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
for (k = size-1; k > 0; k--) {
    b[k] = poll();  // i.e., take max element out of heap.
}

Goal: sort array in place
Approach:
1. turn array into a heap
2. poll repeatedly
Heapsort

// Make b[0..n-1] into a max-heap (in place)
// inv: b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
for (k = size-1; k > 0; k--) {
    b[k] = poll();  // i.e., take max element out of heap.
}

Goal: sort array in place
Approach:
1. turn array into a heap
2. poll repeatedly