CSE 250: Dijkstra's Algorithm

Lecture 24

Oct 25, 2023

Reminders

- PA2: Implement **Map Routing**
  1. Create an adjacency list (discussed today)
  2. Find a path from A to B with the fewest intersections
  3. Find a path from A to B with the shortest distance
- PA2 implementation due Sun, Nov 5 at 11:59 PM

New ADT: Priority Queue

`PriorityQueue<E>` (`E` must be `Comparable`)

- `public void add(E e)`: Add `e` to the queue.
- `public E peek()`: Return the *least* element added.
- `public E remove()`: Remove and return the *least* element added.

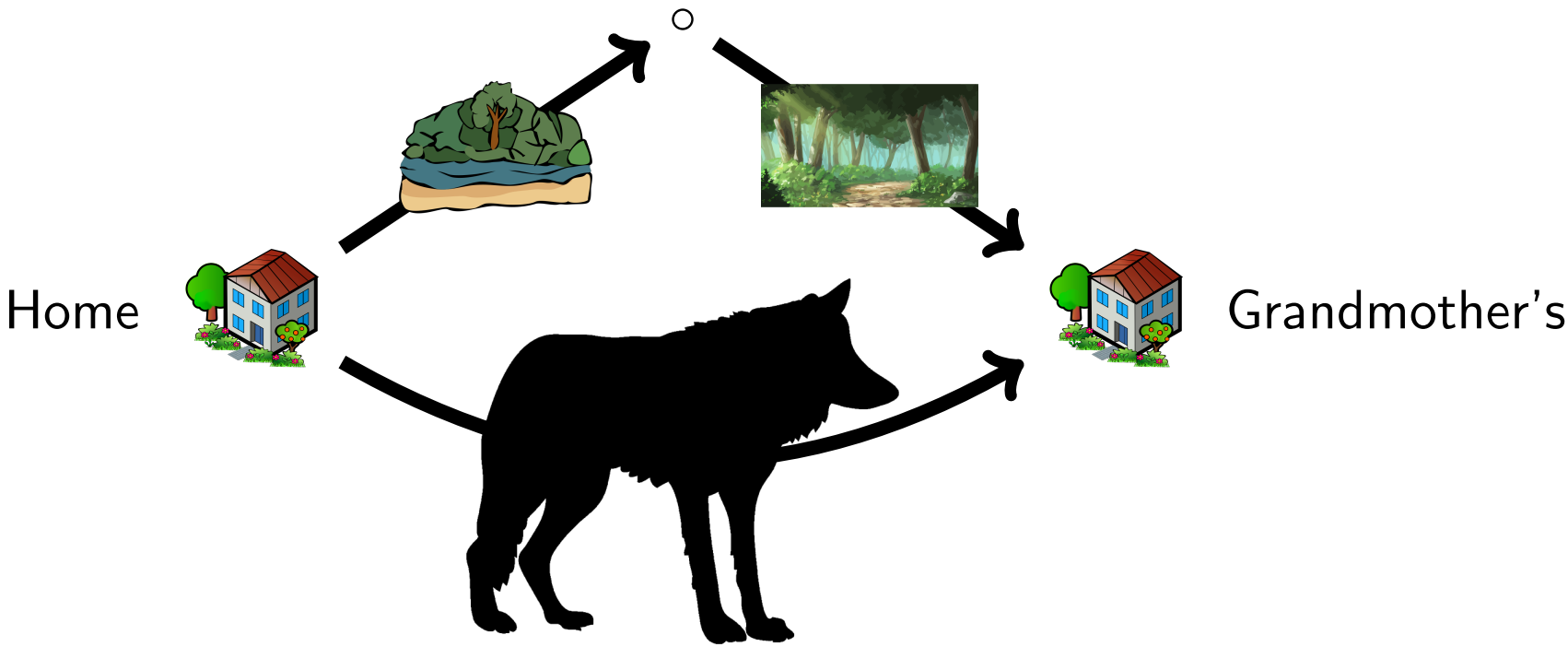
The Heap Data Structure

- `public void pushHeap(T element)` $O(\log(N))$ (Amortized)
  1. Put the item in the 'first empty' slot.
  2. `fixUp` to move the item to the correct place.
- `public T popHeap()`
  1. Remove the item at the root.
  2. Move the item in the 'last' slot to the root
  3. `fixDown` to move the item to the right place.
- `public T peekHeap()`
  1. Look at the item at the root.

Priority Queues

Operation	Lazy	Proactive	Heap
`add`	$O(1)$	$O(N)$	$O(\log(N))$
`remove`	$O(N)$	$O(1)$	$O(\log(N))$
`peek`	$O(N)$	$O(1)$	$O(1)$

Shortest Path



openclipart.org

Shortest Path

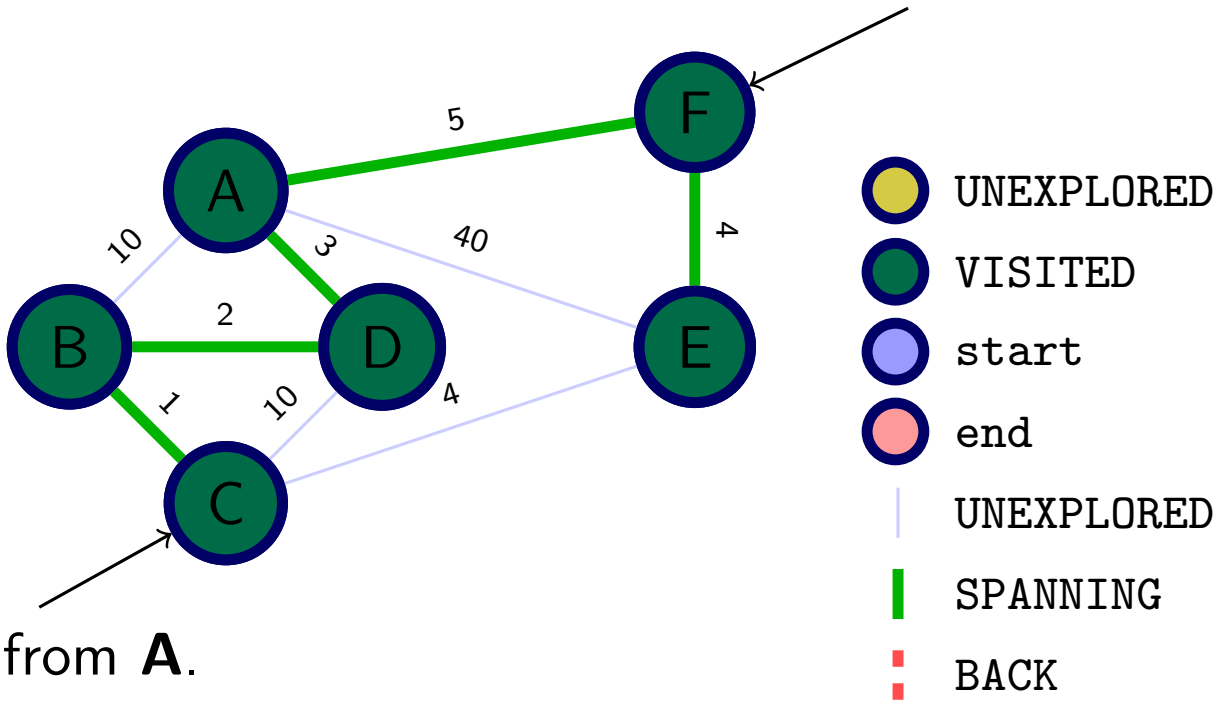
Problem: BFS always finds the path with the *Fewest Edges*

Graph Search - Queue

```
1  public void BFS(Graph<V, E> graph, start: Vertex) {
2    Queue<Pair<Vertex, Integer>> todos = new Queue<>();
3    todos.add( new Pair(start, 0) );
4    start.setLabel(VertexLabel.VISITED)
5    while( !todos.isEmpty() ){
6      Pair curr = todos.remove(); Vertex v = curr.first;
7                                  int d = curr.second;
8      for(edge : v.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(curr);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           todos.add( new Pair(opposite, d + 1) );
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

Finding The Shortest Edge

F is 5 away, it is closer.



C is 6 away from **A**.

Graph Search - Queue (with distance)

```
1  public void BFSIsh(Graph<V, E> graph, start: Vertex) {
2    Queue<Pair<Vertex, Integer>> todos = new Queue<>();
3    todos.add( new Pair(start, 0) );
4    start.setLabel(VertexLabel.VISITED)
5    while( !todos.isEmpty() ){  We want to dequeue in distance order... but how?
6      Pair curr = todos.remove(); Vertex v = curr.first;
7                                  int d = curr.second;
8      for(edge : v.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(curr);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           todos.add( new Pair(opposite, d + 1edge.distance) );
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

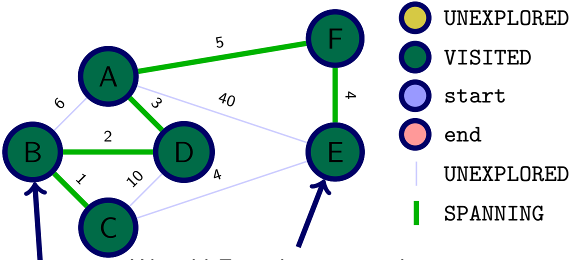
Graph Search - Priority Queue

```
1  public void BFSIsh(Graph<V, E> graph, Vertex start) {
2    Queue<Pair<Vertex, Integer>> todos = new PriorityQueue<>();
3    todos.add( new Pair(start, 0) );
4    start.setLabel(VertexLabel.VISITED)
5    while( !todos.isEmpty() ){
6      Pair curr = todos.remove(); Vertex v = curr.first;
7                                  int d = curr.second;
8      for(edge : v.getIncident()){
9        if(edge.getLabel() == EdgeLabel.UNEXPLORED){
10         Vertex opposite = edge.getOpposite(curr);
11         if(opposite.getLabel() == VertexLabel.UNEXPLORED){
12           todos.add( new Pair(opposite, d + edge.distance) );
13           opposite.setLabel(VertexLabel.VISITED);
14           edge.setLabel(EdgeLabel.SPANNING);
15         } else {
16           edge.setLabel(EdgeLabel.BACK);
17 } } } } }
```

Finding The Shortest Edge

Priority Queue

- ~~**A** via ∅, 0~~
- ~~**D** via **A**, 3~~
- ~~**B** via **D**, 5~~
- ~~**F** via **A**, 5~~
- ~~**B** via **A**, 6~~
- ~~**C** via **B**, 6~~
- ~~**E** via **F**, 9~~
- **C** via **D**, 13
- **E** via **C**, 17
- **E** via **A**, 40



We add **E** to the pqueue, but
this isn't the shortest path

Graph Search - Dijkstra's Algorithm

```
public void Dijkstra(Graph<V, E> graph, Vertex start) {
  PriorityQueue<Pair<Vertex, Integer>> todos = new PriorityQueue<>();

  todos.add( new Pair(start, 0) );                          ← Init priority queue at (start, dist=0)
  start.setLabel(VertexLabel.VISITED);

  while( !todos.isEmpty() ){                                ← As long as there is still work

    Pair curr = todos.remove();                             ← Dequeue the next vertex
    Vertex v = curr.first;                                  ← This will be the closest vertex
    int d = curr.second;                                    ← since we are using a PQ

    if(v.getLabel() == VertexLabel.UNEXPLORED){
      v.setLabel(VertexLabel.VISITED);                      ← Ignore already visited vertices

      for(edge : v.getIncident()){
        Vertex opposite = edge.getOpposite(curr);

        if(opposite.getLabel() == VertexLabel.UNEXPLORED){ ← Enqueue every unvisited adjacent vertex
          todos.add( new Pair(opposite, d + edge.distance) );
        }

      }
    }

  }
}
```

Graph Search - Dijkstra's Algorithm Complexity

```
public void Dijkstra(Graph<V, E> graph, Vertex start) {
  PriorityQueue<Pair<Vertex, Integer>> todos = new PriorityQueue<>();

  todos.add( new Pair(start, 0) );
  start.setLabel(VertexLabel.VISITED);

  while( !todos.isEmpty() ){

    Pair curr = todos.remove();                                  ← Dequeue is O(log(M))
    Vertex v = curr.first;
    int d = curr.second;

    if(v.getLabel() == VertexLabel.UNEXPLORED){                  ← Each vertex is visited once
      v.setLabel(VertexLabel.VISITED);

      for(edge : v.getIncident()){                               ← O(deg(v) work per visit
        Vertex opposite = edge.getOpposite(curr);

        if(opposite.getLabel() == VertexLabel.UNEXPLORED){       ← Each vertex is enqueued 1×/edge
          todos.add( new Pair(opposite, d + edge.distance) );    ← Enqueue is O(log(M))
        }

      }
    }

  }
}
```

Graph Search - Dijkstra's Algorithm Complexity

- Once per item enqueued (at most $O(M)$ times)
  - Enqueue: $O(\log(M))$
  - Dequeue: $O(\log(M))$

  Total: $O(M \cdot \log(M))$
- Once per vertex v
  - Visit $O(\deg(v))$
  - Set `VISITED` $O(1)$

  Total: $O(\sum_v \deg(v) + 1) = O(\sum_v \deg(v) + \sum_v 1)$
  $= O(M + N)$

Total: $O(M \cdot \log(M) + M + N) = O(M \cdot \log(M) + N)$
$\approx O(M \cdot \log(M))$ (if $M > N$)

Dijkstra's Algorithm

Tweaks to the Algorithm

- What if enqueueing the vertex we just saw, we just update the value in the heap?

- How do we track the actual path?
  - Store the 'via' in a hash map
  - Store a path in the 'via' slot