A* Search and Design Decisions
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston    Ethan Knutson    Nathan Lipiarski
Amanda Park       Farrell Fileas    Sam Long
Anish Velagapudi  Howard Xiao      Yifan Bai
Brian Chan        Jade Watkins     Yuma Tou
Elena Spasova     Lea Quan
Announcements

❖ Midterm is this Friday
  ▪ If your student number ends in an odd number, go to KNE 210
  ▪ If your student ends in an even number, go to KNE 220
  ▪ Workshops and review session will be focused on your midterm questions – bring your questions and practice midterms!
  ▪ Review session Thursday night: 4:30-6:30 @ ARC 147

❖ HW6 is released
  ▪ Yes, HW5 and HW6 are both currently released
  ▪ Please prefix your Piazza posts with “HW5: ...” or “HW6: ...”

❖ 20sp instructors want current students to TA next quarter!
  ▪ Check Piazza or course webpage for more details
Feedback from Reading Quiz

❖ If we add diagonals, is it still the Manhattan distance? What is the Euclidean distance?

❖ I still need a walkthrough of Dijkstra’s

❖ Does Dijkstra’s still work if the grid had different weights?
Lecture Outline

❖ Dijkstra’s Algorithm, Reviewed

❖ A* Search
  ▪ Introducing A*
  ▪ A* Heuristics

❖ Design Decisions
Dijkstra’s Algorithm

- Demo: https://docs.google.com/presentation/d/1_bw2z1ggUkquPdh77gwdVBtA0JmaZdpkV6MoAgxlJc/pub?start=false&loop=false&delayms=300

```java
dijkstras(Node s, Graph g) {
    PriorityQueue unvisited;
    unvisited.addAll(g.allNodes(), ∞)
    unvisited.changePriority(s, 0);
    Map<Node, Integer> distances;
    Map<Node, Node> previousNode

    while (!unvisited.isEmpty()) {
        Node n = unvisited.removeMin();
        for (Node i : n.neighbors) {
            if (distances[i] < distances[n] + g.edgeWeight(n, i)) {
                continue;
            } else {
                distances[i] = distances[n] + g.edgeWeight(n, i);
                unvisited.changePriority(i, distances[i]);
                previousNode[i] = n;
            }
        }
    }
}
```

Diagram of the graph with nodes labeled and edges weighted.
Which of the following statements are true?

- Dijkstra’s Algorithm becomes Breadth-first Search if all the edges have the same weight
- Dijkstra’s can find the shortest path from the source to every node in the graph
- At each step of the algorithm, Dijkstra’s only considers the path length from the source

A. True / True / True
B. True / True / False
C. False / True / True
D. False / True / False
E. False / False / False
F. I’m not sure ...
Dijkstra’s Algorithm’s Flaws

- Demo: [https://qiao.github.io/PathFinding.js/visual/](https://qiao.github.io/PathFinding.js/visual/)

- If we want a *single shortest path* (instead of *all shortest paths*), Dijkstra’s and BFS does unnecessary work
  - The answer is still correct, but we did unnecessary computation
Lecture Outline

❖ Dijkstra’s Algorithm, Reviewed

❖ A* Search
  ▪ Introducing A*
  ▪ A* Heuristics

❖ Design Decisions
Single-Pair Shortest Path Problem
Single-Pair Shortest Path: Dijkstra’s Algorithm
How should we hint to Dijkstra’s that we want it to concentrate its search southward?

BFS -> Dijkstra’s switched the Queue for a Priority Queue
  - Can we change our idea of a “priority”?
Introducing A* Search

❖ Idea:

▪ Visit vertices in order of $d(\text{Ravenna Park}, v) + h(v, \text{Japanese Garden})$, where $h(v, \text{Japanese Garden})$ is an estimate of the distance from $v$ to our goal

▪ In other words, prefer a location $v$ if:
  • We already know the fastest way to reach $v$
  • AND we suspect that $v$ might be the fastest way to get to our goal

❖ Dijkstra’s only considers $d(\text{Ravenna Park}, v)$

❖ Demo: [http://qiao.github.io/PathFinding.js/visual/](http://qiao.github.io/PathFinding.js/visual/)
A* Demo

- Source = 0; Destination = 6
- Use the following estimates:

<table>
<thead>
<tr>
<th>Vertex ID</th>
<th>h(v, dest)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Demo:
https://docs.google.com/presentation/d/177bRUTdCa60fjExdr9eO04NHm0MRfPtCzvEup1iMccM/edit
Dijkstra’s Algorithm vs A* Search

**dijkstras** (Node s, Graph g) {
  PriorityQueue unvisited;
  unvisited.addAll(g.allNodes(), ∞)
  unvisited.changePriority(s, 0);

  Map<Node, Integer> distances;
  Map<Node, Node> previousNode

  while (!unvisited.isEmpty()) {
    Node n = unvisited.removeMin();
    for (Node i : n.neighbors) {
      if (distances[i] < distances[n] + g.edgeWeight(n, i)) {
        continue;
      } else {
        distances[i] = distances[n] + g.edgeWeight(n, i);
        unvisited.changePriority(i, distances[i]);
        previousNode[i] = n;
      }
    }
  }
}

**astar** (Node s, Node t, Graph g) {
  PriorityQueue unvisited;
  unvisited.addAll(g.allNodes(), ∞)
  unvisited.changePriority(s, 0);

  Map<Node, Integer> distances;
  Map<Node, Node> previousNode

  while (!unvisited.isEmpty()) {
    Node n = unvisited.removeMin();
    for (Node i : n.neighbors) {
      if (distances[i] < distances[n] + g.edgeWeight(n, i)) {
        continue;
      } else {
        distances[i] = distances[n] + g.edgeWeight(n, i);
        unvisited.changePriority(i, distances[i] + h(i, t));
        previousNode[i] = n;
      }
    }
  }
}
Lecture Outline

❖ Dijkstra’s Algorithm, Reviewed

❖ A* Search
  ▪ Introducing A*
  ▪ A* Heuristics

❖ Design Decisions
Heuristics

- We call this “estimate function” a **heuristic**
  - **Definition:** *a solution or choice or judgement that is “good enough” for a purpose, but which could be optimized*
  - In other words: *it doesn’t have to be perfect*

- What is a good heuristic for this map?
Euclidean and Manhattan Distances

- Assume the entire map can be represented as a grid

- Manhattan distance: $\Delta x + \Delta y$

- Euclidean distance: $\sqrt{\Delta x^2 + \Delta y^2}$
Will A* Search return the correct shortest path if $h(v, \text{dest}) = 10$ for every $v$ in the graph?

A. Always
B. Sometimes
C. Never
D. Not enough information
E. I’m not sure …
But What If We Have a Lousy Heuristic?

- $h(v, \text{dest}) = 0$
  - That’s just Dijkstra’s

- $h(v, \text{dest}) = 1,000,000$
  - Still just Dijkstra’s

- $h(\text{Montlake Bridge, dest}) = 1,000,000$
  - Inconsistent results!
Good Heuristics are Hard!

❖ You’ll frequently hear that “A* Search is hard”
  ▪ As we’ve seen, A* Search is an incremental update to Dijkstra’s
  ▪ What’s hard with A* Search is designing a good heuristic

❖ In this class, we’ll give you a (good) heuristic for HuskyMaps
  ▪ Hint: Manhattan and Euclidean distances are both good heuristics

❖ If you take an AI class, you’ll learn all about designing heuristics
  ▪ Sneak preview: good heuristics have the following characteristics:
    • \( h(v, \text{dest}) \leq \text{true distance from } v \text{ to destination} \) ("admissible")
    • \( h(v, \text{dest}) \leq \text{dist}(v, w) + h(w, \text{dest}) \) ("consistent")
Lecture Outline

❖ Dijkstra’s Algorithm, Reviewed

❖ A* Search
  ▪ Introducing A*
  ▪ A* Heuristics

❖ Design Decisions
Two Key Skills

❖ In Software Engineering, two important skills to have are:
  ▪ Identifying the requirements (ie, selecting an ADT)
  ▪ Making tradeoffs (ie, selecting the data structure for that ADT)

❖ So let’s review the ADTs’ functionality and the performance characteristics of each data structure
List Functionality

**List ADT.** A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index.
- A list has a size defined as the number of elements in the list.
- Elements can be added to the front, back, or *any index in the list*.
- Optionally, elements can be removed from the front, back, or *any index in the list*.

Possible Implementations:
- ArrayList
- LinkedList
### List Performance Tradeoffs

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFront</td>
<td>linear</td>
<td>constant</td>
</tr>
<tr>
<td>removeFront</td>
<td>linear</td>
<td>constant</td>
</tr>
<tr>
<td>addBack</td>
<td>constant*</td>
<td>linear</td>
</tr>
<tr>
<td>removeBack</td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td>get(idx)</td>
<td>const</td>
<td>linear</td>
</tr>
<tr>
<td>put(idx)</td>
<td>linear</td>
<td>linear</td>
</tr>
</tbody>
</table>

* constant for most invocations
Stack and Queue Functionality

**Stack ADT.** A collection storing an ordered sequence of elements.
- A stack has a size defined as the number of elements in the stack.
- Elements can only be added and removed from the top (“LIFO”)

- Possible Implementations:
  - ArrayStack, LinkedStack

**Queue ADT.** A collection storing an ordered sequence of elements.
- A queue has a size defined as the number of elements in the queue.
- Elements can only be added to one end and removed from the other (“FIFO”)

- Possible Implementations:
  - ArrayQueue, LinkedQueue
# Stack and Queue Performance Tradeoffs

- **Stack (LIFO):**

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayStack</th>
<th>LinkedStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>pop</td>
<td>constant</td>
<td>constant</td>
</tr>
</tbody>
</table>

* constant for most invocations

- **Queue (FIFO):**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array Queue (v2)</th>
<th>LinkedQueue (v2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>dequeue</td>
<td>constant</td>
<td>constant</td>
</tr>
</tbody>
</table>

* constant for most invocations
Deque Functionality

Deque ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index.
- A deque has a size defined as the number of elements in the deque.
- Elements can be added to the front or back.
- Optionally, elements can be removed from the front or back.

Possible Implementations:
- ArrayDeque, LinkedDeque
### Deque Performance Tradeoffs

<table>
<thead>
<tr>
<th>Operation</th>
<th>CircularArrayDeque</th>
<th>LinkedDeque</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>removeFirst</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>addLast</td>
<td>constant*</td>
<td>constant</td>
</tr>
<tr>
<td>removeLast</td>
<td>constant</td>
<td>constant</td>
</tr>
</tbody>
</table>

* constant for most invocations
Set and Map Functionality

**Set ADT.** A collection of values.
- A set has a size defined as the number of elements in the set.
- You can add and remove values.
- Each value is accessible via a “get” or “contains” operation.

**Map ADT.** A collection of keys, each associated with a value.
- A map has a size defined as the number of elements in the map.
- You can add and remove (key, value) pairs.
- Each value is accessible by its key via a “get” or “contains” operation.

- Possible Implementations:
  - Unbalanced BST
  - LLRB Tree
  - B-Tree (eg, 2-3 Tree)
  - Hash Tables
## Set and Map Performance Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>Find</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resizing Separate</td>
<td>$\Theta(Q) \in \Theta(N)$</td>
<td>$\Theta(Q) \in \Theta(N)$</td>
<td>$\Theta(Q) \in \Theta(N)$</td>
</tr>
<tr>
<td>Chaining Hash Table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(worst case)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resizing Separate</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
</tr>
<tr>
<td>Chaining Hash Table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(best/average cases)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLRB Tree</td>
<td>$\Theta(h) \in \Theta(\log N)$</td>
<td>$\Theta(h) \in \Theta(\log N)$</td>
<td>$\Theta(h) \in \Theta(\log N)$</td>
</tr>
<tr>
<td>B-Tree</td>
<td>$\Theta(h) \in \Theta(\log N)$</td>
<td>$\Theta(h) \in \Theta(\log N)$</td>
<td>$\Theta(h) \in \Theta(\log N)$</td>
</tr>
<tr>
<td>BST</td>
<td>$\Theta(h) \in \Theta(N)$</td>
<td>$\Theta(h) \in \Theta(N)$</td>
<td>$\Theta(h) \in \Theta(N)$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
</tr>
</tbody>
</table>
Priority Queue Functionality

Priority Queue ADT. A collection of values.
• A PQ has a size defined as the number of elements in the set.
• You can add values.
• You cannot access or remove arbitrary values, only the max value.

❖ Possible Implementations:
  ▪ Balanced BST with “max” pointer
  ▪ Binary Heap
  ▪ (and a ton of others we didn’t discuss)
## Priority Queue Performance Tradeoffs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Balanced BST (worst case)</th>
<th>Binary Heap (worst case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>O(\log N)</td>
<td>O(\log N)**</td>
</tr>
<tr>
<td>max</td>
<td>O(1)*</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeMax</td>
<td>O(\log N)</td>
<td>O(\log N)</td>
</tr>
</tbody>
</table>

* If we keep a pointer to the largest element in the BST
** Average case is constant
Graph Functionality

**Graph ADT.** A collection of vertices and the edges connecting them.

- We can query for vertices connected to, or edges leaving from, a vertex v
- Edges are specified as pairs of vertices
- We can add/remove edges from the graph

- Possible Implementations:
  - Adjacency Matrix
  - Edge Set
  - Adjacency List
## Graph Performance Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th><code>getAllEdgesFrom(v)</code></th>
<th><code>hasEdge(v, w)</code></th>
<th><code>getAllEdges()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adjacency Matrix</strong></td>
<td>$\Theta(V)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(V^2)$</td>
</tr>
<tr>
<td><strong>Edge Set</strong></td>
<td>$\Theta(E)$</td>
<td>$\Theta(E)$</td>
<td>$\Theta(E)$</td>
</tr>
<tr>
<td><strong>Adjacency List</strong></td>
<td>$O(V)$</td>
<td>$\Theta(\text{degree}(v))$</td>
<td>$\Theta(E + V)$</td>
</tr>
</tbody>
</table>
tl;dr

❖ Dijkstra’s is great for \textit{all-pairs shortest path}

❖ A* is great for \textit{single-pair shortest path}
  - But you need to be careful about picking a good heuristic