Today's Topics - More Graphs!

- Reviewing DFS and BFS
- Comparing DFS and BFS
- Making weighty decisions using Dijkstra's algorithm
- Looking into the future with A*
- Google Maps
REVIEWING DFS AND BFS
DEPTH FIRST SEARCH

- Find a path from A to B using _iterative_ depth first search

  (Assume that nodes are pushed onto the stack in _alphabetic order_)
DEPTH FIRST SEARCH (ITERATIVE PSEUDOCODE)

- create a path with just start node and push onto stack \( s \)
- while \( s \) is not empty
  - \( p = s.pop() \)
  - \( v = \text{last node of } p \)
  - if \( v \) is end, you're done
  - mark \( v \) as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - push new path onto \( s \)
Find a path from A to B using iterative depth first search

(Assume that nodes are pushed onto the stack in alphabetic order)

A ➔ E ➔ F ➔ D ➔ B
Find a path from A to B using *iterative* depth first search

(Assume that nodes are pushed onto the stack in *alphabetic order*)

A ➔ E ➔ F ➔ D ➔ B

Is this the shortest path?
DEPTH FIRST SEARCH

Paths to Consider (Stack)

A
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path

THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path
DEPTH FIRST SEARCH

Paths to Consider (Stack)

Current Path
Find a path from A to B using breadth first search

(Assume that nodes are pushed onto the queue in alphabetic order)
create a path with just start node and enqueue into queue q

while q is not empty
  p = q.dequeue()
  v = last node of p
  if v is end, you're done
  mark v as visited
  for each unvisited neighbor:
    create new path and append neighbor
    enqueue new path into q
BREADTH FIRST SEARCH

- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)
- A ➔ C ➔ B
The Life Changing Magic of Dijkstra and A*

Breadth First Search

- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in alphabetic order)
  - A ➔ C ➔ B
  - Is this the shortest path?
BREADTH FIRST SEARCH

- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)
  - A → C → B
- Is *this* the shortest path?
  - Yes
BREADTH FIRST SEARCH

Paths to Consider (Queue)

Current Path
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

BREADTH FIRST SEARCH

Paths to Consider (Queue)

Current Path
BREADTH FIRST SEARCH

Paths to Consider (Queue)

Current Path
BREADTH FIRST SEARCH

Paths to Consider (Queue)

Current Path
YOU NEVER CONSIDER A PATH OF LENGTH K + 1 UNTIL YOU'VE CONSIDERED ALL PATHS OF LENGTH K OR SHORTER
COMPARING DFS AND BFS
COMPARING DFS AND BFS

**DFS**
- create a path with just start node and push onto stack \( s \)
- while \( s \) is not empty:
  - \( p = s.pop() \)
  - \( v = \) last node of \( p \)
  - if \( v \) is end node, you're done
  - mark \( v \) as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - push new path onto \( s \)

**BFS**
- create a path with just start node and enqueue into queue \( q \)
- while \( q \) is not empty:
  - \( p = q.dequeue() \)
  - \( v = \) last node of \( p \)
  - if \( v \) is end node, you're done
  - mark \( v \) as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into \( q \)
COMPARING DFS AND BFS

**DFS**
- create a path with just start node and push onto stack s
- while s is not empty:
  - p = s.pop()
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - push new path onto s

**BFS**
- create a path with just start node and enqueue into queue q
- while q is not empty:
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

THE GRAPH SEARCH TO-DO LIST
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

THE GRAPH SEARCH TO-DO LIST
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

THE GRAPH SEARCH TO-DO LIST
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

THE GRAPH SEARCH TO-DO LIST
WEIGHTY DECISIONS
DEALING WITH WEIGHTY TOPICS
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

DEALING WITH WEIGHTY TOPICS
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

DEALING WITH WEIGHTY TOPICS
DEALING WITH WEIGHTY TOPICS
DEALING WITH WEIGHTY TOPICS
DEALING WITH WEIGHTY TOPICS
DEALING WITH WEIGHTY TOPICS
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

DEALING WITH WEIGHTY TOPICS
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

DEALING WITH WEIGHTY TOPICS

Diagram of a weighted graph with vertices A, B, C, D, E, F and edges connecting them with weights 1, 4, 7.
In Dijkstra's algorithm, the todo list is a priority queue.
DIJKSTRA'S ALGORITHM (PSEUDOCODE)

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q
DIJKSTRA'S ALGORITHM (PSEUDOCODE)

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength
**DIJKSTRA'S ODDS AND ENDS**

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength
- What do you initialize the weight of the path to?
DIJKSTRA'S ODDS AND ENDS

- Create a path with just start node and enqueue into priority queue q

while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

- What do you initialize the weight of the path to?
  - Zero should be fine
DIJKSTRA'S ODDS AND ENDS

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength
- Can't I just return the path as soon as I find the end node? Why wait until I dequeue?
DIJKSTRA'S ODDS AND ENDS

- Create a path with just start node and enqueue into priority queue q
- While q is not empty
  - p = q.dequeue()
  - v = last node of p
  - If v is end node, you're done
  - If you've seen v before, skip it
  - Mark v as visited
  - For each unvisited neighbor:
    - Create new path and append neighbor
    - Enqueue new path into q with priority pathLength
- Can't I just return the path as soon as I find the end node? Why wait until I dequeue?
  - This is one of the most common mistakes people make with Dijkstra's!
  - It's possible a path with a lower priority gets enqueued in the meantime.
DIJKSTRA'S ODDS AND ENDS

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

- Why would you skip the node just because you've seen it before?
DIJKSTRA'S ODDS AND ENDS

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
- mark v as visited
- for each unvisited neighbor:
  - create new path and append neighbor
  - enqueue new path into q with priority pathLength

- Why would you skip the node just because you've seen it before?
  - If you've seen the node before, that means you've already found a shorter path to it.
  - Any path that follows from this one already has a shorter equivalent
  - The first path you find to v will be the shortest path to v
NEGATIVE EDGES

THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

- Edge from A to B: 1
- Edge from B to C: 1
- Edge from C to D: 5
- Edge from D to A: -10
- Edge from C to Goal: 1
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

NEGATIVE CYCLES

A

B

C

D

E

GOAL

2

1

1

1

1

-10

-1

-1

-1
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
<td><strong>2</strong></td>
</tr>
</tbody>
</table>
DIJKSTRA'S MEASURES THE DISTANCE FROM THE START NODE TO THE CURRENT NODE.

WE WANT THE DISTANCE FROM THE CURRENT NODE TO THE DESTINATION.
SEEING THE FUTURE
FORMAL DEFINITIONS

distance(s, u) futureCost(u, t)
FORMAL DEFINITIONS

DIJKSTRA’S

$$\text{priority}(u) = \text{distance}(s, u)$$
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

FORMAL DEFINITIONS

DIJKSTRA’S

\[ \text{priority}(u) = \text{distance}(s, u) \]

IDEAL

\[ \text{priority}(u) = \text{distance}(s, u) + \text{futureCost}(u, t) \]
function futureCost(u, t)
    return abs(u.row - t.row) + abs(u.col - t.col)
<table>
<thead>
<tr>
<th>1 + 5</th>
<th>2 + 4</th>
<th>3 + 3</th>
<th>4 + 2</th>
<th>5 + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + 5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1 + 5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
MAKING GOOD LIFE DECISIONS
**THE LIFE CHANGING MAGIC OF DIJKSTRA AND A**

**FORMAL DEFINITIONS**

.priority(u) = distance(s, u) + futureCost(u, t)

**IDEAL**
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

FORMAL DEFINITIONS

**IDEAL**

\[
priority(u) = \text{distance}(s, u) + \text{futureCost}(u, t)
\]

**A***

\[
priority(u) = \text{distance}(s, u) + \text{heuristic}(u, t)
\]

![Diagram](image-url)

- distance\((s, u)\)
- heuristic\((u, t) \leq \text{futureCost}(u, t)\)
A heuristic is a function that **underestimates** the cost of traveling from $u$ to $t$. It's a "relaxation" heuristic.
columns apart

rows apart
<table>
<thead>
<tr>
<th></th>
<th>2 + 7</th>
<th>1 + 6</th>
<th>2 + 5</th>
<th>3 + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 7</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

 Stars: 2
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2+7</td>
<td>1</td>
<td>2</td>
<td>3+4</td>
</tr>
<tr>
<td>2+7</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2+7</td>
<td>1</td>
<td>2</td>
<td>3+4</td>
</tr>
<tr>
<td>2+7</td>
<td>3+6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The numbers represent sums of the values in the corresponding squares.
<table>
<thead>
<tr>
<th></th>
<th>2 + 7</th>
<th>3 + 6</th>
<th>4 + 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 8</td>
<td>2 + 7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3 + 8</td>
<td>2 + 7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3 + 8</td>
<td>2 + 7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2 + 7</td>
<td>3 + 6</td>
<td>4 + 5</td>
<td></td>
</tr>
</tbody>
</table>

The stars indicate the correct placement of the numbers.
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3+8</td>
<td>3+8</td>
<td>3+8</td>
<td>3+8</td>
<td>3+8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5+4</td>
<td>5+4</td>
<td>5+4</td>
<td>5+4</td>
<td>5+4</td>
</tr>
</tbody>
</table>
The image shows a grid of numbers and stars. The grid contains a pattern of addition problems, with the numbers 3, 4, 5, and 6, and the stars indicate a specific pattern or rule that is to be followed. The stars are placed in a way that suggests a particular alignment or sequence in the numbers.
<table>
<thead>
<tr>
<th>3 + 8</th>
<th>4 + 7</th>
<th>5 + 6</th>
<th>6 + 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 8</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3 + 8</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3 + 8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3 + 8</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3 + 8</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3 + 8</td>
<td>4 + 7</td>
<td>5 + 6</td>
<td>6 + 5</td>
</tr>
</tbody>
</table>
A* (PSEUDOCODE)

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + heuristic
Comparing Dijkstra and A*

**Dijkstra**
- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

**A**
- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + heuristic
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

COMPARING DIJKSTRA AND A*

DIJKSTRA

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

A*

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + heuristic
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

COMPARING DIJKSTRA AND A*

DIJKSTRA

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

A*

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + 0
YOU WANT YOUR HEURISTIC TO BE AS LARGE AS POSSIBLE

BUT YOU NEVER WANT IT TO BE LARGER THAN THE ACTUAL COST.
GOOGLE MAPS
WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million

- How many sets of directions would they need to generate?
  - $N^2$

- How long would that take?
  - $6 \times 10^{15}$ seconds
  - Or... 190 million years
WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million

- How many sets of directions would they need to generate?
  - $N^2$

- How long would that take?
  - $6 \times 10^{15}$ seconds

  - Or... 190 million years
WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million

- How many sets of directions would they need to generate?
  - (roughly) $N^2$
Why Doesn't Google Maps Precompute Directions?

- How many nodes are in the Google Maps graph?
  - About 75 million

- How many sets of directions would they need to generate?
  - (roughly) $N^2$

- How long would that take?
  - 6 x $10^{15}$ seconds
  - Or... 190 million years
THE LIFE CHANGING MAGIC OF DIJKSTRA AND A*

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million

- How many sets of directions would they need to generate?
  - (roughly) $N^2$

- How long would that take?
  - $6 \times 10^{15}$ seconds
WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million

- How many sets of directions would they need to generate?
  - (roughly) $N^2$

- How long would that take?
  - $6 \times 10^{15}$ seconds
  - Or... 190 million years
WHAT HEURISTICS COULD GOOGLE USE?
WHAT HEURISTICS COULD GOOGLE USE?

- As the crow flies
  - Calculate the straight-line distance from A to B, and divide by the speed on the fastest highway
WHAT HEURISTICS COULD GOOGLE USE?

- As the crow flies
  - Calculate the straight-line distance from A to B, and divide by the speed on the fastest highway
- Landmark heuristic
  - Find the distance from A and B to a landmark, calculate the difference (distance < abs(A - B))
WHAT HEURISTICS COULD GOOGLE USE?

- As the crow flies
  - Calculate the straight-line distance from A to B, and divide by the speed on the fastest highway

- Landmark heuristic
  - Find the distance from A and B to a landmark, calculate the difference (distance < abs(A - B))

- All of these and more?
  - You can use multiple heuristics and choose the max