Dijkstra’s Shortest Path Algorithm

Instructors: Sam McCauley and Dan Barowy
May 13, 2022
• Final: Sunday, May 22, 9:30 AM, in Physics 203

• Lab 8 back later today
  • If you think you want to use lab 9 for resubmission let me know. (Can’t use lab 10—mentioned in syllabus)

• “Practice exam” (really just sample exam questions) posted on handouts page. Solutions today or tomorrow

• All practice quizzes also on handouts page!

• Any questions?
Heaps and Priority Queues
Heap vs priority queue

- Priority queue is the interface
Heap vs priority queue

- Priority queue is the interface

- Heap is the specific implementation
Heap vs priority queue

- Priority queue is the interface

- Heap is the specific implementation

- Like Map vs Hashtable. There are other ways to implement a Map; similarly, there are other ways to implement a priority queue
Summary

- In short: heaps are much simpler and have much better constants
Summary

- In short: heaps are much simpler and have much better constants
- Extremely common in practice!
Summary

• In short: heaps are much simpler and have much better constants

• Extremely common in practice!

• HeapSort is one of the most common sorting methods, especially if you want $O(n \log n)$ guaranteed worst-case running time
Summary

• In short: heaps are much simpler and have much better constants

• Extremely common in practice!

• HeapSort is one of the most common sorting methods, especially if you want $O(n \log n)$ guaranteed worst-case running time

• We saw min heaps. Can get a “max heap” by flipping the requirement: the root element must be largest in the heap. Then can get good $\text{removeMax}$ performance
Dijkstra’s Algorithm
Shortest Path

- Goal: given a graph $G = (V, E)$ and a vertex $v$ in $V$, find the shortest path from $v$ to all vertices in $V$.
Shortest Path

- Goal: given a graph $G = (V, E)$ and a vertex $v$ in $V$, find the shortest path from $v$ to all vertices in $V$

- (It turns out: to find the shortest path from $v$ to $u$, we may need to search the entire graph, find the shortest path to all other vertices)
Shortest Path

• Goal: given a graph $G = (V, E)$ and a vertex $v$ in $V$, find the shortest path from $v$ to all vertices in $V$

• (It turns out: to find the shortest path from $v$ to $u$, we may need to search the entire graph, find the shortest path to all other vertices)

• Assume all edges have positive numbers as a label
Shortest Path

• Goal: given a graph $G = (V, E)$ and a vertex $v$ in $V$, find the shortest path from $v$ to all vertices in $V$

• (It turns out: to find the shortest path from $v$ to $u$, we may need to search the entire graph, find the shortest path to all other vertices)

• Assume all edges have positive numbers as a label

• Where to start?
Base Case

- Given \( v \), is there *any* vertex in \( G \) where we can find the shortest path?
Base Case

- Given \( v \), is there any vertex in \( G \) where we can find the shortest path?
- Sure: the shortest path to \( v \) has length 0. (Why?)
Base Case

• Given \( v \), is there any vertex in \( G \) where we can find the shortest path?

• Sure: the shortest path to \( v \) has length 0. (Why?)

• Where do we go from here? Can we find the shortest path to any other vertex?
Short Proof For Intuition

- To Prove: The closest vertex to $v$ is a neighbor of $v$. 

- Consider the shortest path from $v$ to a vertex $u$ that is not a neighbor of $v$: $v, e_1, v_1, e_2, v_2, \ldots, u$. This path has length $\sum_i e_i > e_1$. Then $v_1$ is closer to $v$ than $u$ is.

- So the shortest path to any other vertex is one of the neighbors.

- Idea: adding more edges only makes the path longer.
Short Proof For Intuition

- To Prove: The closest vertex to $v$ is a neighbor of $v$.

- Consider the shortest path from $v$ to a vertex $u$ that is not a neighbor of $v$:
  \[ v, e_1, v_1, e_2, v_2, \ldots, u \]
  This path has length $\sum_i e_i > e_1$. Then $v_1$ is closer to $v$ than $u$ is.
Short Proof For Intuition

• To Prove: The closest vertex to $v$ is a neighbor of $v$.

• Consider the shortest path from $v$ to a vertex $u$ that is not a neighbor of $v$:

$v, e_1, v_1, e_2, v_2, \ldots, u$

This path has length $\sum_i e_i > e_1$. Then $v_1$ is closer to $v$ than $u$ is.

• So the shortest path to any other vertex is one of the neighbors.
Short Proof For Intuition

• To Prove: The closest vertex to $v$ is a neighbor of $v$.

• Consider the shortest path from $v$ to a vertex $u$ that is not a neighbor of $v$:

$$v, e_1, v_1, e_2, v_2, \ldots, u$$

This path has length $\sum_i e_i > e_1$. Then $v_1$ is closer to $v$ than $u$ is.

• So the shortest path to any other vertex is one of the neighbors.

• **Idea**: adding more edges only makes the path longer
Growing the Shortest Paths

- Let’s say we have the shortest path to some collection of vertices in the graph
Growing the Shortest Paths

Let’s say we have the shortest path to some collection of vertices in the graph.

- We’ll mark these vertices as *visited*.
Growing the Shortest Paths

• Let’s say we have the shortest path to some collection of vertices in the graph
  
  • We’ll mark these vertices as *visited*

• Can we find the shortest path to some other vertex?
Growing the Shortest Paths

- Let’s say we have the shortest path to some collection of vertices in the graph
  - We’ll mark these vertices as *visited*

- Can we find the shortest path to some other vertex?
  - *Idea:* must be a neighbor of one of the vertices we’ve already visited
Keeping Track of the State

- What do we need to keep track of?

- All vertices that we have the shortest path for

- And their unexplored incident edges. (Need to keep finding some neighbor of a visited vertex.)

- What do we do when we find the shortest path to a vertex?

  - As we said: mark it as visited

  - Also need to add its incident edges to the list of edges

  - Add them to a priority queue; priority based on the total length of the path: length of the path to this vertex, plus the length of the outgoing edge
Keeping Track of the State

• What do we need to keep track of?
  • All vertices that we have the shortest path for

• What do we do when we find the shortest path to a vertex?
  • As we said: mark it as visited
  • Also need to add its incident edges to the list of edges
    • Add them to a priority queue; priority based on the total length of the path: length of the path to this vertex, plus the length of the outgoing edge
Keeping Track of the State

• What do we need to keep track of?
  • All vertices that we have the shortest path for
  • And their unexplored incident edges. (Need to keep finding some neighbor of a visited vertex.)
Keeping Track of the State

- What do we need to keep track of?
  - All vertices that we have the shortest path for
  - And their unexplored incident edges. (Need to keep finding some neighbor of a visited vertex.)

- What do we do when we find the shortest path to a vertex?
Keeping Track of the State

- What do we need to keep track of?
  - All vertices that we have the shortest path for
  - And their unexplored incident edges. (Need to keep finding some neighbor of a visited vertex.)

- What do we do when we find the shortest path to a vertex?
  - As we said: mark it as visited
Keeping Track of the State

- What do we need to keep track of?
  - All vertices that we have the shortest path for
  - And their unexplored incident edges. (Need to keep finding some neighbor of a visited vertex.)

- What do we do when we find the shortest path to a vertex?
  - As we said: mark it as visited
  - Also need to add its incident edges to the list of edges
Keeping Track of the State

- What do we need to keep track of?
  - All vertices that we have the shortest path for
  - And their unexplored incident edges. (Need to keep finding some neighbor of a visited vertex.)

- What do we do when we find the shortest path to a vertex?
  - As we said: mark it as visited
  - Also need to add its incident edges to the list of edges
  - Add them to a priority queue; priority based on the total length of the path: length of the path to this vertex, plus the length of the outgoing edge
Visualizing Dijkstra’s Algorithm

- Keep track of visited vertices, and their incident edges (that we haven’t explored)
Visualizing Dijkstra’s Algorithm

- Keep track of visited vertices, and their incident edges (that we haven’t explored)

- Start by marking $v$ as visited, and adding its incident edges to the priority queue
Visualizing Dijkstra’s Algorithm

- Keep track of visited vertices, and their incident edges (that we haven’t explored)
- Start by marking $v$ as visited, and adding its incident edges to the priority queue
- Each time step:
Visualizing Dijkstra’s Algorithm

- Keep track of visited vertices, and their incident edges (that we haven’t explored)

- Start by marking $v$ as visited, and adding its incident edges to the priority queue

- Each time step:
  - Remove an edge from the priority queue; if other vertex is unvisited, visit it and add its incident edges to the priority queue
Visualizing Dijkstra’s Algorithm

- Keep track of visited vertices, and their incident edges (that we haven’t explored)

- Start by marking \( v \) as visited, and adding its incident edges to the priority queue

- Each time step:
  - Remove an edge from the priority queue; if other vertex is unvisited, visit it and add its incident edges to the priority queue

- Let’s see an example of how Dijkstra’s runs
Visualizing Dijkstra’s Algorithm

- Keep track of visited vertices, and their incident edges (that we haven’t explored)

- Start by marking $v$ as visited, and adding its incident edges to the priority queue

- Each time step:
  - Remove an edge from the priority queue; if other vertex is unvisited, visit it and add its incident edges to the priority queue

- Let’s see an example of how Dijkstra’s runs

- Now, let’s look at the actual code
Dijkstra’s Analysis

• Graph with $n$ vertices and $m$ edges
Dijkstra’s Analysis

- Graph with $n$ vertices and $m$ edges
- Visit each vertex once. Each time we visit a vertex takes $O(1)$ time. So in total, visiting vertices is $O(n)$ time.
Dijkstra’s Analysis

- Graph with \( n \) vertices and \( m \) edges

- Visit each vertex once. Each time we visit a vertex takes \( O(1) \) time. So in total, visiting vertices is \( O(n) \) time.

- Each edge is added to the priority queue at most once. That means the priority queue has size at most \( m \). Therefore, adding or removing an edge is \( O(\log m) \). Totalling over all edges is \( O(m \log m) \).
Dijkstra’s Analysis

• Graph with \( n \) vertices and \( m \) edges

• Visit each vertex once. Each time we visit a vertex takes \( O(1) \) time. So in total, visiting vertices is \( O(n) \) time.

• Each edge is added to the priority queue at most once. That means the priority queue has size at most \( m \). Therefore, adding or removing an edge is \( O(\log m) \). Totalling over all edges is \( O(m \log m) \).

• Summing, Dijkstra’s algorithm takes \( O(n + m \log m) \) time. Since \( m > n - 1 \) in a connected graph, this is often written \( O(m \log m) \).
Dijkstra’s Analysis

- Graph with \( n \) vertices and \( m \) edges

- Visit each vertex once. Each time we visit a vertex takes \( O(1) \) time. So in total, visiting vertices is \( O(n) \) time.

- Each edge is added to the priority queue at most once. That means the priority queue has size at most \( m \). Therefore, adding or removing an edge is \( O(\log m) \). Totalling over all edges is \( O(m \log m) \).

- Summing, Dijkstra’s algorithm takes \( O(n + m \log m) \) time. Since \( m > n - 1 \) in a connected graph, this is often written \( O(m \log m) \).

- Dijkstra’s demo!
Review!
Balanced Binary Search Trees

- What you should know:

  - Difference between a BST and a BBST
  - How to add to a binary search tree; how to search in a binary search tree
  - Balanced binary search trees maintain height $O(\log n)$
  - AVL trees use rotations to maintain balance.
  - It is possible to delete from a binary search tree, and a balanced binary search tree. (But you don't need to know how!)
Balanced Binary Search Trees

• What you should know:

  • Difference between a BST and a BBST
Balanced Binary Search Trees

- What you should know:
  - Difference between a BST and a BBST
  - How to add to a binary search tree; how to search in a binary search tree
Balanced Binary Search Trees

• What you should know:
  • Difference between a BST and a BBST
  • How to add to a binary search tree; how to search in a binary search tree
  • Balanced binary search trees maintain height $O(\log n)$
Balanced Binary Search Trees

- What you should know:
  - Difference between a BST and a BBST
  - How to add to a binary search tree; how to search in a binary search tree
  - Balanced binary search trees maintain height $O(\log n)$
  - AVL trees use rotations to maintain balance.
Balanced Binary Search Trees

- What you should know:
  - Difference between a BST and a BBST
  - How to add to a binary search tree; how to search in a binary search tree
  - Balanced binary search trees maintain height $O(\log n)$
  - AVL trees use rotations to maintain balance.
  - It is possible to delete from a binary search tree, and a balanced binary search tree. (But you don’t need to know how!)
Tree Rotation: Rotate Left

This rotation is on the orange nodes (18 and 24); for a left rotation one must be a right child of the other. We rearrange the children of these nodes (in blue).
This rotation is on the orange nodes (18 and 24); for a left rotation one must be a right child of the other. We rearrange the children of these nodes (in blue).
Last Week’s Practice Quiz
Any Other Questions?
Wrapping Up CSCI 136
What we Learned

• Lots of data structures:
What we Learned

• Lots of data structures:
  • Linear structures like vectors, linked lists
What we Learned

• Lots of data structures:
  • Linear structures like vectors, linked lists
  • Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
What we Learned

• Lots of data structures:
  • Linear structures like vectors, linked lists
  • Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  • Structures to store relationships like trees and graphs

• Java!
  • Useful programming language
  • Object oriented programming, inheritance, formal commenting
  • Breaking down problems into smaller pieces
  • Induction, recursion
  • Proofs

• How can we be sure that our methodology is correct?
What we Learned

- Lots of data structures:
  - Linear structures like vectors, linked lists
  - Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  - Structures to store relationships like trees and graphs
- Java!
What we Learned

- Lots of data structures:
  - Linear structures like vectors, linked lists
  - Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  - Structures to store relationships like trees and graphs
- Java!
  - Useful programming language
What we Learned

- Lots of data structures:
  - Linear structures like vectors, linked lists
  - Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  - Structures to store relationships like trees and graphs
- Java!
  - Useful programming language
  - Object oriented programming, inheritance, formal commenting
What we Learned

• Lots of data structures:
  • Linear structures like vectors, linked lists
  • Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  • Structures to store relationships like trees and graphs

• Java!
  • Useful programming language
  • Object oriented programming, inheritance, formal commenting

• Breaking down problems into smaller pieces
What we Learned

• Lots of data structures:
  • Linear structures like vectors, linked lists
  • Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  • Structures to store relationships like trees and graphs

• Java!
  • Useful programming language
  • Object oriented programming, inheritance, formal commenting

• Breaking down problems into smaller pieces
  • Induction, recursion
What we Learned

- Lots of data structures:
  - Linear structures like vectors, linked lists
  - Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  - Structures to store relationships like trees and graphs

- Java!
  - Useful programming language
  - Object oriented programming, inheritance, formal commenting

- Breaking down problems into smaller pieces
  - Induction, recursion
  - Proofs
What we Learned

• Lots of data structures:
  • Linear structures like vectors, linked lists
  • Structures for efficiency like balanced binary search trees, dictionaries, and priority queues
  • Structures to store relationships like trees and graphs
• Java!
  • Useful programming language
  • Object oriented programming, inheritance, formal commenting
• Breaking down problems into smaller pieces
  • Induction, recursion
  • Proofs
  • How can we be *sure* that our methodology is correct?