Dijkstra’s Algorithm

Sam McCauley

March 5, 2025

Welcome Back!


shutterstock.com · 1633503538

- Problem Set 1 back
  - Please come to office hours with questions!
  - I did my best to explain any issues I found with proofs, but it's much easier if it's a two-way discussion
- Reminder: should only use textbooks and slides; discuss with your partner and instructor/TAs. (No serious problems! Just a reminder.)
- Sample student solutions for selected problems on Glow
- Problem Set out tonight; last one before Midterm 1
- We'll discuss Midterm 1 on Monday; it is on March 10 (a week from Monday)
- Any questions?

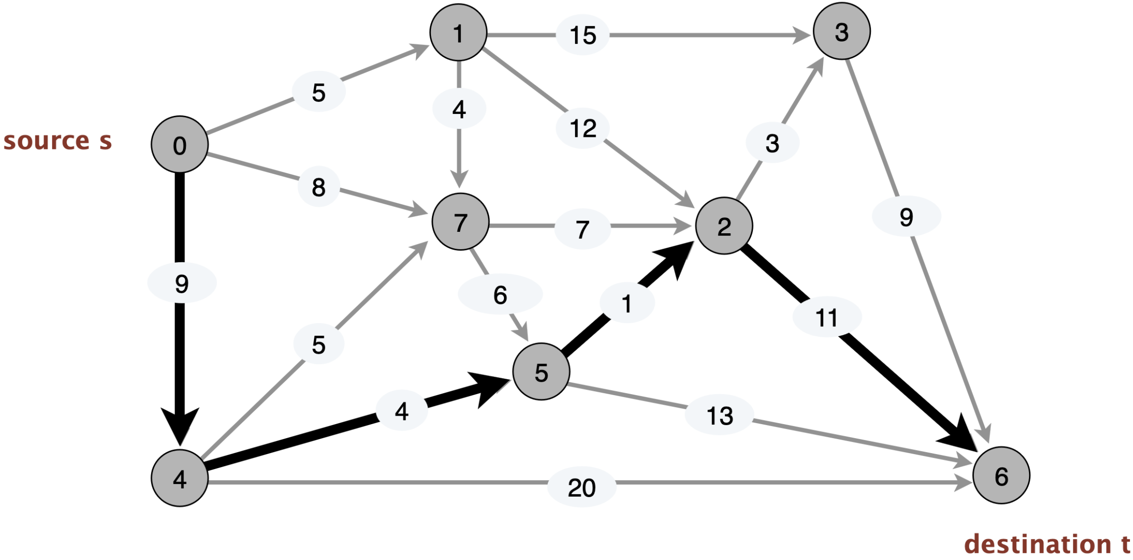
Greedy Algorithms Takeaway

- Greedy algorithms are a *sometimes* thing
- Usually fast; *Correctness* is the main question!
- Only use a greedy algorithm when you can show that it is correct
    - Starting in March we'll look at more sophisticated problem-solving techniques

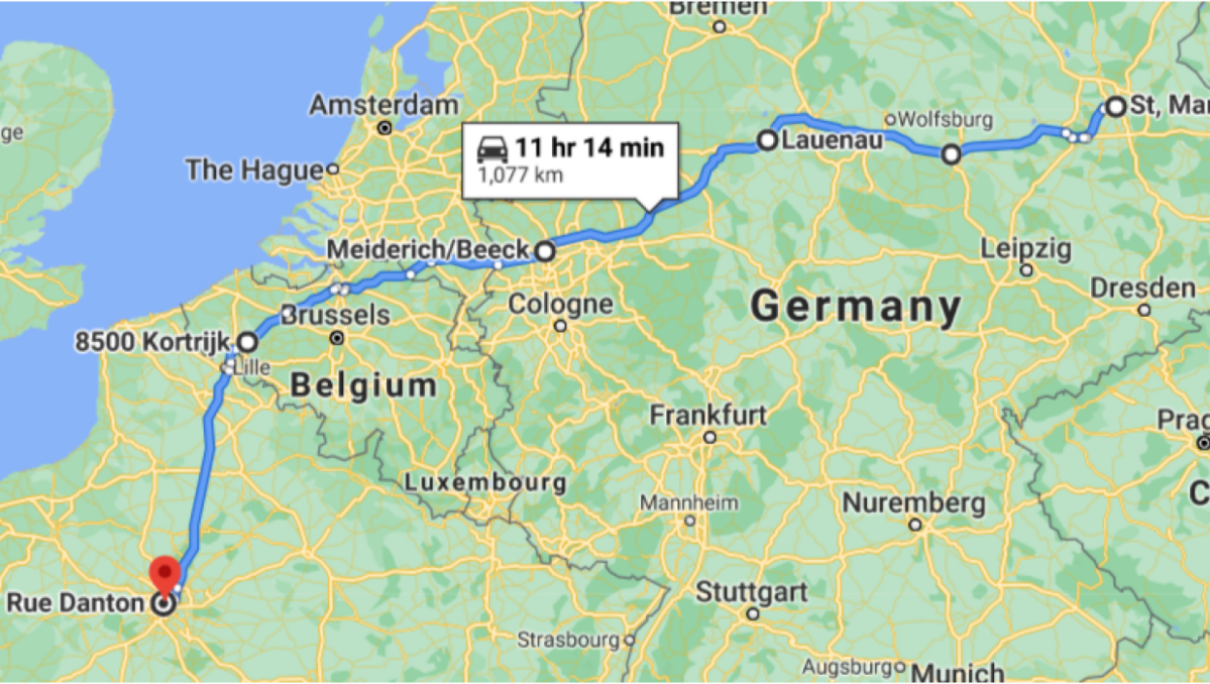
Dijkstra’s Algorithm

Shortest Path in Weighted Graphs

- Given a directed graph G with *positive* edge weights
- Find the *shortest path* from s to t
- Path p from s to t minimizing $\sum_{e \in p} w_e$



length of path = 9 + 4 + 1 + 11 = 25


Bremen
Amsterdam
The Hague
11 hr 14 min
1,077 km
Lauenau
Wolfsburg
St, Ma
Meiderich/Beeck
Leipzig
Dresden
Cologne
Germany
Brussels
8500 Kortrijk
Lille
Belgium
Frankfurt
Prag
Luxembourg
Mannheim
Nuremberg
Rue Danton
Stuttgart
Strasbourg
Augsburg
Munich

Shortest Path Applications

- Map routing
- Robot navigation
- Texture mapping
- Latex typesetting
- Traffic Planning
- Scheduling
- Network routing protocols
- We’ll revisit later in class as well (to allow for negative weights in the graph)

Shortest Path: Plan

- Greedy algorithm
- Goal: find shortest path from s to *all* vertices of the graph
    - Therefore, we get the shortest path to t
    - Assume G is connected to keep things simple. (If there is no path from s to t we will detect that anyway)
- Each time we add a new vertex, *guarantee* that we've found the shortest distance to that vertex
- Greedily grow the vertices until we've found the shortest path to all vertices
- Denote the *actual* shortest path $d(s, v)$. We will store the shortest path we find in an array $d[]$; so our goal is $d[v] = d(s, v)$.
- As we did with BFS/DFS, we can build a tree of shortest paths by defining each vertex's parent to be the one that added it to the queue
- Let's start building the algorithm [Blackboard]

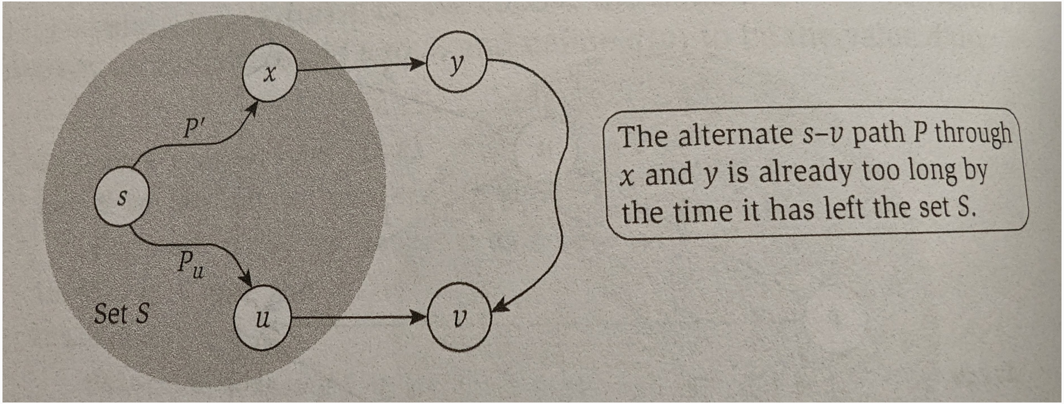
Dijkstra's Algorithm

We want to get the distance to every vertex; we'll store the distances in an array `d[]`. Idea:

1. There are some finished vertices where we've found the shortest path
2. The fringe consists of all (unfinished) neighbors of all finished vertices
3. We find the fringe vertex v with the shortest *total* path length
    - Shortest path from s to some finished v', plus the weight of the edge from v' to v
4. This path length *is* the distance from s to v! Store that distance in $d[v]$. We add v to the finished vertices, and update the fringe. The parent of v in the shortest path tree is v'.

How can we prove that this is correct? (Then: how can we implement this?)

Dijkstra’s Proof Intuition

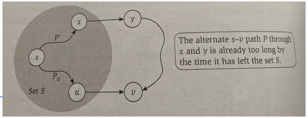


The alternate s–v path P through x and y is already too long by the time it has left the set S.

Dijkstra's Algorithm Proof Strategy

- By induction
- I.H.: after k vertices are marked finished, for any finished vertex v, $d[v]$ stores the distance from s to v.
- Base case?
    - $k = 1$; $d[s] = 0$
    - We are done because all edge lengths are positive so no path can have length less than 0.

Dijkstra's Algorithm Inductive Step



- Assume: after k vertices are finished, for all finished vertices w, $d[w] = d(s,w)$
- We find the edge $e = (u,v)$ between a finished u and and unfinished v that minimizes $d[u] + w_e$; mark v finished; set $d[v] = d[u] + w_e$. To show: $d[v] = d(s,v)$
- Now: there cannot be a path p' to v with length less than $d[u] + w_e$
  - Assume contrary. Let y be the first vertex in p' not finished, and let $e' = (x,y)$ be the edge to y in p'. Then the length of p' is at least $d(s,x) + w_{e'} + d(y,v)$
  - We have $d(s,x) = d[x]$ by I.H., and $d[x] + w_{e'} \geq d[u] + w_e$ by definition of Dijkstra's
  - $d(y,v) \geq 0$ since all edge weights are positive
  - So length of p' is:

$$d(s,x) + w_{e'} + d(y,v) \geq d[x] + w_{e'} \geq d[u] + w_e.$$

Implementing Dijkstra's Algorithm

- Notice: we don't need to keep track of which vertices are "finished". We mark a vertex as finished exactly when we fill in the array d. So we start with $d[v] = \infty$ for all v, and the finished vertices are those with $d[v] \leq \infty$.
- How do we keep track of the fringe? How do we find the fringe vertex with smallest path length?
- We can keep the fringe in a linked list, and scan through it every time. But that's very slow.
- What operations do we want to do on the fringe?
  - Insert a new path to a vertex into the fringe
    - Like we saw with BFS/DFS: some vertices might wind up in the fringe multiple times. That's OK; if we remove a finished vertex from the fringe we ignore it
  - Remove the smallest-path-length vertex from the fringe

Priority Queues

(Lengthy) Aside: Priority Queues

- **Priority queue:** A data structure that can store a set of items, each with some *priority*, with the following operations:
    - **Insert**(i, p)**:** Insert a new item i with priority p into the priority queue
    - **RemoveMin**$()$**:** Remove (and return) the item i with the smallest priority in the queue
- Has anyone seen a priority queue before? How to implement a priority queue before?

Heap

- Let’s define a data structure using a tree
- Then: we’ll significantly simplify it into an array data structure
- This data structure is called a *heap*
- Invariant: Each element in the heap is smaller than its children
  - This implies: each element in the heap is smaller than all of its descendants
- Invariant 2: The heap is a *complete* binary tree: all levels but the last are “full”; last is filled left to right
- Let’s draw a heap [Blackboard]
- Does *not* necessarily satisfy Binary Search Tree property (left child is less than right child)

Inserting into a Heap

In pairs: How can we insert into a heap to maintain these invariants?

- Where does the new item go?
  - Last level must be filled in left to right; let's put it there
- How can we ensure that this satisfies the heap property?
  - Swap with parent until the heap property is satisfied
  - Called "sift up"
- Why does this work? [Blackboard]

Inserting into a Heap: Analysis

- What is the height of a complete binary tree with n nodes?
    - $O(\log n)$
- Each swap takes $O(1)$ time, so insert takes $O(\log n)$ time

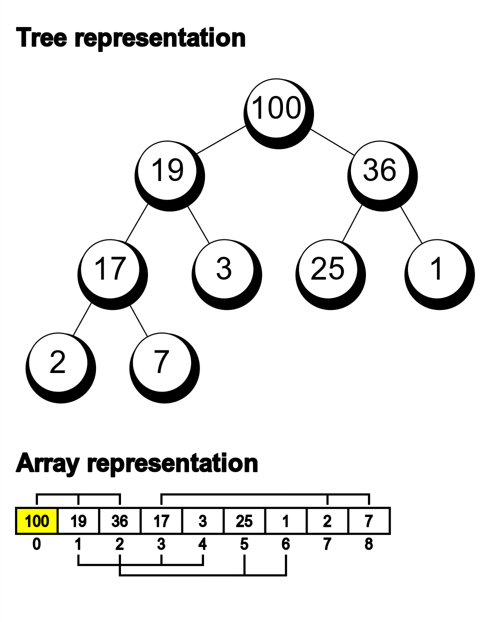
Removing the Minimum Element of a Heap

- How can we remove the minimum element?
  - It would be a lot nicer to remove the last element of the last level
  - Idea: Swap the root and the last element. Then we can safely remove it. Then, we'll "sift down" to preserve the heap property [Blackboard]
- To sift down: swap the element with its *smaller* child (why?). Repeat until heap property is maintained
- Also $O(\log n)$

Heap as array

- Observation: The shape of our tree is super restricted. Can we store it in an easier way?
- Let’s number the nodes starting at 1, in level order
- If a node has number i, what numbers are its children?
  - $2i$ and $2i + 1$
- If a node has number i, what number is its parent?
  - $\lfloor i/2 \rfloor$
- (Can prove by induction.)
- Let’s throw out the tree and do a heap operation using just the array! [Blackboard]

Priority Queue



100	19	36	17	3	25	1	2	7
0	1	2	3	4	5	6	7	8

- Insert a new item (Insert)
- Remove minimum weight item (ExtractMin)
- Done using a heap
- *Extremely* efficient; used extensively in practice
- $O(\log n)$ time to insert or remove minimum item
- Will help us out with Dijkstra’s algorithm

Heaps (Reference)

- Heap property: each item in the tree is smaller than either of its children
- Tree has minimum height; filled in left to right (“full” tree)
- Maintain implicitly in an array (do not need pointers!)
- Extract min, or insert a new item, in $O(\log n)$ time
- Fun fact: Can build a heap (even on unsorted data) in $O(n)$ time (!)

Implementing Dijkstra's Algorithm: Revisit

- Notice: we don't need to keep track of which vertices are "finished". We mark a vertex as finished exactly when we fill in the array d. So we start with $d[v] = \infty$ for all v, and the finished vertices are those with $d[v] \leq \infty$.
- How do we keep track of the fringe? How do we find the fringe vertex with smallest path length?
- We keep the fringe in a priority queue
  - Insert a new path to a vertex into the fringe
  - Remove the smallest-path-length vertex from the fringe

Dijkstra's Algorithm

We want to get the distance to every vertex; we'll store the distances in an array `d[]`. All finished vertices v will have $d[v] \leq \infty$. Then:

1. We store a priority queue fringe consists of all (unfinished) neighbors of all finished vertices
2. We use RemoveMin() to find the fringe vertex v with smallest path length
    - If v has $d[v] < \infty$ we ignore it
    - Otherwise: this path length is the distance to v. Store that distance in $d[v]$.
    - We add v to the finished vertices, and add all neighbors v' of v to the priority queue, with priority $d[v] + w(v, v')$.
3. Repeat the above until the fringe is empty

Dijkstra's Algorithm Analysis

- Each vertex is finished once
- Each time a vertex is finished, we add all of its unfinished neighbors to the fringe
- How large can the fringe get?
    - $O(m)$: a vertex with d_v neighbors is in the fringe $\leq d_v$ times
- Time per vertex: $O(\log m + d_v \log m)$
- Summing over all vertices: $O(n \log m + m \log m) = O(m \log n)$ Substitution: $m \geq n - 1$ because the graph is connected; $m \leq n^2$ in all graphs

Improving Dijkstra’s Algorithm

- We are being wasteful with our edge storage!
- Only need to store one edge to each fringe node
- Only need a priority queue of n items!
- But: what happens when we find a new edge to a vertex that was already in the fringe (i.e. in the priority queue)?
  - Need to *update* the best path length to the vertex
  - Must modify the priority priority queue! How can we update the weight of a vertex in a heap?
  - Just sift up; $O(\log n)$ time
- In practice: queue is usually much smaller than n; runs quite quickly
- In theory: using a Fibonacci heap can insert and decrease key in $O(1)$; extract minimum in $O(\log n)$
  - Gives $O(m + n \log n)$ running time for Dijkstra’s algorithm
  - Can we do better? *Open problem.* (?)
  - If edge weights are integers can get $O(m)$ running time

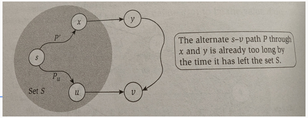
Dijkstra’s Algorithm Implementation Pseudocode

```
function Dijkstra(Graph, source):
    for all v:
        initialize dist[v] $\leftarrow \infty$ and prev[v] $\leftarrow 0$
    dist[source] $\leftarrow 0$
    add source to Q
    while Q is not empty:
        remove u with minimum priority from Q
        dist[u] $\leftarrow$ priority of u in Q
        for each neighbor v of u with dist[v] $= \infty$:
            alt $\leftarrow$ dist[u] + Graph.Edges(u, v)
            if v is in Q:
                if alt < current priority of v:
                    reduce priority of v in Q to alt
                    prev[v] $\leftarrow$ u
            if v is not in Q:
                add v to Q with priority alt
                prev[v] $\leftarrow$ u
    return dist[], prev[]
```

Negative Edge Weights

- Why doesn’t Dijkstra’s algorithm work if edge weights are negative?
- Definitely can’t have any *cycles* whose total weight is negative. (Why is that?)
- Let’s look at the proof to give us a hint

Reminder: Dijkstra's Algorithm Inductive Step



The alternate s–v path P through x and y is already too long by the time it has left the set S.

- Assume: after k vertices are finished, for all finished vertices w, $d[w] = d(s, w)$
- We find the edge $e = (u, v)$ with u finished and v unfinished that minimizes $d[u] + w_e$; mark v finished; set $d[v] = d[u] + w_e$. To show: $d[v] = d(s, v)$
- Now: there cannot be a path p' to v with length less than $d[u] + w_e$
    - Assume contrary. Let y be the first vertex in p' not finished, and let $e' = (x, y)$ be the edge to y in p'. Then the length of p' is at least $d(s, x) + w_{e'} + d(y, v)$.
    - We have $d(s, x) = d[x]$ by I.H., and $d[x] + w_{e'} \geq d[u] + w_e$ by definition of Dijkstra's
    - $d(y, v) \geq 0$ since all edge weights are positive
    - So length of p' is:

$$d(s, x) + w_{e'} + d(y, v) \geq d[x] + w_{e'} \geq d[u] + w_e.$$

Negative Edge Weights

- Why doesn't Dijkstra's algorithm work if edge weights are negative?
- Definitely can't have any *cycles* whose total weight is negative. (Why is that?)
- If there are negative edge weights, then the alternate path might be better! Take a larger cost to get to the fringe, then a negative path to recover that cost to get to our vertex
- Let's try to draw an example **In pairs (if we have time)** then [Blackboard]