Dijkstra's Algorithm and Divide and Conquer

Sam McCauley

March 4, 2024

Welcome Back!

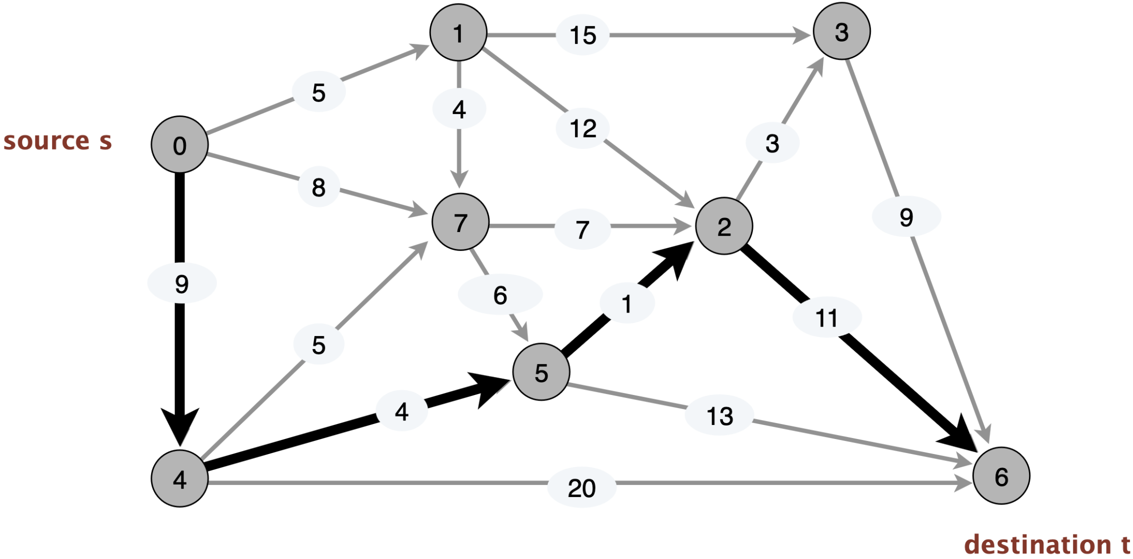
```
     6 7 5
   x 1 4 4
  ---------
   2 7 0 0
 2 7 0 0 0
+6 7 5 0 0
 ---------
 9 7 2 0 0
```

- Midterm back soon
- Assignment released Wednesday
- Current plan: Assignments 3 and 5 in groups; Assignment 4 solo
- Today we’ll have a fun discussion of how to optimize Kruskal’s, then Dijkstra’s algorithm, and finally intro divide and conquer
- Hand-multiplying integers?

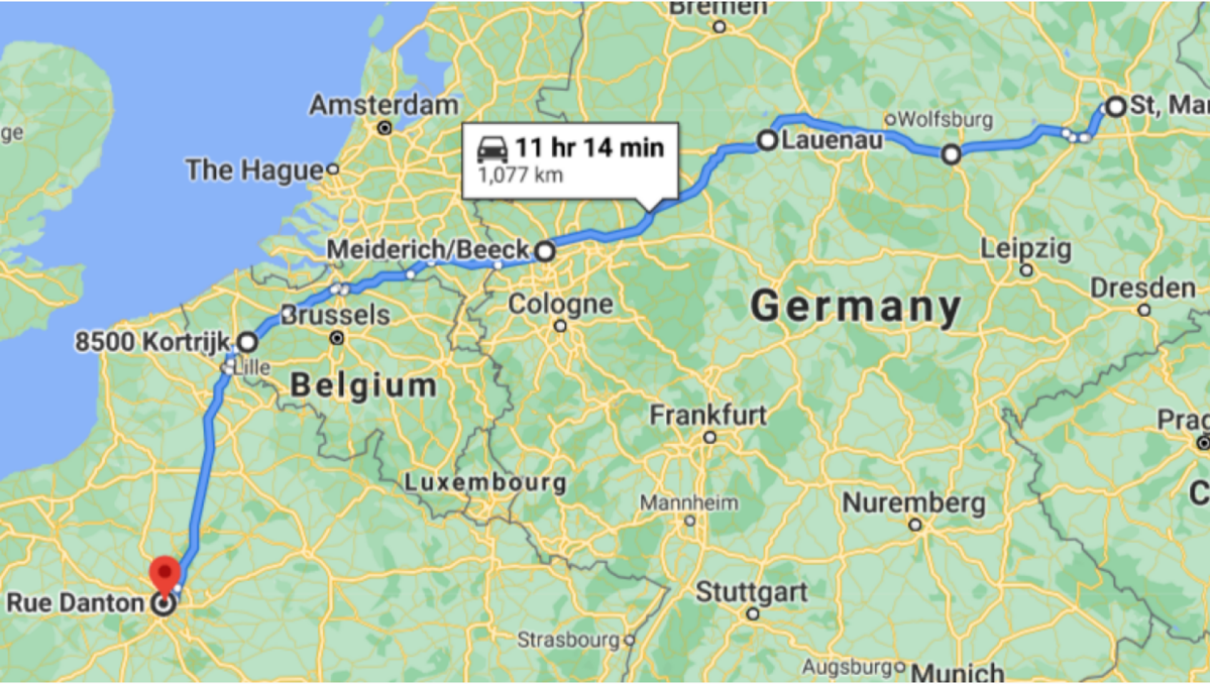
Dijkstra’s Algorithm

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
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

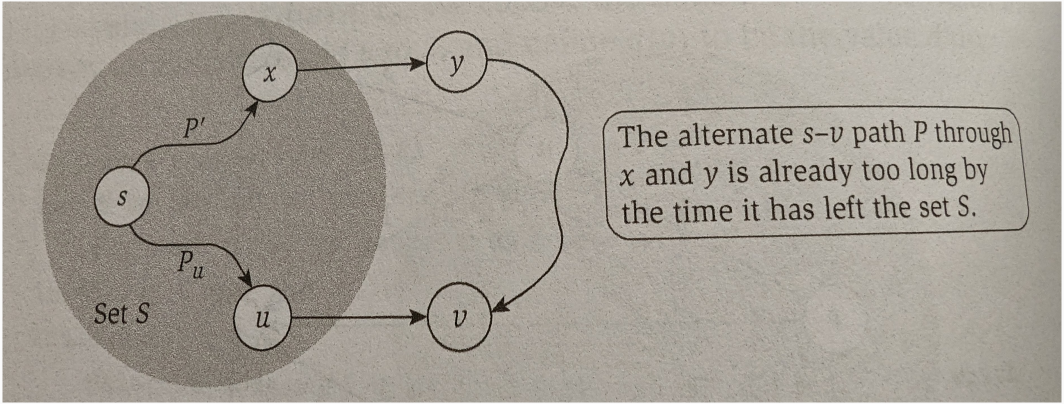
- Greedy algorithm much like Prim's
- Find shortest path from s to *all* vertices of the graph
  - Therefore, we get the shortest path to t
  - Assume G is connected to keep things simple. (If there is no path from s to t we will detect that anyway)
- Each time we add a new vertex, *guarantee* that we've found the shortest distance to that vertex
- Greedily grow the vertices we've found the shortest path to
- Denote the *actual* shortest path $d(s, v)$. We will store the shortest path we find in an array $d[]$; so our goal is $d[v] = d(s, v)$.
- Let's start building the algorithm [On Board #1]

Dijkstra's Algorithm

- Maintain a set S of vertices we have found the shortest path to; array d of shortest paths
- Start with $S \leftarrow \{s\}$; $d[s] = 0$; $d[v] = \infty$ for all $v \neq s$
- To add a new vertex to S:
  - Among all cut edges C of S
  - Find the edge $e = (u, v) \in C$ minimizing $d[u] + w_e$
  - Set $d[v] = d[u] + w_e$; add v to S

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

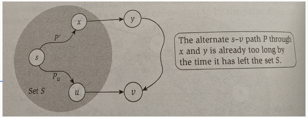
Dijkstra's Proof Intuition



Dijkstra's Algorithm Proof Strategy

- By induction
- I.H.: for any k, if $|S| = k$, then for any $v \in S$, $d[v]$ stores the length of the shortest path 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 that for some set S of size k, for all $w \in S$, $d[w] = d(s, w)$
- We find cut edge $e = (u, v)$ minimizing $d[u] + w_e$; add v to S; set $d[v] = d[u] + w_e$. To show: $d[v] = d(s, v)$
- Claim: for *any* vertex $y \notin S$, $d(s, y) \geq d[u] + w_e$
  - Idea: If there is a shorter path to y, there must be a smaller cut edge [On Board #2]
- Now: there cannot be a path p' to v with length less than $d[u] + w_e$
  - Idea: assume contrary. Let y be the first vertex in p' not in S. Then the length of p' is at least $d(s, y) + d(y, v)$
  - $d(s, y) \geq d[u] + w_e$ from claim above; $d(y, v) \geq 0$

Implementing Dijkstra's Algorithm

- Dijkstra's is correct by induction (see above)
- How can we find the smallest cut edge?
- Same technique as Prim's algorithm!
- Keep a priority queue Q of cut edges; "priority" of an edge $e = (u, v)$ is $d[u] + w_e$
- Remove smallest-weight edge $e' = (x, y)$ from Q. If $y \in S$, skip it. Otherwise, add y to S, and set $d[y] = d[x] + w_{e'}$
- Running time?
  - $O(m \log m)$ (each edge is added to the queue only once; $O(\log m)$ to add it or extract minimum)

Improving Dijkstra's Algorithm

- We are being wasteful with our edge storage!
- Only need to store one edge to each node in $V \setminus S$ [On Board #3]
- Only need a priority queue of n items!
- But: what happens when we find a new edge to a vertex not in S?
  - Need to update the vertex's weight
  - Must modify the priority queue! How can we update the weight of a vertex in a heap?
- In practice: smaller queue means runs faster
- In theory: using a Fibonnacci 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

Divide and Conquer Algorithms

Algorithmic Design Paradigms

- Greedy Algorithms
  - Gas-filling; maximum interval scheduling
  - Prim's, Kruskal's, Dijkstra's
  - Idea: we choose an item to add *permanently* to the solution
  - Proof that each item we have is correct
- Divide and Conquer ⇐ we are here!
  - Divide problem into multiple parts
  - *Combine* solutions into a new correct solution
- Dynamic Programming
- Network Flow

Sorting

- Selection sort: take largest item; place it in last slot; repeat
- Can be viewed as "greedy:" once we place an item, we have proven that it stays there irrevocably
- $\Theta(n^2)$ time (requires $\Omega(i)$ time to find largest of i items)
- Can we do better with divide and conquer?
- Let's revisit Merge Sort, and talk about how to analyze it

Merge Sort

Goal: sort an array A of size n (Assume $|A|$ is a power of 2 for simplicity)

- If $|A| \leq 1$ return A
- Otherwise, sort the left half of A and the right half of A using Merge Sort
- "Merge" the two halves together to create a sorted array

Let's look at how to merge efficiently [On Board #4]

Running time? $O(n)$

Merge Sort

- Classic divide and conquer algorithm; need:
  - A base case
  - A way to divide into smaller instances
  - A way to combine the solution for smaller instances into an overall solution
- What do we need for correctness?
  - Combining smaller solutions must give correct solution for overall instance
  - Base case must be correct
  - Must *reach* the base case!

Divide and Conquer Running Time

- Analyzing D & C algorithms can be initially confusing
- Challenge: the algorithm “jumps” all over the place due to the recursive structure
- Today: *group/categorize* costs to allow us to analyze divide and conquer more effectively

Merge Sort Running Time

What is the running time of Merge Sort on an array of size n?

One answer:

- running time of Merge Sort on an array of size $n/2$, plus
- running time of Merge Sort on a second array of size $n/2$, plus
- $O(n)$ to merge.
- Or, if $n = 1$, then the cost is 1.

Let $T(n)$ be the *exact* cost of Merge Sort on an array of size n. Then:

$$T(n) = 2 \cdot T(n/2) + O(n), \qquad T(1) = 1$$

Recurrences

Recurrences

- To find the running time of a divide and conquer algorithm, we write a *recurrence*
- Let $T(n)$ be the cost of the algorithm on a problem of size n. Can write $T(n)$ as:
  - A base case for small n (oftentimes $T(1) = 1$)
  - A sum of the "divide" recursive calls which can be written in terms of T (e.g. $T(n/2)$), plus the cost to "conquer"
- A solution to this recurrence gives our total running time!

First example: merge sort

- $T(n) = 2T(n/2) + O(n); T(1) = 1$
- First: set constants
- For some c, $T(n) \leq 2T(n/2) + cn; T(1) \leq c$
- How can we solve this?

Recurrence Tree Technique

- Let's draw the recurrence as a tree [On Board #5]
- Idea: this drawing will help us group together the costs of the algorithm
- How does Merge Sort actually run?
- But: can we bound the cost of a given level of the tree?
  - Yes: each level costs cn in total
  - Specifically: level i has 2^i subproblems, each with cost $\leq cn/2^i$
- How many levels are there?
- What is the total cost of Merge Sort?

Recurrence Tree Analysis: Merge Sort

- What is this level-by-level analysis saying about Merge Sort?
- Look at *all* work we do across all subproblems of size $n/2^i$
- Answer: cn total work
- So we do cn total work on the subproblem of size n; cn total work on the 2 subproblems of size $n/2$; cn on the four subproblems of size $n/4, \ldots, n$ on the n subproblems of size 1
- That's $\leq cn(\log_2 n + 1)$ total work!

Double-Checking our Work

- We wanted a solution to:

$$T(n) = 2 \cdot T(n/2) + cn, \qquad T(1) = c$$

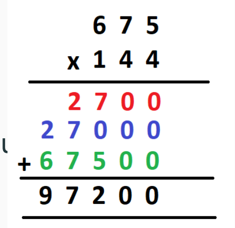
- Does $cn(\log_2 n + 1)$ satisfy this?
  - Yes.

$$\begin{aligned} cn(\log_2 n + 1) &\leq 2\left(\frac{cn}{2}\left(\log_2 \frac{n}{2} + 1\right)\right) + cn \\ &= cn\left(\log_2 \frac{n}{2} + 1\right) + cn \\ &= cn\left(\log_2 n - \log_2 2 + 1\right) + cn \\ &= cn\left(\log_2 n\right) + cn \checkmark \end{aligned}$$

Stepping Back

- Merge Sort divides the array into halves, sorts each half, and then recombines them in $O(n)$ time
- Running time is initially difficult to see
- We wrote the running time as a recurrence
- To solve the recurrence, we drew a tree, which helped us group the costs
- $\log_2 n$ levels, each of cost $O(n)$, means $O(n \log n)$ total cost!

Divide and Conquer: Multiplication



- Let's say we want to multiply two n-digit numbers $a \times b$ (let's assu base 10, but the same idea holds for binary numbers)
  - Let's say n is much larger than 64, so our CPU
- What is the running time of the algorithm you learned in school?
  - For each digit of b, multiply with each digit of a; carry as necessary
  - $O(n)$ time for each digit of b
  - $O(n^2)$ time overall
- Addition is only $O(n)$ however
- Can we do multiplication more efficiently? In 1960, Kolmogorov *conjectured* no; any algorithm takes $\Omega(n^2)$ worst-case time

Divide and Conquer: Multiplication

Assume n is a power of 2 for the moment for simplicity.

- Let's write a as the sum of two $n/2$-bit numbers: $a = 10^{n/2}a_\ell + a_r$
- Let's write b as the sum of two $n/2$-bit numbers: $b = 10^{n/2}b_\ell + b_r$
- Then $a \times b = (10^{n/2}a_\ell + a_r)(10^{n/2}b_\ell + b_r)$
- Using algebra, $a \times b = 10^n(a_\ell + b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$.

Divide and Conquer: Multiplication

$$a \times b = 10^n (a_\ell b_\ell) + 10^{n/2} (a_\ell b_r + b_\ell a_r) + a_r b_r$$

- So we can use divide and conquer! To multiply two n-digit numbers, we first perform four recursive multiplications:
  - $a_\ell \times b_\ell$, $a_\ell \times b_r$, $b_\ell \times a_r$, and $a_r \times b_r$
- And then we add them together in $O(n)$ time.
- Recurrence?
- $T(n) = 4T(n/2) + O(n)$; $T(1) = 1$
- Let's solve this recurrence together on the board!
- Get $\Theta(n^2)$ time, same as before (*for now...*)

Divide and Conquer: A Very Clever Algorithm (Karatsuba's Algorithm)

$$a \times b = 10^n(a_\ell b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$$

- Consider the following *three* recursive multiplications
  - $a_\ell \times b_\ell$, $a_r \times b_r$, and $(a_\ell + a_r) \times (b_\ell + b_r)$
- I claim this is enough! Why?
- $a_\ell b_r + b_\ell a_r = (a_\ell + a_r) \times (b_\ell + b_r) - a_\ell \times b_\ell - a_r \times b_r$
- So after *three* recursive calls of size $n/2$ I can calculate $a \times b$. I used $O(n)$ total time other than the recursive calls
- $T(n) = 3T(n/2) + O(n)$; $T(1) = 1$

Solving the Multiplication Recurrence

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = 1$$

- Let's solve this recurrence [On Board #6]
- We want to ask ourselves: What is the height of the tree? What is the cost of each level?
- Solution: $O(n^{\log_2 3}) = O(n^{1.58})$ time
- Much better than n^2!
- Reflect: why did changing a *constant* from 3 to 4 have such an impact on the running time?

Multiplying Numbers Efficiently

- Kolmogorov conjectured that $\Omega(n^2)$ time is needed; stated this conjecture in a seminar at Moscow State University in 1960
- Karatsuba, a student figured out this $O(n^{\log_2 3})$ time algorithm in the next week
- Kolmogorov cancelled the whole seminar and then published the result on Karatsuba's behalf without telling him
- Can we do better?
- Best known: $O(n \log n)$ [Harvey, van der Hoeven 2019]
- Are these speedups useful in practice?
    - Sometimes! Karatsuba's is used in some libraries