

Dijkstra's Algorithm and Graph Modeling

CS/MCS 401: Computer Algorithms I

Lecture 11

Professor: Jonathan Liu

Introduce yourself!

Icebreaker: What's your favorite beverage?

Announcements

I owe you a lot of things that will be released today:

- HW 2 Solutions
- HW 2 Grades
- HW 3
 - Relevant to, and fair game for, the midterm!

Lesson Goals

Today we're going to recap Dijkstra's algorithm, and talk in more detail about why it works.

Then, we're going to get practice with the major skill/outcome from this unit: graph modeling.

Dijkstra's Algorithm



Dijkstra, on Dijkstra's Algorithm

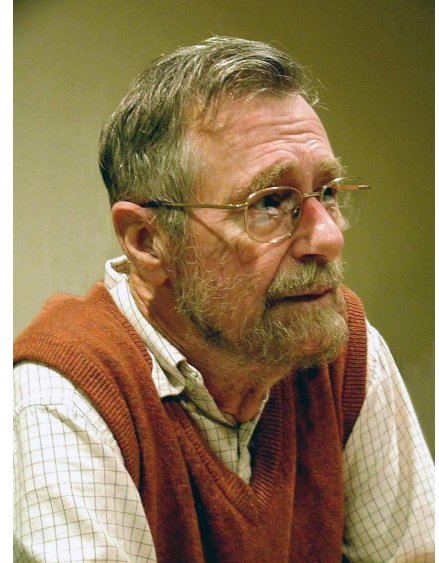
"What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame."

- Edsger Dijkstra, 2001



Dijkstra, on Software Engineering

A number of these phenomena have been bundled under the name "Software Engineering". As economics is known as "The Miserable Science", software engineering should be known as "The Doomed Discipline", doomed because it cannot even approach its goal since its goal is self-contradictory. Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has accepted as its charter "How to program if you cannot."



<https://www.cs.utexas.edu/~EWD/ewd10xx/EWD1036.PDF>

Dijkstra's Algorithm for Single Source Shortest Paths

Dijkstra's($G=(V,E)$, starting vertex s):

$S \leftarrow \{s\}$ $\backslash \backslash$ set of visited vertex

$d[s] \leftarrow 0$ $\backslash \backslash$ distances from s to every vertex

while $S \neq V$:

 get the unvisited vertex v minimizing $d[u] + l_{(u,v)}$

 add v to S

$d[v] \leftarrow d[u] + l_{(u,v)}$

$l_{(u,v)}$: length of edge (u, v)

Dijkstra's algorithm returns the shortest path from s to every point on the graph.

BFS vs DFS vs Dijkstra's Algorithm!

WhateverFirstSearch($G=(V,E)$, starting vertex s):

put s into the bag

while the bag is not empty:

take v from the bag

if v is unvisited:

mark v as visited

for each edge (v, w)

put w into the bag

BFS: The “bag” is a queue. First in, first out.

DFS: The “bag” is a stack. First in, last out.

Dijkstra's: The “bag” is a priority queue that takes out the vertex closest to the visited vertices.

Dijkstra's Algorithm for Single Source Shortest Paths

Dijkstra's($G=(V,E)$, starting vertex s):

$S \leftarrow \{s\}$ $\backslash\backslash$ set of visited vertex

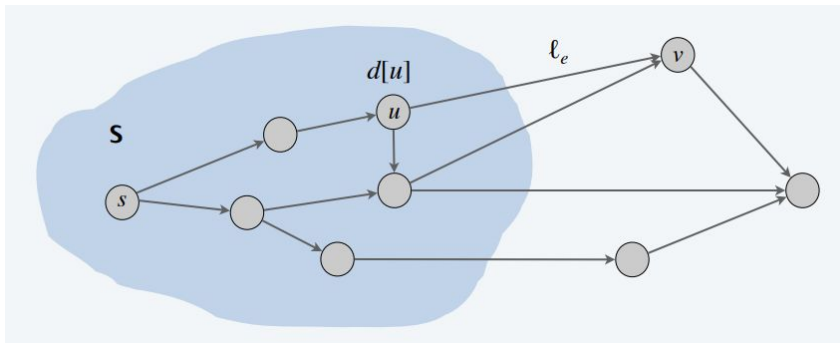
$d[s] \leftarrow 0$ $\backslash\backslash$ distances from s to every vertex

while $S \neq V$:

 get the unvisited vertex v minimizing $d[u] + \ell_{(u,v)}$

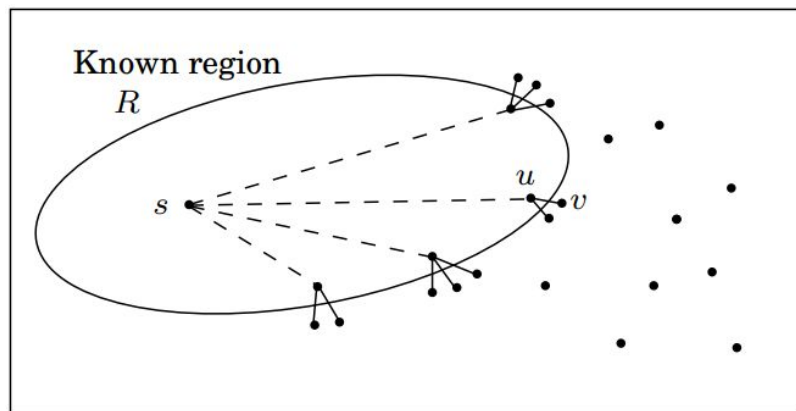
 add v to S

$d[v] \leftarrow d[u] + \ell_{(u,v)}$



Interpretation:

Slowly create a region of vertices whose distances from s you already know. At each step, take one edge from the region to a new vertex, prioritizing the closest ones to s .



Dijkstra's Algorithm for Single Source Shortest Paths

Proof of Correctness: Want to prove: returns the shortest path from start to any vertex.

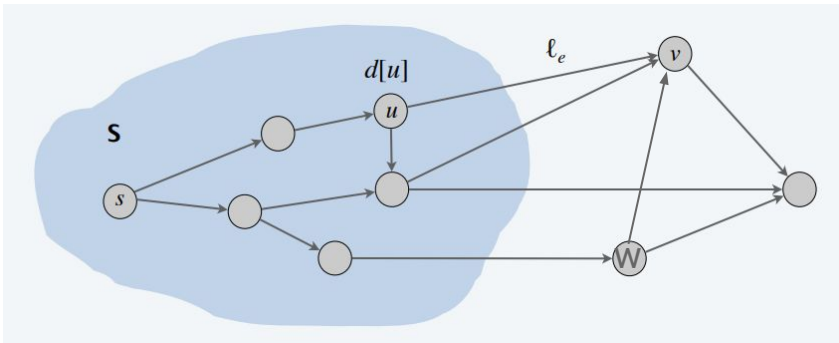
Claim: For each v , when we add v to S , $d[u] + l_{(u,v)}$ is the length of the shortest path from s to v .

Induct on $|S|$. Base Case: works. IH: Visited region so far is computed correctly.

IS: Need to show: new v is also computed correctly. Take some other path to v . It must have more than one step from the known region to v . By the minimization, that first step already has higher distance than $d[u] + l_{(u,v)}$. So it can not find a shorter path to v .

Dijkstra's($G=(V,E)$, starting vertex s):

```
S ← {s}           \\ set of visited vertex
d[s] ← 0           \\ distances from s to every vertex
while S ≠ V:
    get the unvisited vertex v minimizing d[u] + l(u,v)
    add v to S
    d[v] ← d[u] + l(u,v)
```



Dijkstra's Algorithm for Single Source Shortest Paths

Runtime:

Dijkstra's($G=(V,E)$, starting vertex s):

$S \leftarrow \{s\}$ $\backslash\backslash$ set of visited vertex

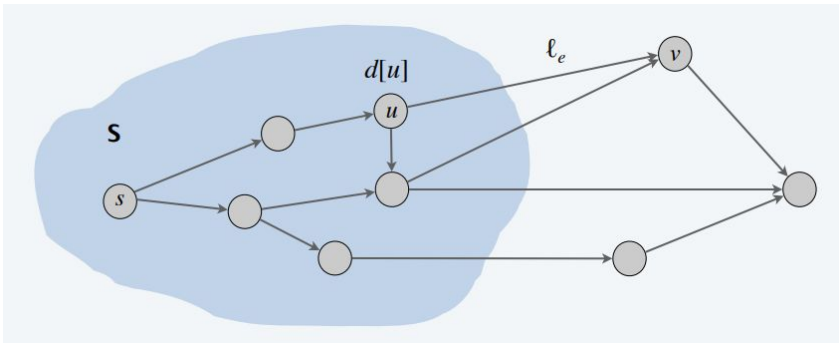
$d[s] \leftarrow 0$ $\backslash\backslash$ distances from s to every vertex

while $S \neq V$:

 get the unvisited vertex v minimizing $d[u] + \ell_{(u,v)}$

 add v to S

$d[v] \leftarrow d[u] + \ell_{(u,v)}$



Dijkstra's Algorithm for Single Source Shortest Paths

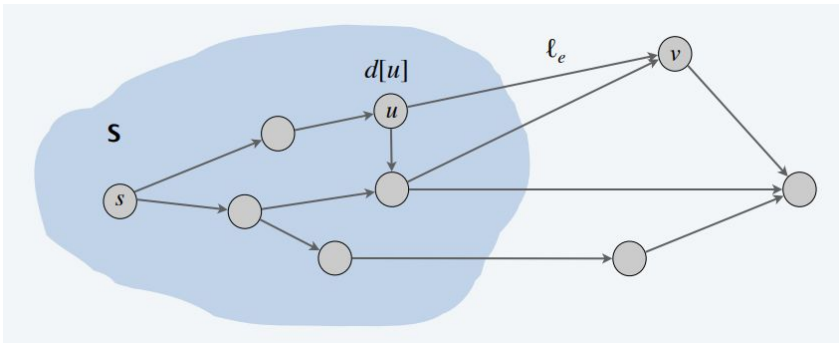
Runtime:

The runtime of Dijkstra's Algorithm depends basically entirely on how quickly you can find this unvisited vertex!

With different priority queue implementations, you can bring it down. Dijkstra himself said you should just scan all vertices, giving a $O(V^2)$ runtime. **A normal priority queue brings the runtime to $O(E \log V)$, which is the standard for this course.** With better data structures, you can bring it down to $O(E + V \log V)$.

Dijkstra's($G=(V,E)$, starting vertex s):

```
S ← {s}           \\ set of visited vertex
d[s] ← 0           \\ distances from s to every vertex
while S ≠ V:
    get the unvisited vertex v minimizing  $d[u] + \ell_{(u,v)}$ 
    add v to S
    d[v] ← d[u] +  $\ell_{(u,v)}$ 
```



New Runtime Record!

Breaking the Sorting Barrier for Directed Single-Source Shortest Paths

Ran Duan^{*} Jiayi Mao^{*} Xiao Mao[†] Xinkai Shu[‡] Longhui Yin^{*}

July 31, 2025

Abstract

We give a deterministic $O(m \log^{2/3} n)$ -time algorithm for single-source shortest paths (SSSP) on directed graphs with real non-negative edge weights in the comparison-addition model. This is the first result to break the $O(m + n \log n)$ time bound of Dijkstra's algorithm on sparse graphs, showing that Dijkstra's algorithm is not optimal for SSSP.

Our algorithm is based on the idea above, but instead of using a Dijkstra-like method where the frontier is dynamic and thus intractable, we use a divide-and-conquer procedure that consists of $\log n/t$ levels, each containing a set of frontier vertices and an upper bound B , such that a naive implementation would still spend $\Theta(t)$ time per frontier vertex and the running time would remain $\Theta(\log n)$ per vertex. We can however apply the aforementioned frontier reduction on each level so that the $\Theta(t)$ work only applies to the pivots, about $1/\log^{\Omega(1)}(n)$ of the frontier vertices. Thus the running time per vertex gets reduced to $\log n/\log^{\Omega(1)}(n)$, which is a significant speedup.

Graph Algorithm Design



An Overview of Graph Algorithm Design

Why reinvent the wheel?

As you've seen in the homework, proofs are a lot easier when we can call algorithms that we already are familiar with.

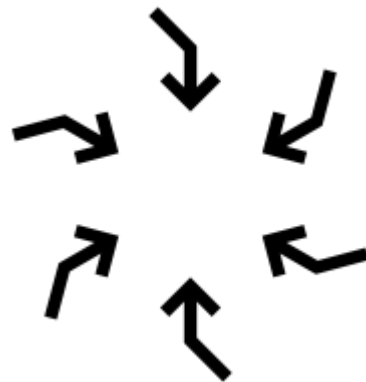
For that reason, *designing* algorithms for graph problems often **making a reduction** to an algorithm we know:

- Create the graph
- Adjust the graph in some way
- **Call an algorithm that we know**
- Use the output to figure out what it is we wanted to return

We say A reduces to B if you can use B to solve A.

Example Problem

You and your k friends live in different places, but you've agreed to meet at a party in the city. Find the shortest paths from everyone to the party.



Example Problem

You and your k friends live in different places, but you've agreed to meet at a party in the city. Find the shortest paths from everyone to the party.

Brute Force: Run Dijkstra's once for each friend. $O(kE \log V)$.

Smarter:

- Adjust the graph in some way
- **Call an algorithm that we already know**
- Use the output to figure out what it is we wanted to return

Example Problem

You and your k friends live in different places, but you've agreed to meet at a party in the city. Find the shortest paths from everyone to the party.

Brute Force: Run Dijkstra's once for each friend. $O(kE \log V)$.

Smarter:

- Reverse the graph!
- **Call Dijkstra's algorithm once, from the source.**
- Use all the output paths and return them reversed.
- $O(E \log V)$

Takeaway: You should know how to modify graphs!

A big takeaway here is that, while it's important to know the graph algorithms, it's just as important to be comfortable modifying graphs and understanding the implications of these modifications.

Worksheet:

- These problems are designed to be a bit tricky. Work with a partner! Answer each question yourself, then discuss with a partner before moving on. If you can't come to an agreement, come talk to me!
- These problems are really good midterm practice ;)
- Answers will be available on the course website by the end of this week.

Graph Modeling



An Overview of Graph Algorithm Design

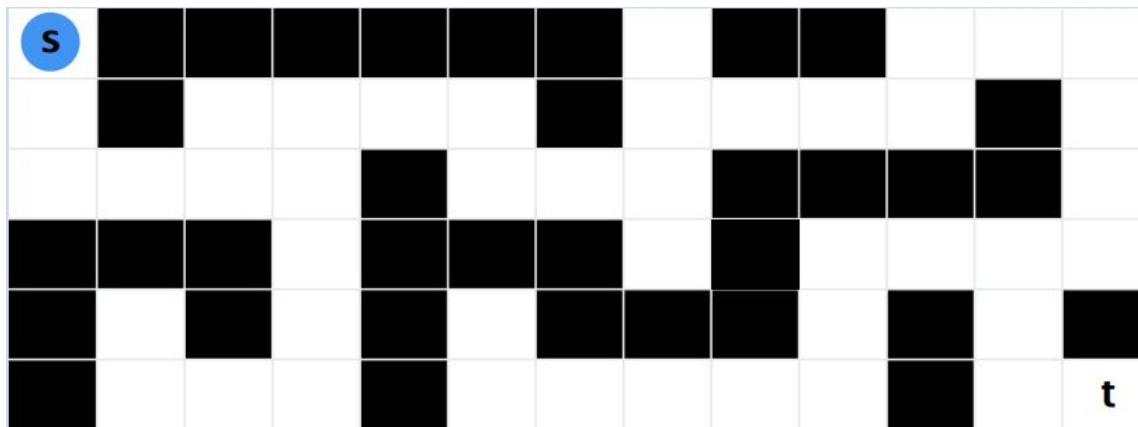
Here are the steps for a reduction that I named earlier.

- Create the graph
- Adjust the graph in some way
- Call an algorithm that we already know
- Use the output to figure out what it is we wanted to return

One thing to note is that you won't always be given a graph!

Example Problem

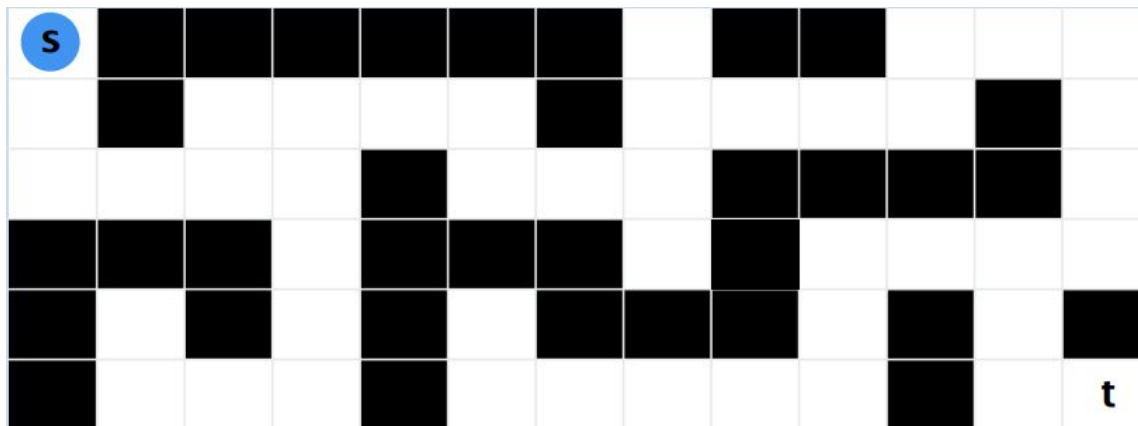
Find the shortest path from s to t.



Example Problem

Find the shortest path from s to t.

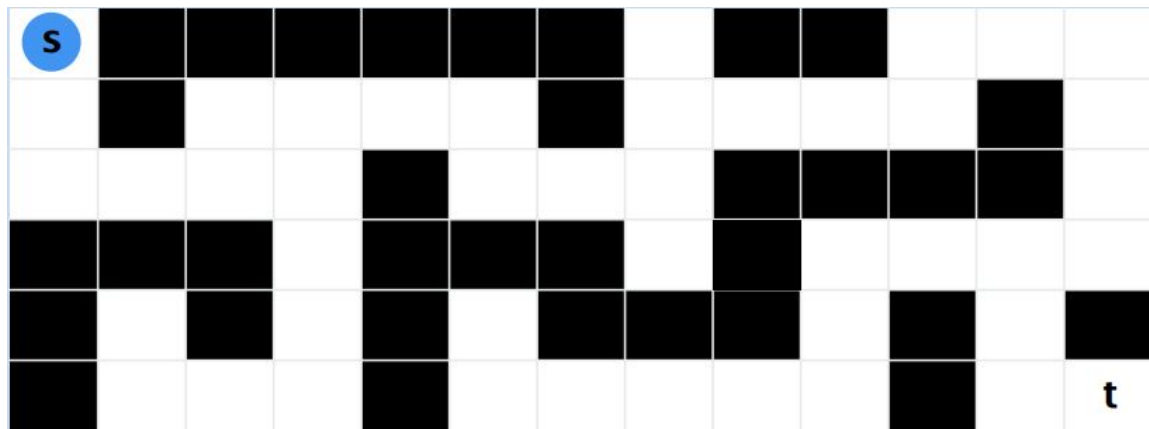
- Step 1: Create the graph



Example Problem

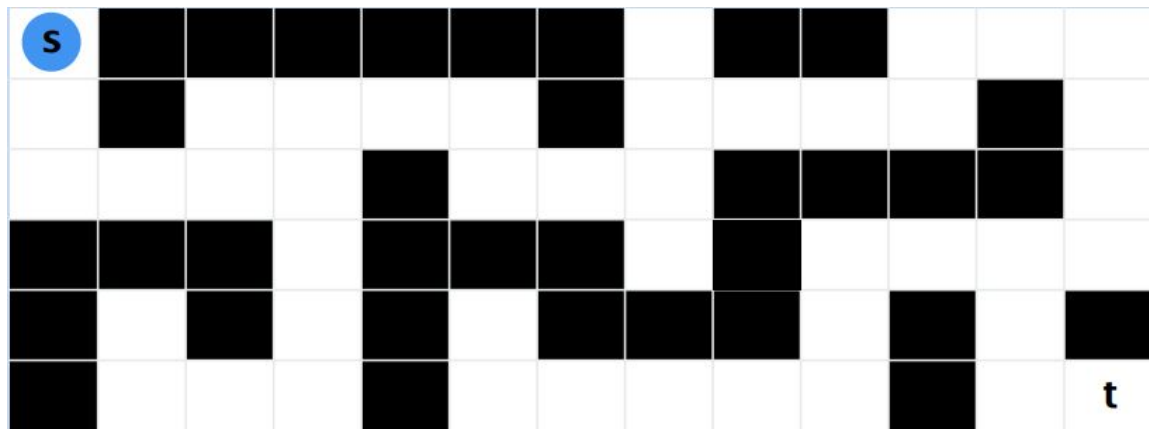
Find the shortest path from s to t .

- Step 1: Create the graph
 - Each vertex is a white square in the grid. Each edge connects two adjacent squares.
 - Step 2: Adjust the graph
 - Unnecessary here.
 - Step 3: Run an algorithm.
 - Run BFS
 - Step 4:
 - Return shortest path.
-



Example Problem: Google's foo.bar challenge

Find the shortest path from s to t, but you're allowed to remove one wall.

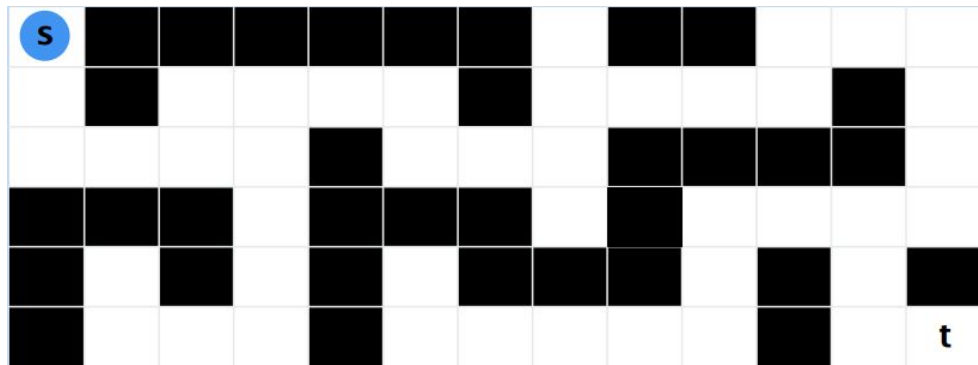


Example Problem: Google's foo.bar challenge

Find the shortest path from *s* to *t*, but you're allowed to remove one wall.

[Here's one, of many, linear time solutions]

- Step 1: Create the graph
 - Each vertex is a white square in the grid. Each edge connects two adjacent squares.
- Step 2: Adjust the graph
 - Add the walls! But make each wall a leaf, so you only get distance from start.
- Step 3: Run an algorithm
 - Run BFS from *s* and from *t*
- Step 4: Use the result
 - Find the wall with lowest sum of BFS values



Takeaway: You should know how to create graphs!

Another big takeaway from this unit is that many problems won't be modeled so neatly as graph problems – you'll have to figure out the graph yourself.

Worksheet:

- These problems are also really good midterm practice ;)

Summary

Today we were introduced to DFS, and saw a surprising application for it beyond just graph traversal: decomposing a graph into its strongly connected components.

This adds to the lesson from Tuesday, that the standard graph algorithms can do much more interesting tasks than they were designed for.

We also learned about pre-ordering and post-ordering, two useful metrics for graph traversal that help us understand the graph's structure better.