Graph Theory I

Workshop – Term 1, Week 10

Susan He, Austin Song, Matthew Gu

Outline

1. **Housekeeping**
2. **Roads, cities, wires, and kidneys**
3. **A graph in code?**
4. **Exploration!**
5. **Shortest paths**
6. **Funny tricks**
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. **Challenge problems >:)**
8. **Appendix and resources**

Attendance!

Please scan the attendance form so you can get free food



VJudge For Workshop

If you're confident with the content for today, feel free to get started on problems!

Link: https://vjudge.net/contest/806361

Outline

1. Housekeeping
2. **Roads, cities, wires, and kidneys**
3. A graph in code?
4. Exploration!
5. Shortest paths
6. Funny tricks
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. Challenge problems >:)
8. Appendix and resources

The Basics:

Definitions

The Basics:

Definitions

A graph is made up of nodes and edges:

The Basics:

Definitions

A graph is made up of nodes and edges:

- **Nodes** (aka vertices) represent entities in the graph

The Basics:

Definitions

A graph is made up of nodes and edges:

- **Nodes** (aka vertices) represent entities in the graph
- **Edges** represent the relationships between entities

The Basics:

Definitions

A graph is made up of nodes and edges:

- **Nodes** (aka vertices) represent entities in the graph
- **Edges** represent the relationships between entities
  - Can be directed (one way relationship) or undirected (mutual relationship)

The Basics:

Definitions

A graph is made up of nodes and edges:

- **Nodes** (aka vertices) represent entities in the graph
- **Edges** represent the relationships between entities
  - Can be directed (one way relationship) or undirected (mutual relationship)
  - On weighted graphs, each edge has some cost

The Basics:

Definitions

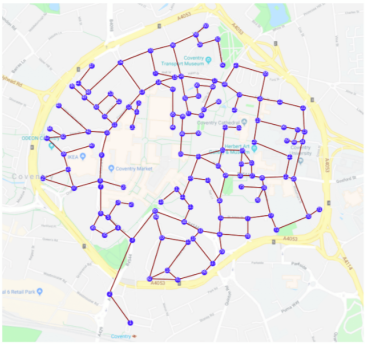
A graph is made up of nodes and edges:

- **Nodes** (aka vertices) represent entities in the graph
- **Edges** represent the relationships between entities
  - Can be directed (one way relationship) or undirected (mutual relationship)
  - On weighted graphs, each edge has some cost

Examples:

- Cities connected by roads
- Game states
- Nervous system

Examples



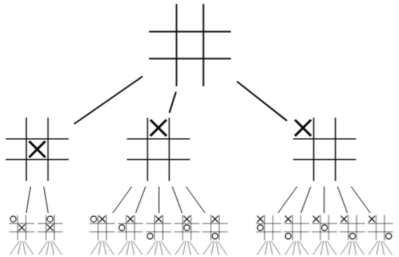


Figure: Graphs

The Basics:

More definitions

- Two nodes are **neighbours** (or adjacent) if there is at least one edge connecting them
- A **path** on a graph is a sequence of nodes, where each pair of adjacent nodes are neighbours
- A cycle is a path that starts and ends at the same node

Paths

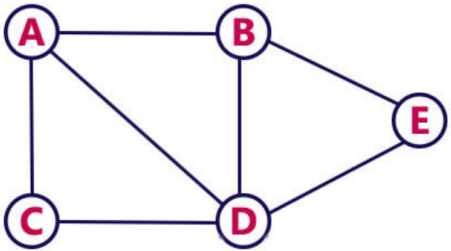


Figure: Simple unweighted, undirected graph

- A, D, E, B is a valid path
- D, B, E, D is a cycle
- A, E, B, D is not a valid path

Outline

1. Housekeeping
2. Roads, cities, wires, and kidneys
3. **A graph in code?**
4. Exploration!
5. Shortest paths
6. Funny tricks
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. Challenge problems >:)
8. Appendix and resources

Representing Graphs

How do we store a graph in code? There are three main ways:

Representing Graphs

How do we store a graph in code? There are three main ways:

1. Adjacency list
   - For every node, we simply store its direct neighbours.
   - Very common representation method

Representing Graphs

How do we store a graph in code? There are three main ways:

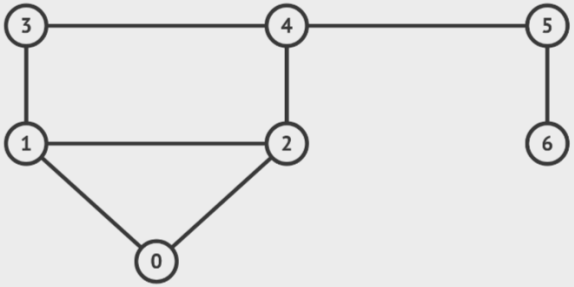
1. Adjacency list
   - For every node, we simply store its direct neighbours.
   - Very common representation method
2. Adjacency matrix
   - The value of cell $A_{u,v}$ is 1 if there is a path from node u to v.
   - If the graph is weighted, the cell can store the weight instead.

Representing Graphs

How do we store a graph in code? There are three main ways:

1. Adjacency list
   - For every node, we simply store its direct neighbours.
   - Very common representation method
2. Adjacency matrix
   - The value of cell $A_{u,v}$ is 1 if there is a path from node u to v.
   - If the graph is weighted, the cell can store the weight instead.
3. Edge list
   - List of all edges. Not too useful for us.

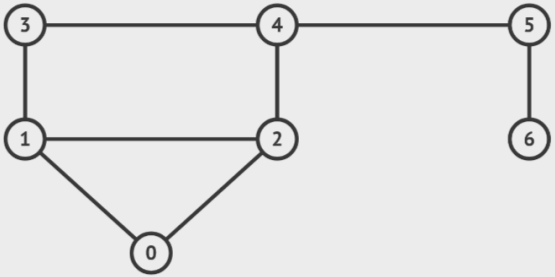
Adjacency List



Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Courtesy of visualalgo.net

Adjacency Matrix



Adjacency Matrix							
	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Courtesy of visualalgo.net

Representing Graphs

Which method should we choose?

	Adjacency Matrix	Adjacency List
Memory	$\|V\|^2$	$\|V\| + \|E\|$
Edge lookup	$O(1)$	$O(E)$
Loop through all neighbours	$O(V)$	$O(E)$

In most beginner graph problems, the adjacency list is enough!

Representing Graphs

Which method should we choose?

	Adjacency Matrix	Adjacency List
Memory	$\|V\|^2$	$\|V\| + \|E\|$
Edge lookup	$O(1)$	$O(E)$
Loop through all neighbours	$O(V)$	$O(E)$

In most beginner graph problems, the adjacency list is enough!
Sometimes we have implicit graphs, e.g. a grid problem.

Outline

1. Housekeeping
2. Roads, cities, wires, and kidneys
3. A graph in code?
4. **Exploration!**
5. Shortest paths
6. Funny tricks
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. Challenge problems >:)
8. Appendix and resources

Connectivity

Connectivity

Definitions

A node u is **connected** to a node v if and only if one can take a sequence of edges from u to v.

Connectivity

Definitions

A node u is **connected** to a node v if and only if one can take a sequence of edges from u to v.

In an **undirected graph**, a maximal set of pairwise connected nodes is referred to as a **connected component**.

Connectivity

Definitions

A node u is **connected** to a node v if and only if one can take a sequence of edges from u to v.

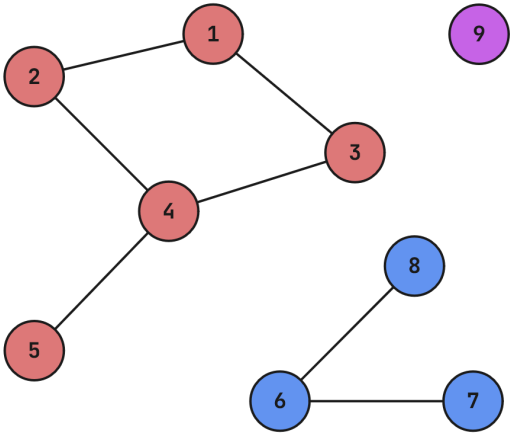
In an **undirected graph**, a maximal set of pairwise connected nodes is referred to as a **connected component**.

Making Friends I

There are N people. M pairs of these people are friends.

Rumors spread between friends... if person 1 has a rumor, to whom will it spread?

Making Friends – Example



Making Friends – Discussion

It's not sufficient to just consider adjacencies. We need to find whether a node is reachable from another.

- We're trying to find the nodes which are in the same connected component as 1.

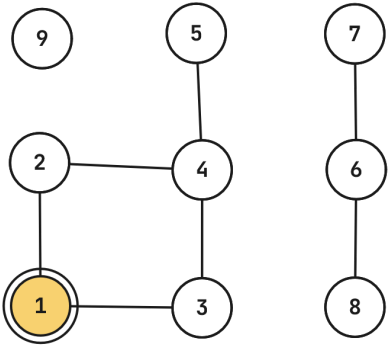
How can we do this?

Traversing Our Graph

We can simulate how our rumor spreads!

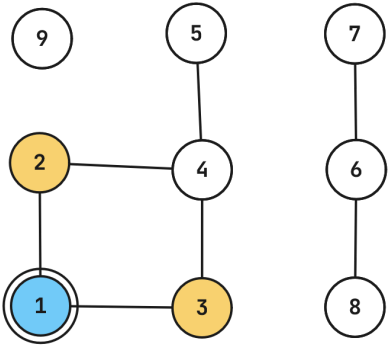
Traversing Our Graph

We can simulate how our rumor spreads!



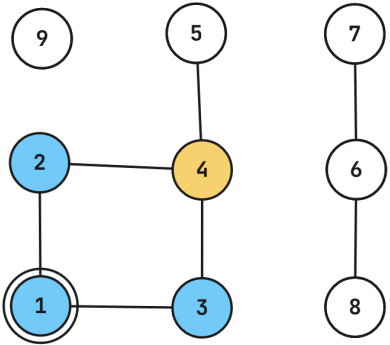
Traversing Our Graph

We can simulate how our rumor spreads!



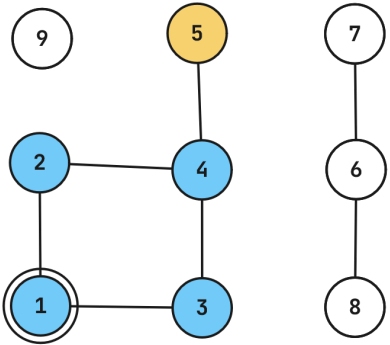
Traversing Our Graph

We can simulate how our rumor spreads!



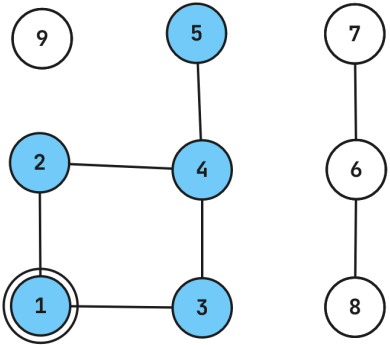
Traversing Our Graph

We can simulate how our rumor spreads!



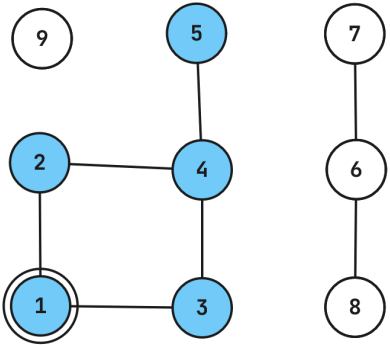
Traversing Our Graph

We can simulate how our rumor spreads!



Traversing Our Graph

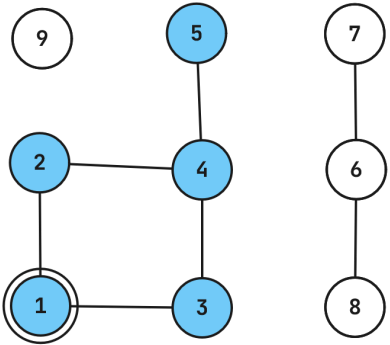
We can simulate how our rumor spreads!



We visit all nodes at the same distance together.

Traversing Our Graph

We can simulate how our rumor spreads!



We visit all nodes at the same distance together.

Since we explore outward layer by layer, this is called **Breadth-First Search**.

BFS

We start at some starting node $s = 1$.

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

- We **visit** all the nodes $v \in Q$ one by one.

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

- We **visit** all the nodes $v \in Q$ one by one.
- We extend the boundary by **peeking** at any **unpeeked** neighbours of v, placing them in some new Q'.

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

- We **visit** all the nodes $v \in Q$ one by one.
- We extend the boundary by **peeking** at any **unpeeked** neighbours of v, placing them in some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

- We **visit** all the nodes $v \in Q$ one by one.
- We extend the boundary by **peeking** at any **unpeeked** neighbours of v, placing them in some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.
- We keep going until Q' is empty.

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

- We **visit** all the nodes $v \in Q$ one by one.
- We extend the boundary by **peeking** at any **unpeeked** neighbours of v, placing them in some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.
- We keep going until Q' is empty.

Question

What's the time complexity of this implementation of BFS?

BFS

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer:

- We **visit** all the nodes $v \in Q$ one by one.
- We extend the boundary by **peeking** at any **unpeeked** neighbours of v, placing them in some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.
- We keep going until Q' is empty.

Question

What's the time complexity of this implementation of BFS?

We process (visit) at most N nodes. At each node, we peek at all it's neighbours. Over the entire traversal, we peek at $O(2M)$ nodes. It's $O(N + M)$!

BFS and Queues

The way BFS is typically implemented is with a **queue**.

BFS and Queues

The way BFS is typically implemented is with a **queue**.

Here, elements of Q' are directly appended to the **end** of Q, while elements of Q are popped off from the **front**.

BFS and Queues

The way BFS is typically implemented is with a **queue**.

Here, elements of Q' are directly appended to the **end** of Q, while elements of Q are popped off from the **front**.

This gives the exact same time complexity, but it's much easier to do!

I'll demonstrate it now!

Properties of BFS

BFS has one key property that makes it useful:

Properties of BFS

BFS has one key property that makes it useful:

- Consider all the nodes $\{u_d\}$ which have distance d from s.

Properties of BFS

BFS has one key property that makes it useful:

- Consider all the nodes $\{u_d\}$ which have distance d from s.
- Because we visit nodes layer-by-layer, they're all visited before $\{u_{d+1}\}$.

Properties of BFS

BFS has one key property that makes it useful:

- Consider all the nodes $\{u_d\}$ which have distance d from s.
- Because we visit nodes layer-by-layer, they're all visited before $\{u_{d+1}\}$.
- This means the first time we visit a node is the **earliest** we'll ever get to it!

Properties of BFS

BFS has one key property that makes it useful:

- Consider all the nodes $\{u_d\}$ which have distance d from s.
- Because we visit nodes layer-by-layer, they're all visited before $\{u_{d+1}\}$.
- This means the first time we visit a node is the **earliest** we'll ever get to it!

BFS essentially greedily chooses all the nodes in reach, since it knows no path can beat it.

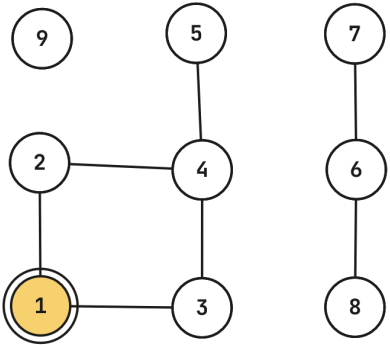
Keep this property in mind...

Traversing Our Graph II

But there's another strategy we can use.

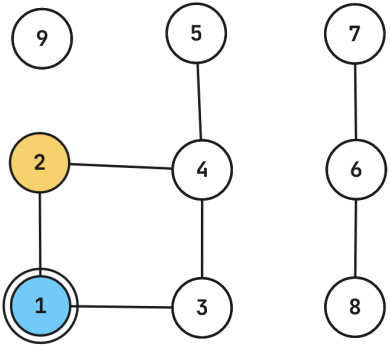
Traversing Our Graph II

But there's another strategy we can use.



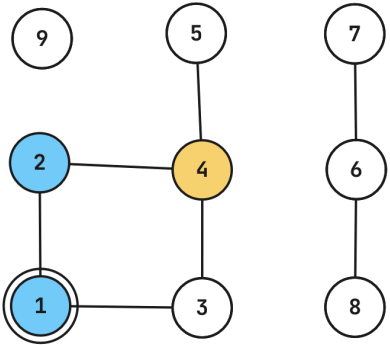
Traversing Our Graph II

But there's another strategy we can use.



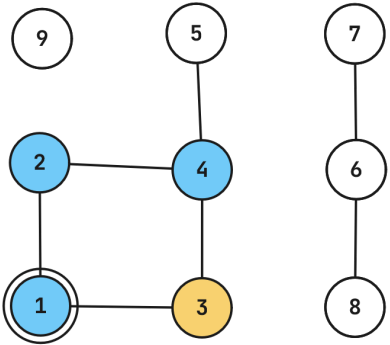
Traversing Our Graph II

But there's another strategy we can use.



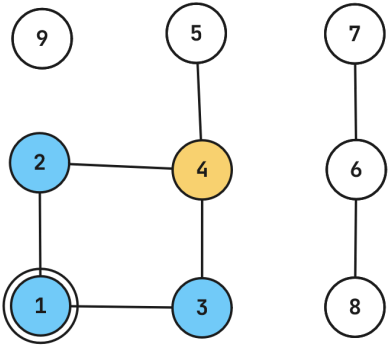
Traversing Our Graph II

But there's another strategy we can use.



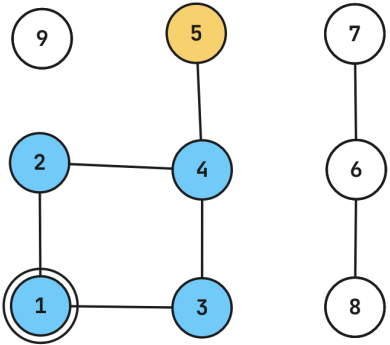
Traversing Our Graph II

But there's another strategy we can use.



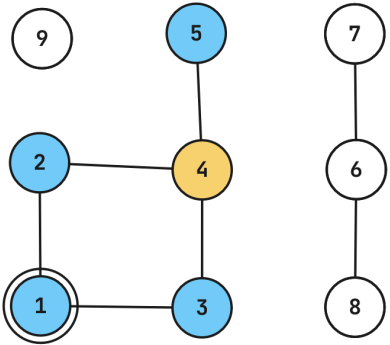
Traversing Our Graph II

But there's another strategy we can use.



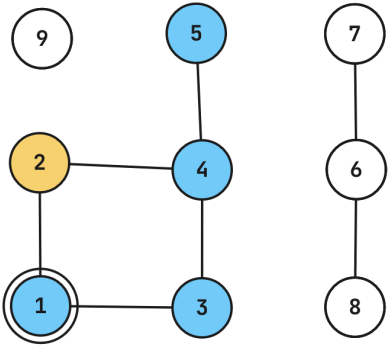
Traversing Our Graph II

But there's another strategy we can use.



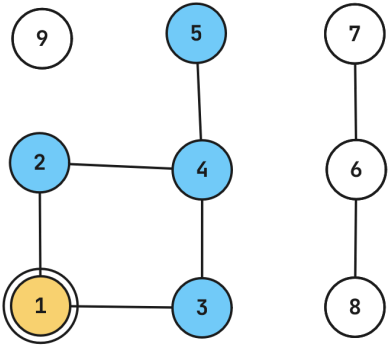
Traversing Our Graph II

But there's another strategy we can use.



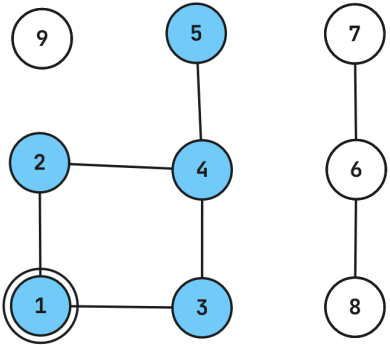
Traversing Our Graph II

But there's another strategy we can use.



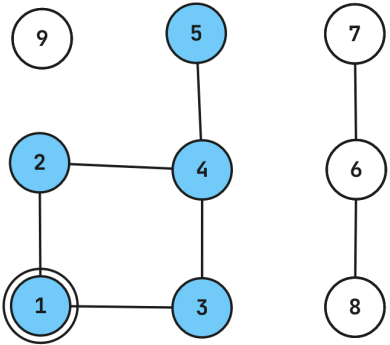
Traversing Our Graph II

But there's another strategy we can use.



Traversing Our Graph II

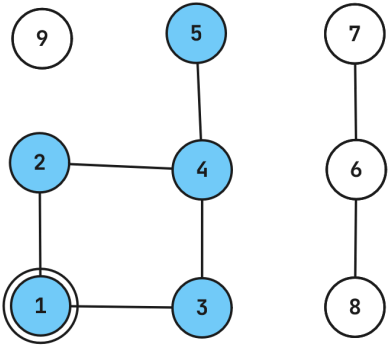
But there's another strategy we can use.



We're walking the graph like a maze, going to the very end before earlier branching routes.

Traversing Our Graph II

But there's another strategy we can use.



We're walking the graph like a maze, going to the very end before earlier branching routes.

Since we explore to the end, this is called **Depth-First Search**.

DFS

We start at some starting node $s = 1$.

DFS

We start at some starting node $s = 1$.

As we process our current node u:

DFS

We start at some starting node $s = 1$.

As we process our current node u:

- We mark it as visited (in processing).

DFS

We start at some starting node $s = 1$.

As we process our current node u:

- We mark it as visited (in processing).
- We visit some unvisited neighbour v of u by recursively entering v.

DFS

We start at some starting node $s = 1$.

As we process our current node u:

- We mark it as visited (in processing).
- We visit some unvisited neighbour v of u by recursively entering v.
- When we return from v, we continue dealing with the remaining neighbours of u.

DFS

We start at some starting node $s = 1$.

As we process our current node u:

- We mark it as visited (in processing).
- We visit some unvisited neighbour v of u by recursively entering v.
- When we return from v, we continue dealing with the remaining neighbours of u.
- Once everything adjacent to u is visited, we exit from u.

DFS

We start at some starting node $s = 1$.

As we process our current node u:

- We mark it as visited (in processing).
- We visit some unvisited neighbour v of u by recursively entering v.
- When we return from v, we continue dealing with the remaining neighbours of u.
- Once everything adjacent to u is visited, we exit from u.

Question

What's the time complexity of this implementation of DFS?

DFS

We start at some starting node $s = 1$.

As we process our current node u:

- We mark it as visited (in processing).
- We visit some unvisited neighbour v of u by recursively entering v.
- When we return from v, we continue dealing with the remaining neighbours of u.
- Once everything adjacent to u is visited, we exit from u.

Question

What's the time complexity of this implementation of DFS?

We process (visit) at most N nodes. We traverse each edge at most twice (forward when visiting, backwards when returning). It's $O(N + M)$ again!

Properties of DFS

DFS has two key properties that make it useful:

Properties of DFS

DFS has two key properties that make it useful:

- It's really fast to implement if you don't really care about the order you visit things.

Properties of DFS

DFS has two key properties that make it useful:

- It's really fast to implement if you don't really care about the order you visit things.
- There's an interesting property regarding the spanning tree formed by a DFS traveral – we won't go through this today.

Properties of DFS

DFS has two key properties that make it useful:

- It's really fast to implement if you don't really care about the order you visit things.
- There's an interesting property regarding the spanning tree formed by a DFS traveral – we won't go through this today.

I'll showcase the first property now :)

Outline

1. Housekeeping
2. Roads, cities, wires, and kidneys
3. A graph in code?
4. Exploration!
5. **Shortest paths**
6. Funny tricks
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. Challenge problems >:)
8. Appendix and resources

Unweighted Graphs

We already know how to find the shortest path between a and b on an unweighted graph!

Unweighted Graphs

We already know how to find the shortest path between a and b on an unweighted graph!

We'll be looking at weighted graphs in this section.

Unweighted Graphs

We already know how to find the shortest path between a and b on an unweighted graph!

We'll be looking at weighted graphs in this section. Notably, all weights will be:

- non-equal.
- non-negative.

Minimum Cost Paths

Minimum Cost Paths

Rob the Race-ist II

Minimum Cost Paths

Rob the Race-ist II

Once again, Rob is exercising his favourite hobby – racing!

Minimum Cost Paths

Rob the Race-ist II

Once again, Rob is exercising his favourite hobby – racing!

This time, he's picked a racecourse with N intersections, and M bidirectional paths between them. His job is to start at 1, and make his way to N.

Minimum Cost Paths

Rob the Race-ist II

Once again, Rob is exercising his favourite hobby – racing!

This time, he's picked a racecourse with N intersections, and M bidirectional paths between them. His job is to start at 1, and make his way to N.

Unfortunately, Rob also happens to be exercising his second favourite hobby. The i'th path contains some number of *undesirables*, w_i. If Rob runs along path i, he'll encounter all w_i 'people'.

Minimum Cost Paths

Rob the Race-ist II

Once again, Rob is exercising his favourite hobby – racing!

This time, he's picked a racecourse with N intersections, and M bidirectional paths between them. His job is to start at 1, and make his way to N.

Unfortunately, Rob also happens to be exercising his second favourite hobby. The i'th path contains some number of *undesirables*, w_i. If Rob runs along path i, he'll encounter all w_i 'people'.

Help Rob find the minimum number of undesirables he needs to encounter on his race!

Minimum Cost Paths

Rob the Race-ist II

Once again, Rob is exercising his favourite hobby – racing!

This time, he's picked a racecourse with N intersections, and M bidirectional paths between them. His job is to start at 1, and make his way to N.

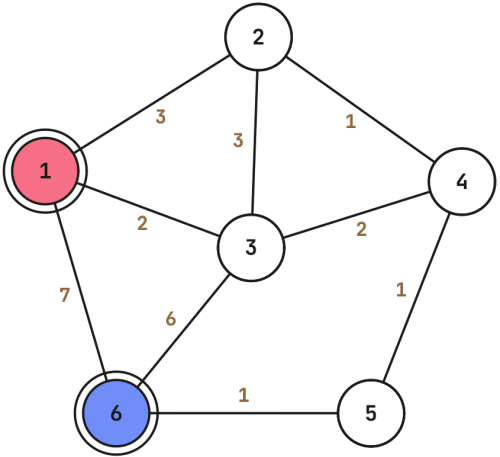
Unfortunately, Rob also happens to be exercising his second favourite hobby. The i'th path contains some number of *undesirables*, w_i. If Rob runs along path i, he'll encounter all w_i 'people'.

Help Rob find the minimum number of undesirables he needs to encounter on his race!

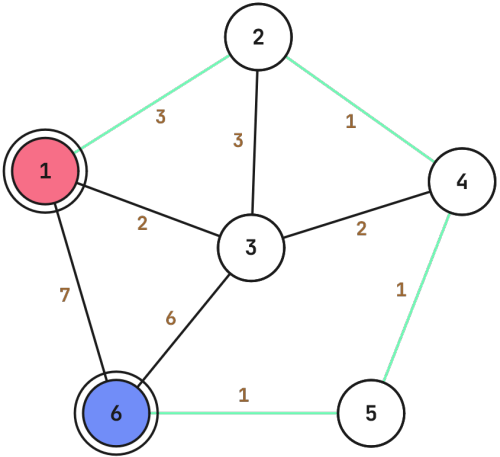
Constraints:

- $1 \leq N, M \leq 2 \times 10^5$, $0 \leq w_i \leq 10^9$.

Rob the Race-ist II – Example



Rob the Race-ist II – Example



BFS II

We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

BFS II

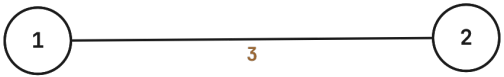
We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

- For each edge with weight w_i, replace it with $w_i - 1$ intermediary nodes connected by edges of weight 1.

BFS II

We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

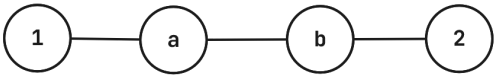
- For each edge with weight w_i, replace it with $w_i - 1$ intermediary nodes connected by edges of weight 1.



BFS II

We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

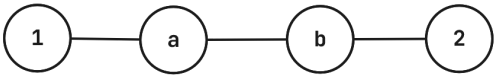
- For each edge with weight w_i, replace it with $w_i - 1$ intermediary nodes connected by edges of weight 1.



BFS II

We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

- For each edge with weight w_i, replace it with $w_i - 1$ intermediary nodes connected by edges of weight 1.

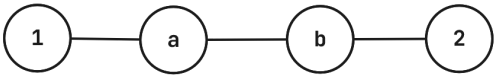


Now that our graph is unweighted, we can run BFS on it!

BFS II

We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

- For each edge with weight w_i, replace it with $w_i - 1$ intermediary nodes connected by edges of weight 1.



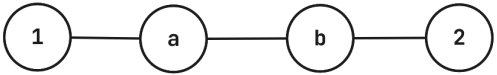
Now that our graph is unweighted, we can run BFS on it!

Exercise: What's the time complexity?

BFS II

We know how to solve the problem when the graph is unweighted – let's try and construct an unweighted graph from this!

- For each edge with weight w_i, replace it with $w_i - 1$ intermediary nodes connected by edges of weight 1.



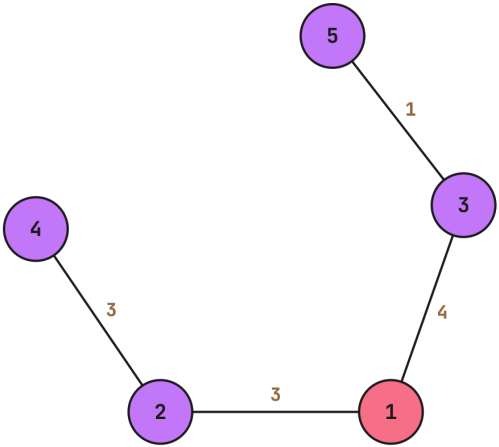
Now that our graph is unweighted, we can run BFS on it!

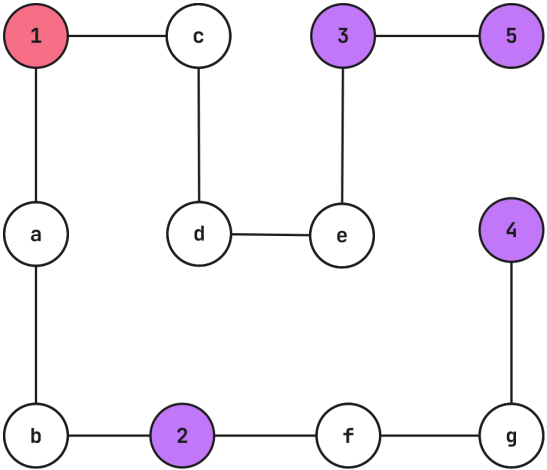
Exercise: What's the time complexity?

Solution: There are now $10^9 M$ nodes and edges in the worst case. It's $O(10^9 M)$:(

Speeding It Up

Consider a graph and its expanded version:





Speeding It Up

We only care about the order which the purple vertices (ones that actually exist) are visited.

What is this order?

Speeding It Up

We only care about the order which the purple vertices (ones that actually exist) are visited.

What is this order?

- Remember that our 'BFS' visits all the nodes with distance d before those with distance $d+1$.

Speeding It Up

We only care about the order which the purple vertices (ones that actually exist) are visited.

What is this order?

- Remember that our 'BFS' visits all the nodes with distance d before those with distance $d+1$.
- So the current "closest node" will always be visited first.
  - i.e. we visit greedily, based on distance so far from our source s!

Speeding It Up

We only care about the order which the purple vertices (ones that actually exist) are visited.

What is this order?

- Remember that our 'BFS' visits all the nodes with distance d before those with distance $d + 1$.
- So the current "closest node" will always be visited first.
  - i.e. we visit greedily, based on distance so far from our source s!

At each point, we simply pick the closest node found so far to process.

- We're just accelerating the process of BFS to only consider the interesting points.

BFS++

We start at some starting node $s = 1$.

BFS++

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer, and additionally maintain the best distance for each node:

BFS++

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer, and additionally maintain the best distance for each node:

- We **visit** all the nodes $v \in Q$ one by one, picking the smallest distance node first.

BFS++

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer, and additionally maintain the best distance for each node:

- We **visit** all the nodes $v \in Q$ one by one, picking the smallest distance node first.
- We extend the boundary by **peeking** at any **unvisited** neighbours of v:
  - If we found a better distance for it, update the distance.
  - Else, insert it into some new Q'.

BFS++

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer, and additionally maintain the best distance for each node:

- We **visit** all the nodes $v \in Q$ one by one, picking the smallest distance node first.
- We extend the boundary by **peeking** at any **unvisited** neighbours of v:
  - If we found a better distance for it, update the distance.
  - Else, insert it into some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.

BFS++

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer, and additionally maintain the best distance for each node:

- We **visit** all the nodes $v \in Q$ one by one, picking the smallest distance node first.
- We extend the boundary by **peeking** at any **unvisited** neighbours of v:
  - If we found a better distance for it, update the distance.
  - Else, insert it into some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.
- We keep going until Q' is empty.

BFS++

We start at some starting node $s = 1$.

We keep a set Q of nodes in our current layer, and additionally maintain the best distance for each node:

- We **visit** all the nodes $v \in Q$ one by one, picking the smallest distance node first.
- We extend the boundary by **peeking** at any **unvisited** neighbours of v:
  - If we found a better distance for it, update the distance.
  - Else, insert it into some new Q'.
- After everything in Q has been **visited**, replace Q with Q'.
- We keep going until Q' is empty.

Question

What's the time complexity of this algorithm?

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

- A priority queue can be thought of as a queue which maintains its elements in sorted order.
- Insertion of an element and removal of the minimum element are both $O(\log N)$.

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

- A priority queue can be thought of as a queue which maintains its elements in sorted order.
- Insertion of an element and removal of the minimum element are both $O(\log N)$.

Instead of searching through Q manually, replace it with a priority queue!

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

- A priority queue can be thought of as a queue which maintains its elements in sorted order.
- Insertion of an element and removal of the minimum element are both $O(\log N)$.

Instead of searching through Q manually, replace it with a priority queue!

- We store a pair $(d[u], u)$ for each node $u \in Q$.

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

- A priority queue can be thought of as a queue which maintains its elements in sorted order.
- Insertion of an element and removal of the minimum element are both $O(\log N)$.

Instead of searching through Q manually, replace it with a priority queue!

- We store a pair $(d[u], u)$ for each node $u \in Q$.
- Whenever we update the distance from $d[u]$ to $d'[u]$, we push $(d'[u], u)$ into Q.

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

- A priority queue can be thought of as a queue which maintains its elements in sorted order.
- Insertion of an element and removal of the minimum element are both $O(\log N)$.

Instead of searching through Q manually, replace it with a priority queue!

- We store a pair $(d[u], u)$ for each node $u \in Q$.
- Whenever we update the distance from $d[u]$ to $d'[u]$, we push $(d'[u], u)$ into Q.
- Since $d'[u] < d[u]$, we'll always process the newest (most updated) version of u first.

Dijkstra and Priority Queues

Spoiler: this algorithm is called **Dijkstra's Algorithm** :)

The time complexity of Dijkstra's algorithm depends on how quickly we can find the minimum distance node in Q. We often use a **heap** or **priority queue** for this!

- A priority queue can be thought of as a queue which maintains its elements in sorted order.
- Insertion of an element and removal of the minimum element are both $O(\log N)$.

Instead of searching through Q manually, replace it with a priority queue!

- We store a pair $(d[u], u)$ for each node $u \in Q$.
- Whenever we update the distance from $d[u]$ to $d'[u]$, we push $(d'[u], u)$ into Q.
- Since $d'[u] < d[u]$, we'll always process the newest (most updated) version of u first.
- If we extract a node we've already **visited**, we skip over it.

This is $O((N + M) \log N)$. Time to implement!

Dijkstra's Shortcomings

Remember how we derived Dijkstra's algorithm. There's one big issue:

Dijkstra's Shortcomings

Remember how we derived Dijkstra's algorithm. There's one big issue:

- $w_i < 0$: In this case, we cannot 'construct' our BFS representation.

Dijkstra's Shortcomings

Remember how we derived Dijkstra's algorithm. There's one big issue:

- $w_i < 0$: In this case, we cannot 'construct' our BFS representation.
  - Specifically, the greedy claim fails. The closest node's distance isn't always the best.
  - So, we have to continually update all our nodes (even the ones 'visited').

Dijkstra's Shortcomings

Remember how we derived Dijkstra's algorithm. There's one big issue:

- $w_i < 0$: In this case, we cannot 'construct' our BFS representation.
  - Specifically, the greedy claim fails. The closest node's distance isn't always the best.
  - So, we have to continually update all our nodes (even the ones 'visited').

Exercise: Can you think of potential other issues with the algorithm correctness?

Outline

1. Housekeeping
2. Roads, cities, wires, and kidneys
3. A graph in code?
4. Exploration!
5. Shortest paths
6. **Funny tricks**
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. Challenge problems >:)
8. Appendix and resources

Multisource – Problem

Multisource – Problem

Crossing the River

This problem was adapted from a recent COMP2521 supplementary exam.

Multisource – Problem

Crossing the River

This problem was adapted from a recent COMP2521 supplementary exam.

You're attempting to cross a river, modelled as a grid of $N \times N$ cells. You can start anywhere along the upstream shore (cells in row 1) and you must finish anywhere along the downstream shore (cells in row N).

Multisource – Problem

Crossing the River

This problem was adapted from a recent COMP2521 supplementary exam.

You're attempting to cross a river, modelled as a grid of $N \times N$ cells. You can start anywhere along the upstream shore (cells in row 1) and you must finish anywhere along the downstream shore (cells in row N).

You are allowed to move in any of the four cardinal directions. It takes one second to move from a cell to an adjacent one.

Multisource – Problem

Crossing the River

This problem was adapted from a recent COMP2521 supplementary exam.

You're attempting to cross a river, modelled as a grid of $N \times N$ cells. You can start anywhere along the upstream shore (cells in row 1) and you must finish anywhere along the downstream shore (cells in row N).

You are allowed to move in any of the four cardinal directions. It takes one second to move from a cell to an adjacent one.

To make the problem marginally more challenging, there are K blocked cells which you cannot pass through, and cannot start on.

Multisource – Problem

Crossing the River

This problem was adapted from a recent COMP2521 supplementary exam.

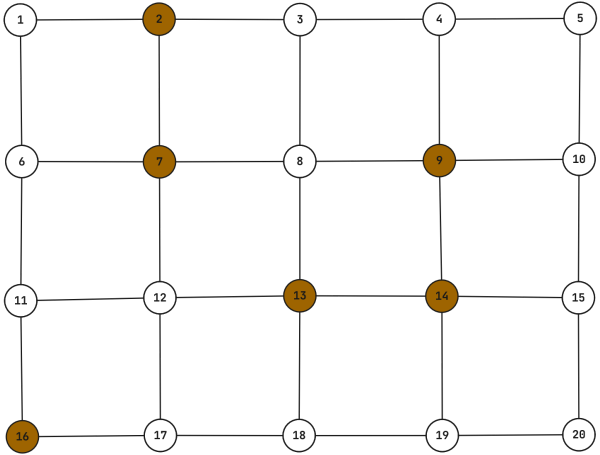
You're attempting to cross a river, modelled as a grid of $N \times N$ cells. You can start anywhere along the upstream shore (cells in row 1) and you must finish anywhere along the downstream shore (cells in row N).

You are allowed to move in any of the four cardinal directions. It takes one second to move from a cell to an adjacent one.

To make the problem marginally more challenging, there are K blocked cells which you cannot pass through, and cannot start on.

Given this map of the river, determine the shortest time to cross it.

Multisource – Example



Multisource – Solution

If there was a single point s to start at, we could just BFS from there. But what do we do when there's multiple source vertices?

Multisource – Solution

If there was a single point s to start at, we could just BFS from there. But what do we do when there's multiple source vertices?

Construct a dummy node 0 which connects to all the nodes on the top row with weight 0.

Multisource – Solution

If there was a single point s to start at, we could just BFS from there. But what do we do when there's multiple source vertices?

Construct a dummy node 0 which connects to all the nodes on the top row with weight 0.

- Equivalently, we can just add all the top row vertices to our BFS queue to begin with.

Multisource – Solution

If there was a single point s to start at, we could just BFS from there. But what do we do when there's multiple source vertices?

Construct a dummy node 0 which connects to all the nodes on the top row with weight 0.

- Equivalently, we can just add all the top row vertices to our BFS queue to begin with.

Then, the answer is just the first time that we hit a vertex on the bottom!

This trick works for general graphs too; not just grids!

0-1 BFS – Problem

Crossing the River II

Now, each non-blocked cell also has a current, pointing in one of the four cardinal directions.

To travel in this current direction takes no time at all for you! You can still fight the current to take any other direction in one second.

Now what's the shortest time to cross from one side to the other?

0-1 BFS – Solution

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

$$Q = \underbrace{v}_{d[v]}, \ldots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \cdots \underbrace{n}_{d[v]+1}$$

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

$$Q = \underbrace{v}_{d[v]}, \ldots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \cdots \underbrace{n}_{d[v]+1}$$

- In this case, since the structure is so simple, we can just use a **deque**!

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

$$Q = \underbrace{v}_{d[v]}, \ldots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \cdots \underbrace{n}_{d[v]+1}$$

- In this case, since the structure is so simple, we can just use a **deque**!
  - When we peek at u through an edge of 0, we insert u at the front.

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

$$Q = \underbrace{v}_{d[v]}, \ldots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \cdots \underbrace{n}_{d[v]+1}$$

- In this case, since the structure is so simple, we can just use a **deque**!
  - When we peek at u through an edge of 0, we insert u at the front.
  - Else, we insert u at the back.

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

$$Q = \underbrace{v}_{d[v]}, \ldots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \cdots \underbrace{n}_{d[v]+1}$$

- In this case, since the structure is so simple, we can just use a **deque**!
  - When we peek at u through an edge of 0, we insert u at the front.
  - Else, we insert u at the back.
  - Both these operations are $O(1)$ with a deque!

0-1 BFS – Solution

Since the weights are not equal, we can just use Dijkstra's algorithm!

- This gives us a time complexity of $O(N^2 \log N)$.

But what if we can't afford the log factor?

- Notice that each edge weight is either 0 or 1.
- What does the priority queue look like in this instance? Well...

$$Q = \underbrace{v}_{d[v]}, \ldots, \underbrace{u}_{d[v]}, \underbrace{m}_{d[v]+1} \cdots \underbrace{n}_{d[v]+1}$$

- In this case, since the structure is so simple, we can just use a **deque**!
  - When we peek at u through an edge of 0, we insert u at the front.
  - Else, we insert u at the back.
  - Both these operations are $O(1)$ with a deque!
- Alternatively, you can think of it as just chucking u right back into Q, instead of Q'.

Dijkstra++

Minimise Maximum Miseries

Dijkstra++

Minimise Maximum Miseries

Miserable Momo must make a mandatory midnight march from mansion 1 to mansion N through a misty metropolis, modelled as N mansions and M meandering motorways. Each motorway has a non-negative **misery measure**.

Dijkstra++

Minimise Maximum Miseries

Miserable Momo must make a mandatory midnight march from mansion 1 to mansion N through a misty metropolis, modelled as N mansions and M meandering motorways. Each motorway has a non-negative **misery measure**.

When Momo moves along a motorway, the motorway's misery measure melds into the march's misery. Momo is not measuring the mere sum of miseries, minding only the most monstrous moment: the maximum misery measure 'mongst motorways marched on.

Dijkstra++

Minimise Maximum Miseries

Miserable Momo must make a mandatory midnight march from mansion 1 to mansion N through a misty metropolis, modelled as N mansions and M meandering motorways. Each motorway has a non-negative **misery measure**.

When Momo moves along a motorway, the motorway's misery measure melds into the march's misery. Momo is not measuring the mere sum of miseries, minding only the most monstrous moment: the maximum misery measure 'mongst motorways marched on.

Momo's mission must make the march's maximum misery as minimal as mathematically manageable.

Dijkstra++ – Solution

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Why?

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Why?

- The greedy claim underpinning Dijkstra assumes the **smallest distance node** at any point is accurate and the best place to expand (thus visiting it and cementing its value).

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Why?

- The greedy claim underpinning Dijkstra assumes the **smallest distance node** at any point is accurate and the best place to expand (thus visiting it and cementing its value).
- This is still true, even for this cost function!

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Why?

- The greedy claim underpinning Dijkstra assumes the **smallest distance node** at any point is accurate and the best place to expand (thus visiting it and cementing its value).
- This is still true, even for this cost function!
  - The `max` function keeps the path cost monotonically increasing.

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Why?

- The greedy claim underpinning Dijkstra assumes the **smallest distance node** at any point is accurate and the best place to expand (thus visiting it and cementing its value).
- This is still true, even for this cost function!
  - The `max` function keeps the path cost monotonically increasing.
  - So, no path found in the future can beat the current best distance node (every other node in the queue has a cost at least as bad as your current best!)

Dijkstra++ – Solution

Perhaps surprisingly, we can simply use Dijkstra as usual, replacing

$$d[v] = d[u] + w_{uv}$$

with

$$d[v] = \max(d[u], w_{uv}).$$

Why?

- The greedy claim underpinning Dijkstra assumes the **smallest distance node** at any point is accurate and the best place to expand (thus visiting it and cementing its value).
- This is still true, even for this cost function!
  - The `max` function keeps the path cost monotonically increasing.
  - So, no path found in the future can beat the current best distance node (every other node in the queue has a cost at least as bad as your current best!)

This trick even extends to more complex cost functions (e.g. delays/windows at nodes, etc).

Outline

1. Housekeeping
2. Roads, cities, wires, and kidneys
3. A graph in code?
4. Exploration!
5. Shortest paths
6. Funny tricks
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. **Challenge problems >:)**
8. Appendix and resources

Problem A

Down Under

Problem A

Down Under

"I come from a land down under."

Problem A

Down Under

"I come from a land down under."

Agartha needs your help!

Problem A

Down Under

"I come from a land down under."

Agartha needs your help!

Despite living in a mythical underground kingdom with UFO technology and an infinite supply of raw milk, the Agarthans still haven't figured out the basics of highway engineering.

Problem A

Down Under

"I come from a land down under."

Agartha needs your help!

Despite living in a mythical underground kingdom with UFO technology and an infinite supply of raw milk, the Agarthans still haven't figured out the basics of highway engineering.

Agartha's road network consists of N sacred chambers, with M bidirectional tunnels connecting them. The high priest has ordered you to ensure that there exists a route between any two chambers.

Problem A

Down Under

"I come from a land down under."

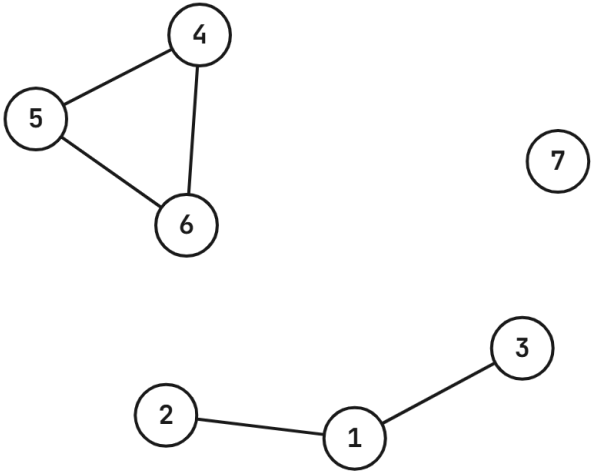
Agartha needs your help!

Despite living in a mythical underground kingdom with UFO technology and an infinite supply of raw milk, the Agarthans still haven't figured out the basics of highway engineering.

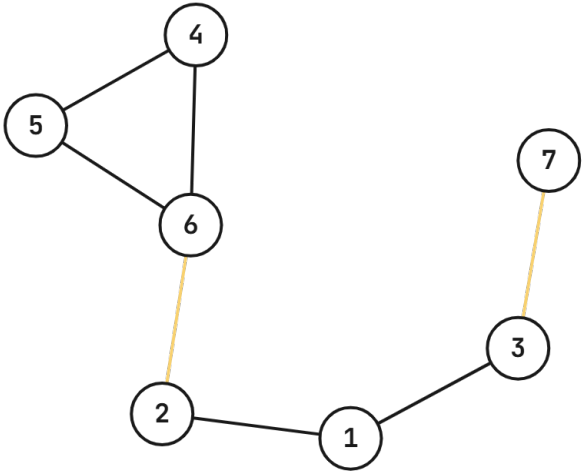
Agartha's road network consists of N sacred chambers, with M bidirectional tunnels connecting them. The high priest has ordered you to ensure that there exists a route between any two chambers.

What's the minimum number of new tunnels you must dig to fulfill his orders?

Visualisation



Visualisation



Solution

What determines the minimum number of edges to add?

Solution

What determines the minimum number of edges to add?

- We can visualise each connected component of our graph as a single node (since everything inside it is connected).

Solution

What determines the minimum number of edges to add?

- We can visualise each connected component of our graph as a single node (since everything inside it is connected).
- This reduces the question to: how many edges are required to connect N nodes?

Solution

What determines the minimum number of edges to add?

- We can visualise each connected component of our graph as a single node (since everything inside it is connected).
- This reduces the question to: how many edges are required to connect N nodes?
- The answer is $N - 1$, the number of nodes in a tree.

Solution

What determines the minimum number of edges to add?

- We can visualise each connected component of our graph as a single node (since everything inside it is connected).
- This reduces the question to: how many edges are required to connect N nodes?
- The answer is $N - 1$, the number of nodes in a tree.
- So, our answer is simply the number of connected components -1.

Solution

What determines the minimum number of edges to add?

- We can visualise each connected component of our graph as a single node (since everything inside it is connected).
- This reduces the question to: how many edges are required to connect N nodes?
- The answer is $N - 1$, the number of nodes in a tree.
- So, our answer is simply the number of connected components -1.

Exercise: How do we count connected components?

Solution

What determines the minimum number of edges to add?

- We can visualise each connected component of our graph as a single node (since everything inside it is connected).
- This reduces the question to: how many edges are required to connect N nodes?
- The answer is $N - 1$, the number of nodes in a tree.
- So, our answer is simply the number of connected components -1.

Exercise: How do we count connected components?

Exercise: What's the final time complexity?

Problem B

Hot N Cold

Problem B

Hot N Cold

"You change your mind
Like a girl changes clothes."

Problem B

Hot N Cold

"You change your mind
Like a girl changes clothes."

is what you would like to tell the high priest of Agartha, after he orders you to completely redesign the road network, once again.

Problem B

Hot N Cold

"You change your mind
Like a girl changes clothes."

is what you would like to tell the high priest of Agartha, after he orders you to completely redesign the road network, once again.

Agartha's road network still consists of N sacred chambers and M tunnels connecting them.

Problem B

Hot N Cold

"You change your mind
Like a girl changes clothes."

is what you would like to tell the high priest of Agartha, after he orders you to completely redesign the road network, once again.

Agartha's road network still consists of N sacred chambers and M tunnels connecting them. However, this time, the high priest has ordered that each tunnel can only be traversed in a single direction (for spiritual reasons, of course).

Problem B

Hot N Cold

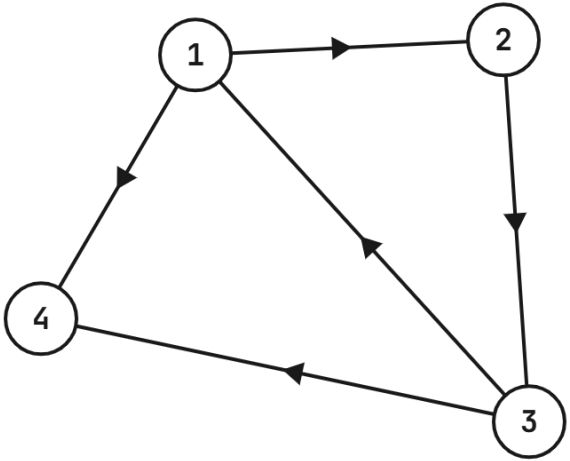
"You change your mind
Like a girl changes clothes."

is what you would like to tell the high priest of Agartha, after he orders you to completely redesign the road network, once again.

Agartha's road network still consists of N sacred chambers and M tunnels connecting them. However, this time, the high priest has ordered that each tunnel can only be traversed in a single direction (for spiritual reasons, of course).

Given the new directed network, the high priest asks you whether it's still possible to travel from any chamber to any other chamber.

Visualisation



Getting Started

How could we solve this problem in $O(N^2)$ time?

Getting Started

How could we solve this problem in $O(N^2)$ time?

Slow solution

- A simple brute-force idea is to try every node as a starting point.

Getting Started

How could we solve this problem in $O(N^2)$ time?

Slow solution

- A simple brute-force idea is to try every node as a starting point.
- For each node u, run DFS/BFS from u.

Getting Started

How could we solve this problem in $O(N^2)$ time?

Slow solution

- A simple brute-force idea is to try every node as a starting point.
- For each node u, run DFS/BFS from u.
- If some traversal misses a node, we can already output `NO`.

Getting Started

How could we solve this problem in $O(N^2)$ time?

Slow solution

- A simple brute-force idea is to try every node as a starting point.
- For each node u, run DFS/BFS from u.
- If some traversal misses a node, we can already output `NO`.
- If all N traversals reach everyone, output `YES`.

Getting Started

How could we solve this problem in $O(N^2)$ time?

Slow solution

- A simple brute-force idea is to try every node as a starting point.
- For each node u, run DFS/BFS from u.
- If some traversal misses a node, we can already output `NO`.
- If all N traversals reach everyone, output `YES`.

But can we do better?

An Observation

Observation

If there exists a node u such that

- u can reach every other node, and
- every other node can reach u,

then the graph is valid.

An Observation

Observation

If there exists a node u such that

- u can reach every other node, and
- every other node can reach u,

then the graph is valid.

Exercise: Why???

An Observation

Observation

If there exists a node u such that

- u can reach every other node, and
- every other node can reach u,

then the graph is valid.

Exercise: Why???

Exercise: How to check that **every** node can reach some u?

Solution

- Let $u = 1$.

Solution

- Let $u = 1$.
- Run DFS/BFS from u on the original graph.

Solution

- Let $u = 1$.
- Run DFS/BFS from u on the original graph.
- If some node v is unvisited, then u cannot reach v, so output `NO`.

Solution

- Let $u = 1$.
- Run DFS/BFS from u on the original graph.
- If some node v is unvisited, then u cannot reach v, so output `NO`.
- Build the reverse graph by replacing each edge $a \to b$ with $b \to a$.

Solution

- Let $u = 1$.
- Run DFS/BFS from u on the original graph.
- If some node v is unvisited, then u cannot reach v, so output `NO`.
- Build the reverse graph by replacing each edge $a \to b$ with $b \to a$.
- Run DFS/BFS from u on the reverse graph.

Solution

- Let $u = 1$.
- Run DFS/BFS from u on the original graph.
- If some node v is unvisited, then u cannot reach v, so output `NO`.
- Build the reverse graph by replacing each edge $a \to b$ with $b \to a$.
- Run DFS/BFS from u on the reverse graph.
- If some node v is unvisited now, then in the original graph v cannot reach u, so output `NO`.

Solution

- Let $u = 1$.
- Run DFS/BFS from u on the original graph.
- If some node v is unvisited, then u cannot reach v, so output `NO`.
- Build the reverse graph by replacing each edge $a \to b$ with $b \to a$.
- Run DFS/BFS from u on the reverse graph.
- If some node v is unvisited now, then in the original graph v cannot reach u, so output `NO`.
- If both traversals visit every node, output `YES`.

Problem C

Last Friday Night

Problem C

Last Friday Night

"Last Friday night, yeah, I think we broke the law."

Problem C

Last Friday Night

"Last Friday night, yeah, I think we broke the law."

Last Friday night, while you were busy working on the new road design, an infamous droid pulled off the greatest heist in Agarthan history.

Problem C

Last Friday Night

"Last Friday night, yeah, I think we broke the law."

Last Friday night, while you were busy working on the new road design, an infamous droid pulled off the greatest heist in Agarthan history.

The criminal **stole** Agartha's entire White Monster supply, and escaped into the road network, with a plan to defect to Hyperborea.

Problem C

Last Friday Night

"Last Friday night, yeah, I think we broke the law."

Last Friday night, while you were busy working on the new road design, an infamous droid pulled off the greatest heist in Agarthan history.

The criminal **stole** Agartha's entire White Monster supply, and escaped into the road network, with a plan to defect to Hyperborea.

Again, there are N sacred chambers and M **bidirectional** tunnels connecting them (you didn't finish the redesign). It's been K hours since the heist, and in each hour, you know the criminal traversed exactly one edge, starting from the Monster Vault in chamber 1.

Problem C

Last Friday Night

"Last Friday night, yeah, I think we broke the law."

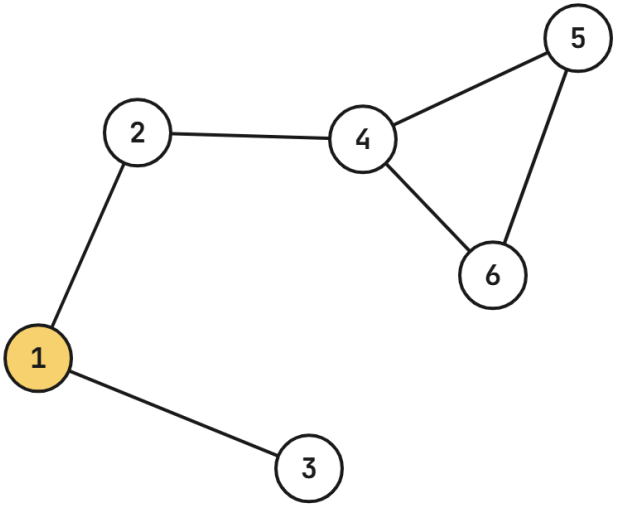
Last Friday night, while you were busy working on the new road design, an infamous droid pulled off the greatest heist in Agarthan history.

The criminal **stole** Agartha's entire White Monster supply, and escaped into the road network, with a plan to defect to Hyperborea.

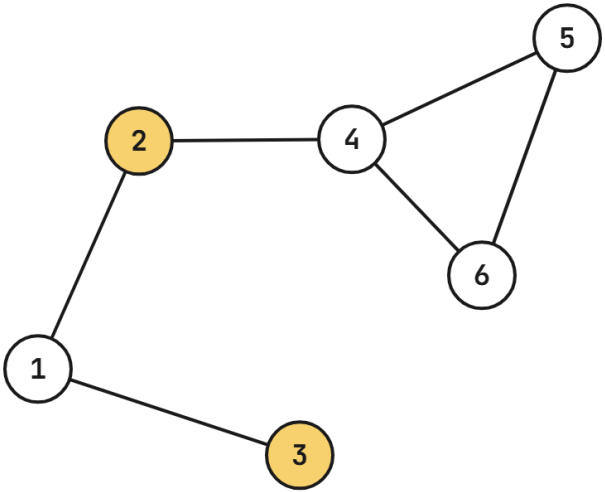
Again, there are N sacred chambers and M **bidirectional** tunnels connecting them (you didn't finish the redesign). It's been K hours since the heist, and in each hour, you know the criminal traversed exactly one edge, starting from the Monster Vault in chamber 1.

The high priest urgently asks: how many chambers could the criminal be in now?

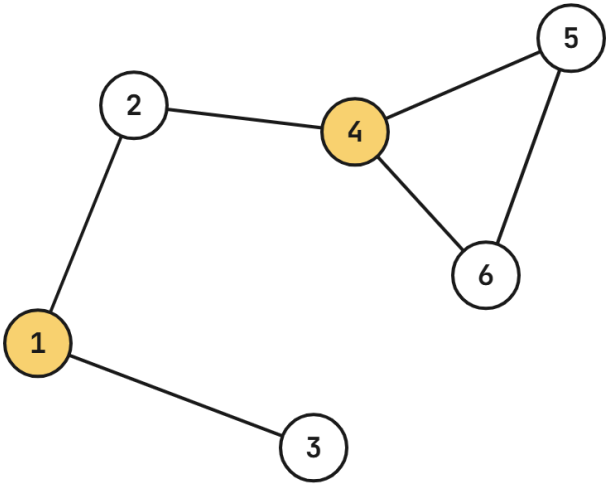
Visualisation



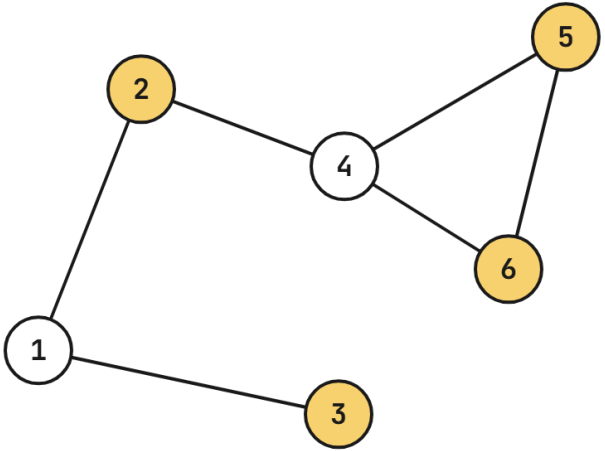
Visualisation



Visualisation



Visualisation



Getting Started

Getting Started

What if we didn't **have** to move every minute?

Getting Started

What if we didn't **have** to move every minute?

- Then after K moves, we could finish at **any** node u within distance K of 1.

Getting Started

What if we didn't **have** to move every minute?

- Then after K moves, we could finish at **any** node u within distance K of 1.
- So we would just run a BFS from 1 and count how many cities u have

$$\text{dist}_u \leq K.$$

Getting Started

What if we didn't **have** to move every minute?

- Then after K moves, we could finish at **any** node u within distance K of 1.
- So we would just run a BFS from 1 and count how many cities u have

$$\text{dist}_u \leq K.$$

What goes wrong?

In the real problem, we must make **exactly** K moves.

Getting Started

What if we didn't **have** to move every minute?

- Then after K moves, we could finish at **any** node u within distance K of 1.
- So we would just run a BFS from 1 and count how many cities u have

$$\text{dist}_u \leq K.$$

What goes wrong?

In the real problem, we must make **exactly** K moves.

So, if we arrive at u to early, we are forced to "waste" some extra moves and return back to u.

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.
- So what really matters is the **parity** of the number of moves.

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.
- So what really matters is the **parity** of the number of moves.
- For each node u, let:
  - $\text{dist}_{0,u}$ = shortest time to reach u in an even number of moves.
  - $\text{dist}_{1,u}$ = shortest time to reach u in an odd number of moves.

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.
- So what really matters is the **parity** of the number of moves.
- For each node u, let:
  - $\text{dist}_{0,u}$ = shortest time to reach u in an even number of moves.
  - $\text{dist}_{1,u}$ = shortest time to reach u in an odd number of moves.
- We can compute these with a state-explosion BFS on $(\text{node}, \text{parity})$.

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.
- So what really matters is the **parity** of the number of moves.
- For each node u, let:
  - $\text{dist}_{0,u}$ = shortest time to reach u in an even number of moves.
  - $\text{dist}_{1,u}$ = shortest time to reach u in an odd number of moves.
- We can compute these with a state-explosion BFS on $(\text{node}, \text{parity})$.
- At the end, node u is valid iff $\text{dist}_{K \bmod 2,u} \leq K$.

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.
- So what really matters is the **parity** of the number of moves.
- For each node u, let:
  - $\text{dist}_{0,u}$ = shortest time to reach u in an even number of moves.
  - $\text{dist}_{1,u}$ = shortest time to reach u in an odd number of moves.
- We can compute these with a state-explosion BFS on $(\text{node}, \text{parity})$.
- At the end, node u is valid iff $\text{dist}_{K \bmod 2,u} \leq K$.

Why?

Solution

- Since the graph is undirected, we can always waste 2 moves by going $u \to v \to u$.
- So what really matters is the **parity** of the number of moves.
- For each node u, let:
  - $\text{dist}_{0,u}$ = shortest time to reach u in an even number of moves.
  - $\text{dist}_{1,u}$ = shortest time to reach u in an odd number of moves.
- We can compute these with a state-explosion BFS on $(\text{node}, \text{parity})$.
- At the end, node u is valid iff $\text{dist}_{K \bmod 2,u} \leq K$.

Why?

If we can reach u with the correct parity and in $\leq K$ hops, we can add extra back-and-forth moves in chunks of 2 until we use exactly K hops.

Outline

1. Housekeeping
2. Roads, cities, wires, and kidneys
3. A graph in code?
4. Exploration!
5. Shortest paths
6. Funny tricks
   - Multisource
   - 0-1 BFS
   - Variations on Dijkstra
7. Challenge problems >:)
8. **Appendix and resources**

Graph Theory – Resources

The **USACO Guide** has good introductory resources (and problems) on graph theory.

https://usaco.guide/silver/graph-traversal

For more beginner problems, CSES has a problemset, starting with

https://cses.fi/problemset/task/1192

The variety problems at the end are best emulated from various past contests.

To get a good intution of graphs, there are plenty of high-quality visualisers online!

https://anacc22.github.io/another_graph_editor/

Feedback!

Please fill out feedback form so we know how to run future workshops :D



(a) Feedback form



(b) Attendance form

Photo time :)





UNSW CPMSoc