6 Graph Algorithms

6.1 Graph Definitions

Definition 6.1 (Graph)**.** A undirected graph $G = (V, E)$ is a set of vertices V and edges $E \subseteq \binom{V}{2}$ (the set of pairs of elements of V). Notationally, we write $n = |V|$ and $m = |E|$. A directed graph $G = (V, E)$ is a set of vertices V and edges $E \subseteq V \times V$.

Definition 6.2 (Weighted Graph)**.** A weighted graph $G = (V, E, w)$ is a graph with a weight $w : E \to \mathbb{R}$ assigned to each edge. Both undirected and directed graphs can be weighted.

Definition 6.3 (Path and Cycle)**.** A path p from $v_0 \rightsquigarrow v_k$ in G is a list of edges $\big((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\big)$. The length of a path is the number of edges, i.e. k. If th graph is weighted, then the length of the paths is the sum of the weights of the edges.

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \tag{6.1}$$

A cycle is a path with $v_0 = v_k$; a *simple cycle* has no repeated vertices.

Definition 6.4 (Connected)**.** A graph is connected if there is a path between every pair of vertices.

Definition 6.5 (Subgraph)**.** A subgraph of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E \cap \binom{V'}{2}$. The weight of the subgraph G' is

$$w(G') = \sum_{(u,v) \in E'} w(u, v) \tag{6.2}$$

Definition 6.6 (Forest and Trees)**.** A forest in G is a subgraph of G without simple cycles and a tree is a connected forest. We often refer to the distinct connected components of the forest as the trees of the forest. A *spanning tree* is a tree in G that connects all the vertices in G.

6.2 Single-Source Shortest Paths

Definition 6.7 (Single-Source Shortest Paths)**.** Given a directed weighted graph $G = (V, E, w)$ with non-negative weights $w : E \to \mathbb{R}^+$ and a vertex $s \in V$, the single-source shortest paths is the family of shortest paths $s \rightsquigarrow v$ for every vertex $v \in V$.

The natural question of course is what is an efficient algorithm to calculate single-source shortest paths. We first notice that the family of single-source shortest paths has a compact representation as a directed tree rooted at s.

Lemma 6.8. *If $(s = v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k = v)$ is a shortest path $s \rightsquigarrow v$, then for any $i \leq k$, the shortest path $s \rightsquigarrow v_i$ is $(s = v_0, v_1), (v_1, v_2), \ldots, (v_{i-1}, v_i)$.*

Proof. Assume for some $i \leq k$, that there was a shorter path $p' : s \rightsquigarrow v$. Then the path p' with $(v_i, v_{i+1}), \ldots, (v_{k-1}, v_k)$ would be a shorter path, contradicting the assumed optimality. □

As a consequence of Lemma 6.8, we can see that the set of single-source shortest paths form a directed tree rooted at s. Furthermore, this offers us an efficient method of storing the single-source shortest paths. We could store them as the $n-1$ edges forming the tree, or for each vertex $v \in V$, we can store parent(v). The latter also allow efficient lookup of the shortest-path for any vertex: Recursively call parent$(\cdot)$ until s is reached.

We begin with a special case where $w = 1$ for all edges. The Breadth First Search Algorithm provides an efficient method for finding the single-source shortest paths in this scenario.

Algorithm 6.9 (Breadth First Search)**.** Let G be a directed graph and u a vertex in G. Initialize all the vertices as unmarked and let Q be an empty queue. Mark u and push u onto Q. While Q isn't empty,

- Pop a vertex from Q; call it a.
- For every vertex b s.t. $(a, b) \in E$, if b is unmarked,
  - Push b onto Q.
  - Set parent$(b) \leftarrow a$.
  - Set dist$(u, b) \leftarrow$ dist$(u, a) + 1$.

Complexity. If G is given by adjacency lists, the runtime is $O(n + m)$. If G is given by adjacency matrices, it costs us to look for the neighbors yielding a runtime of $O(n^2)$.

Definition 6.10. Let $V(u, k) = \{v \in V : \text{dist}(u, v) = k\}$, the set of vertices of distance k from u. We define distance in this case to be the length of the shortest path $u \rightsquigarrow v$.[1]

Lemma 6.11. *There exist times $t_0 < t_1 < \ldots < t_k < \ldots$ s.t. at time t_k,*

1. *The queue Q contains exactly $V(u, k)$.*
2. *The set of marked vertices is precisely $\bigcup_{i=0}^{k} V(u, i)$.*

[1] This is a well-defined metric.

Proof. Proceed by induction. The base case for $k = 0$ is trivial. For $k > 0$, at time t_{k-1}, Q contains exactly $V(u, k-1)$. Define z as the last element in the queue. Define t_k as the time after z has been popped and all its neighbors added to Q.

Then at t_k, we can guarantee that no vertex of $V(u,0) \cup \ldots \cup V(u,k-1)$ is in Q because of the marking of vertices. Furthermore, every vertex in the Q had a neighbor in $V(u,k)$ because of how vertices are added to the queue. These two statement imply that for every $v \in Q$ at t_k, $\text{dist}(u,v) > k-1$ and $\text{dist}(u,v) \leq k$, proving $\text{dist}(u,v) = k$ and $v \in V(u,k)$. Lastly, the vertices marked between t_{k-1} and t_k are $V(u,k)$ proving the second statement. □

To move to general weight functions, $w : E \to \mathbb{R}^+$ or even $w : E \to \mathbb{R}$ (in certain cases), we will need to do something slightly more tricky. But luckily these algorithms will be greedy just like this one.

6.3 Dijkstra's Algorithm

We can adapt the previous algorithm to solve $w : E \to \mathbb{R}^+$. We need to adjust the queue from Algorithm 6.9 to a *priority queue*.

Algorithm 6.12 (Dijkstra's Algorithm)**.** Let $G = (V, E, w)$ be a directed graph with $w : E \to \mathbb{R}^+$ and $u \in V$. Generate a priority queue P and push onto P evey element $v \in V$ with key ∞. Decrease key of u to 0. While P isn't empty,

- Pop the min weight element from P; call it a.
- For every vertex b s.t. $(a,b) \in E$, if $\text{key}(b) > \text{key}(a) + w(a,b)$,
  - Decrease key of b to $\text{key}(a) + w(a,b)$.
  - Set $\text{parent}(b) \leftarrow a$.
- Set $\text{dist}(u,a) \leftarrow \text{key}(a)$.

The resulting try by $\text{parent}(\cdot)$ is a single-source shortest paths function (not-necessarily unique) and $\text{dist}(u,\cdot)$ is the distance function.

Lemma 6.13. $\text{dist}(u,\cdot)$ *is the correct distance function.*

Proof. The proof is by induction and similar to that of Lemma 6.11. Let $\tilde{d}(u,\cdot)$ be the actual distance function. Clearly $\text{key}(u) = 0$ and is never changed so $\text{dist}(u,u) = 0 = \tilde{d}(u,u)$.

We claim that $\text{dist}(u,v) \geq \tilde{d}(u,v)$ for any $v \in V$. Let y be the *first* vertex popped with $\text{dist}(u,y) > \tilde{d}(u,y)$. Define $U \leftarrow$ set of vertices popped before y. Let $u \rightsquigarrow y$ be the shortest path (the one

corresponding to $\tilde{d}(u, y)$). Define v as the last element of U in the path. Let x be the next element in the path.

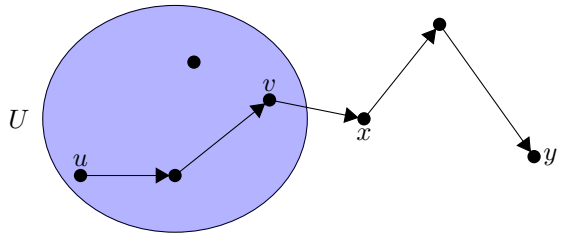

Figure 6.1: Diagram of Lemma 6.13.

By induction, we argue correctness of every element in U. Since x is on the shortest path $u \rightsquigarrow y$ then this is also the shortest path to x (Lemma 6.8). Therefore,

$$\tilde{d}(u, x) = \text{dist}(u, v) + w(v, x). \tag{6.3}$$

Then the decrease key call after popping v will guarantee that $\text{dist}(u, x) = \tilde{d}(u, x)$. If $x = y$, then y isn't a counterexample. So assume $x \neq y$. But as y was the *first* element popped after U, then y was popped before x. So,

$$\text{dist}(u, y) = \text{key}(y) < \text{key}(x) = \tilde{d}(u, x) \leq \tilde{d}(u, y) < \text{key}(y) \tag{6.4}$$

This is a contradiction, so no such first y exists proving correctness. □

Theorem 6.14 (Dijkstra's Runtime)**.** *If Dijkstra's is implemented with a binary heap, the runtime is* $O((n+m)\log n)$*. If Dijkstra's is implemented with a Fibonacci heap, the runtime is* $\Theta(m+n\log n)$*.*

Proof. From the algorithm, we can find that the runtime is $O(m \cdot T_{\text{decrease key}} + n \cdot T_{\text{pop min}})$. The time to decrease a key in a binary heap is $O(\log n)$ and to pop min is $O(\log n)$. Hence the statement. If we use a Fibonacci heap, the cost to decrease key drops to $O(1)$ explaining the runtime speedup. □

6.4 Bellman-Ford Algorithm

We now want to extend this to negative weights as well. However, this can run into some difficulties.

In Figure 6.4, we notice that traversing the triangle yields a weight of -2. Its not difficult to see that the only paths $s \rightsquigarrow t$ are $s \to x$, traversing the triangle n times, and then $x \to t$. The weight of this path is $5 - 2n$ and so the minimum weight path is $-\infty$.

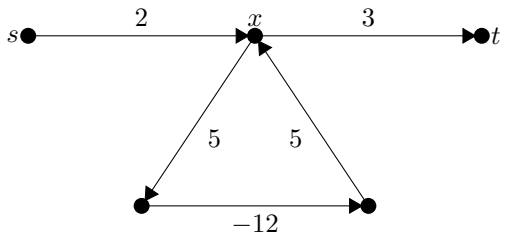

Figure 6.2: Example of an issue with negative weights.

We call a cycle such as the triangle a *negative weight cycle* and we want to identify if these exist in the problem. Therefore, we can adjust the problem definition to:

Exercise 6.15. Let G be a directed weight graph with $w : E \to \mathbb{R}$ and $u \in V$. For every $v \in V$, report if there is a negative weight cycle on some path $u \rightsquigarrow v$ or if not the weight of the shortest path $u \rightsquigarrow v$.

The Bellman-Ford algorithm will fix this issue. Both Bellman-Ford and Dijkstra's work be relaxing the distance function. Meaning, gradually the estimate of the distance is reduced and reduced until the optimal is achieved. However, Dijkstra's algorithm is a *greedy* algorithm and this strategy cannot work for our problem. (There are plenty of counterexamples you can draw.) We are going to go back and apply a dynamic programming approach to catch the negative weight cycles.

Algorithm 6.16 (Bellman-Ford)**.** Given a directed graph $G = (V, E, w)$ with $w : E \to \mathbb{R}$ and a vertex $u \in V$, set $\text{key}(u) = 0$ and $\text{key}(v) = \infty$ for $v \neq u$. Set the iteration number $I = 0$. Repeating until no key values change:

1. If $I = n$, halt and report that a negative cycle was found.
2. For each edge $(a, b) \in E$ (in any order)
   (a) Define $c \leftarrow \text{key}(a) + w(a, b)$.
   (b) If $c < \text{key}(b)$, then set $\text{key}(b) = c$ and $\text{parent}(b) = a$.
3. Increase $I \leftarrow I + 1$.

There are two principle differences between Dijkstra's and Bellman-Ford. One is that the edges we used to consider per iteration of the algorithm were the greedy choice of edges and now we need to consider all edges. The second is the halting condition if keys are decreasing for more than n rounds.

Proof of Correctness: By similar arguments to what we made for Dijkstra's, $\text{key}(v)$ is always an upper bound for $\text{dist}(u, v)$ (the true distance). Let k be the length of the shortest least weight path

$u \rightsquigarrow v$. Due to the tree structure, by the kth iteration, $\text{key}(v) = \text{dist}(u, v)$.

If the graph lacks negative-weight cycles then as the length of any path without cycles it at most $n-1$, by the $n-1$th iteration, the key values should halt changing.

On the other hand, if there is a negative-weight cycle, then there is a path from u to every vertex on that negative weight cycle and that path has at most $n-1$ edges. So by the $n-1$th iteration, every key $< \infty$. The negative weight cycle must contain at least one edge of negative weight (call it (a, b)). Then on the nth iteration, $\text{key}(b)$ will decrease by at least $\left|w(a, b)\right|$. Then the alg. halts appropriately. □

Complexity. If the graph is given as adjacency lists, the runtime is $O(mn)$ as it runs n iterations of testing each edge. If the graph is given as an adjacency matrix, it takes $O(n^2)$ time per iteration for a total runtime of $O(n^3)$. A more nuanced analysis shows that for graphs lacking negative cycles with the length of least weight paths at most ℓ, the runtime is $O(m\ell)$.

6.5 Semi-Rings

We take a brief interlude from graph problems to introduce some more math to make the analysis nicer.

Definition 6.17 (Semi-ring)**.** A semi-ring is a set K along with two operations which we call $(+), (\times) : K \times K \to K$. These operations must satisfy:

- $(+)$ is commutative and associative. Meaning for $a, b, c \in K$, $a(+)b = b(+)a$, $(a(+)b) + c = a(+)(b(+)c)$.
- There is an identity element for $(+)$ called 0. For $a \in K$, $a(+)0 = a$.
- $(\times)$ is commutative and associative. Meaning for $a, b, c \in K$, $a(\times)b = b(\times)a$, $(a(\times)b)(\times)c = a(\times)(b(\times)c)$.
- There is an identity element for $(\times)$ called 1. For $a \in K$, $a(\times)1 = a$.
- There is a distributive law. For $a, b, c \in K$,

$$a(\times)(b(+)c) = (a(\times)b)(+)(a(\times)c)$$

A ring is a semi-ring with the additional constraint that $(+)$ has inverses. Given the notation of $(+), (\times)$ a natural realization of a semi-ring is with $K = \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{Z}/k\mathbb{Z}$ and $(+)$ and $(\times)$ being

the operations that we traditionally define them as.[2] But, we can define $(+)$ and $(\times)$ as other functions as well and still produce semi-rings.

Take for example, $K = \mathbb{R} \cup \infty$, $(+) = \min$, and $(\times) = +$. We can verify that this is a semi-ring. The minimization operator is certainly commutative and associative as $\min(a, b) = \min(b, a)$ and $\min(a, \min(b, c)) = \min(a, b, c) = \min(\min(a, b), c)$. The identity element for $(+)$ is ∞ as $\min(a, \infty) = a$. As $(\times)$ is addition, then the criteria hold with identity element of 0. The distributive law is a fun exercise to check.

Definition 6.18 (Min-Sum Semi-Ring)**.** The min-sum semi-ring is defined by $K = \mathbb{R} \cup \infty$, $(+) = \min$, and $(\times) = +$.

Why do we care about semi-rings? We find that certain computational rpboelms can be expressed as known problems over a semi-ring. For example, we will see that the all-pairs shortest path problem can be solved rather easily via matrix multiplication over the min-sum semi-ring K.

6.6 All Pairs Shortest Paths

Definition 6.19 (All Pairs Shortest Paths)**.** Given a directed weighted graph $G = (V, E, w)$, for each $u, v \in V$, find the shortest path $u \rightsquigarrow v$.

Definition 6.20 (Adjacency Matrix)**.** Given a weighted directed graph $G = (V, E, w)$, the adjacency matrix A is the unique $n \times n$ matrix defined by

$$A_{ij} = \begin{cases} 0 & (i, j) \notin E \\ w_{ij} & (i, j) \in E \end{cases} \tag{6.5}$$

Lemma 6.21. *For a weighted directed graph $G = (V, E, w)$ with adjacency matrix A, the shortest path $i \rightsquigarrow k$ of length $\leq \ell$ has weight $(A^\ell)_{ik}$ where the matrix product is computed over the min-sum semi-ring.*

Proof. We prove by induction. For $\ell = 1$, this is trivially true: The path $i \rightsquigarrow k$ only exists if $(i, k) \in E$ and has weight $w_{ik} = A_{ik}$. Assume true now up to ℓ. Since we are computng over the min-sum semi-ring

$$(A^{\ell+1})_{ik} = (A \cdot A^\ell)_{ik} = \min_j \left(A_{ij} + (A^\ell)_{jk} \right) \tag{6.6}$$

If a path exists $i \rightsquigarrow k$ of length $\leq \ell + 1$, it must have a second vertex j. The path weight will be the weight of $i \to j$ plus the weight of $j \rightsquigarrow k$ under the condition that the length $\leq \ell$. The equation above, exhaustively minimizes over all possible j proving correctness. □

[2] All of these examples are rings.

This gives a natural algorithm for calculating the solution to the all-paths problem due to the following lemma you are probably familiar with.

Lemma 6.22. *For a graph G with n vertices, then the length of the shortest path $u \rightsquigarrow v$ is at most n (if it exists).*

Proof. If a path longer than n exists, it must have a repeated vertex by the pidgeon-hole principle. We can then cut out the loop from the path and decrease the length. We can iteratively apply this until we get a path of length $\leq n$. Therefore, the shortest path must have length $\leq n$. □

As the longest path has length $\leq n$, we only need to look at A^n to calculate the single-source shortest paths. This presents the following first-take algorithm:

Algorithm 6.23 (All-pairs Shortest Path)**.** For a graph G, calculate the adjacency matrix A. Using repeated-squaring calculate A^n under the min-sum semi-ring. Then A_{ij} is the minimum weiht of the path $i \rightsquigarrow j$. This algorithm runs in $O(n^\omega \log n)$ where ω is the exponent of matrix multiplication (naïvely $\omega = 3$).

Proof. The prior lemmas tell us that we onl need to look at A^n. Repeated matrix multiplication will take $O(\log n)$ matrix multiplications and their runtime is each going to be $O(n^\omega)$. □

There are a lot of interesting open questions about this approach. For one, how fast is min-sum matrix multiplication Can we improve past $O(n^\omega)$?

6.7 Floyd-Warshall Algorithm

In practice, we know $\omega < 2.373$ (thanks Virginia Williams!) but the constant associated with the runtime is gigantic. We will see how to prove that $\omega < \log_2 7$ later but for now, we can build a quick Dynamic Programming algorithm for all-pairs shortest path.

In the previous example, we generated 'subproblems' decided by if there was a path of $u \rightsquigarrow v$ that contained $\leq k$ vertices. For the next part, let's label the vertices with $1, \ldots, n$ for simplicity. We can define a subproblem as the least weight path from $i \rightsquigarrow j$ with only internal vertices from $\{1, \ldots, k\}$. See Figure 6.7 for an example.

We can define the solution to a subproblem in therms of $O(1)$ subsubproblems. Let $D_k(i, j)$ be the least weight path $i \rightsquigarrow j$ with only internal vertices from $\{1, \ldots, k\}$.

$$D_k(i, j) = \min\{D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)\} \tag{6.7}$$

The proof of this statement's correctness comes from its exhaustiveness. Either the minimal path includes vertex k or it doesn't. If it doesn't, then the answer is iven by the subproblem $D_{k-1}(i,j)$. If it does include k, then we can write it in terms of the two subproblems $D_{k-1}(i,k)$ and $D_{k-1}(k,j)$. The base cases are $D_0(i,j) = w_{ij}$.

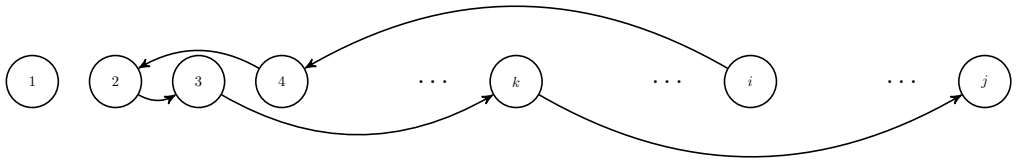

Figure 6.3: Example path $i \rightsquigarrow j$ that only includes vertices from $\{1, \ldots, k\}$

This yields the following Dynamic Programming Algorithm:

Algorithm 6.24 (Floyd-Warshall Algorithm for All-Pairs Shortest Path)**.** Initialize $D_0(i,j) = w_{ij}$. For $k = 1, \ldots, n$, we apply (6.7). The shortest-path $i \rightsquigarrow j$ has weight $D_n(i,j)$.

Proof of Correctness: We proved the inductive correctness of the definition earlier. Inductively, we consider every path when we consider $D_n(i,j)$. □

Complexity: The time and space is $O(n^3)$ due to the size of the table and each subproblem has $O(1)$ additional runtime over its subproblems. □

6.8 Johnson's Algorithm

Can we do better if we know some properties of the graph? For example, a very common problem is what can we say if the graph is sparse? We know m, the number of edges, is bounded above by $O(n^2)$. However, our runtime for Floyd-Warshall was not dependent on m.

Assume we had a potential function $\psi : V \to \mathbb{R}$ on the vertices. We can define a new weight function that accounts for the potential by:

$$\tilde{w}_{uv} = w_{uv} + \psi(u) - \psi(v) \tag{6.8}$$

We can see that shifting our weight function by this potential doesn't change our problems too much.

Lemma 6.25. *For any path $\gamma = (v_0, \ldots, v_k)$ be a path in the graph. And define the weight of a*

path as the sum of the weights of the edges in the path. Then

$$\tilde{w}_\gamma = \tilde{w}_\gamma + \psi(v_0) - \psi(v_k) \tag{6.9}$$

Proof. The sum telescopes:

$$\begin{aligned} \tilde{w}_\gamma &= \sum_{j=0}^{k-1} \tilde{w}_{v_j v_{j+1}} \\ &= \sum_{j=0}^{k-1} w_{v_j v_{j+1}} + \psi(v_j) - \psi(v_{j+1}) \\ &= w_\gamma + \psi(v_0) - \psi(v_k) \end{aligned} \tag{6.10}$$

□

Corollary 6.26. *As the weight of any path $u \rightsquigarrow v$ is augmented by $\psi(u) - \psi(v)$, then solving the least weight problem over metric w or $\tilde{w}$ is equivalent.*

The question is then, what potential ψ can we write to solve this problem? A nice potential would be one with which we could argue that $\tilde{w}_{ij} \geq 0$. This way, we could just treat the problem as n single-source shortest path problems and apply Dijkstra's (one from every vertex). We show that such a potential function necessarily exists.

Lemma 6.27. *For any graph $G = (V, E, w)$ without negative-cycles, there exists a potential function ψ such that $\tilde{w}_{ij} \geq 0$ for all i, j. Furthermore, this potential can be calculated in $O(nm)$ time.*

Proof. Pick a vertex $s \in V$ as the start node. set $\psi(v)$ as the weight of the shortest path $s \rightsquigarrow v$. This can be calculated in $O(nm)$ with Belman-Ford algorithm. Therefore, for all $u, v \in V$,

$$\psi(u) + w_{u,v} \geq \psi(v) \tag{6.11}$$

as this equivalent to the triangle inequality statement, $w_{s,u} + w_{u,v} \geq w_{s,v}$. From here, rearranging the previous equation shows that $\tilde{w}$ is non-negative. □

We can combine all these parts together to form the following algorithm:

Algorithm 6.28 (Johnson's)**.** Given a graph $G = (V, E, w)$ we calculate a potential function ψ as described in Lemma 6.27. If the subroutine Bellman-Ford declares that negative weight cycles exist, then abort. Otherwise, using the potential function, calculate $\tilde{w}$. Then, for each vertex $u \in V$, run Dijkstra's algorithm starting at v.

Proof of Correctness: The correctness follows as a direct consequence of Lemma 6.27 and the correctness of Bellman-Ford and Dijkstra. □

Complexity: The runtime is $O(mn + n^2 \log n)$ because Bellman-Ford runs in $O(mn)$ and we run n copies of Dijkstra's which run in $O(m + n \log n)$. □

6.9 Cut for Space: Savitch's Algorithm

We end this somewhat long excursion into shortest-path algorithms with a seemingly simple problem about how to calculate if a path exists but use very little space.

Exercise 6.29. What is the minimal space complexity of an algorithm to decide for a graph G with vertices s, t if there exists a path $s \rightsquigarrow t$?

We will show that there is a $O(\log^2 n)$-space algorithm for deciding this problem. This problem is actual an **NL**-complete problem, meaning that it is a complete problem that can be solved non-deterministically using $O(\log n)$ space. If someone can show that this problem can be solved deterministically in $O(\log n)$ space, this will show the complexity class collapse $\mathbf{L} = \mathbf{NL}$.

First a word on how small $O(\log n)$ space is. The space it takes to write down a specific vertex in memory requires $O(\log n)$ space. So, using $O(\log n)$ space means that the algorithm is only writing down effectively a constant number of vertices. To me that is an absurdly small! $O(\log^2 n)$ is a little nicer, this means we can write down logarithmically many vertices. Furthermore, remember that space and time have a trade-off. We expect that the time complexity is going to be pretty bad...
Do you remember Zeno's paradox? In order for the tortoise to complete th race, he must first go halfway. But to go halfway, he must first go a quarter-way and so on...

What if we guessed (as in tried all n possibilities) for a halfway vertex of the path $s \rightsquigarrow t$. Then we can recursively guess a quarterway point, etc. As the path has length at most n, and this recursive problem has depth $O(\log n)$ and at each recursive step we only need to store $O(\log n)$ information. Let's formalize:

Algorithm 6.30 (Savitch's)**.** Define $\text{Savitch}(u, v, k)$ if there exists a path $u \rightsquigarrow v$ of length $\leq 2^k$. We return $\text{Savitch}(u, v, \lceil \log n \rceil)$. We can define $\text{Savitch}(u, v, 0) = \text{TRUE}$ if only if u and v are connected. Recursively,

$$\text{Savitch}(u, v, k) = \bigvee_{m \in V} \big(\text{Savitch}(u, m, k-1) \wedge \text{Savitch}(m, v, k-1)\big) \tag{6.12}$$

Proof of Correctness: Since we exhaustively consider all midpoints recursively, if a path exists it will be found. □

Complexity: As the algorithm is run with $k = O(\log n)$ at the top level, then the total depth is $O(\log n)$. Furthermore, each subproblem only requires storing the vertices defining the subproblem in the stack. This is a constant number of vertices, so $O(\log n)$ space. Applying a depth-first search (which we cover next), we can solve this problem in $O(\log^2 n)$ space.

6.10 Depth-First Search

The idea of Depth-First Search is relatively simple; we want an efficient method to traverse the tree and quickly explore each node.

Algorithm 6.31 (Depth-First Search)**.** Given a graph G, Initially mark every vertex as *unvisited*.