On a graph that is simply a linear list, or a graph consisting of a "root" node v that is connected to all other nodes, but such that no other edges are in the graph, breadth first and depth first traversals can visit the nodes of the graph in the same order. On other graphs, however, an ordering that is possible with depth first search is not possible with breadth first search and vice versa. Can you construct an example of such a graph?

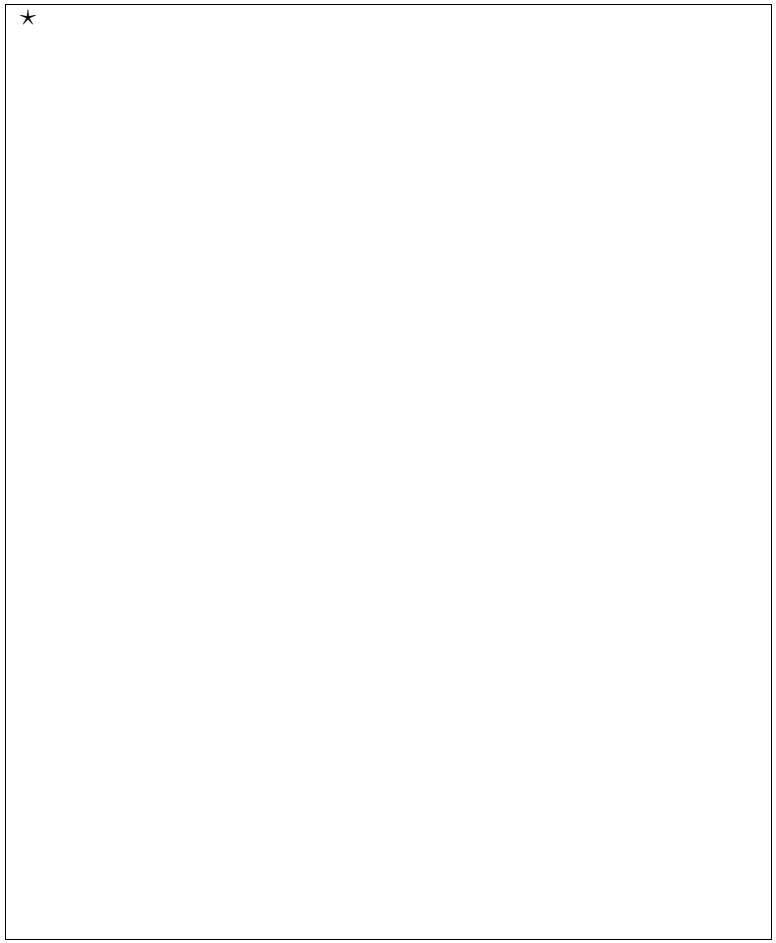
Game Trees

Trees are useful in the analysis of games like tic-tac-toe, chess, and some economic games.

Consider a version of a game called *nim*. Initially, there are n piles, each containing a number of identical tokens. Players alternate moves. A move consists of removing one or more tokens from one pile. The player who removes the last token loses. As an example, let's play with 2 piles: one containing 3 tokens and one containing 2 tokens. The **game**

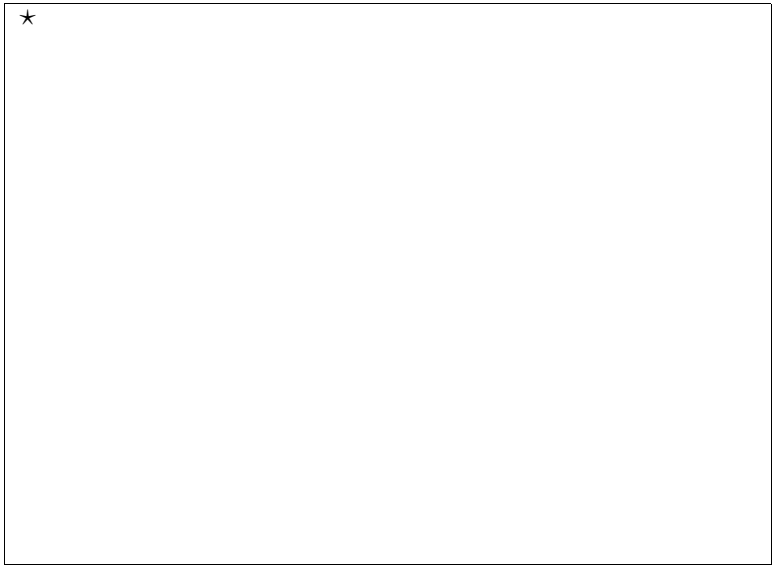
tree follows.

The corresponding **minimax tree** is



The Minimum Spanning Tree Problem

Given a connected graph with n nodes in which the edges are weighted, the goal is find a tree with n nodes which is a subgraph of the original graph, and such that the sum of weights of edges in the tree is minimized. Such a tree is called a minimum spanning tree of the graph. Here is an example:



Efficient Algorithms for MST

1 Kruskal's Algorithm

This algorithm first sorts the edges by weight. Then, the edges are considered in order of increasing weight. When an edge is considered, it is added to the tree (with its endpoints) if it does not create a cycle with previously added edges.

2 Prim's Algorithm

1. Pick arbitrary starting node.
2. Let e be an edge of least cost that connects a node already in the tre to a node not yet in the tree. Add e (along with its endpoint) to the
3. Repeat step 2 until all of the nodes are in the tree.

This is an example of a **greedy algorithm**: on each iteration of step 2, the tree is expanded to include a new node by choosing the cheapest possible edge.

⋆ Prim's example

⋆ Kruskal's example

Correctness of Prim's Algorithm

We now explain why Prim's algorithm correctly returns a minimum spanning tree of the input graph. This is our first look at a proof that an algorithm is correct. In this case, the algorithm consists of a sequence of steps, where in each step we are adding a new edge to the MST. It is natural to structure the proof of correctness so that it reasons about one step at a time. The following proof does this as follows: it shows that if the algorithm is "correct" after i steps, in the sense that the partial tree constructed at that point is a subtree of some MST, then the algorithm is also correct after $i + 1$ steps.

Claim: *If the partially constructed tree T, constructed after i edges have been added, is a subtree of some minimum spanning tree, then so is the tree T', constructed after $i + 1$ edges are added.*

⋆ **Proof of claim:** Let T_{final} be some minimum spanning tree that contains T as a subtree. Let $\{x, y\}$ be the $(i+1)^{st}$ edge added by the algorithm. If T_{final} contains $\{x, y\}$, we are done. Otherwise, let P be the path in T_{final} connecting x with y.

Let e be the first edge on path P that is not in the tree T. (In our example, e is the edge $\{b, d\}$.) Let T'_{final} be the tree obtained from T_{final} by removing e and adding $\{x, y\}$. Then T'_{final} is also a spanning tree of our graph.

We now show that the cost of T'_{final} is $\leq$ the cost of T_{final}. This is because the weight of (x, y) must be $\leq$ the weight of e, by the construction of our algorithm. Hence,

$$\begin{aligned} \text{cost}(T'_{final}) &= \text{cost}(T_{final}) + \text{weight}((x, y)) - \text{weight}(e) \\ &\leq \text{cost}(T_{final}) \end{aligned}$$

Thus, T'_{final} is a minimum spanning tree which contains T' as a subtree. ■

Here is one way to think about this proof that may be helpul. Suppose that person A is applying Prim's algorithm to build up a minimum spanning tree. Suppose that person B happens to have a copy of some minimum spanning tree T_{final} of the graph. Each time person A adds a new edge to the tree T being built up, A asks B if the tree T is ok. Person B compares T to T_{final} and if T is a subtree of T_{final}, person B says "looks good." But now suppose that when edge $\{x, y\}$ to the tree T, the resulting tree T' is no longer a subtree of T_{final}. This time, B's response to A is: "well, I don't know

whether what you are doing is correct or no, but what you have now is not consistent with the tree that I happen to know is a MST of the graph."

Now, A is anxious to convince B that, even though the tree that A is constructing is going to look different than B's tree, it still is going to be a MST. (Remember that there may be lots of distinct MST's for a given graph.)

What A can do to convince B is the following: A takes B's tree, adds the edge $\{x, y\}$ and breaks the resulting cycle as described in the proof above (i.e. by removing the first edge e on the path in B's tree from x to y that is not in A's tree). A thus obtains a new tree, call it T'_{final}, that is also a MST of the graph. Since T'_{final} differs from T_{final} by just two edges, B can easily see that the total cost of the edges in A's tree is indeed no more than the total cost of the edges in B's tree, and is therefore a MST.

Implementation and running time of Prim's Algorithm

A good data structure for storing the nodes that are not yet in the MST is a heap (also known as a priority queue), where the key used to order the heap elements is the cost, $cost(u)$. The main heap operations needed are:

- **insert** all elements into the heap (Time: $O(n)$)
- **extract** the minimum element (Time: $O(\log n)$)

We can now analyze the running time of the algorithm.

- We initially build the heap once. Total time: $O(n)$.
- While looping over all nodes, we do exactly one "extract the minimum element" operation each iteration through the loop. Total time: $O(n \log n)$.

There are m edges. Total time: $O(n \log n + m)$. For a more detailed analysis of the running time anc pseudocode see the textbook.

Implementation of Kruskal's MST Algorithm

In Kruskal's algorithm, initially the edges are sorted; the time to do this is $O(m \log m)$. Then, for each edge in increasing order of weight (cost), if the edge does not create a cycle with the edges added so far, then add the edge to the tree.

Shortest Path Algorithms

Description of the Shortest Path (SP) Problem

Let $G = (V, E)$ be a directed graph, with weights $w(u, v)$ on the edges (negative edges are allowed). Suppose we wish to find the shortest path between two points in a graph. Unfortunately, the shortest path between two points may be undefined if there is a "negative cycle" from u to v. For example, consider the graph:

There are several variations of the shortest path problem:

1. single source, single destination
2. single source, all destinations
3. all sources, all destinations

All known algorithms that solve (1) actually solve (2). There are two possible outputs of an algorithm for variation (2). They are:

- If the graph has no negative cycle then the output is a directed spanning tree giving the shortest path to all nodes reachable from the source. This is known as **Shortest Path Tree**.
- If, on the other hand, the graph has negative cycles then the algorithm reports a "negative cycle" and terminates.