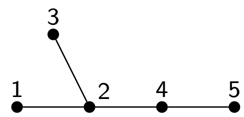
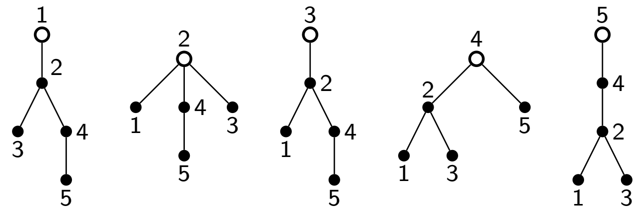
Rooted trees

Recall that a tree is an acyclic connected graph. A rooted tree is a (labeled) tree, together with a choice of special vertex.

For example, if T is the tree



then there are 5 associated *rooted* trees:



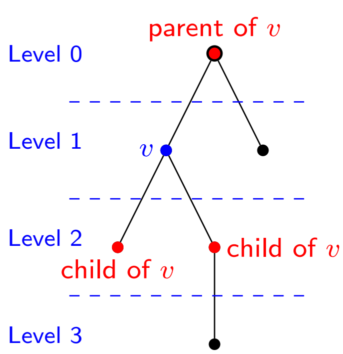
(We typically draw trees with the root at the top.)

Aside: How many rooted trees are there with vertex set $V = \{1, \ldots, n\}$?

(Hint: recall Prüfer code)

Rooted trees

Recall that a tree is an acyclic connected graph. A rooted tree is a (labeled) tree, together with a choice of special vertex.



By convention, draw the root at the top.

The choice of root determines a grading on the rooted tree, given by the distance from the root. (Distance between vertices u and v is the length of a shortest walk from u to v.)

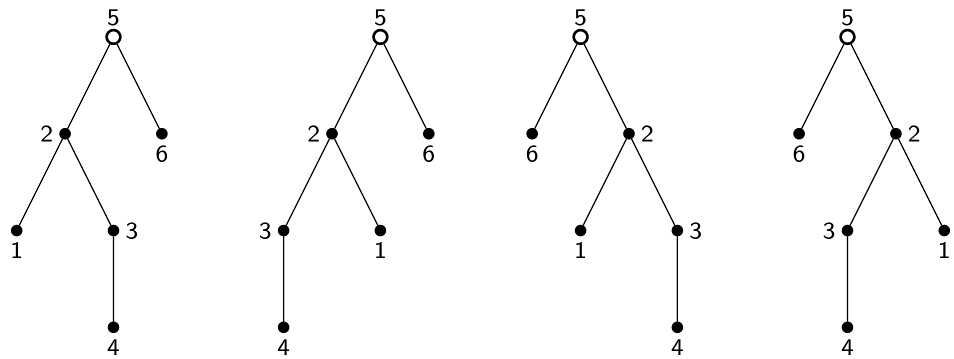
Give a choice of vertex v, the parent of v is the neighbor that is one level up (unique!). A child of v is any neighbor one level down.

Recall a leaf is a vertex of degree 1; note that leaves are the vertices with no children. Everything else is called an internal vertex.

Ordered rooted trees

Recall that a tree is an acyclic connected graph. A rooted tree is a (labeled) tree, together with a choice of special vertex. An ordered rooted tree (ORT) is a rooted tree, together on a choice of order on each of the children of each vertex.

For example, the following are all equal as rooted trees, but are distinct as *ordered* rooted trees:



Given a rooted tree, there are $\boxed{}$ associated ORT's.

Application 1: Binary search trees

Goal: Searching for items in an ordered set.

Building the tree: Let U be a totally ordered set (e.g. words, integers, etc.), and let $S \subseteq U$ be a finite subset. To build the search tree for S...

1. Pick some $r \in S$ to be the root.
2. Given a partial tree, insert a new vertex $s \in S$ by starting with the root, compare s to vertices already in place. When at vertex v...
   - if $s < v$, look for a left child...
     - if v has a left child, then move there and return to 2;
     - if not, insert s as a left child to v.
   - if $s > v$, look for a right child...
     - if v has a right child, then move there and return to 2;
     - if not, insert s as a right child to v.

Theorem. The height of a binary search trees for a set S of size n is between $\lceil \log_2(n+1) \rceil - 1$ and $n - 1$. (The more "balanced", the shorter.)

Remark: There are algorithms for balancing search trees as they get built. (See: "data structures")

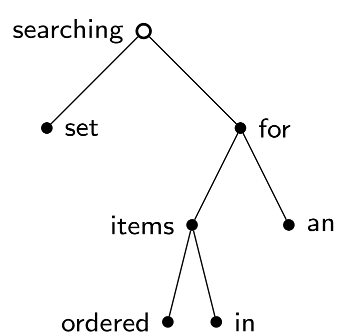
Moral: Building the tree takes some work, but once it's built, it reduces the computational complexity of finding items.

1. Pick some $r \in S$ to be the root.
2. Given a partial tree, insert a new vertex $s \in S$ by starting with the root, compare s to vertices already in place. When at vertex v...
   - if $s < v$, look for a left child...
     - if v has a left child, then move there and return to 2;
     - if not, insert s as a left child to v.
   - if $s > v$, look for a right child...
     - if v has a right child, then move there and return to 2;
     - if not, insert s as a right child to v.

Example: Build a search tree for

{ searching, for, items, in, an, ordered, set }

(ordered alphabetically).

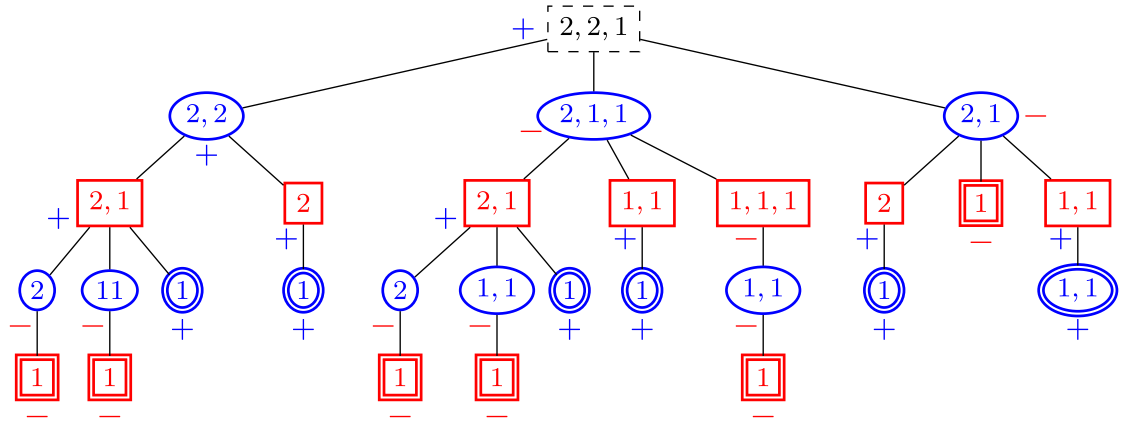


You try: add the words { build, the, search, tree } to the above ORT.

Application 2: Game trees

Given a game played in turns, build a decision tree for that game.

Nim: 2 players start with piles of stones. Taking turns, each player takes one or more stones from any one pile. The player to take the last stone loses. Example: Start with 2, 2, and 1. The associated game tree is. . .

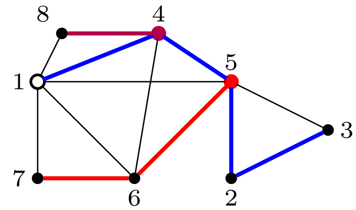


Next, assign $+1$ to leaves where player 1 wins, and -1 to leaves where player 2 wins. For internal vertices: give player 1 the min value of its children, give player 2 (and the root) the max value of its children.

Thm: The value says who will win if each player follows a min/max strategy.

Application 3: Searching graphs

Let G be a simple connected graph, and put an order on the vertices.

Ex: $G =$ 

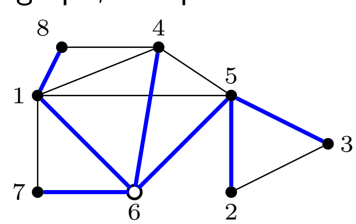
Depth-first search:

1. Pick a vertex to start at.
2. Walk away from that vertex, never repeating previously visited vertices, always picking the least available neighboring vertex, until the walk cannot be extended.
3. Tracing backwards along your last walk, stop at the last vertex that had an available neighbor. Repeat 2 from that vertex.
4. Stop when you're out of vertices.

The result is a *rooted spanning tree*.

Searching graphs

Let G be a simple connected graph, and put an order on the vertices.

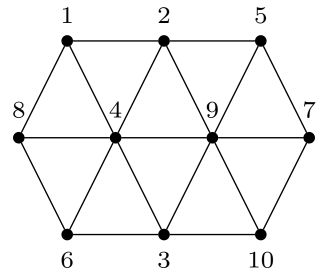
Ex: $G =$ 

Breadth-first search:

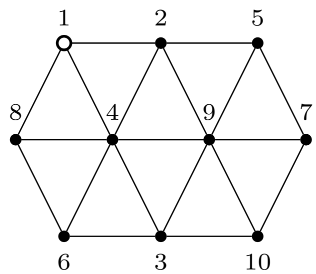
1. Pick a vertex to start at.
2. Walk one step to each of the available neighbors.
3. Of the vertices visited in the previous step, moving in order, repeat step 1.
4. Stop when you're out of vertices.

The result is also a *rooted spanning tree*. Note that at each recursion, you're building all of the vertices at a given level.

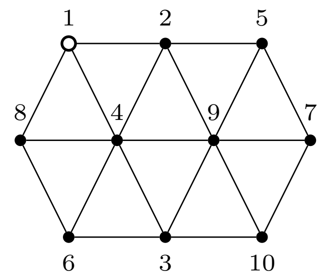
You try: Starting from vertex 1, compute the depth-first and breadth-first search trees for the graph



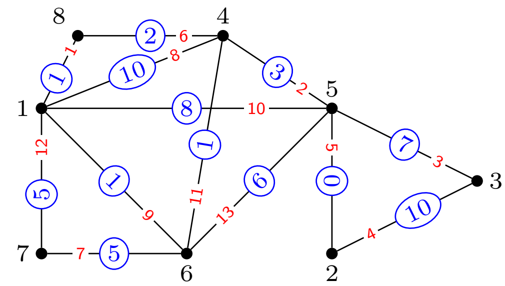
Depth:



Breadth:



A weighted graph is a graph together with numerical weighting on the edges.

Ex: $G =$ 

Relevant questions:

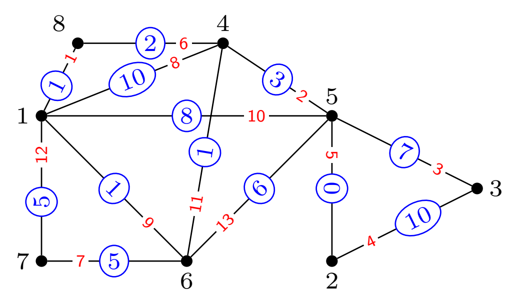
- Given vertices u and v, what's the smallest-weight walk from u to v? (Think: flights, cab rides, production lines, etc.)
- What is the smallest-weight spanning tree?

See also: Traveling salesman problem.

Prim's algorithm: Order the edges. $O(m \log n)$

1. Pick an edge of minumum weight and add it (and its vertices) to your tree.
2. Of all edges incident to vertices already included, moving in order, add all edges of minimum available weight (without creating cycles).
3. Stop when you've covered all vertices.

A weighted graph is a graph together with numerical weighting on the edges.

Ex: $G =$ 

Relevant questions:

- Given vertices u and v, what's the smallest-weight walk from u to v? (Think: flights, cab rides, production lines, etc.)
- What is the smallest-weight spanning tree?

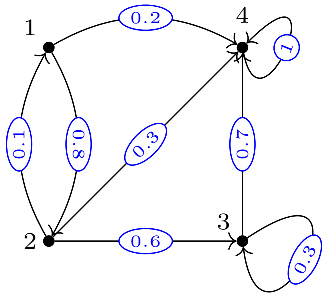
See also: Traveling salesman problem.

Kruskal's algorithm: Order the edges. $O(m \log m)$

1. Moving in order, add the first addable edge of minimum available weight. (Of all addable edges of minimum weight, pick the first according to your order.)
2. Repeat until you've covered all vertices.

Random walks

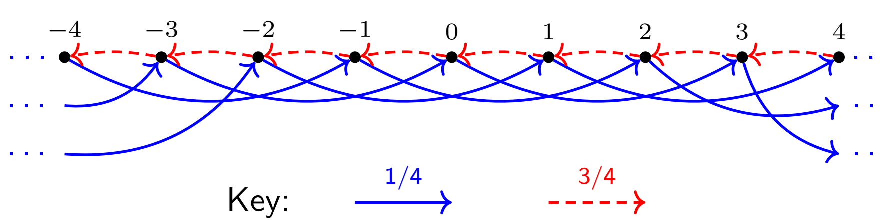
Let G be a weighted directed graph (assume no multiple arrows), satisfying the property that for any vertex v, the sum of the weights on the out-arrows is 1.

Ex: $G =$ 

A random walk is a walk generated iteratively, where each step is taken with probability determined by the weight of the out arrows.

Random walks

Let G be a weighted directed graph (assume no multiple arrows), satisfying the property that for any vertex v, the sum of the weights on the out-arrows is 1.
A random walk is a walk generated iteratively, where each step is taken with probability determined by the weight of the out arrows.
Example: Suppose you play a game of dice, where at each turn you roll two six-sided dice. If you roll a multiple of 4 (prob $1/4$), you get \$3; if you don't, you pay \$1 (prob $3/4$).



Relevant questions: Starting at vertex u...

1. What's the probability that you'll reach vertex v?
2. After n steps, what's the probability that you've landed at v?
3. Is it more likely for a walk gravitate toward any one vertex? Is it more likely that a random walk wanders off in any particular direction?