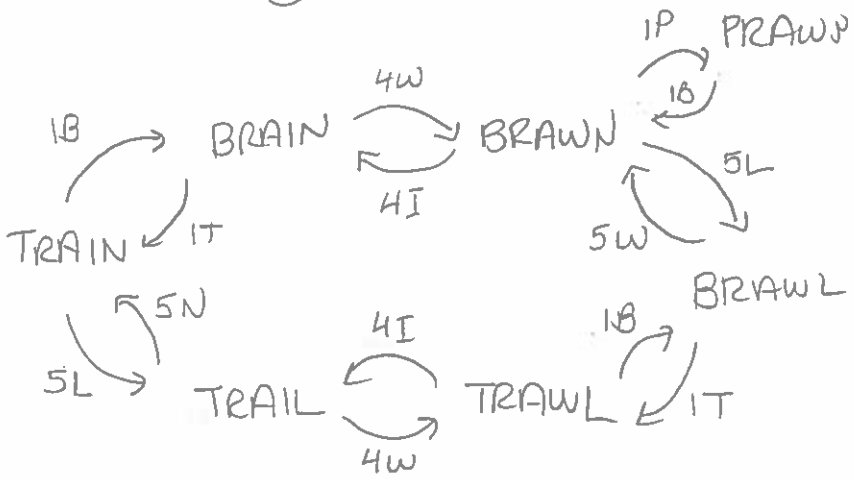
Uninformed Search

① Let's think about how we might find a solution to the word ladder problem using our state machine formulation.

TRAIN
_____
_____
_____
PRAWN



② We'll define a <u>search node</u> as a triple $\langle q, g, h \rangle$ where:

- $q \in Q$ is a state of our state machine
- $g \in \mathbb{R}$ is the "cost" of the search node (roughly speaking)
- $h \in \mathbb{R}$ is a guess at the "cost" of finding a final state from the current state q. (roughly speaking)

If we have a search node $n = \langle q, g, h \rangle$, we will sometimes use the notation $q(n)$, $g(n)$, $h(n)$ to refer to its components.

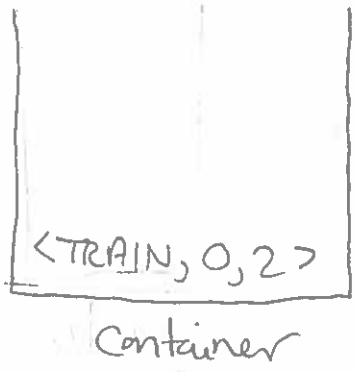
Uninformed Search

(3) Let's go through a general-purpose search strategy. We'll start by making a search node out of the initial state ("TRAIN"):

$$n_0 = \langle \text{TRAIN}, 0, 2 \rangle$$

Since we haven't done anything yet, we'll say the cost $g(n_0) = 0$. We'll guess that the cost $h(n_0)$ of finding a final state from TRAIN is 2, because only two letters of TRAIN are different than PRAWN

(4) We put this node into a container.



(5) Next, we get a node from the container. There's only one; of course, so we get that one:

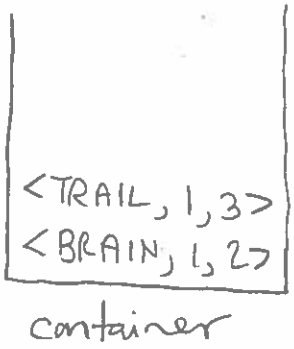
⑥ We now want to look at its "successors". TRAIN has two neighbors in the state machine: BRAIN and TRAIL. Each of these states costs 1 to reach (assuming the weight of any transition is 1 for word ladder). Because BRAIN has 2 letters that are different from PRAWN, we add node

<BRAIN, 1, 2>

to the container. Because TRAIL has 3 letters that are different from PRAWN, we add node

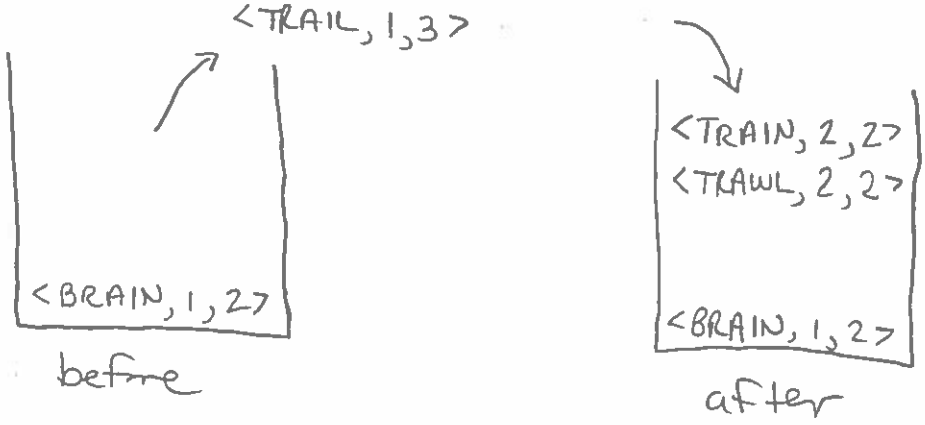
<TRAIL, 1, 3>

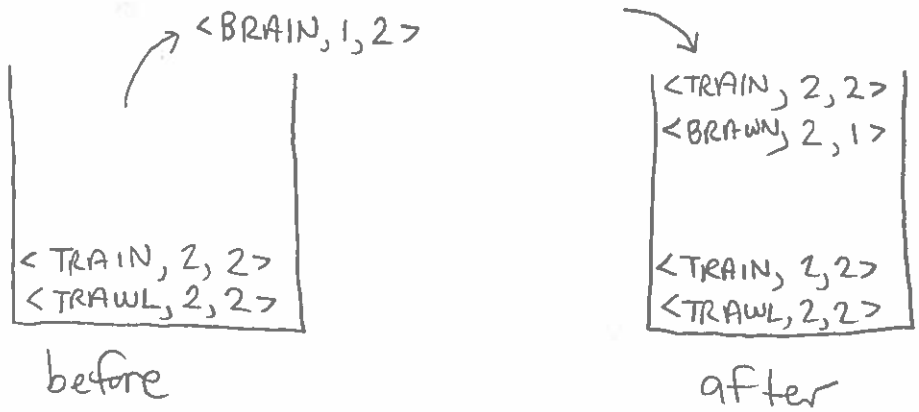
to the container.

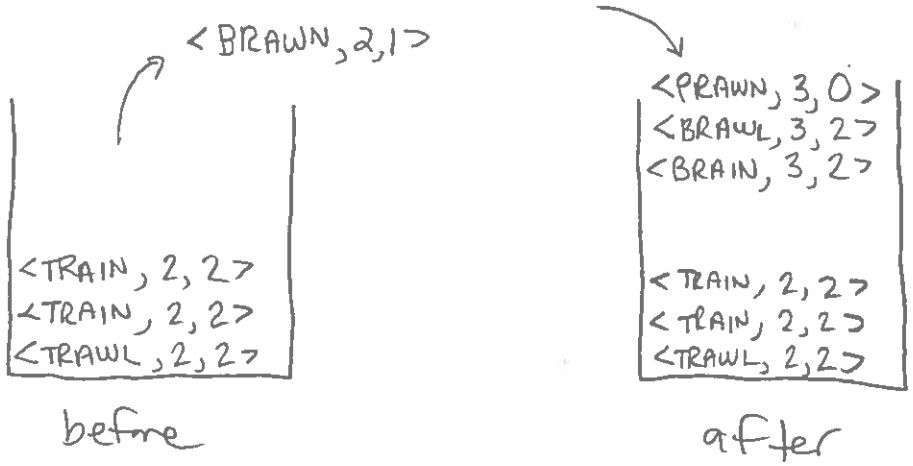


Uninformed Search

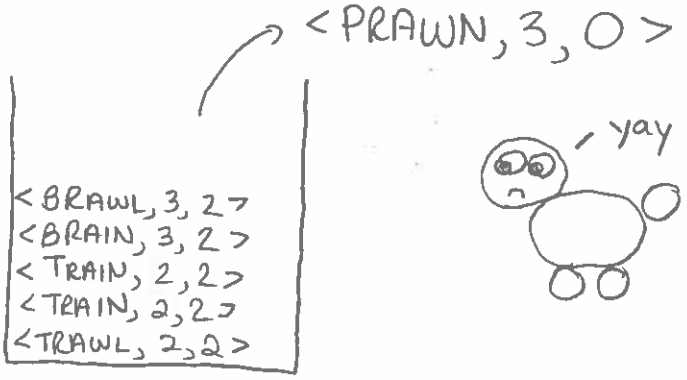
(7) Now we just repeat this process, getting a search node from the container, generating search nodes ("successors") for its neighbors, and putting them into the container.







(8) At some point, we get a final state from the container.



Thus we have found that there exists a solution of cost 3 for our word ladder.

---

(9) There are two important details:
- how do we decide which search node to get from the container?
- how do we compute the h-value for each search node?

---

(10) We'll let the container decide. It will be required to have methods .get() and .put(n), which "get" (and remove) a search node from the container (chosen by the container) and "put" a search node n into the container.

Uninformed Search

(11) How do we compute the h-value for each search node? We'll assume we have a heuristic function $H : Q \to \mathbb{R}$ that maps each state to an estimated cost. In our example, H computed the number of letters that were different from PRAWN, so $H(\text{BRAWL}) = 2$ and $H(\text{CLAPS}) = 4$.

(12) This gives us the following preliminary "master search algorithm":

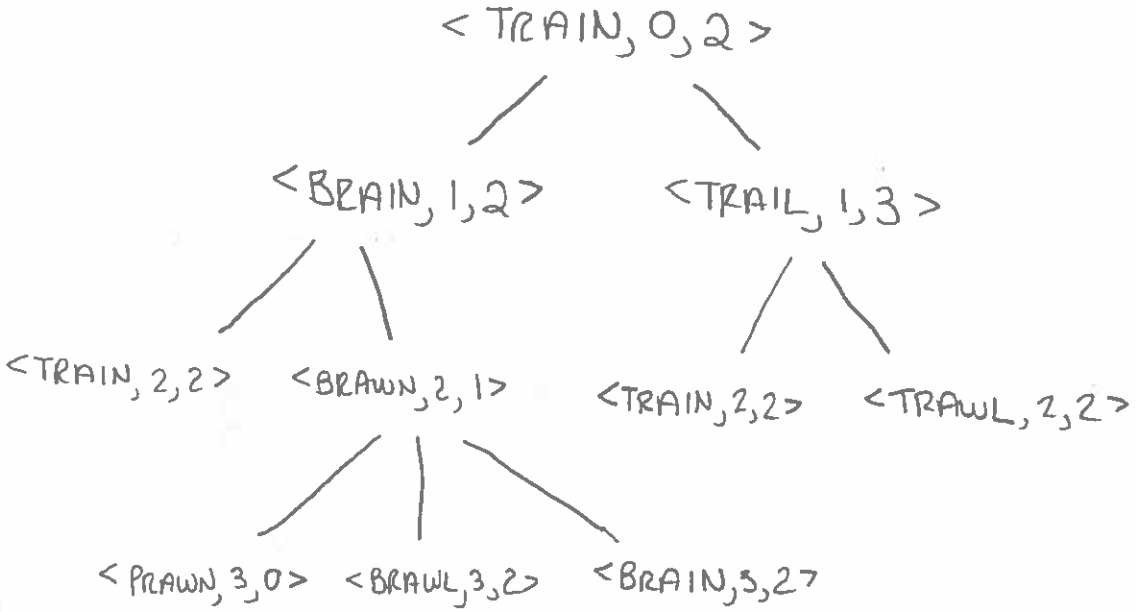
```
SEARCH(M = (Q, Σ, Δ, q0, F, w), H):
    container = new Container()
    container.put(<q0, 0, H(q0)>)
    repeat:
        if container.empty() return ∞
        n = container.get()
        if q(n) ∈ F return q(n)
        VISIT(n, container, M, H)

VISIT(n, container, M, H):
    for n' in successors_{M,H}(n):
        container.put(n')
```

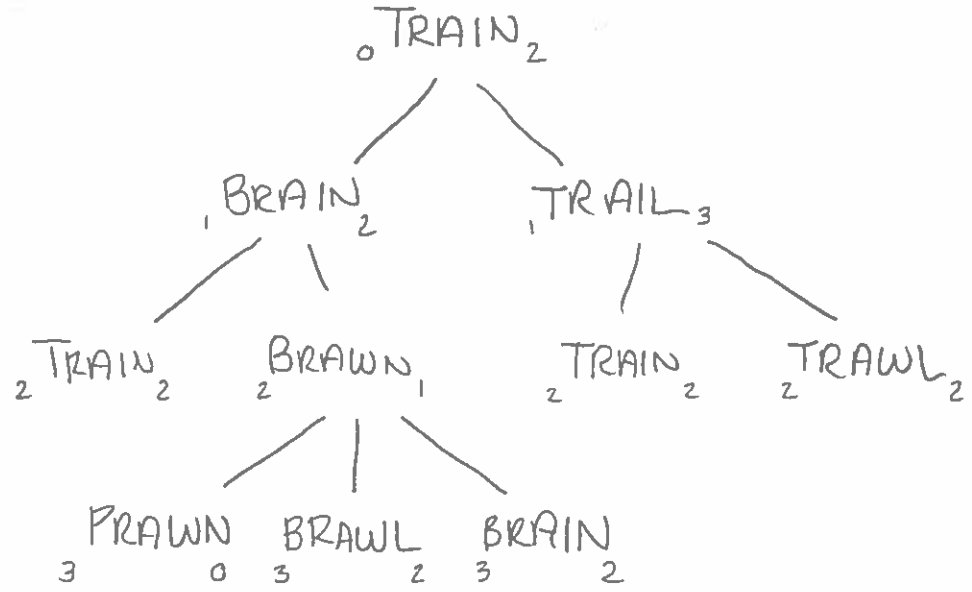
where $\text{successors}_{M,H}(\langle q, g, h \rangle) = \left\{ \langle q', g + w(q, \sigma, q'), H(q') \rangle \mid \sigma \in \Sigma, (q, \sigma, q') \in \Delta \right\}$

Uninformed Search

(13) This repeated process of "visiting" nodes and generating their successors can be visualized with a tree where the vertices are search nodes, and the children of a search node are its successors:



(14) To make this a little easier to write, let's write search node $\langle q, g, h \rangle$ as ${}_{g}q_{h}$ when we write the tree:



(15) Each search node corresponds to a search path of state machine M (see the handout STATE SPACES, (6)).

For instance, $_2$BRAWN$_1$ corresponds to the search path $\langle$(TRAIN, 1B, BRAIN), (BRAIN, 4W, BRAWN)$\rangle$.

The g-value of a search node is the cost of that search path, e.g.

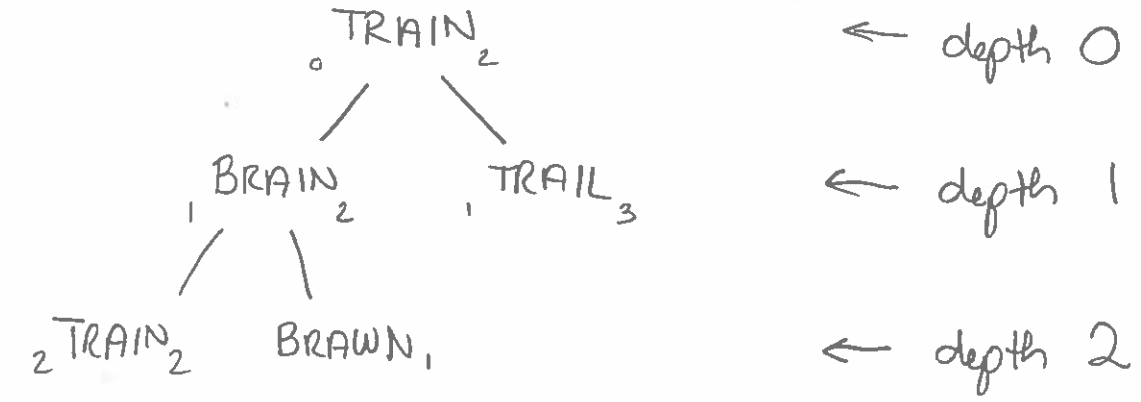
$$\begin{aligned} g(_1\text{BRAIN}_2) &= g(_0\text{TRAIN}_2) + w(\text{TRAIN}, 1\text{B}, \text{BRAIN}) \\ &= 0 + 1 \\ &= 1 \\ g(_2\text{BRAWN}_1) &= g(_1\text{BRAIN}_2) + w(\text{BRAIN}, 4\text{W}, \text{BRAWN}) \\ &= g(_0\text{TRAIN}_2) + w(\text{TRAIN}, 1\text{B}, \text{BRAIN}) + w(\text{BRAIN}, 4\text{W}, \text{BRAWN}) \\ &= 0 + 1 + 1 \\ &= 2 \end{aligned}$$

(16) If the g-value of a search node is a final state, then its search path is a solution to our search problem. We call these <u>goal nodes</u>.

In other words, we want to find a goal node with minimum g-value.

(17) When discussing search trees, some additional terms may be useful.

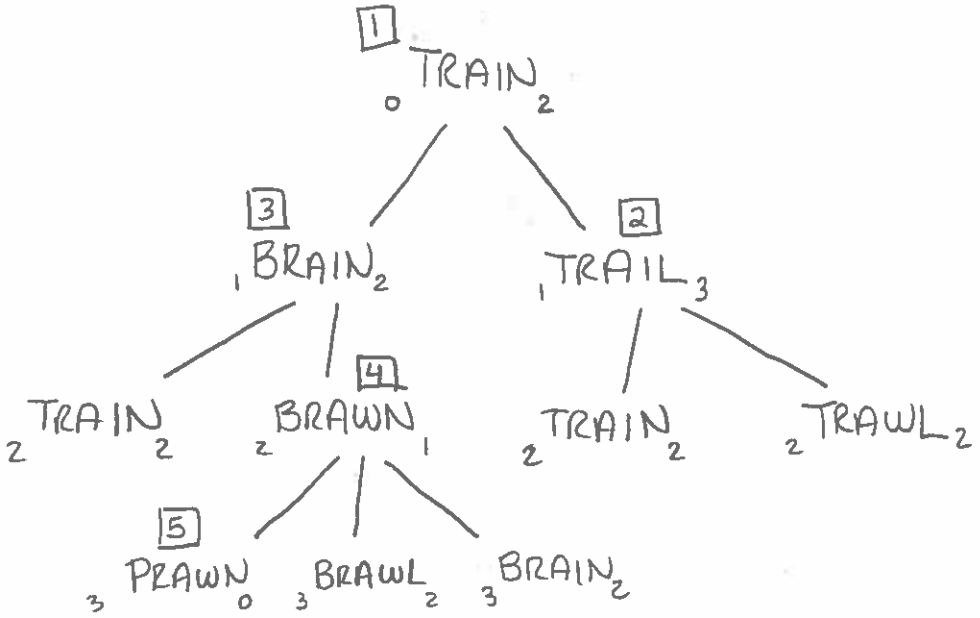
The search depth of a search node is its depth in the tree:



Another way to define search depth is the length of its search path.

Uninformed Search

⑱ In our first example, the container visited search nodes in a rather arbitrary order.

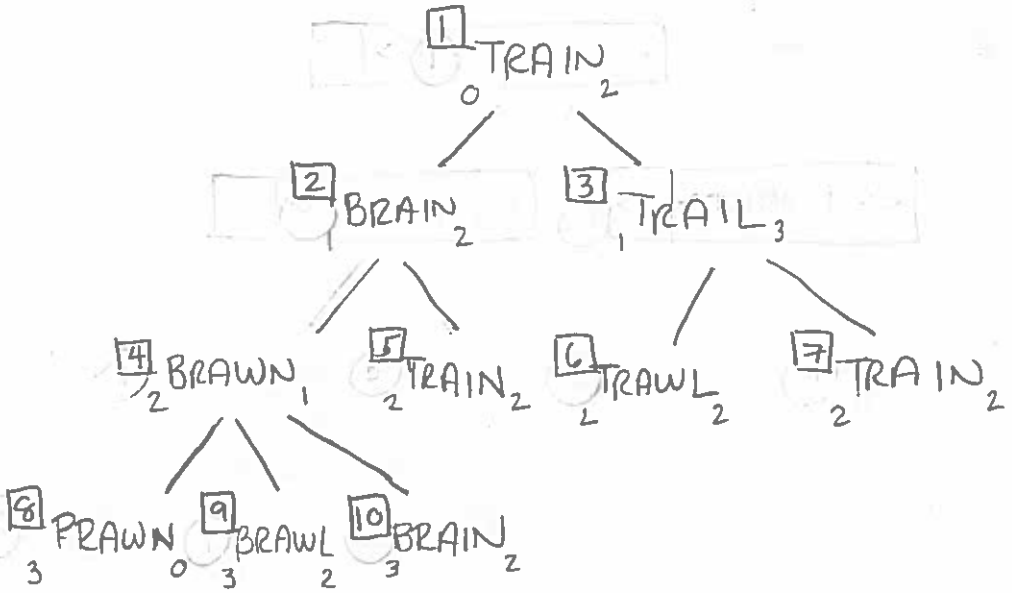


(19) Let's consider the behavior of SEARCH using different containers.

Suppose that we use a queue. Queues are "first in, first out", so we get the following behavior:

	Container
put <TRAIN,0,2>	<TRAIN,0,2>
get <TRAIN,0,2>	
put <BRAIN,1,2>, <TRAIL,1,3>	<BRAIN,1,2> <TRAIL,1,3>
get <BRAIN,1,2>	<TRAIL,1,3>
put <BRAWN,2,1>, <TRAIN,2,2>	<TRAIL,1,3> <BRAWN,2,1> <TRAIN,2,2>
get <TRAIL,1,3>	<BRAWN,2,1> <TRAIN,2,2>
put <TRAWL,2,2>, <TRAIN,2,2>	<BRAWN,2,1> <TRAIN,2,2> <TRAWL,2,2> <TRAIN,2,2>
get <BRAWN,2,1>	<TRAIN,2,2> <TRAWL,2,2> <TRAIN,2,2>
put <PRAWN,3,0>, <BRAWL,3,2>, <BRAIN,3,2>	etc.

(20) If we look at the search tree, this is the order in which states are processed:



We first visit all search nodes at depth 0 (i.e. $_0$TRAIN$_2$), then all search nodes at depth 1 (i.e. $_1$BRAIN$_2$ and $_1$TRAIL$_3$), then all search nodes at depth 2, etc.

This instantiation of the master SEARCH algorithm is called <u>breadth-first search</u> (BFS).

Uninformed Search

(21) BFS will find the optimal solution for word ladder because of the following property:

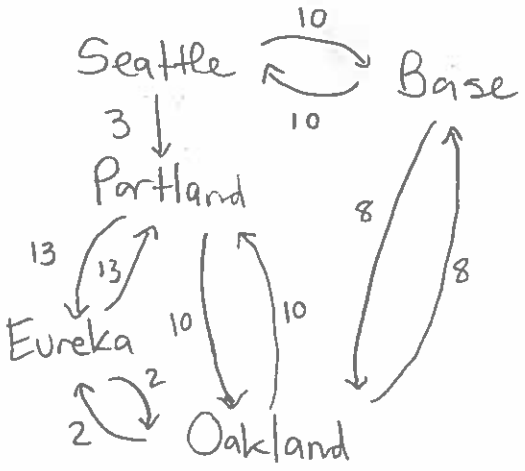
> if node n_1 is visited before node n_2, then $\text{depth}(n_1) \leq \text{depth}(n_2)$

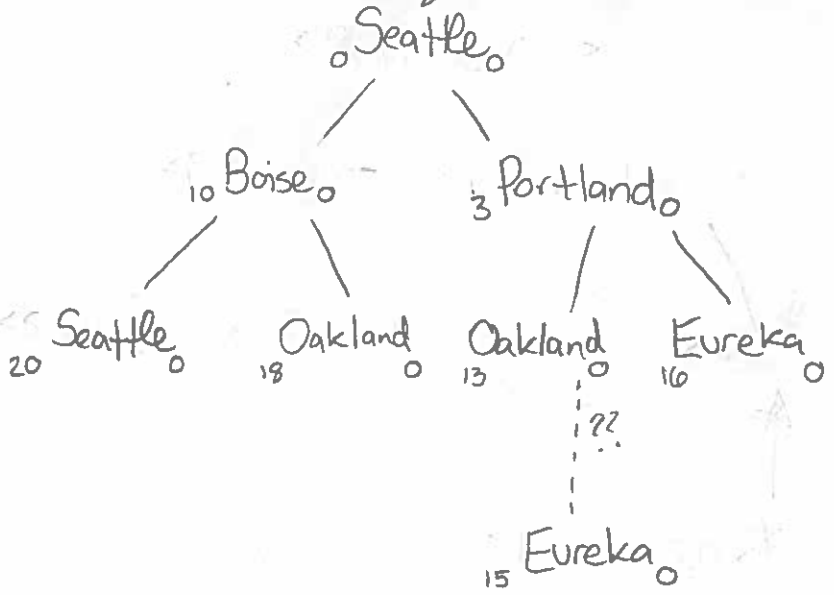
Because depth(n) is equal to its cost g(n) for word ladder, we know that the first goal node n we visit has minimal cost g(n).

Suppose otherwise, that there's a better goal node n' such that $g(n') < g(n)$. Thus $\text{depth}(n') < \text{depth}(n)$, so n' is visited by BFS before n — CONTRADICTION

---

(22) BFS will not necessarily find the optimal solution for state machines with non-uniform weights, like:

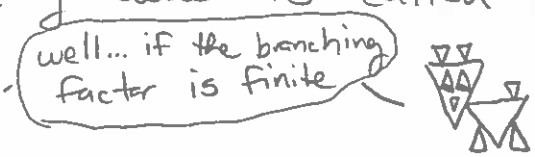




(23) How might we adjust the container so that it finds the optimal solution even for state machines with non-uniform weights? Instead of having a policy of "first in, first out", we could instead prioritize getting search nodes of lowest g-value (corresponding to search paths of lowest cost):

	Container
put $_0$Seattle$_0$	$_0$Seattle$_0$
get $_0$Seattle$_0$	
put $_3$Portland$_0$, $_{10}$Boise$_0$	$_3$Portland$_0$ $_{10}$Boise$_0$
get $_3$Portland$_0$	$_{10}$Boise$_0$
put $_{13}$Oakland$_0$, $_{16}$Eureka$_0$	$_{10}$Boise$_0$ $_{13}$Oakland$_0$ $_{16}$Eureka$_0$
get $_{10}$Boise$_0$	$_{13}$Oakland$_0$ $_{16}$Eureka$_0$
put $_{18}$Oakland$_0$, $_{20}$Seattle$_0$	$_{13}$Oakland$_0$ $_{16}$Eureka$_0$ $_{18}$Oakland$_0$ $_{20}$Seattle$_0$
get $_{13}$Oakland$_0$	$_{16}$Eureka$_0$ $_{18}$Oakland$_0$ $_{20}$Seattle$_0$
put $_{15}$Eureka$_0$, $_{21}$Boise$_0$	$_{16}$Eureka$_0$ $_{18}$Oakland$_0$ $_{20}$Seattle$_0$ $_{15}$Eureka$_0$ $_{21}$Boise$_0$
get $_{15}$Eureka$_0$	$_{16}$Eureka$_0$ $_{18}$Oakland$_0$ $_{20}$Seattle$_0$ $_{21}$Boise$_0$

(24) A queue that gets the next node based on some function of the node (like its g-value) is called a priority queue. The instantiation of the master SEARCH algorithm whose container is a priority queue prioritized by g-value is called Uniform Cost Search (UCS).



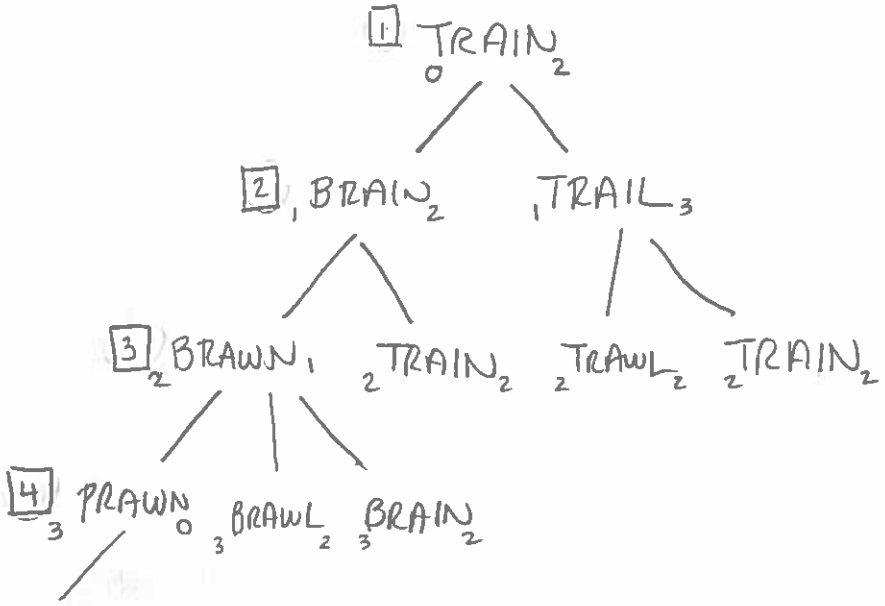
UCS is guaranteed to find the optimal solution for any state machine because of the following property:

if node n_1 is visited before node n_2,
then $g(n_1) \leq g(n_2)$

(25) We could alternatively use a stack as our container. Stacks are "last in, first out," so we get the following behavior:

	Container
put <TRAIN,0,2>	<TRAIN,0,2>
get <TRAIN,0,2>	
put <BRAIN,1,2>	<BRAIN,1,2> <TRAIL,1,3>
put <TRAIL,1,3>	
get <BRAIN,1,2>	<TRAIL,1,3>
put <BRAWN,2,1>	<BRAWN,2,1> <TRAIN,2,2> <TRAIL,1,3>
put <TRAIN,2,2>	
get <BRAWN,2,1>	<TRAIN,2,2> <TRAIL,1,3>
put <PRAWN,3,0>	
put <BRAWL,3,2>	<PRAWN,3,0> <BRAWL,3,2> <BRAIN,3,2> <TRAIN,2,2> <TRAIL,1,3>
put <BRAIN,3,2>	
get <PRAWN,3,0>	etc.

(26) This time, if we look at the search tree, we process states in a different order:

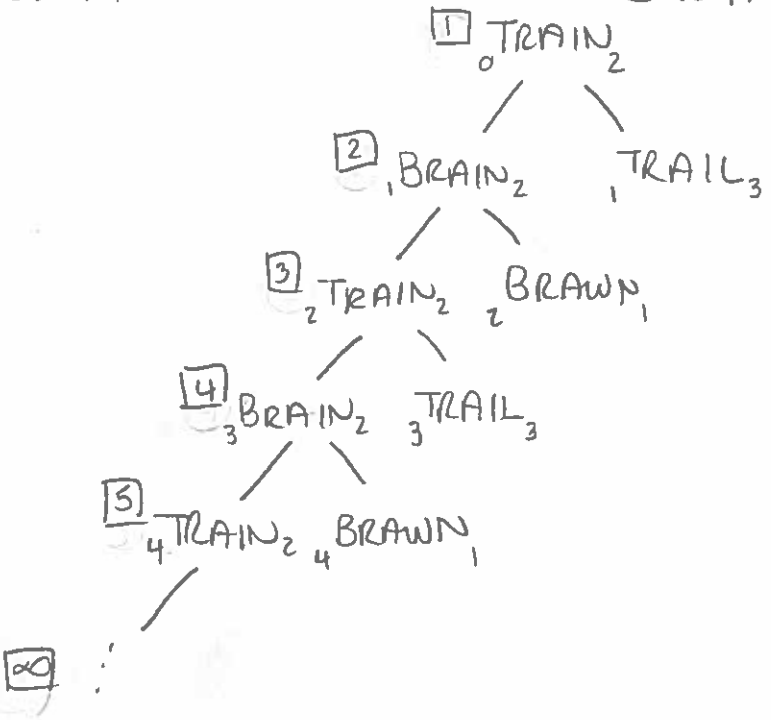


We dive headlong down a single path until we reach a final state or a dead-end (a state with no transition). If we hit a dead-end, we backtrack to the closest unvisited "cousin".

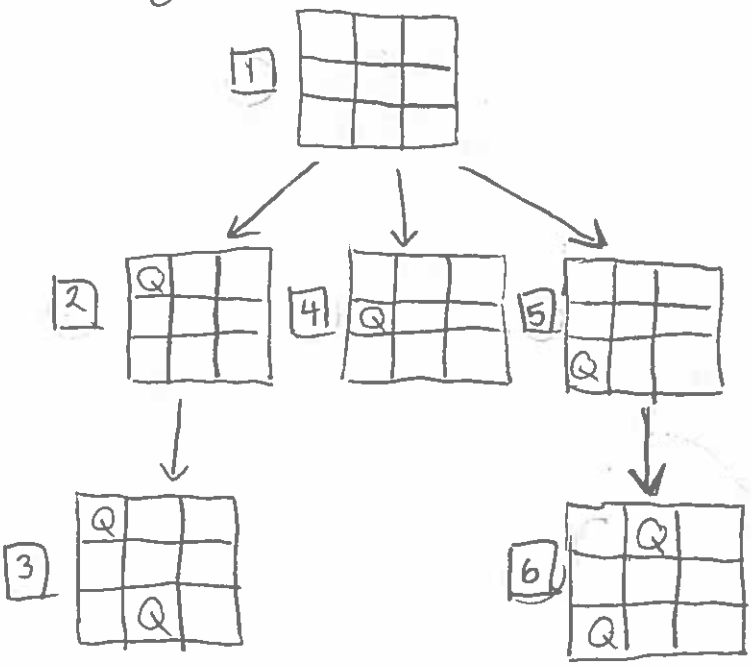
This instantiation of the master SEARCH algorithm is called depth-first search (DFS)

Uninformed Search

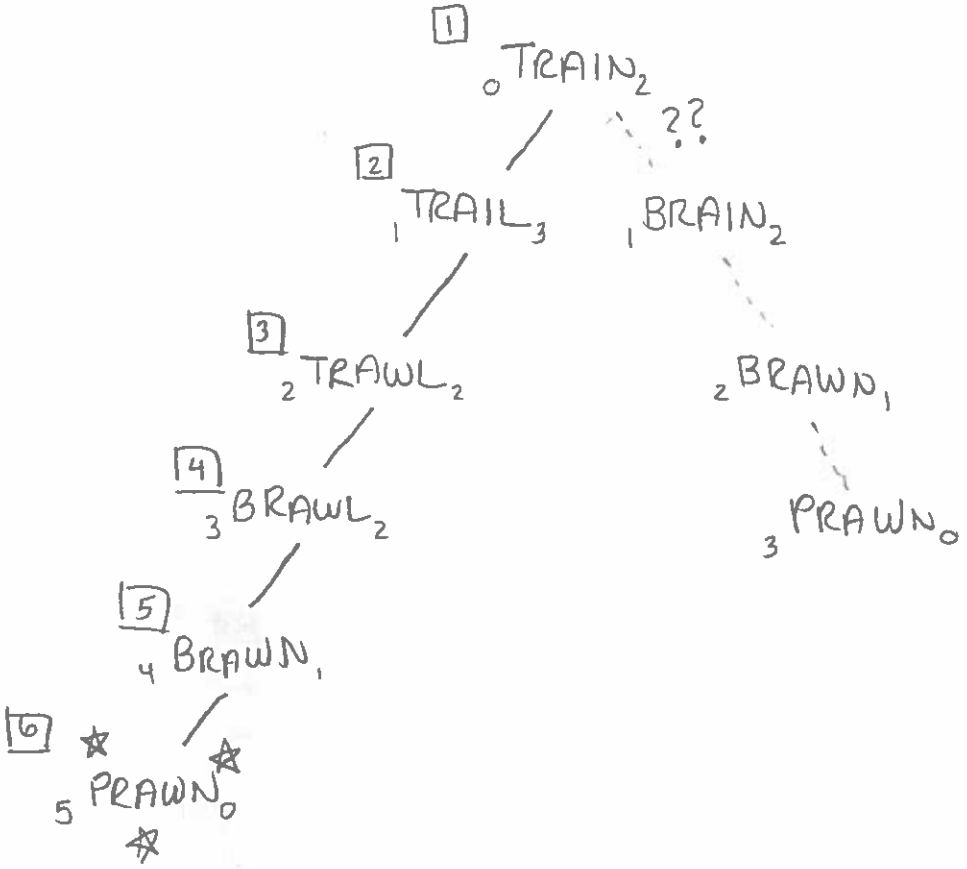
(27) If the state machine is cyclic, then DFS can loop forever and never find a solution



(28) However it will terminate eventually if the state machine is acyclic (and finite state).



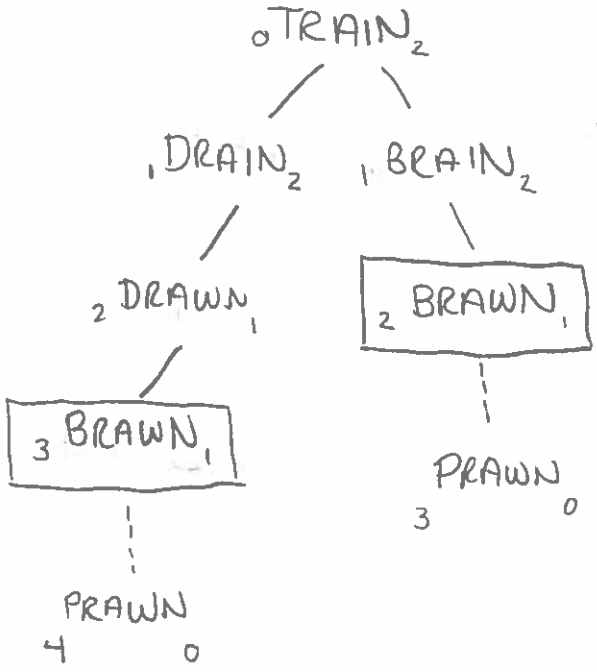
(29) But DFS is not guaranteed (even with uniform weights) to find an optimal solution:



(30) Here's a summary of so-called "uninformed search" techniques:

	container	finds optimal solution?
BFS	queue	yes (with uniform weights)
DFS	stack	no
UCS	priority queue prioritized by g-value	yes

(31) One optimization of the master SEARCH algorithm is to record states that have already been processed. The observation is: no matter how we reached a particular search node, the optimal search path from its state to a final state is the same as any other search node with the same state:



(32) So if we don't find a goal node after visiting the state <u>once</u>, it won't help to visit it again.

(33) This suggests a slightly more sophisticated version of master SEARCH algorithm, which "memoizes" (records) the states it has already visited:

```
MEMOIZEDSEARCH(M=(Q,Σ,Δ,q0,F,ω), H):
    container = new Container()
    container.put(<q0, 0, H(q0)>)
    visited = {}
    repeat:
        if container.empty() return ∞
        n = container.get()
        if q(n) ∈ F return g(n)
        VISIT(n, container, visited, M, H)

VISIT(n, container, visited, M, H):
    visited.add(n)
    for n' in successors_{M,H}(n):
        if n' ∉ visited:
            container.put(n')
```

We do, however, need to be a bit careful with this optimization, as it may affect optimality of certain algorithms (as we will see). However, the optimality of BFS and UCS are unaffected by this optimization.