Analysis of Search

① Different instantiations of SEARCH, like BFS and DFS, have different running times and use different amounts of memory (time and space complexity, respectively). Usually, when these algorithms are analyzed in a theory class, they are analyzed in terms of the number of nodes in the state graph.

This analysis is not particularly useful for most AI applications, since almost any interesting state graph has an enormous and intractible number of states.

② Instead, we analyze the complexity in terms of three properties of the state machine:

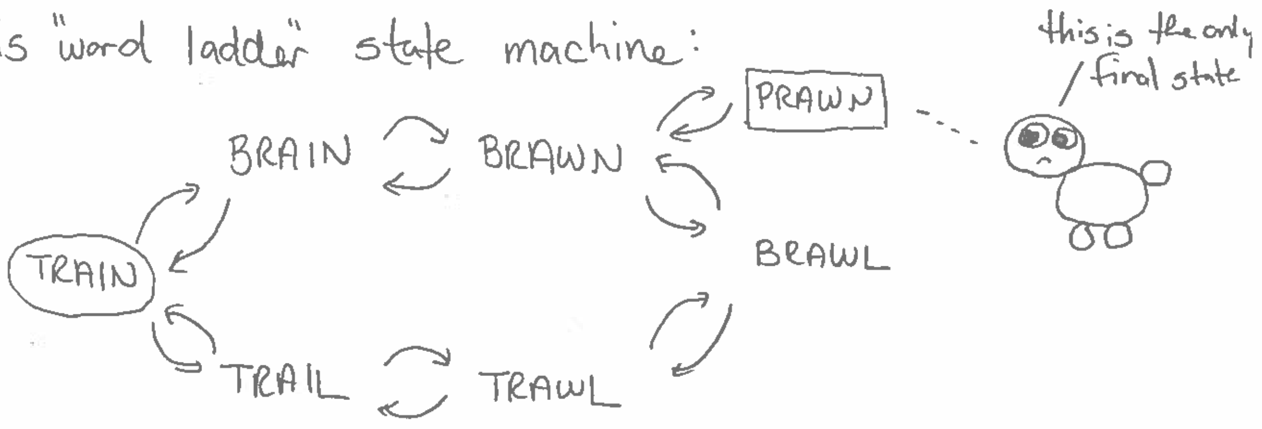
- branching factor: This is the maximum number of states that can be reached from an arbitrary state using a single transition:

$$\max_q \left| \{ q' \mid (q, \sigma, q') \in \Delta(M), \sigma \in \Sigma(M) \} \right|$$

- maximum depth: This is the length of the longest search path from the initial state to any other state. In a cyclic graph, this is often infinite.
- solution depth: This is the length of the shortest search path from the initial state to a final state.

ANALYSIS OF SEARCH

3) Consider this "word ladder" state machine:



What is the branching factor? 3.

$$\begin{tikzcd} & & \text{PRAWN} \; \boxed{2} \\ \overset{\boxed{1}}{\text{BRAIN}} & \text{BRAWN} \arrow[l] \arrow[ur] \arrow[dr] & \\ & & \underset{\boxed{3}}{\text{BRAWL}} \end{tikzcd}$$

What is the maximum depth? ∞.

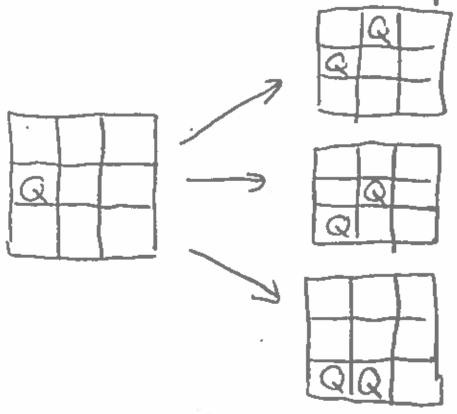
$$\text{TRAIN} \rightarrow \text{BRAIN} \rightarrow \text{TRAIN} \rightarrow \text{BRAIN} \rightarrow \text{TRAIN} \rightarrow \cdots$$

What is the solution depth? 3

$$\text{TRAIN} \xrightarrow{\boxed{1}} \text{BRAIN} \xrightarrow{\boxed{2}} \text{BRAWN} \xrightarrow{\boxed{3}} \text{PRAWN}$$

ANALYSIS OF SEARCH

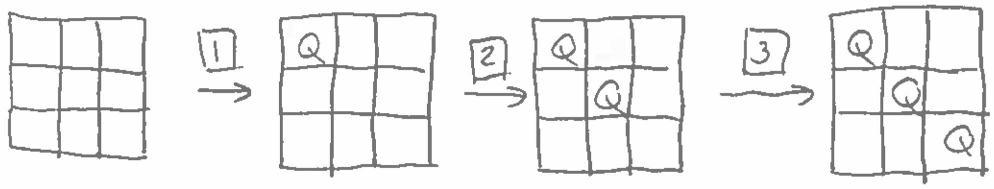
4) Consider the n-queens state machine in which we add a queen to the leftmost empty column.



What is the branching factor? n.

(because the board is nxn, so there are at most n positions per column)

What is the maximum depth? n.



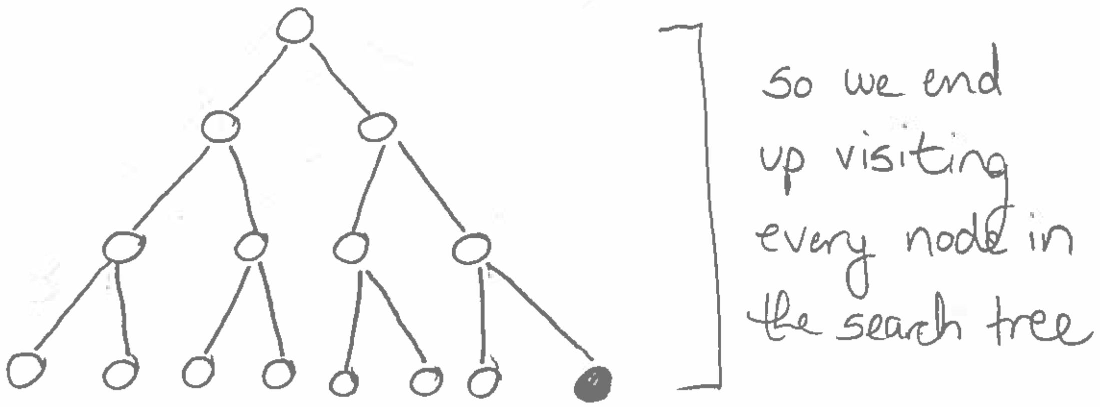
What is the solution depth? n.

ANALYSIS OF SEARCH

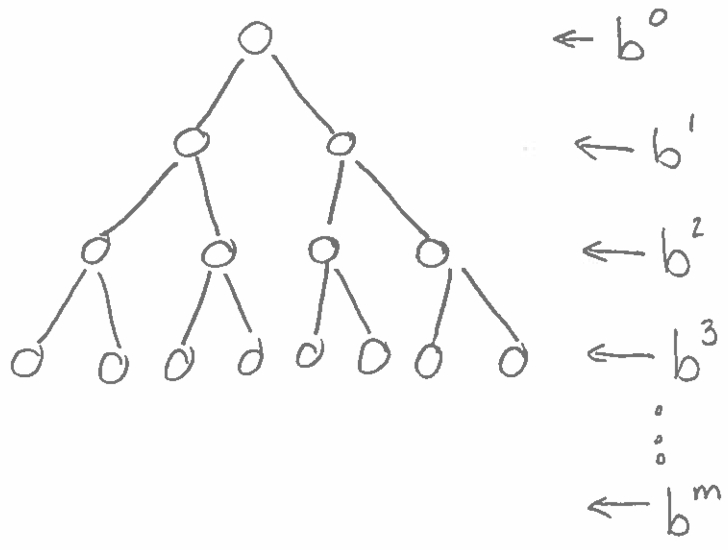
5) The time complexity of BFS and DFS are relatively straightforward to analyze.

For DFS, if the state machine is cyclic (and we don't memoize), it may just never terminate.

If the state machine is acyclic, then in the worst case, the only goal node will be the last node we visit:

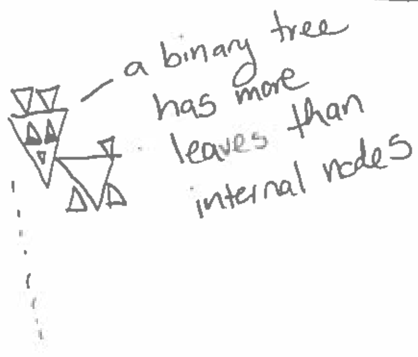


How many nodes are in a tree with branching factor b and maximum depth m?



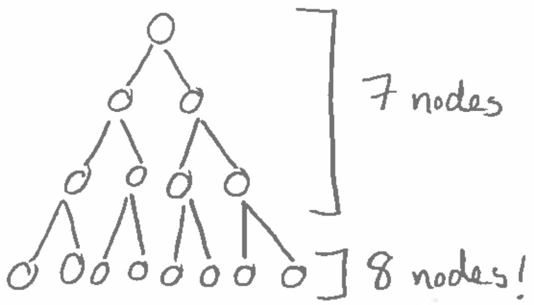
⑥ So there are at most this many nodes:

$$\sum_{i=0}^{m} b^i$$



Interestingly, if $b \geq 2$, then

$$b^m \geq \sum_{i=0}^{m-1} b^{m-1}$$



So $\sum_{i=0}^{m} b^i \leq 2 \cdot b^m$

Thus, assuming it takes a constant amount of time to visit a search node, the running time of DFS takes $O(b^m)$.

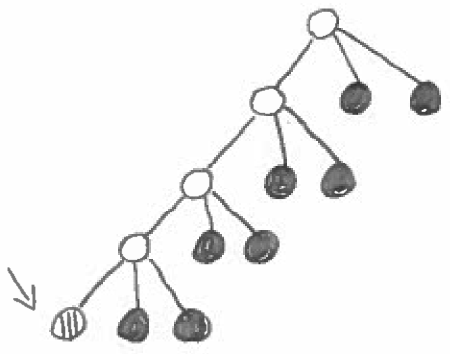
⑦ The running time of BFS is similar to DFS, but benefits from the optimality of BFS:

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

So if there's a goal node at depth d, then BFS will never visit nodes of depth $d+1$ or more. So the running time of BFS is the number of nodes up to solution depth d, which is $O(b^d)$.

Analysis of Search

(8) To analyze the space complexity of DFS and BFS, we just need to figure out how full the container (e.g. the stack or the queue) can get. For DFS, consider the container when it visits a search node at depth k:



$$\left.\begin{array}{r} 2 \\ +2 \\ +2 \\ +2 \\ \hline 8 \end{array}\right] k(b-1) \quad = 4 \cdot 2 = 8$$

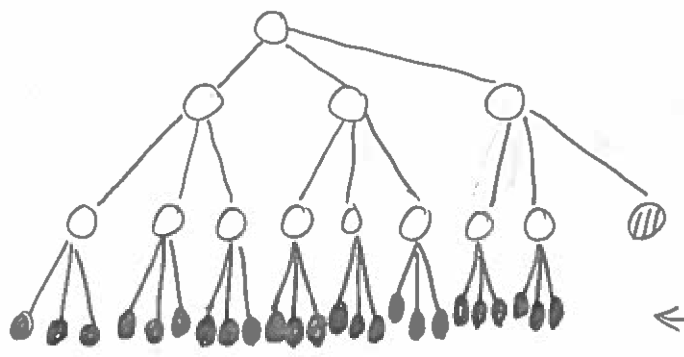
At this point, the container contains all the filled-in circles, i.e. all the successors of the current node's ancestors. There are $\leq k \cdot b$ of these.

Assuming the state machine has a maximum search depth m, then there will be at most $m \cdot b$ nodes in the container at any point.

So the space complexity is $O(m \cdot b)$.

ANALYSIS OF SEARCH

⑨ For BFS, consider the container when we find the goal node of least depth:



$\leftarrow b^{d+1} - b$ nodes

$= 3^{2+1} - 3$

$= 24$

It could contain almost all the nodes at depth $d+1$, which is $O(b^{d+1})$ nodes.

---

⑩ This is actually really bad news.

Consider the formulation of 8-queens where each action adds a queen to the leftmost column. This has a branching factor of 8, and the goal nodes all have depth 8, so at some point BFS will have $O(8^9)$ nodes in its container. This is

1,073,741,824

nodes. Say your computer has 4GB of RAM. This is

4,294,967,296

bytes.

(11) Maybe you could get away with running BFS on 8-queens (if you somehow manage to use less than 4 bytes per search node, and don't have anything else running on your computer).

But for 10-queens, BFS will have $O(10^{11})$ nodes in its container at some point, which would require 100GB of RAM (assuming only 1 byte per node!)

20-queens is beyond imagining.

40-queens starts to get in the realm of number of atoms in the universe.

---

(12) Because you can fill up a hard drive in much less time than an average human life span, space complexity is typically a bigger problem than time complexity.

---

(13) But BFS was so good up until now!

	finds optimal solution?	time complexity	space complexity
BFS	yes, with unit weights	$O(b^d)$	$O(b^{d+1})$ ☹!
DFS	no	$O(b^m)$	$O(m \cdot b)$

ANALYSIS OF SEARCH

---

(14) In practice, we can't use BFS on large-scale search problems. We need a better, more space-efficient technique.

DFS is space efficient, but not optimal. Can we modify it to make it optimal (while keeping the space complexity)?

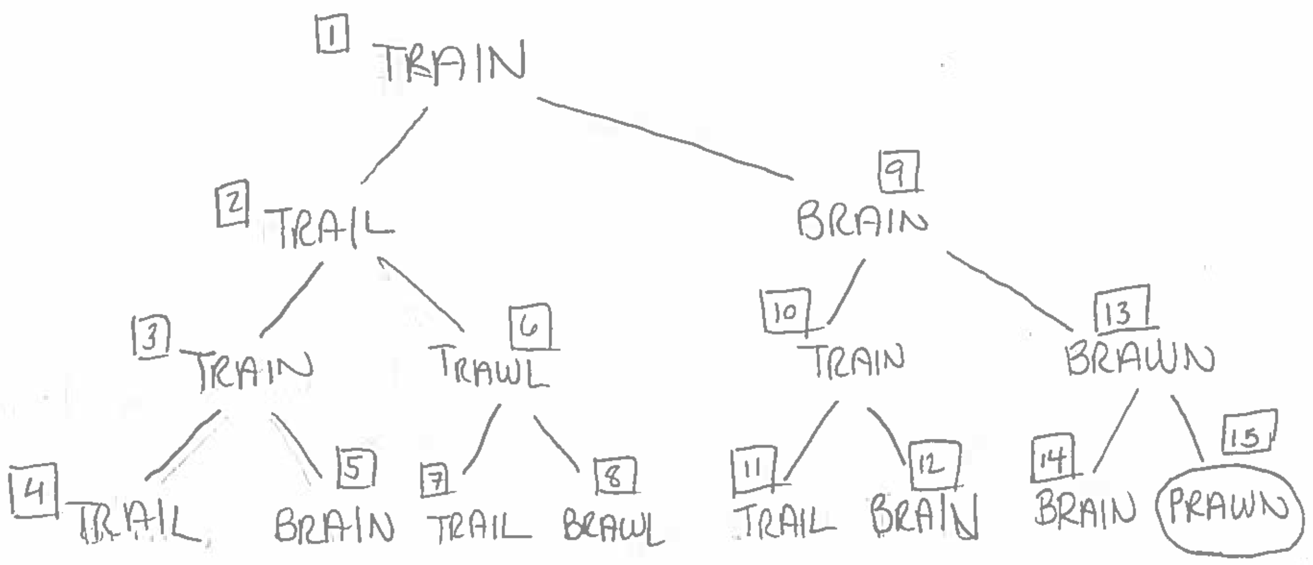
---

(15) Suppose we knew the solution depth d before we ran SEARCH. We could optimize it by discarding any search nodes of g-value > d:

$M = (Q, \Sigma, \Delta, q_0, F, w)$

```
LIMITEDSEARCH (M, H, $\boxed{d}$):
    container = new Container()
    container.put(<$q_0$, 0, $H(q_0)$>)
    repeat:
        if container.empty() return $\infty$
        n = container.get()
        if $q(n) \in F$ return $g(n)$
        VISIT(n, container, M, H, $\boxed{d}$)
VISIT(n, container, M, H, $\boxed{d}$):
    for <q, g, h> $\in$ successors$_{M,H}$(n):
        $\boxed{\text{if } g \leq d:}$
            container.put(<q, g, h>)
```

(16) Surprisingly, this small change allows us to use a stack as our container (and get its linear space complexity) <u>and</u> get optimality (assuming unit weights):



Optimality is assured because any solution of greater cost (i.e. greater search depth) is never added to the container.

(17) This modification (LIMITEDSEARCH using a stack as the container) is called <u>depth-limited search</u>.

It is perfect for the n-queens problem, because all solutions have depth n (and we know this in advance).

18 But we don't always know the solution depth in advance. Consider word ladder (and ignore the fact that often these puzzles do indeed indicate their solution depth).

What can we do? How about:

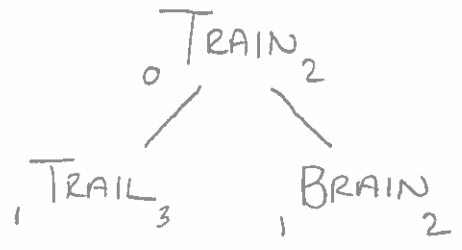
```
ITERATIVEDEEPENINGSEARCH(M, H):
    d = 0
    repeat:
        cost = LIMITEDSEARCH(M, H, d)
        if cost ≠ ∞ return cost
        d = d + 1
```

19 First we use depth-limited search to see if there's a goal node at depth 0:

$_0$TRAIN$_2$
nope

There isn't.

Then we start over, and use depth-limited search to see if there's a goal node at depth 1:



There isn't.

20) Eventually we call depth-limited search at maximum depth 3 and get the optimal solution. We know its optimal because if there was a goal node at an earlier search depth, we would have found it when we called LIMITEDSEARCH with that maximum depth.

21) The space complexity is the same as DFS, i.e. $O(m \cdot b)$. Actually, it's even better really — it's $O(d \cdot b)$, where d is the solution depth.

But what about the time complexity? We run DFS exhaustively $d+1$ times. Isn't that bad?

22) Well, let's see. Each call to LIMITEDSEARCH with maximum depth d visits at most b^d nodes. So in total, ITERATIVEDEEPENING visits:

$$b^0 + b^1 + b^2 + \ldots + b^d \text{ nodes}$$

where b is the solution depth.

(23) But recall (from (6)) that if $b \geq 2$, then:

$$\sum_{i=0}^{d} b^i \leq 2 \cdot b^d$$

So ITERATIVEDEEPENING visits at most $2 \cdot b^d$ search nodes in total. Assuming each node can be visited in a constant amount of time, the running time is $O(b^d)$ which is the <u>same</u> as BFS.

(24) So we can get the best of both worlds:

	finds optimal solution?	time	space
BFS	yes, with unit weights	$O(b^d)$	$O(b^{d+1})$
DFS	no	$O(b^m)$	$O(m \cdot b)$
IDS	yes, with unit weights	$O(b^d)$	$O(d \cdot b)$