Artificial Intelligence

CS4365 --- Spring 2018

Uninformed Search

Reading: Sections 3.1-3.4, R&N

Problem Solving as Search

Search is a central topic in AI

— Originated with Newell and Simon's work on problem solving. Famous book: "Human Problem Solving" (1972)

— Automated reasoning is a natural search task

— More recently: Given that almost all AI formalisms (planning, learning, etc.) are NP-complete or worse, some form of search is generally **unavoidable (no "smarter" algorithm available)**.

Defining a Search Problem

State space – described by

- **initial state** – starting state
- **actions** – possible actions available
- **successor function; operators** – given a particular state *x*, returns a set of *<action, successor>* pairs

A **path** is any sequence of states connected by a sequence of actions.

Goal test – determines whether a given state is a goal state.

Path cost – function that assigns a cost to a path; relevant if more than one path leads to the goal, and we want the shortest path.

Assumption: cost of a path is the sum of the costs of the individual actions along the path; sum of the **step costs**, which must be non-negative.

The 8-Puzzle

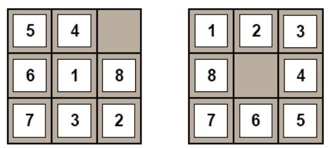
States:

Initial state:

Goal test:

Successor function:

Path cost:



Start State Goal State

Cryptarithmetic

```
   SEND
 + MORE
 ------
  MONEY
```

Find substitution of digits for letters such
that the resulting sum is arithmetically correct.

Each letter must stand for a different digit.

Cryptarithmetic, cont.

States: an 8-tuple indicating a (partial) assignment of digits to letters.

Successor function: represents the act of assigning digits to letters.

Goal test: all letters have been assigned digits and sum is correct.

Path cost: ...all solutions are equally valid; step cost = 0.

Solving a Search Problem: State Space Search

Input:

- Initial state
- Goal test
- Successor function
- Path cost function

Output: path from initial state to goal. Solution quality is measured by the path cost function.

State space is **not** stored in its entirety by the computer.

Generic Search Algorithm

```
L = make-queue(initial-state)
loop
  node = remove-front(L) (and save in order
     to return as part of path to goal)
  if goal-test(node) = true return(path to goal)
  S = successors(node)
  insert(S,L)
until L is empty
return failure
```

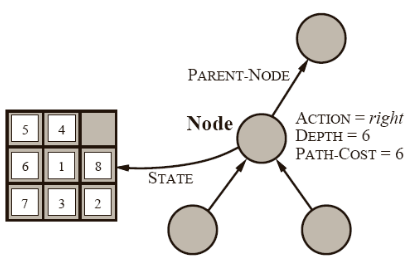

Search procedure defines a search tree

- **root node** — initial state
- **children of a node** — successor states
- **fringe of tree** — L: states not yet expanded

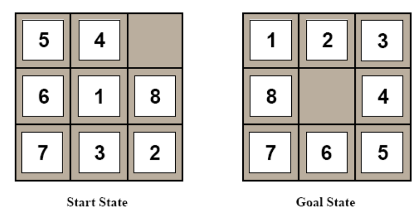
stack: Depth-First Search (DFS).
queue: Breadth-First Search (BFS).

Search strategy — algorithm for deciding which leaf node to expand next.

Node Data Structure



Solving the 8-Puzzle



What would the search tree look like after the start state was expanded?

Evaluating a Search Strategy

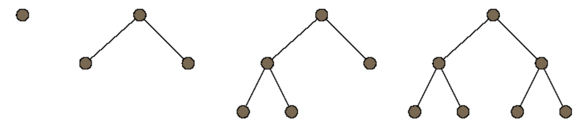
Completeness: is the strategy guaranteed to find a solution when there is one?

Time Complexity: how long does it take to find a solution?

Space Complexity: how much memory does it need?

Optimality: does the strategy find the highest-quality solution when there are several different solutions?

Uninformed Search: BFS

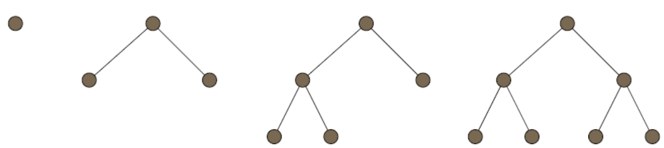


Consider paths of length 1, then of length 2, then of length 3, then of length 4,....

Time and Memory Requirements for BFS – $O(b^{d+1})$

Let b = branching factor, d = solution depth, then the maximum number of nodes *generated* is:

$b + b^2 + ... + b^d + (b^{d+1} - b)$



Time and Memory Requirements for BFS – $O(b^{d+1})$

b = 10

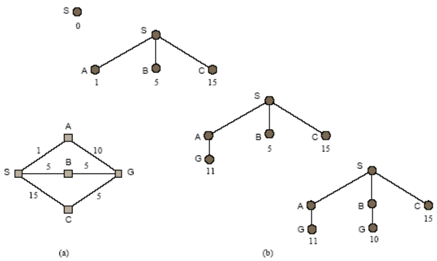
10000 nodes/second

each node requires 1000 bytes of storage

depth	nodes	time	memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	10^7	19 min	10 gig
8	10^9	31 hrs	1 tera
10	10^{11}	129 days	101 tera
12	10^{13}	35 yrs	10 peta
14	10^{15}	3523 yrs	1 exa

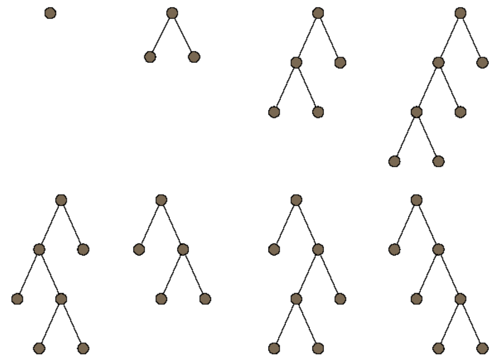
Uniform-Cost Search

Use BFS, but always expand the lowest-cost node on the fringe as measured by path cost $g(n)$.



$g(\text{Successor}(n)) > g(n)$ is a necessary condition for completeness and a sufficient condition for optimality

Uninformed search: DFS



DFS vs. BFS

	Complete?	Optimal?	Time	Space
BFS	YES	YES	$O(b^{d+1})$	$O(b^{d+1})$
DFS	finite depth	NO	$O(b^m)$	$O(bm)$

m is maximum depth

Time

$m = d$ — DFS typically wins

$m > d$ — BFS might win

m **is infinite** — BFS probably will do better

Space

DFS almost always beats BFS

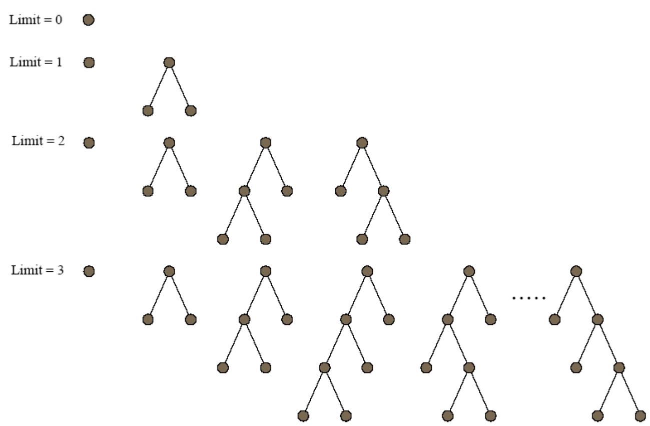
Iterative Deepening [Korf 1985]

Idea:
Use an *artificial* depth cutoff, *c*.

If search to depth *c* succeeds, we're done. If not, increase *c* by 1 and start over.

Each iteration searches using DFS.

Iterative Deepening



Space requirements? Same as DFS. Each search is just a DFS.

Time requirements. Would seem very expensive!! **BUT** not much different from single BFS or DFS to depth *d*.

Reason: Almost all work is in the final couple of layers. E.g., binary tree: 1/2 of the nodes are in the bottom layer. With b=10, 9/10th of the nodes in final layer!

So, repeated runs are on much smaller trees (i.e., exponentially smaller).

Example:
b=10, d=5, the number of nodes generated in a BFS:
$b + b^2 + ... + b^d + b^{d+1} - b =$
10 + 100 + 1000 + 10,000 + 100,000 + 999,990 = 1,111,100

For IDS:
$(d)b + (d - 1)b^2 + ... + (1)b^d =$
50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

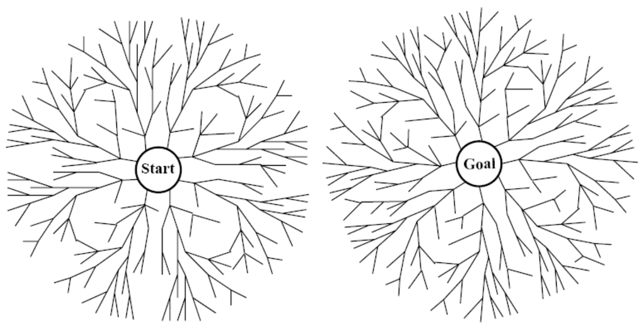
Cost of repeating the work at shallow depths is not prohibitive.

Cost of Iterative Deepening

space: $O(bd)$ as in DFS, **time:** $O(b^d)$

b	ratio of IDS to DFS
2	3
3	2
5	1.5
10	1.2
25	1.08
100	1.02

Bidirectional Search



- Search forward from the start state and backward from the goal state simultaneously and stop when the two searches meet the middle.
- If branching factor = b from both directions, and solution exists at depth d, then need only $O(2b^{d/2}) = O(b^{d/2})$ steps.
- Example b = 10, d = 6 then BFS needs 1,111,111 nodes and bidirectional search needs only 2,222.
  - What does it mean to search backwards from a goal?
  - What if there is more than one goal state? (chess).

Which search should I use?

Depends on the problem.

If there may be infinite paths, then depth-first is probably bad. If goal is at a known depth, then depth-first is good.

If there is a large (possibly infinite) branching factor, then breadth-first is probably bad.

(Could try **nondeterministic** search. Expand an open node at random.)

Comparing Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Iterative Deepening	Bidirectional (if applicable)
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bd	$b^{d/2}$
Optimal?	yes	yes	no	yes	yes
Complete?	yes	yes	no	yes	yes

***Note that many of the “yes’s” above have caveats, which we discussed when covering each of the algorithms.