

Search I

Tuomas Sandholm
Carnegie Mellon University
Computer Science Department

[Read Russell & Norvig Chapter 3]

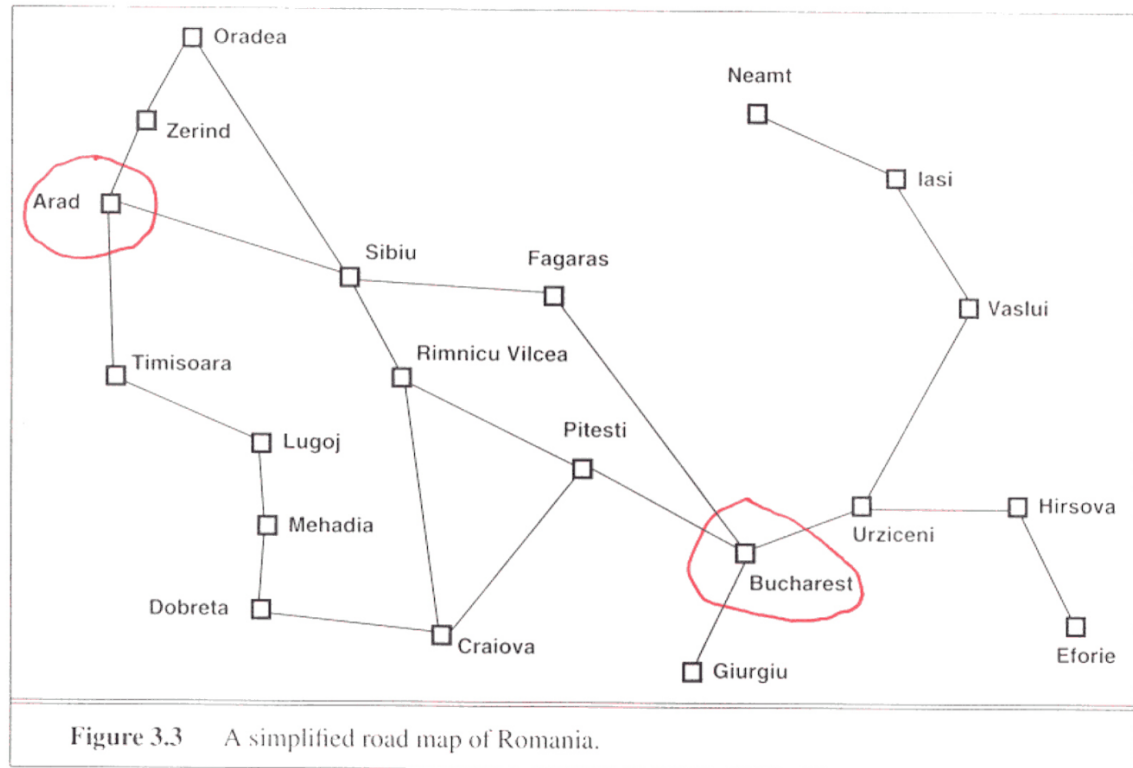
Search I

Goal-based agent (problem solving agent)

Goal formulation (from preferences). Romania example, (Arad → Bucharest)

Problem formulation: deciding what actions & state to consider.
E.g. not “move leg 2 degrees right.”

No map vs. Map
physical vs. deliberative
search search



Search I

“Formulate, Search, Execute” (sometimes interleave search & execution)

For now we assume

full observability = known state
known effects of actions

Data type *problem*

Initial state (perhaps an abstract characterization) \leftrightarrow partial observability (set)

Operators

Goal-test (maybe many goals)

Path-cost-function

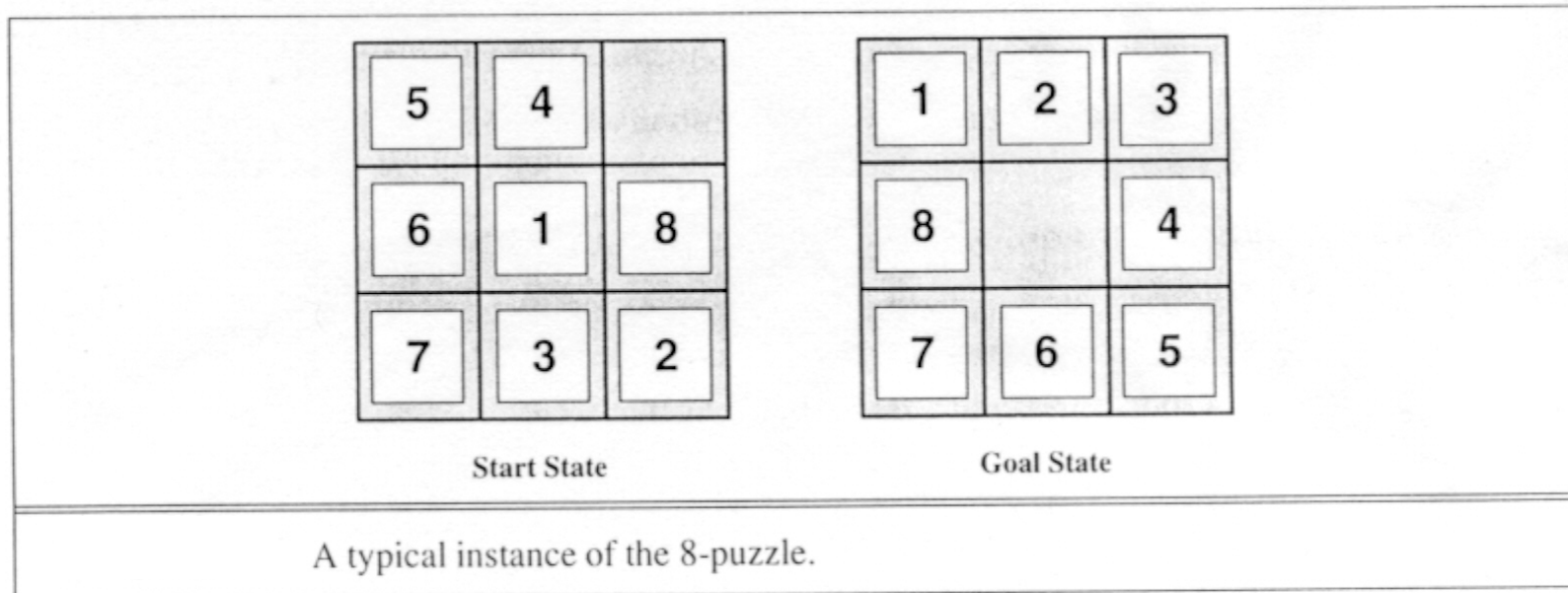
Knowledge representation

Mutilated chess board

Search I

Example problems demonstrated in terms of the problem definition.

I. 8-puzzle (general class is NP-complete)



How to model operators? (moving tiles vs. blank)

Path cost = 1

Search I

II. 8-queens (general class has efficient solution) path cost = 0

Incremental formulation:

(constructive search)

States: any arrangement of 0 to 8 queens on board

Ops: add a queen to any square

sequences = 64^8

Complete State formulation:

(iterative improvement)

States: arrangement of 8 queens, 1 in each column

Ops: move any attacked queen to another square in the same column

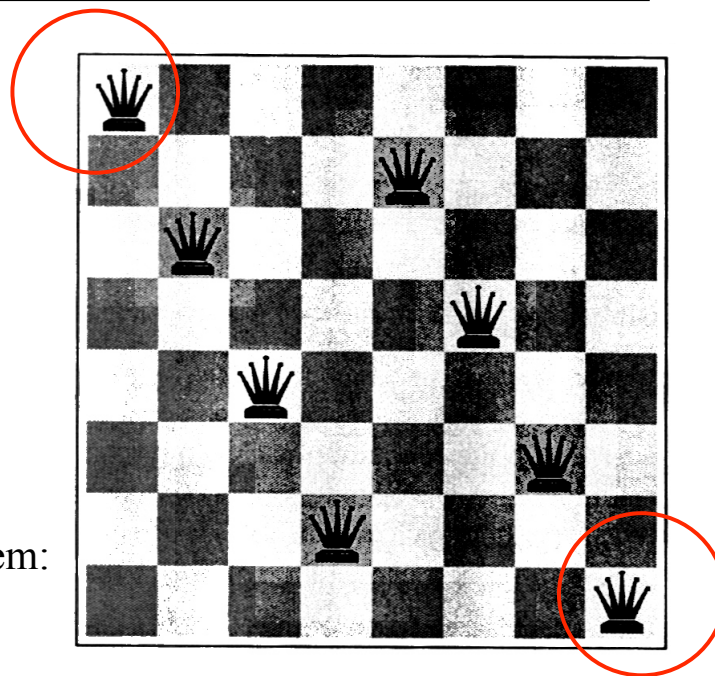
Improved incremental formulation:

States: any arrangement of 0 to 8 queens on board *with none attacked*

Ops: place a queen in the left-most empty column s.t. it is not attacked by any other queen

sequences = 2057

Almost a solution to the 8-queen problem:



Search I

III. Rubik' Cube $\sim 10^{19}$ states

IV. Crypt arithmetic

$$\begin{array}{r} \text{FORTY} \quad 29786 \\ + \quad \text{TEN} \quad + \quad 850 \\ + \quad \text{TEN} \quad + \quad 850 \\ \hline \text{SIXTY} \quad 31486 \end{array}$$

V. Real world problems

1. Routing (robots, vehicles, salesman)
2. Scheduling & sequencing
3. Layout (VLSI, Advertisement, Mobile phone link stations)
4. Winner determination in combinatorial auctions

...

Data type *node*

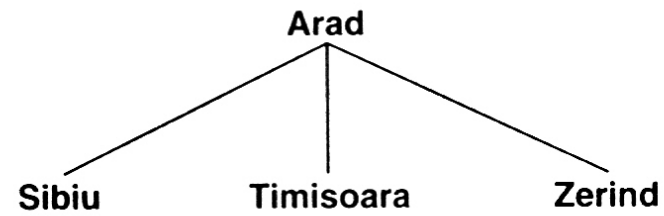
- State
- Parent-node
- Operator
- Depth
- Path-cost

Fringe = frontier = open (as queue)

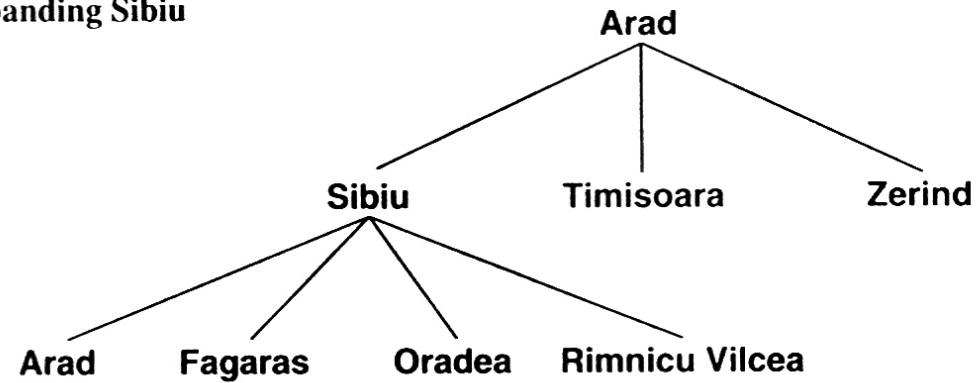
(a) The initial state

Arad

(b) After expanding Arad



(c) After expanding Sibiu



Partial search tree for route finding from Arad to Bucharest.

function GENERAL-SEARCH(*problem*, QUEUING-FN) **returns** a solution, or failure

nodes — MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))

loop do

if *nodes* is empty **then return** failure

node — REMOVE-FRONT(*nodes*)

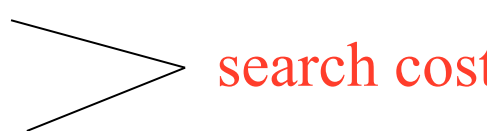
if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*

nodes — QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))

end

The general search algorithm. (Note that QUEUING-FN is a variable whose value will be a function.)

Goodness of a search strategy

- Completeness
 - Time complexity
 - Space complexity
- 
- search cost
- Optimality of the solution found
(path cost = domain cost)
 - Total cost = domain cost + search cost

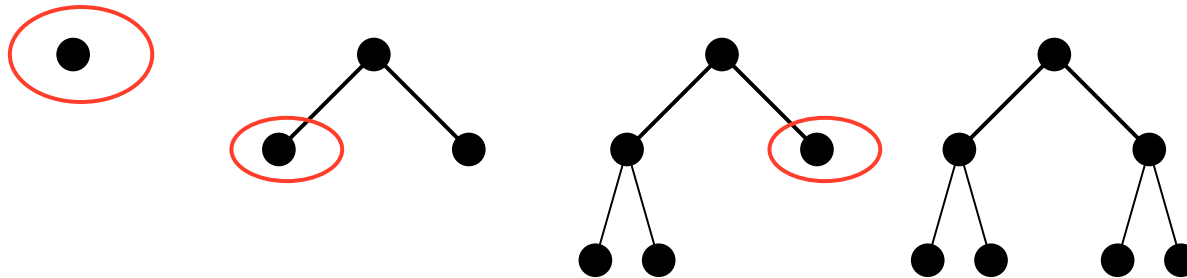
Uninformed vs. informed search



Can only distinguish goal states from non-goal state

Breadth-First Search

```
function BREADTH-FIRST-SEARCH (problem) returns a solution or failure  
return GENERAL-SEARCH (problem, ENQUEUE-AT-END)
```



Breadth-first search tree after 0,1,2 and 3 node expansions

Breadth-First Search ...

Max $1 + b + b^2 + \dots + b^d$ nodes (d is the depth of the shallowest goal)

- Complete
- Exponential time & memory $O(b^d)$
- Finds optimum if path-cost is a non-decreasing function of the depth of the node.

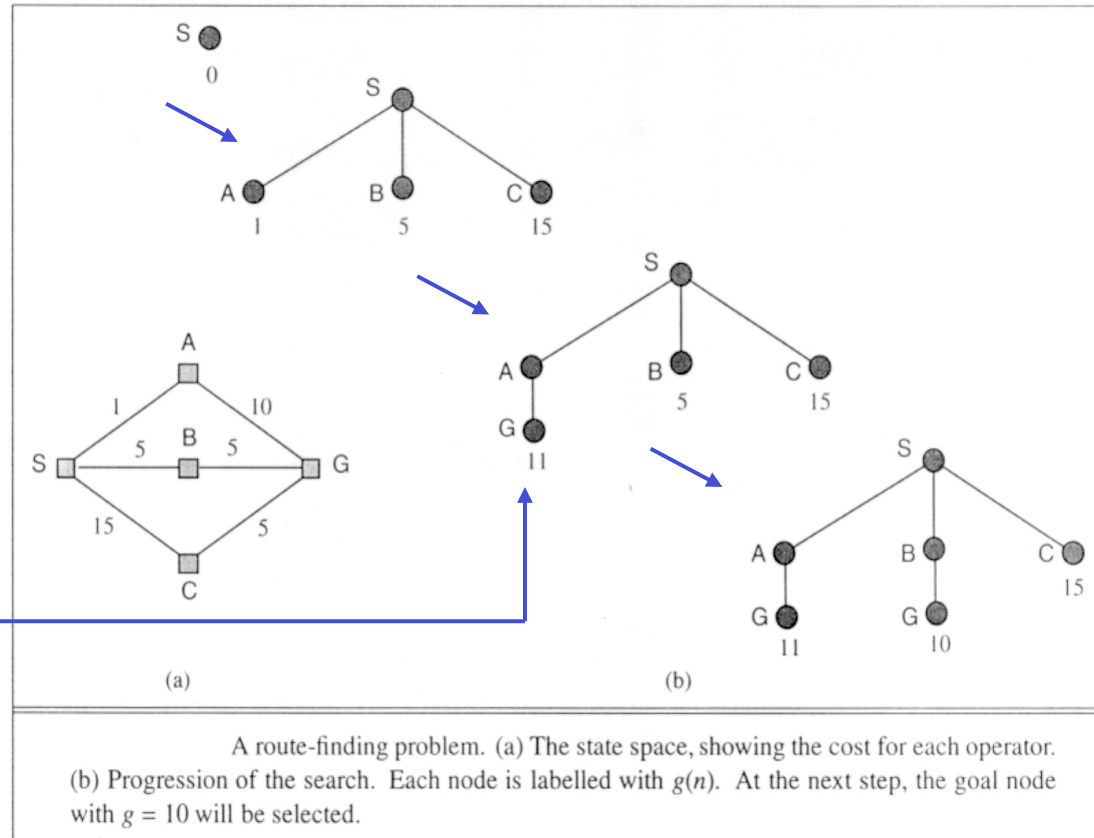
Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Time and memory requirements for breadth-first search. The figures shown assume branching factor $b = 10$; 1000 nodes/second; 100 bytes/node.

Uniform-Cost Search

Insert nodes onto open list in ascending order of $g(n)$.

G inserted into open list although it is a goal state. Otherwise cheapest path to a goal may not be found.



Finds optimum if the cost of a path never decreases as we go along the path.
 $g(\text{SUCCESSORS}(n)) \geq g(n)$

\Leftarrow Operator costs ≥ 0

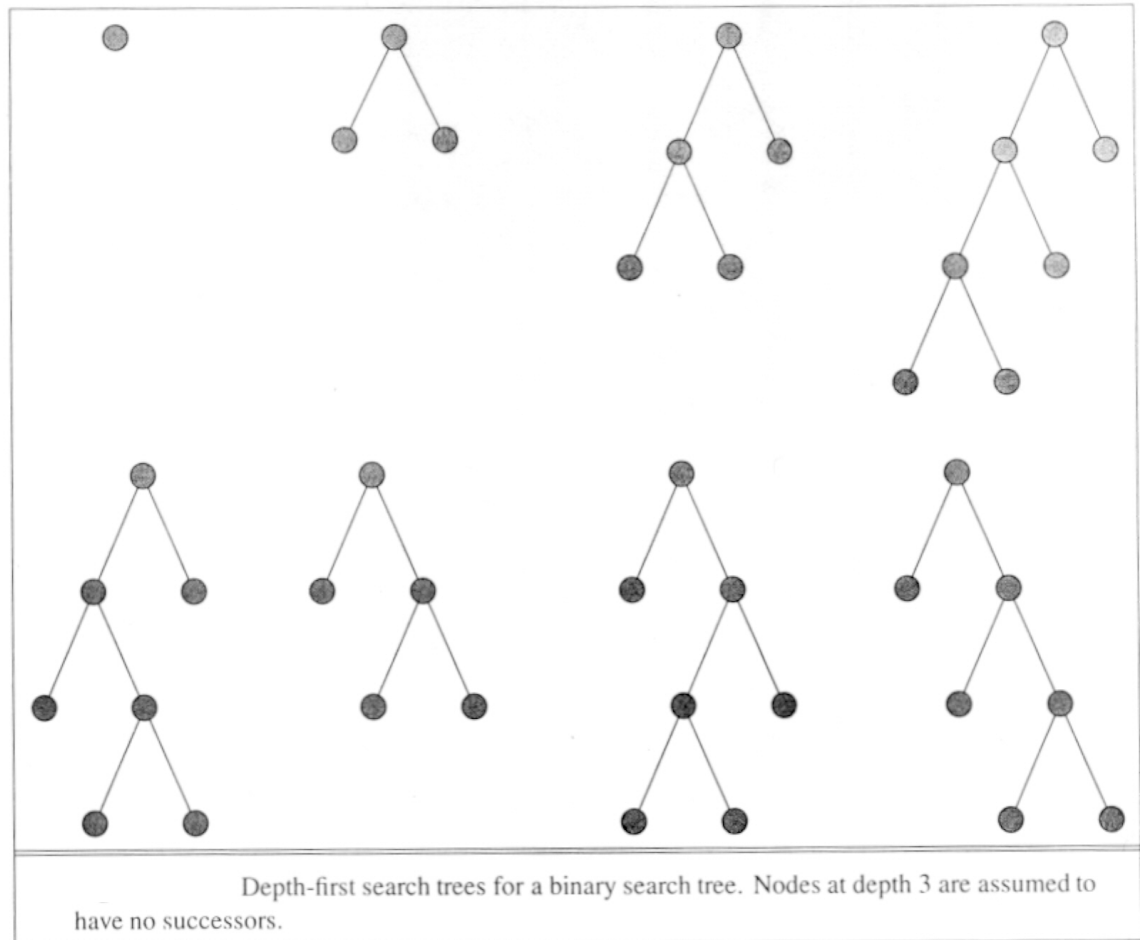
If this does not hold, nothing but an exhaustive search will find the optimal solution.

Depth-First Search

function DEPTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
GENERAL-SEARCH (*problem*, ENQUEUE-AT-FRONT)

Alternatively can
use a recursive
implementation.

- Time $O(b^m)$ (m is the max depth in the space)
- Space $O(bm)$!
- Not complete (m may be ∞)
 - E.g. grid search in one direction
- Not optimal



Depth-Limited Search

- Depth limit in the algorithm, or
- Operators that incorporate a depth limit

L = depth limit

Complete if $L \geq d$ (d is the depth of the shallowest goal)

Not optimal (even if one continues the search after the first solution has been found, because an optimal solution may not be within the depth limit L)

$O(b^L)$ time

$O(bL)$ space

Diameter of a search space?

Iterative Deepening Search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
  end
  return failure
```

Breadth first search :

$$1 + b + b^2 + \dots + b^{d-1} + b^d$$

E.g. $b=10, d=5: 1+10+100+1,000+10,000+100,000 = 111,111$

Iterative deepening search :

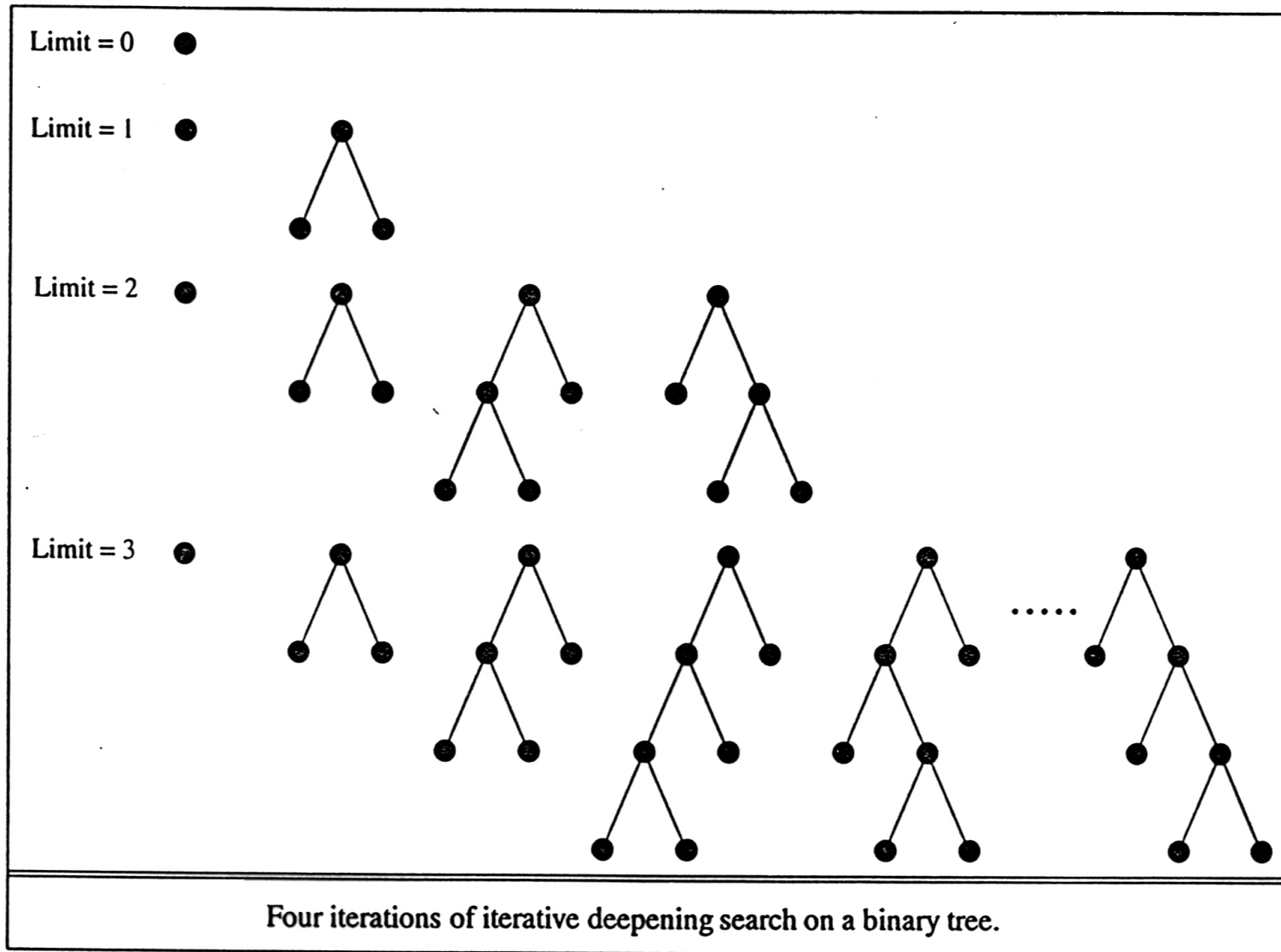
$$(d+1)*1 + (d)*b + (d-1)*b^2 + \dots + 2b^{d-1} + 1b^d$$

E.g. $6+50+400+3000+20,000+100,000 = 123,456$

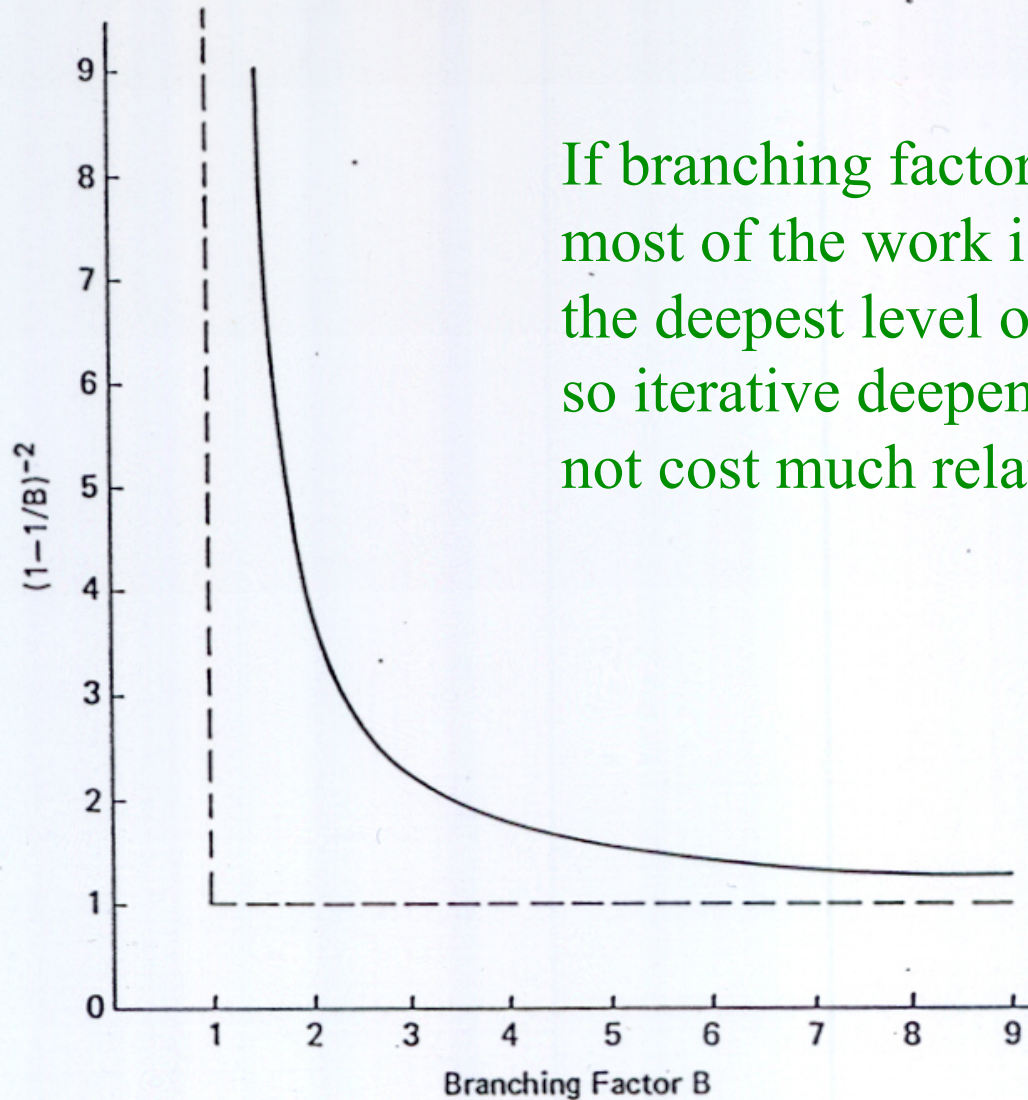
Complete, Optimal, $O(b^d)$ time, $O(bd)$ space

Preferred when search space is large & depth of (optimal) solution is unknown

Iterative Deepening Search...



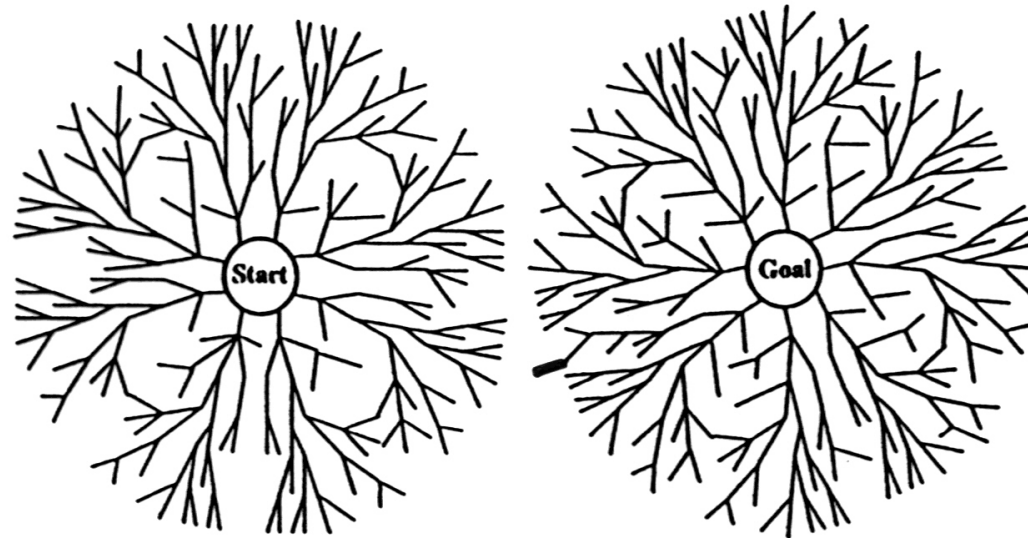
Iterative Deepening Search...



If branching factor is *large*, most of the work is done at the deepest level of search, so iterative deepening does not cost much relatively speaking

Graph of branching factor vs. constant coefficient as search depth goes to infinity.

Bi-Directional Search



A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Time $O(b^{d/2})$

Bi-Directional Search ...

Need to have operators that calculate predecessors.

What if there are multiple goals?

- If there is an explicit list of goal states, then we can apply a predecessor function to the state set just as we apply the successors function in multiple-state forward search.
- If there is only a description of the goal set, it MAY be possible to figure out the possible descriptions of “sets of states that would generate the goal set”.

Efficient way to check when searches meet: hash table

- 1-2 step issue if only one side stored in the table

Decide what kind of search (e.g. breadth-first) to use in each half.

Optimal, complete, $O(b^{d/2})$ time. $O(b^{d/2})$ space (even with iterative deepening) because the nodes of at least one of the searches have to be stored to check matches

Time, Space, Optimal, Complete?

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Evaluation of search strategies. b is the branching factor; d is the depth of solution; m is the maximum depth of the search tree; l is the depth limit.

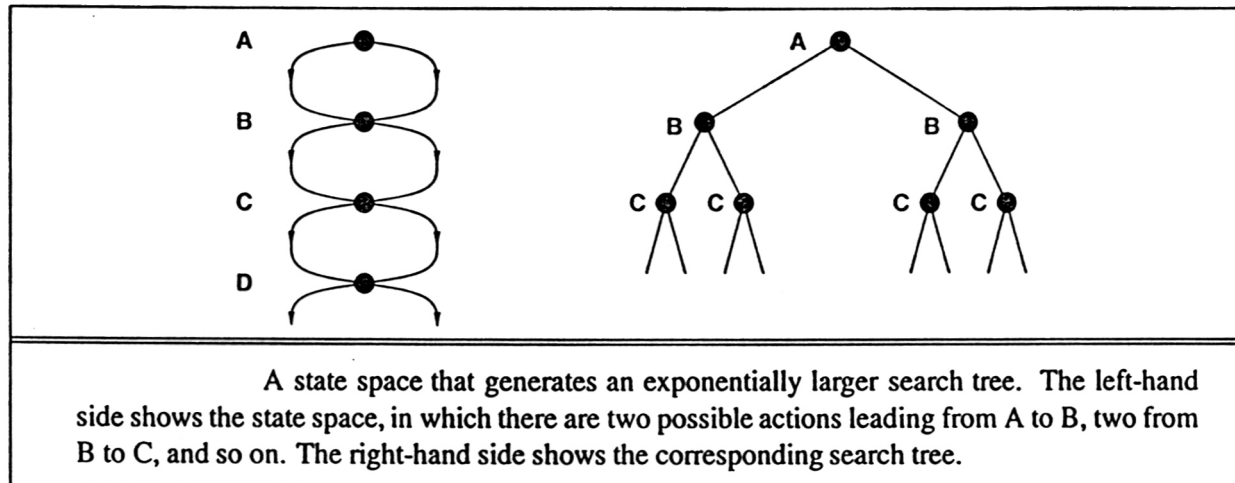
b = branching factor

d = depth of shallowest goal state

m = depth of the search space

l = depth limit of the algorithm

Avoiding repeated states



More effective
& more
computational
overhead

- Do not return to the state you just came from. Have the expand function (or the operator set) refuse to generate any successor that is the same state as the node's parent.
- Do not create paths with cycles in them. Have the expand function (or the operator set) refuse to generate any successor of a node that is the same as any of the node's ancestors.
- Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory, resulting in a space complexity of $O(b^d)$, potentially. It is better to think of this as $O(s)$, where s is the number of states in the entire state space.

To implement this last option, search algorithms often make use of a hash table that stores all the nodes that are generated. This makes checking for repeated states reasonably efficient. The trade-off between the cost of storing and checking and the cost of extra search depends on the problem: the "loopier" the state space, the more likely it is that checking will pay off.

With loops, the search tree may even become infinite