Solving Problems by Searching (Blindly)

R&N: Chap. 3

(many of these slides borrowed from Stanford's AI Class)

Problem Solving Agents

- Decide what to do by finding a sequence of actions that lead to desirable states.

Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest

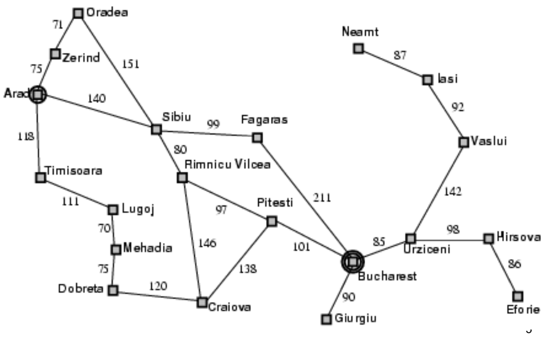
Problem Solving Agent

- Formulate goal:
  - be in Bucharest (in time for flight the next day)
  - Goal formulation is the decision of what you are going to search for - helps us simplify our methods for finding a solution
- Formulate problem: decide what actions, states to consider given a goal
  - states: map with agent in a particular city (location)
  - actions: drive between cities (if there is a road)

Finding a solution…

- Take a road from where I am and see if it takes me to Bucharest…
- Three roads leave Arad, but none go to Bucharest…

Example: Romania



Single-state problem formulation

A problem is defined by three (four) items:

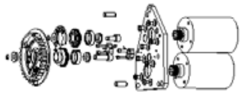
1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of precondition-action pairs where the action returns a state
   - e.g., *S(at Arad)* = {<*at Arad* → *(at Zerind*>, … }
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

State Space

- Each state is an abstract representation of a collection of possible worlds sharing some crucial properties and differing on non-important details only

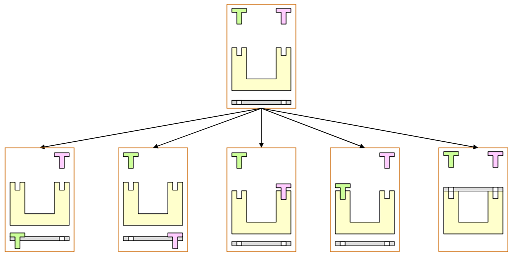
  E.g.: In assembly planning, a state does not define exactly the absolute position of each part



- The state space is discrete. It may be finite, or infinite and is implicit in the problem formulation.

Successor Function

- It implicitly represents all the actions that are feasible in each state



Successor Function

- It implicitly represents all the actions that are feasible in each state
- Only the results of the actions (the successor states) and their costs are returned by the function
- The successor function is a "black box": its content is unknown

  E.g., in assembly planning, the function does not say if it only allows two sub-assemblies to be merged or if it makes assumptions about subassembly stability

Path Cost

- An arc cost is a positive number measuring the "cost" of performing the action corresponding to the arc, e.g.:
  - 1 in the 8-puzzle example
  - expected time to merge two sub-assemblies
- We will assume that for any given problem the cost c of an arc always verifies: $c \geq \varepsilon > 0$, where ε is a constant

  [This condition guarantees that, if path becomes arbitrarily long, its cost also becomes arbitrarily large]

Goal State

- It may be explicitly described:

1	2	3
4	5	6
7	8	

- or partially described:

1	a	a
a	5	a
a	8	a

("a" stands for "any")

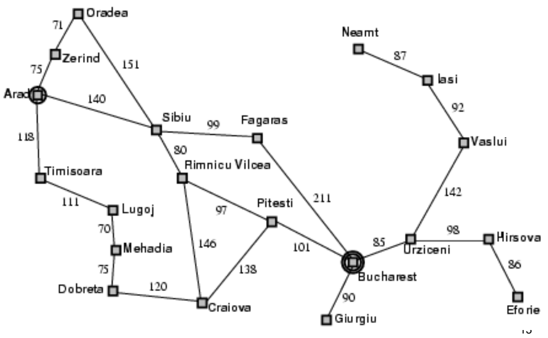
- or defined by a condition, e.g., the sum of every row, of every column, and of every diagonals equals 30

15	1	2	12
4	10	9	7
8	6	5	11
3	13	14	

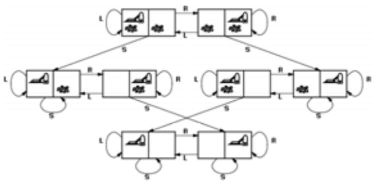
Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: being in various cities
  - initial state: being in Arad
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania

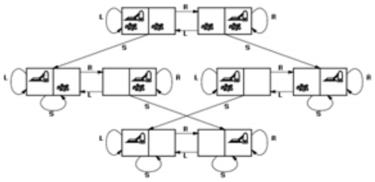


Vacuum world state space graph



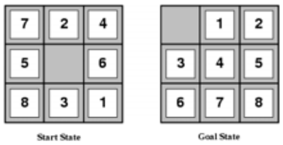
- states?
- Initial state?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



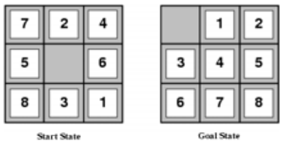
- states? integer dirt and robot location
- Initial state? Dirt in both locations and the vacuum cleaner in one of them
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



- states?
- Initial state?
- actions?
- goal test?
- path cost?

Example: The 8-puzzle



- states? locations of tiles
- Initial state? puzzle in the configuration above
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

GO TO SLIDES

- DO WATERJUG PROBLEM
- Problem Formulation; Search algorithms

Assumptions in Basic Search

- The world is static
- The world is discretizable
- The world is observable
- The actions are deterministic

But many of these assumptions can be removed, and search still remains an important problem-solving tool

---

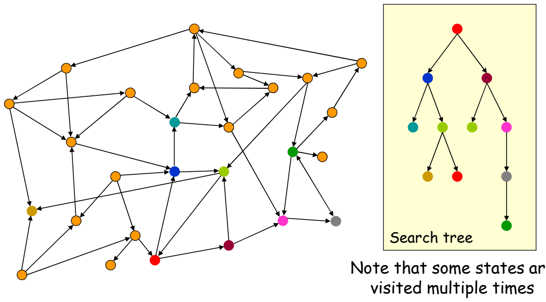
Searching the state

- So far we have talked about how a problem can be looked at so as to form search problems.
- How do we actually do the search?
- (Do search-algorithm slides…)

Simple Problem-Solving-Agent Agent Algorithm

1. s_0 ← sense/read state
2. GOAL? ← select/read goal test
3. SUCCESSORS ← read successor function
4. solution ← **search**(s_0, G, Succ)
5. perform(solution)

Searching the State Space



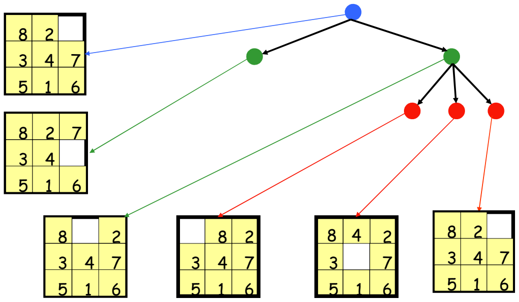
Note that some states are visited multiple times

Basic Search Concepts

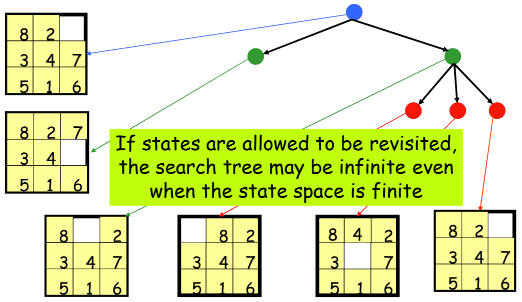
- Search tree
- Search node
- Node expansion
- Fringe of search tree
- Search strategy: At each stage it determines which node to expand

---

Search Nodes ≠ States



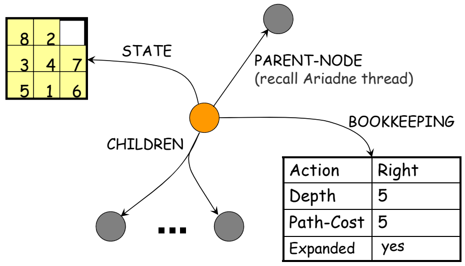
Search Nodes ≠ States



If states are allowed to be revisited, the search tree may be infinite even when the state space is finite

---

Data Structure of a Node



Action	Right
Depth	5
Path-Cost	5
Expanded	yes

Depth of a node N = length of path from root to N

(Depth of the root = 0)

Node expansion

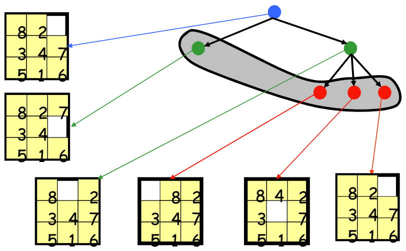
The expansion of a node N of the search tree consists of:

1) Evaluating the successor function on STATE(N)
2) Generating a child of N for each state returned by the function

---

Fringe and Search Strategy

- The fringe is the set of all search nodes that haven't been expanded yet



Is it identical to the set of leaves?

Fringe and Search Strategy

- The fringe is the set of all search nodes that haven't been expanded yet
- It is implemented as a priority queue FRINGE
  - INSERT(node,FRINGE)
  - REMOVE(FRINGE)
- The ordering of the nodes in FRINGE defines the search strategy

---

Search Algorithm

```
1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node,FRINGE)
3. Repeat:
   a. If empty(FRINGE) then return failure
   b. n ← REMOVE(FRINGE)
   c. s ← STATE(n)
   d. If GOAL?(s') then return path or goal state
   e. For every state s' in SUCCESSORS(s)
      i.  Create a new node n' as a child of n
      ii. INSERT(n',FRINGE)
```

Performance Measures

- Completeness
  A search algorithm is complete if it finds a solution whenever one exists
  [What about the case when no solution exists?]
- Optimality
  A search algorithm is optimal if it returns a minimum-cost path whenever a solution exists
  [Other optimality measures are possible]
- Complexity
  It measures the time and amount of memory required by the algorithm

---

Important Parameters

1) Maximum number of successors of any state

   → branching factor b of the search tree

2) Minimal length of a path between the initial and a goal state

   → depth d of the shallowest goal node in the search tree

Important Remark

- Some search problems, such as the (n^2-1)-puzzle, are NP-hard
- One can't expect to solve all instances of such problems in less than exponential time
- One may still strive to solve each instance as efficiently as possible

---

Blind Strategies

- Breadth-first
  - Bidirectional
- Depth-first
  - Depth-limited
  - Iterative deepening

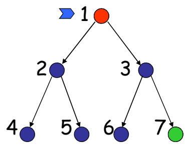
} Arc cost = 1

- Uniform-Cost
  (variant of breadth-first)

} Arc cost $= c(\text{action}) \geq \varepsilon > 0$

Breadth-First Strategy

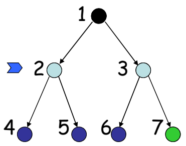
New nodes are inserted at the end of FRINGE



FRINGE = (1)

Breadth-First Strategy

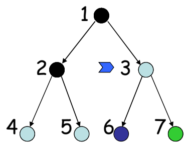
New nodes are inserted at the end of FRINGE



FRINGE = (2, 3)

Breadth-First Strategy

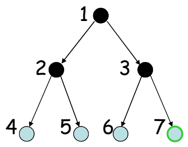
New nodes are inserted at the end of FRINGE



FRINGE = (3, 4, 5)

Breadth-First Strategy

New nodes are inserted at the end of FRINGE



FRINGE = (4, 5, 6, 7)

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  - Complete
  - Optimal if step cost is 1
- Number of nodes generated:
  $1 + b + b^2 + \ldots + b^d = (b^{d+1}-1)/(b-1) = O(b^d)$
- → Time and space complexity is $O(b^d)$

Big O Notation

$g(n) = O(f(n))$ if there exist two positive constants a and N such that:

for all $n > N$: $g(n) \leq a \times f(n)$

Time and Memory Requirements

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim 10^6$	1 sec	100 Mb
8	$\sim 10^8$	100 sec	10 Gbytes
10	$\sim 10^{10}$	2.8 hours	1 Tbyte
12	$\sim 10^{12}$	11.6 days	100 Tbytes
14	$\sim 10^{14}$	3.2 years	10,000 Tbytes

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

Time and Memory Requirements

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim 10^6$	1 sec	100 Mb
8	$\sim 10^8$	100 sec	10 Gbytes
10	$\sim 10^{10}$	2.8 hours	1 Tbyte
12	$\sim 10^{12}$	11.6 days	100 Tbytes
14	$\sim 10^{14}$	3.2 years	10,000 Tbytes

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

Remark

If a problem has no solution, breadth-first may run for ever (if the state space is infinite or states can be revisited arbitrary many times)

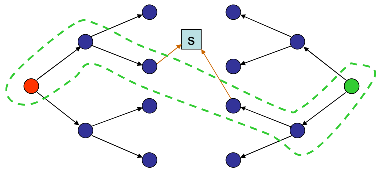
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

? →

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2



Time and space complexity is $O(b^{d/2}) << O(b^d)$ if both trees have the same branching factor b

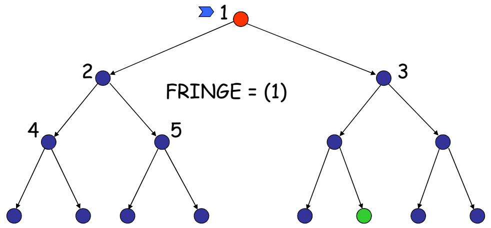
Question: What happens if the branching factor is different in each direction?

Bidirectional Search

- Search forward from the start state and backward from the goal state simultaneously and stop when the two searches meet in the middle.
- If branching factor=b, and solution at depth d, then $O(2b^{d/2})$ steps.
- B=10, d=6 then BFS needs 1,111,111 nodes and bidirectional needs only 2,222.

Depth-First Strategy

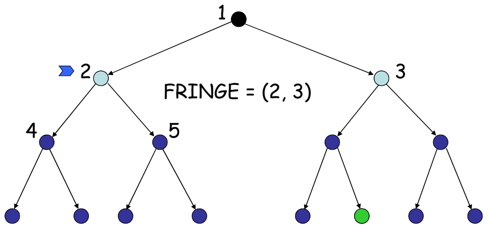
New nodes are inserted at the front of FRINGE



FRINGE = (1)

Depth-First Strategy

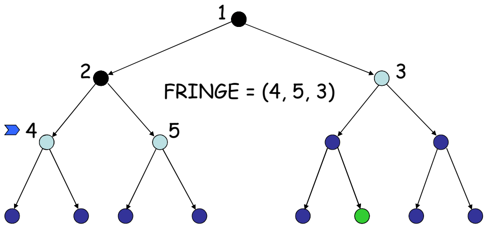
New nodes are inserted at the front of FRINGE



FRINGE = (2, 3)

Depth-First Strategy

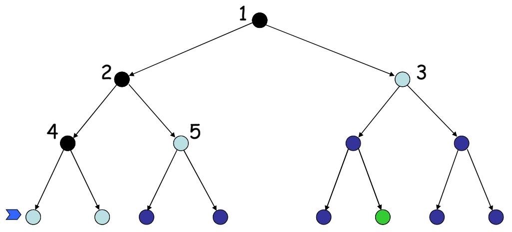
New nodes are inserted at the front of FRINGE



FRINGE = (4, 5, 3)

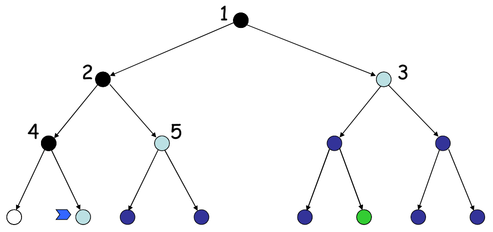
Depth-First Strategy

New nodes are inserted at the front of FRINGE



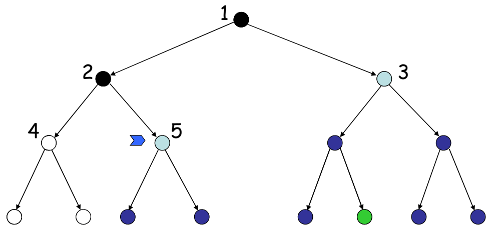
Depth-First Strategy

New nodes are inserted at the front of FRINGE



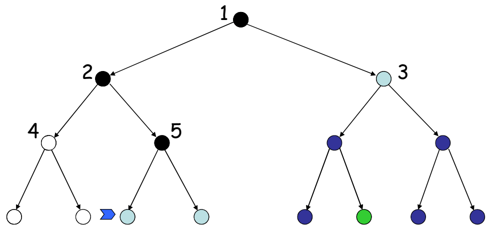
Depth-First Strategy

New nodes are inserted at the front of FRINGE



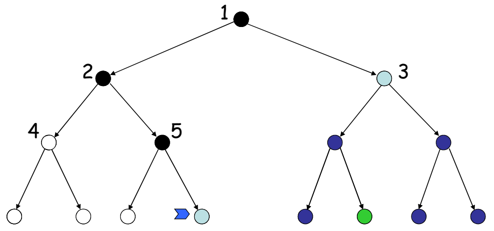
Depth-First Strategy

New nodes are inserted at the front of FRINGE



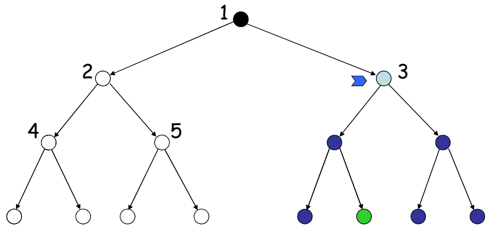
Depth-First Strategy

New nodes are inserted at the front of FRINGE



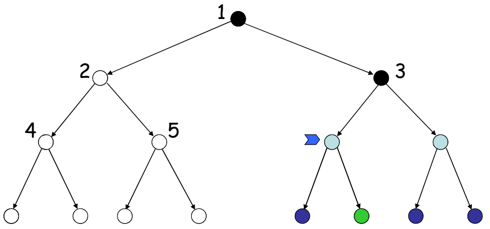
Depth-First Strategy

New nodes are inserted at the front of FRINGE



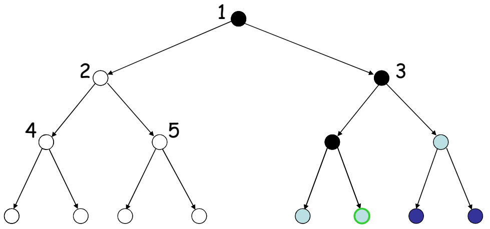
Depth-First Strategy

New nodes are inserted at the front of FRINGE



Depth-First Strategy

New nodes are inserted at the front of FRINGE



Evaluation

- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Depth-first search is:
  - Complete only for finite search tree
  - Not optimal
- Number of nodes generated: $1 + b + b^2 + \ldots + b^m = O(b^m)$
- Time complexity is $O(b^m)$
- Space complexity is $O(bm)$ [or $O(m)$]

[Reminder: Breadth-first requires $O(b^d)$ time and space]

Depth-Limited Search

- Depth-first with depth cutoff k (depth below which nodes are not expanded)
- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)

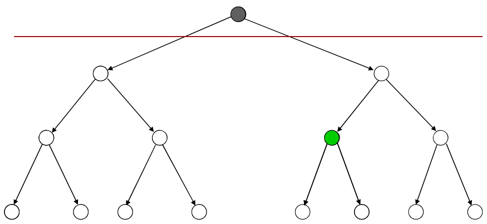
Iterative Deepening Search

Provides the best of both breadth-first and depth-first search

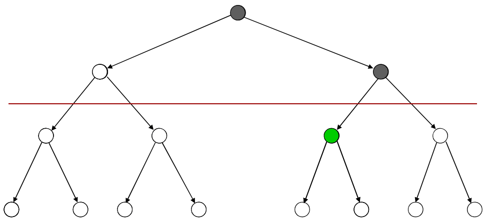
Main idea: Totally horrifying !

```
IDS
For k = 0, 1, 2, … do:
    Perform depth-first search with
    depth cutoff k
```

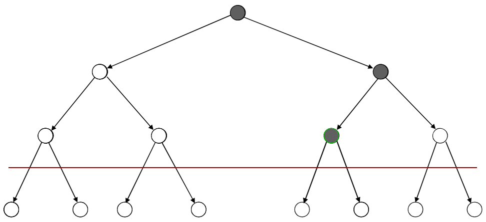
Iterative Deepening



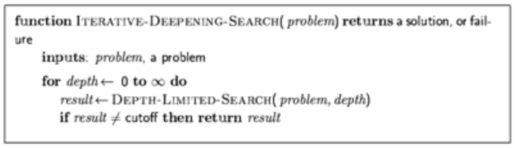
Iterative Deepening



Iterative Deepening



Iterative deepening search

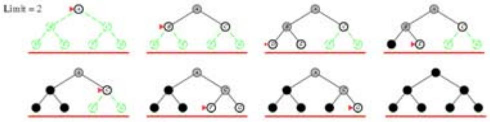


```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

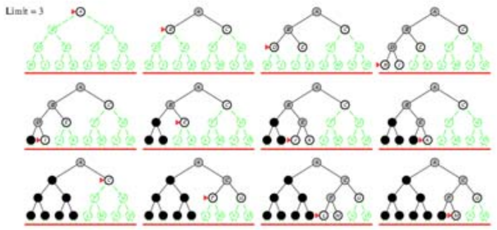
Iterative deepening search *l* =0

Iterative deepening search *l* =1

Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

  $N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

  $N_{IDS} = (d+1)b^0 + d\ b$^$^1 + (d-1)b$^$^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

Complete? Yes

Time? $(d+1)b^0 + d\ b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space? $O(bd)$

Optimal? Yes, if step cost = 1

Performance

- Iterative deepening search is:
  - Complete
  - Optimal if step cost =1
- Time complexity is:

  $(d+1)(1) + db + (d-1)b^2 + \ldots + (1)\ b^d = O(b^d)$
- Space complexity is: $O(bd)$ or $O(d)$

Calculation

$$
\begin{aligned}
&db + (d-1)b^2 + \ldots + (1)\ b^d \\
&\quad = b^d + 2b^{d-1} + 3b^{d-2} + \ldots + db \\
&\quad = (1 + 2b^{-1} + 3b^{-2} + \ldots + db^{-d}) \times b^d \\
&\quad \leq \left(\Sigma_{i=1,\ldots,\infty}\ ib^{(1-i)}\right) \times b^d \ = \ b^d\,(b/(b-1))^2
\end{aligned}
$$

Number of Generated Nodes
(Breadth-First & Iterative Deepening)

d = 5 and b = 2

BF	ID
1	1 x 6 = 6
2	2 x 5 = 10
4	4 x 4 = 16
8	8 x 3 = 24
16	16 x 2 = 32
32	32 x 1 = 32
63	120

120/63 ~ 2

Number of Generated Nodes
(Breadth-First & Iterative Deepening)

d = 5 and b = 10

BF	ID
1	6
10	50
100	400
1,000	3,000
10,000	20,000
100,000	100,000
111,111	123,456

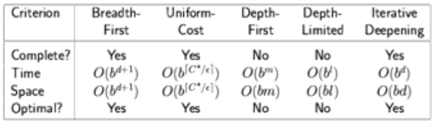
123,456/111,111 ~ 1.111

Comparison of Strategies

- Breadth-first is complete and optimal, but has high space complexity
- Depth-first is space efficient, but is neither complete, nor optimal
- Iterative deepening is complete and optimal, with the same space complexity as depth-first and almost the same time complexity as breadth-first

---

Summary of algorithms



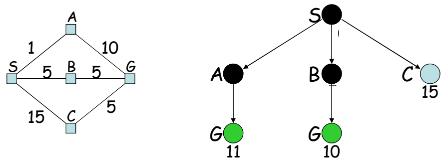
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Avoiding Revisited States

- Let's not worry about it yet… but generally we will have to be careful to avoid states we have already seen…

Uniform-Cost Search

- Each arc has some cost $c \geq \varepsilon > 0$
- The cost of the path to each fringe node N is
  $g(N) = \Sigma$ costs of arcs
- The goal is to generate a solution path of minimal cost
- The queue FRINGE is sorted in increasing cost



- Need to modify search algorithm