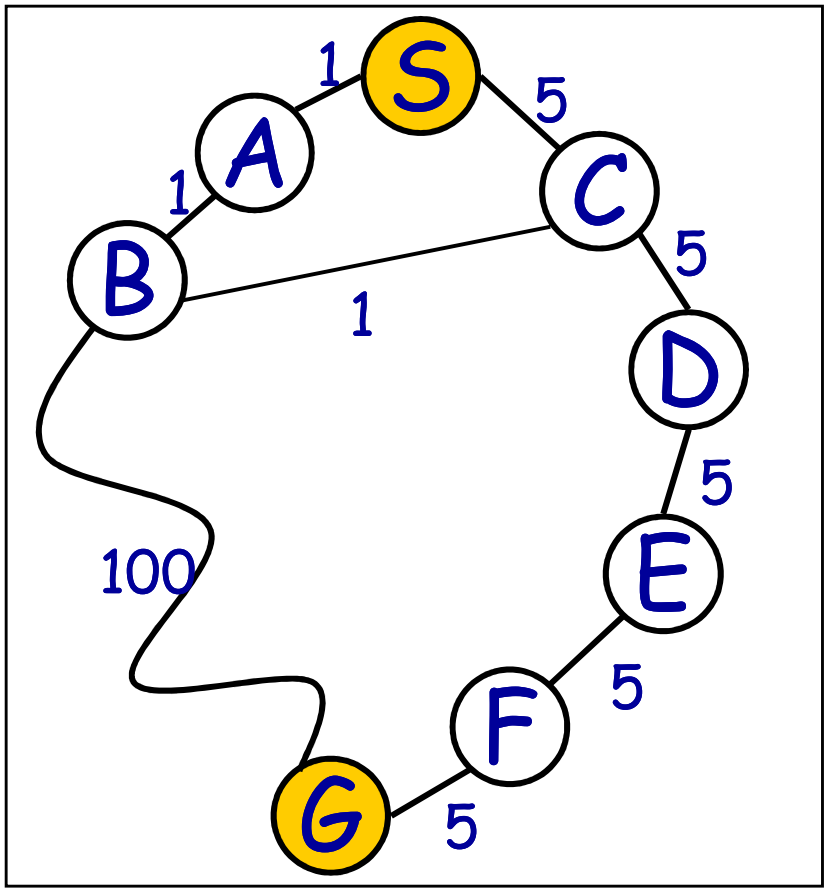
A Clean Robust Uniform Cost Search Algorithm

```
Function UniformCost-Search(problem, Queuing-Fn) returns a solution, or failure
    open ← make-queue(make-node(initial-state[problem]))
    closed ← [empty]
    loop do
        if open is empty then return failure
        currnode ← Remove-Front(open)
        if Goal-Test[problem] applied to State(currnode) then return currnode
        children ← Expand(currnode, Operators[problem])
        while children not empty

                                [... see next slide ...]

        end
        closed ← Insert(closed, currnode)
        open ← Sort-By-PathCost(open)
    end
```

A Clean Robust Algorithm

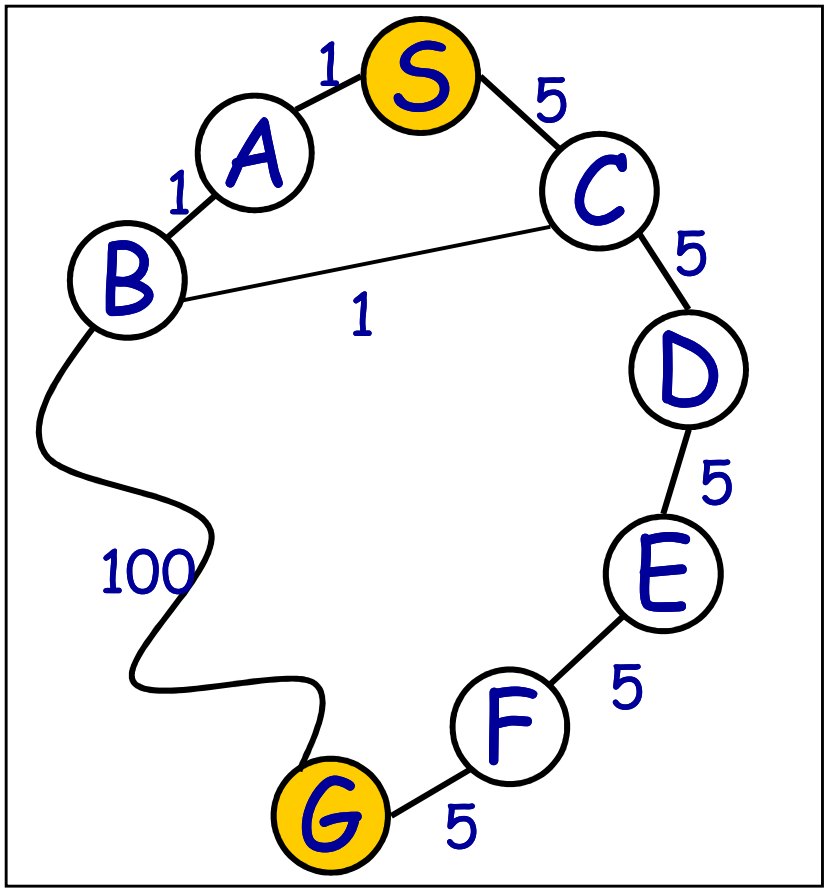
```
[... see previous slide ...]
        children ← Expand(currnode, Operators[problem])
        while children not empty
                child ← Remove-Front(children)
                if no node in open or closed has child's state
                        open ← Queuing-Fn(open, child)
                else if there exists node in open that has child's state
                        if PathCost(child) < PathCost(node)
                                open ← Delete-Node(open, node)
                                open ← Queuing-Fn(open, child)
                else if there exists node in closed that has child's state
                        if PathCost(child) < PathCost(node)
                                closed ← Delete-Node(closed, node)
                                open ← Queuing-Fn(open, child)
        end
[... see previous slide ...]
```

Example



#	State	Depth	Cost	Parent
1	S	0	0	-

Example

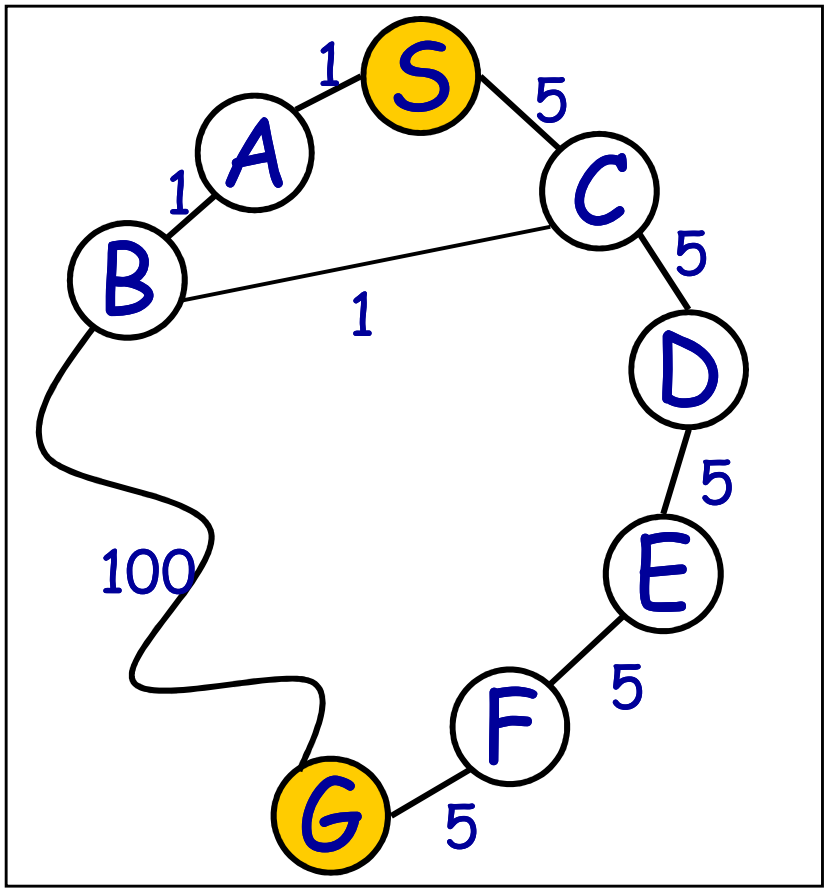


#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
3	C	1	5	1

Black = open queue
Grey = closed queue

Insert expanded nodes
Such as to keep *open* queue sorted

Example

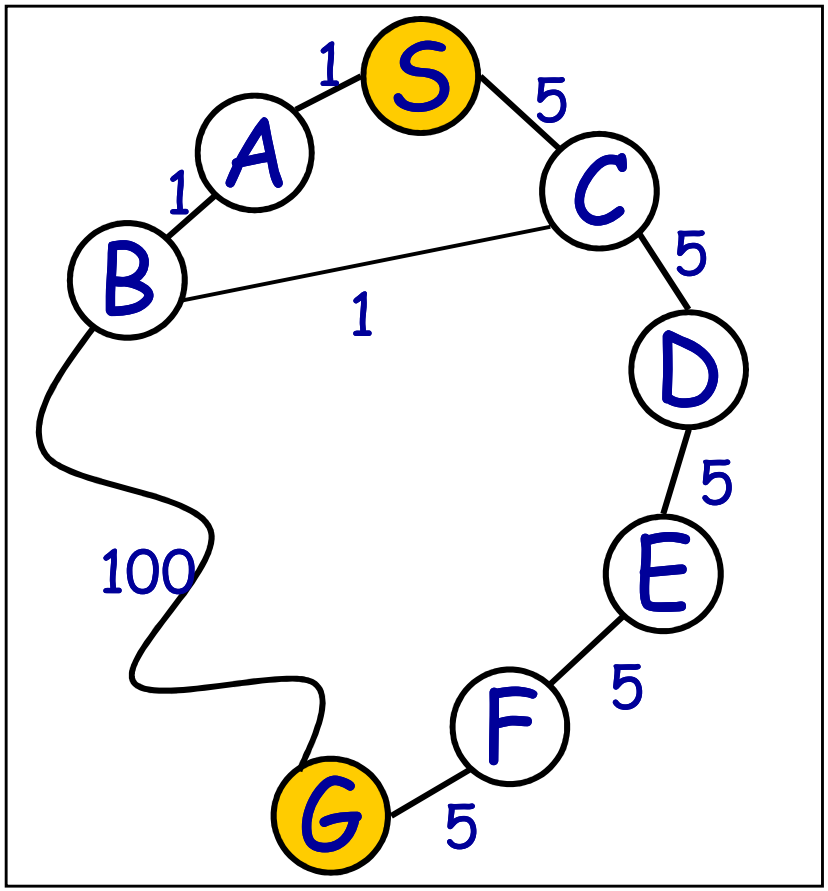


#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
3	C	1	5	1

Node 2 has 2 successors: one with state B and one with state S.

We have node #1 in *closed* with state S; but its path cost 0 is smaller than the path cost obtained by expanding from A to S. So we do not queue-up the successor of node 2 that has state S.

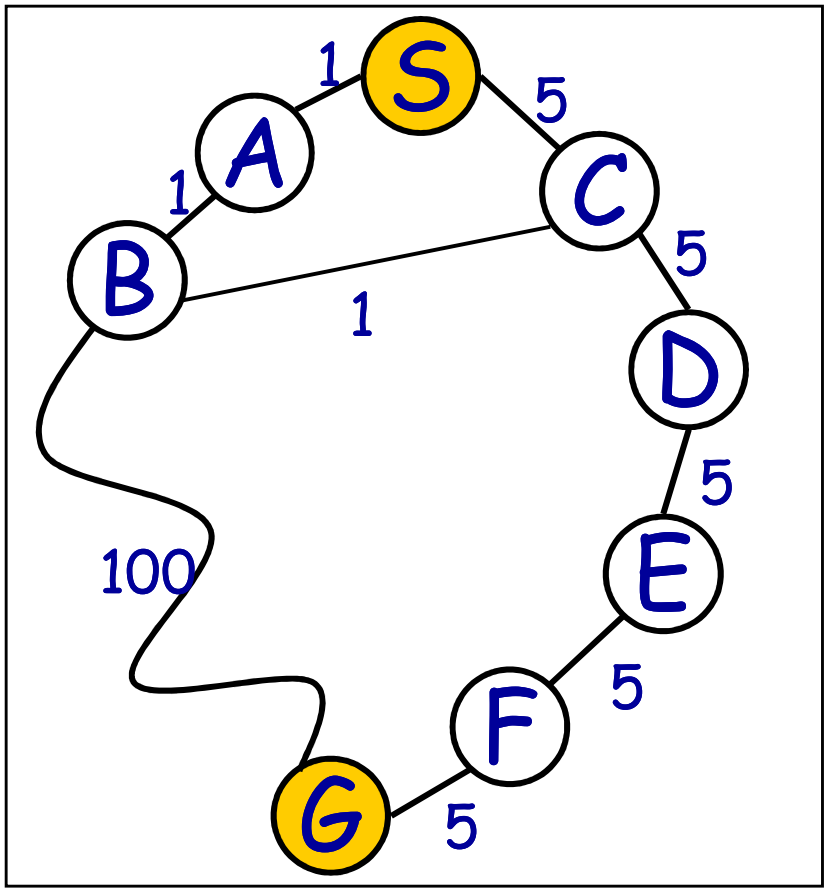
Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
6	G	3	102	4

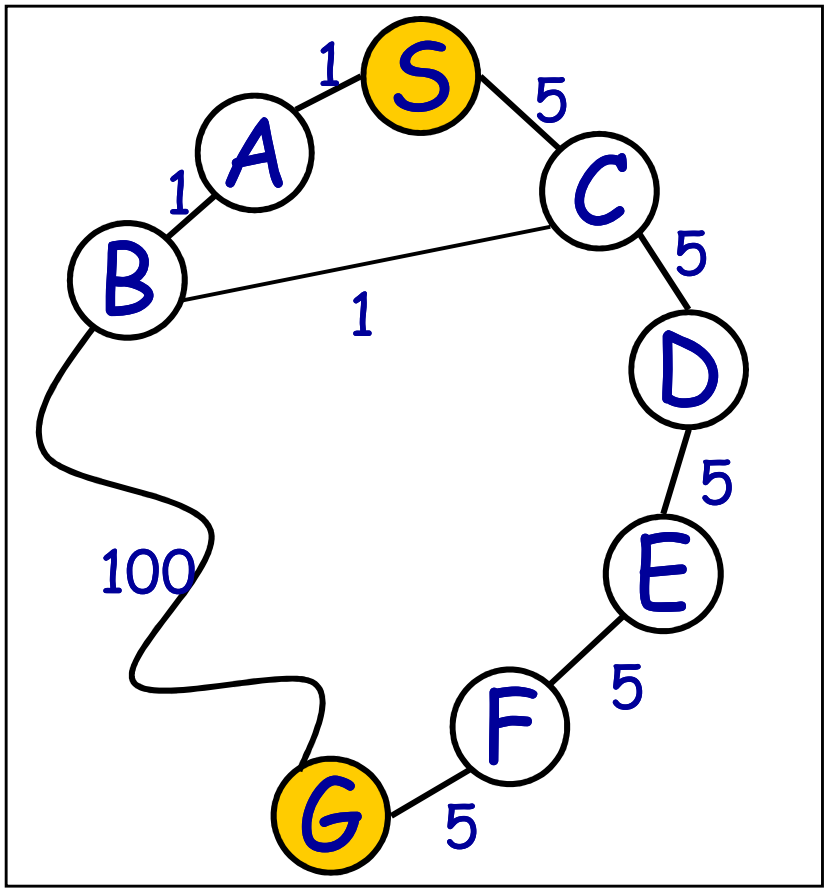
Node 4 has a successor with state C and cost smaller than node #3 in *open* that also had state C; so we update the *open* queue to reflect the shortest path.

Example



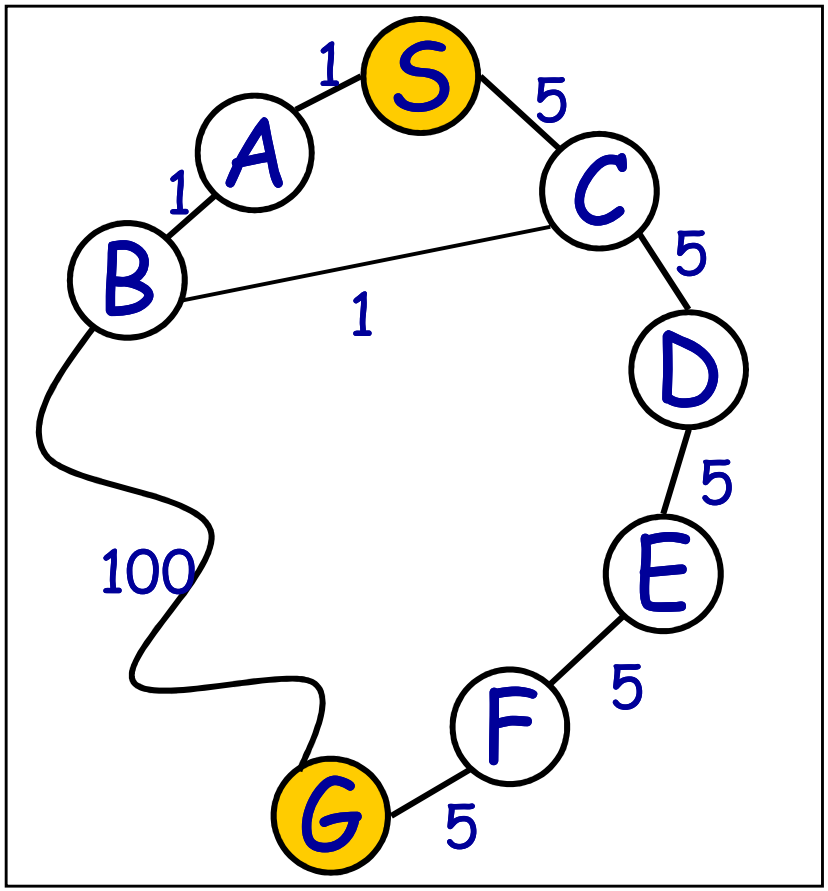
#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
7	D	4	8	5
6	G	3	102	4

Example



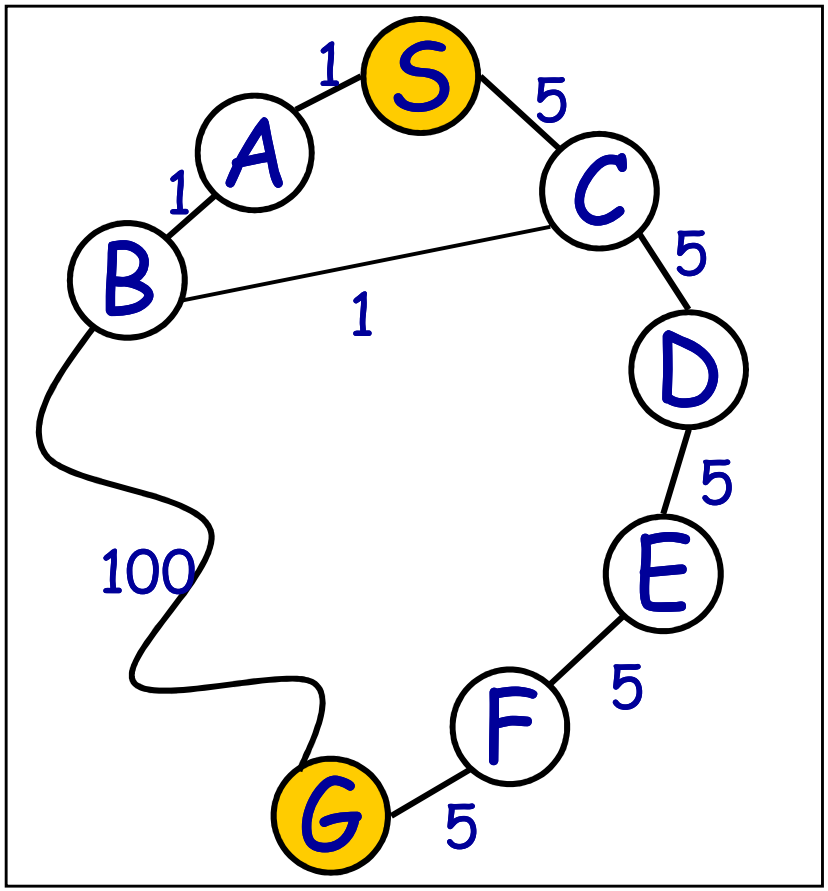
#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
7	D	4	8	5
8	E	5	13	7
6	G	3	102	4

Example



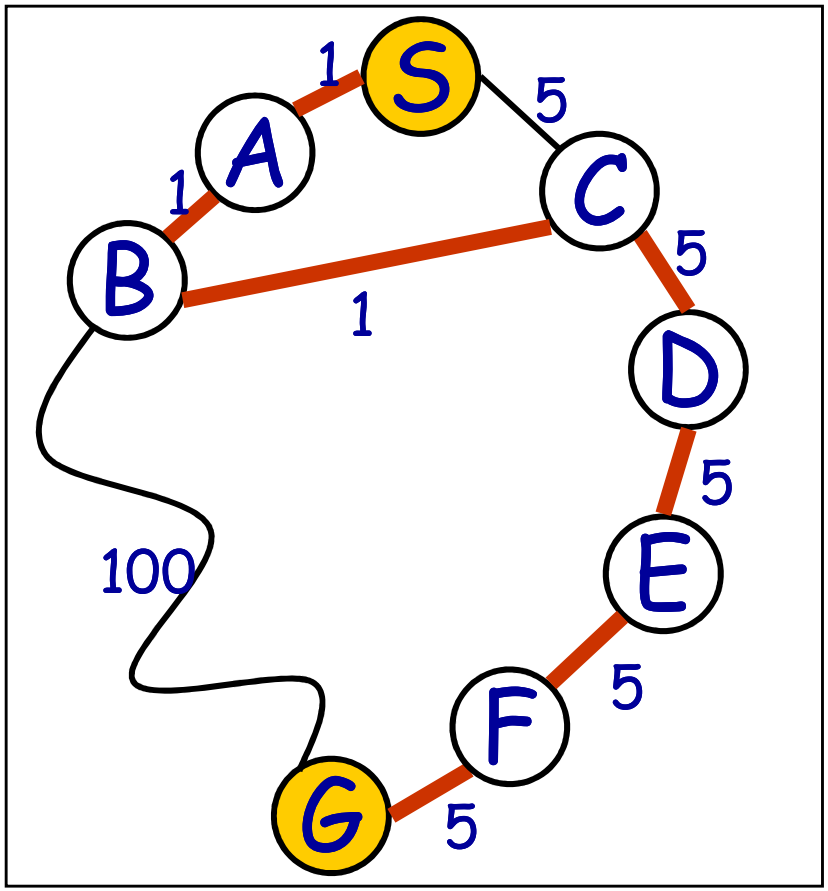
#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
7	D	4	8	5
8	E	5	13	7
9	F	6	18	8
6	G	3	102	4

Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
7	D	4	8	5
8	E	5	13	7
9	F	6	18	8
10	G	7	23	9
6	G	3	102	4

Example



#	State	Depth	Cost	Parent
1	S	0	0	-
2	A	1	1	1
4	B	2	2	2
5	C	3	3	4
7	D	4	8	5
8	E	5	13	7
9	F	6	18	8
10	G	7	23	9
6	G	3	102	4

Goal reached

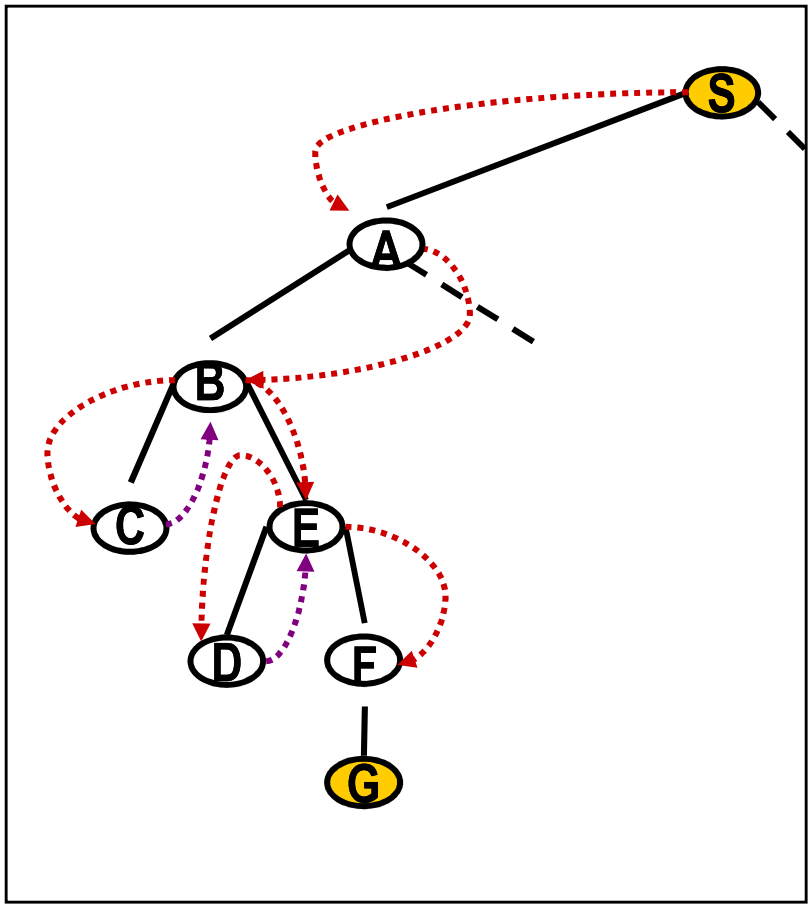
Depth-first search

Expand deepest unexpanded node

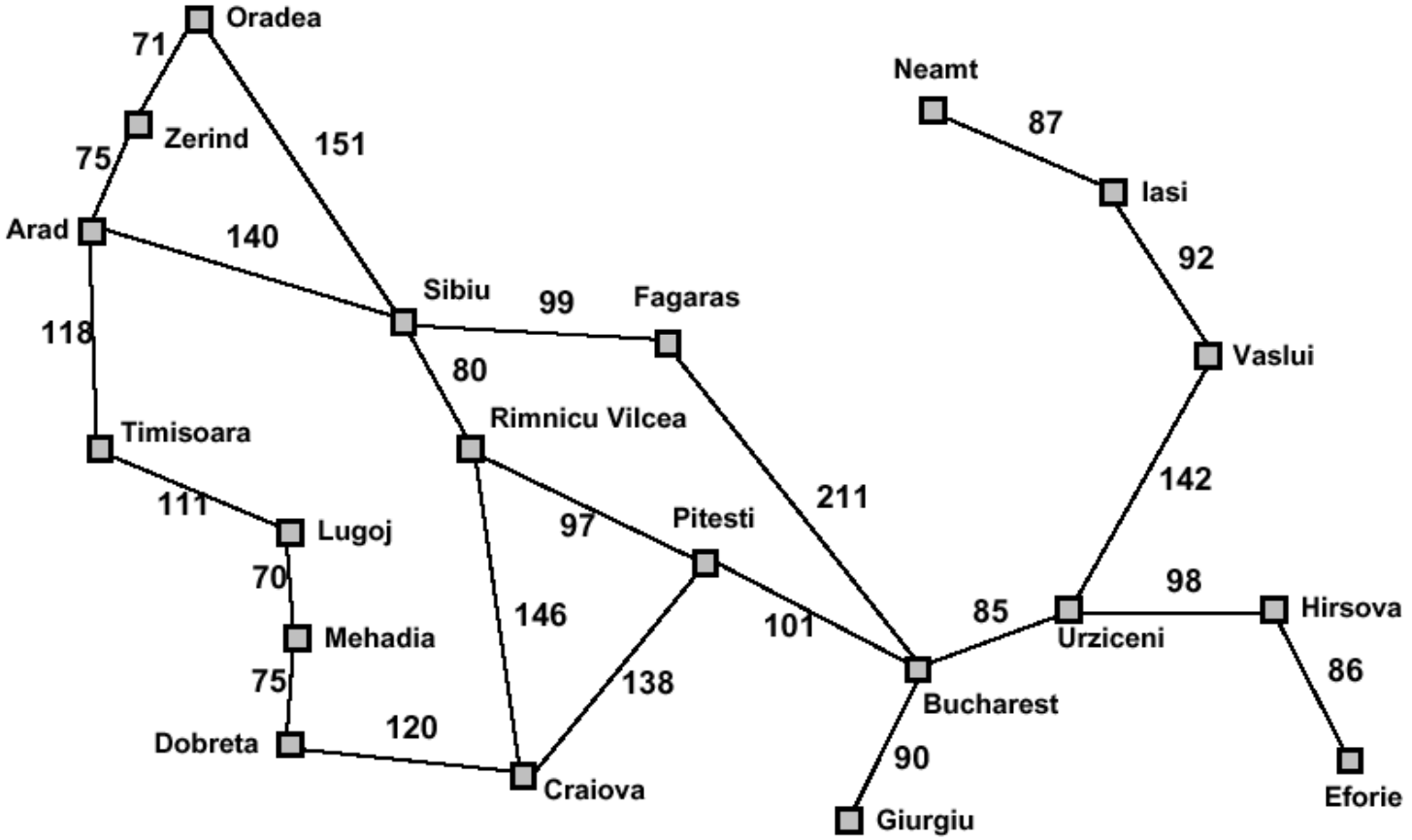
Implementation:

QUEUEINGFN = insert successors at front of queue

Depth First Search

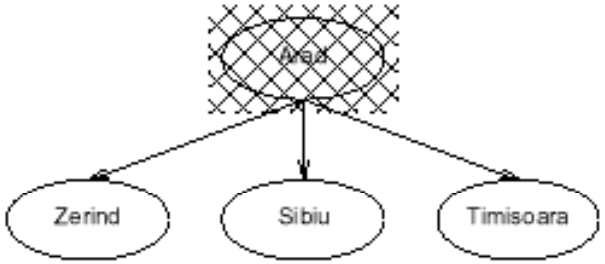


Romania with step costs in km

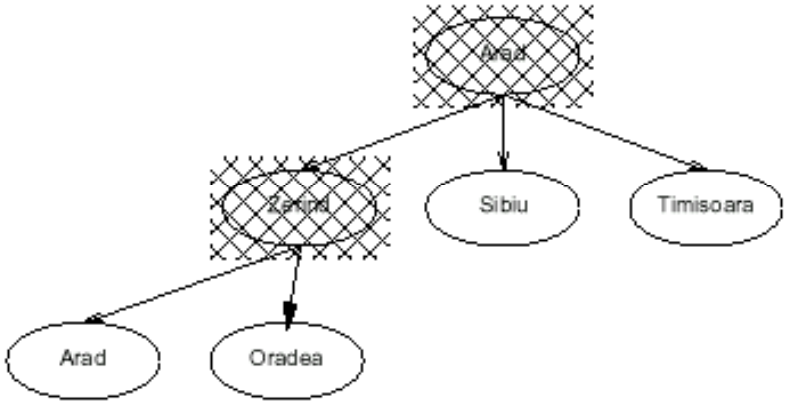


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

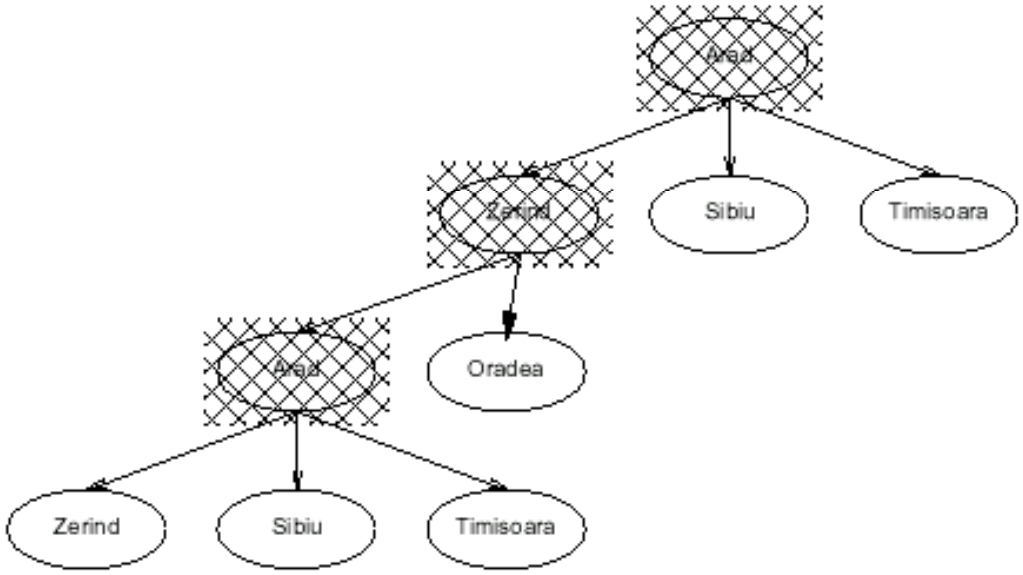
Depth-first search



Depth-first search



Depth-first search



I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

Properties of depth-first search

- Completeness: No, fails in infinite state-space (yes if finite state space)
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$
- Optimality: No

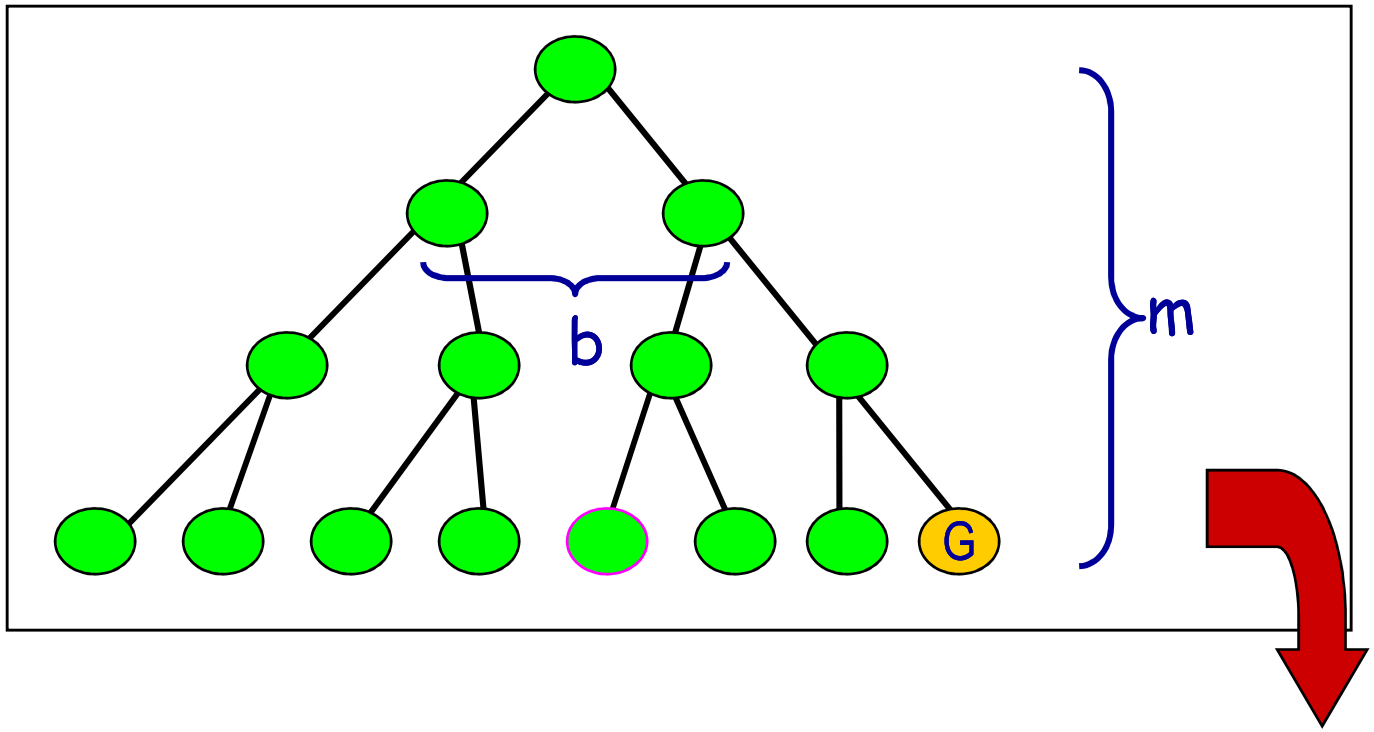
Remember:

b = branching factor

m = max depth of search tree

Time complexity of depth-first: details

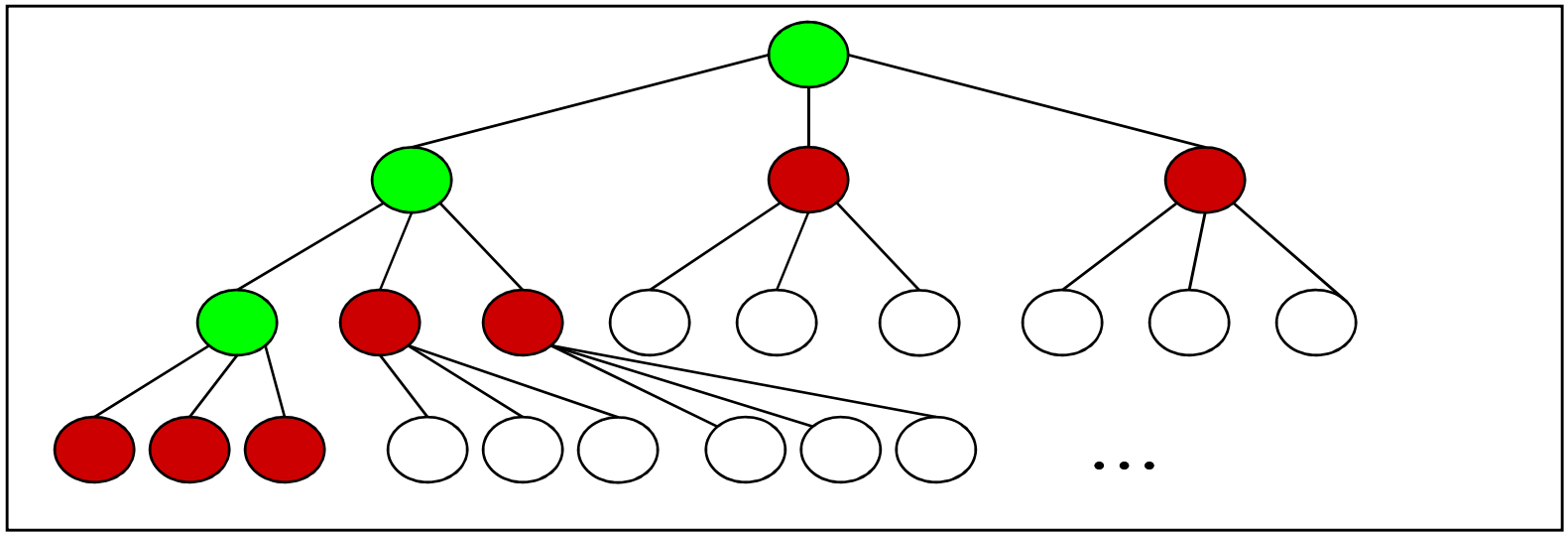
- In the worst case:
  - the (only) goal node may be on the right-most branch,



- Time complexity == $b^m + b^{m-1} + \ldots + 1 = \frac{b^{m+1} - 1}{b - 1}$
- Thus: $O(b^m)$

Space complexity of depth-first

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: m = 3, b = 3 :



- QUEUE contains all  nodes. Thus: 7.
- In General: ((b-1) * m) + 1
- Order: O(m*b)

Avoiding repeated states

In increasing order of effectiveness and computational overhead:

- do not return to state we come from, i.e., expand function will skip possible successors that are in same state as node's parent.
- do not create paths with cycles, i.e., expand function will skip possible successors that are in same state as any of node's ancestors.
- do not generate any state that was ever generated before, by keeping track (in memory) of every state generated, unless the cost of reaching that state is lower than last time we reached it.

Depth-limited search

Is a depth-first search with depth limit l

Implementation:
Nodes at depth l have no successors.

Complete: if cutoff chosen appropriately then it is guaranteed to find a solution (not really a "guarantee" in other words!)

Optimal: it does not guarantee finding the least-cost solution

Iterative deepening search

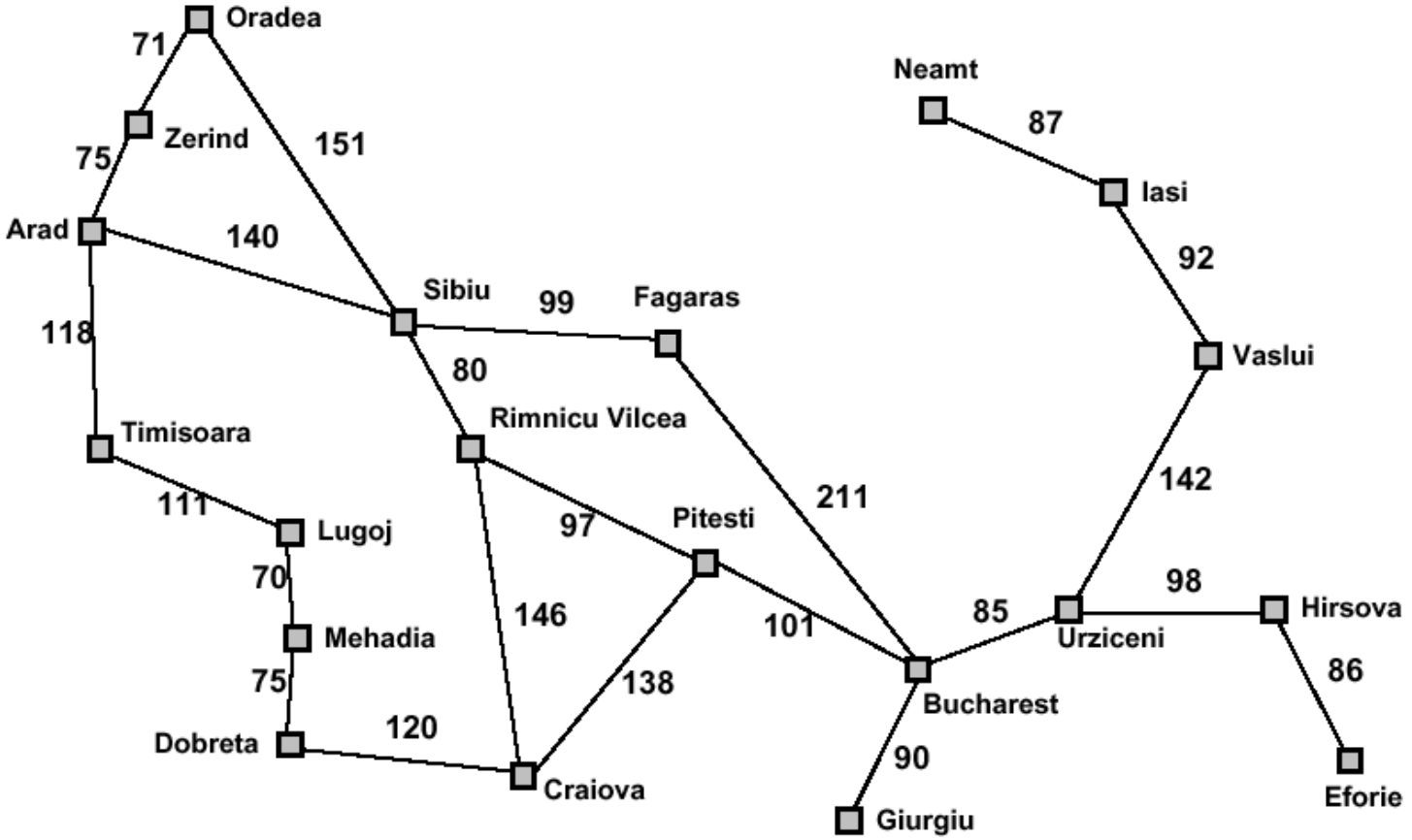
Function Iterative-deepening-Search(*problem*) **returns** a solution,
or failure

for *depth* = 0 **to** ∞ **do**
result ← Depth-Limited-Search(*problem*, *depth*)
if *result* succeeds **then return** *result*
end
return failure

Combines the best of breadth-first and depth-first search strategies.

- Completeness: Yes,
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality: Yes, if step cost = 1

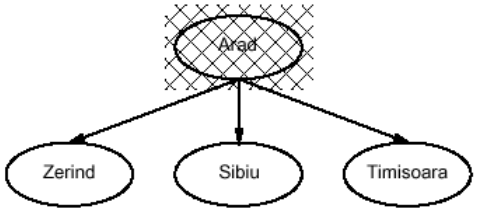
Romania with step costs in km



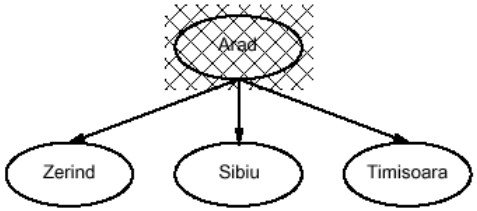
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

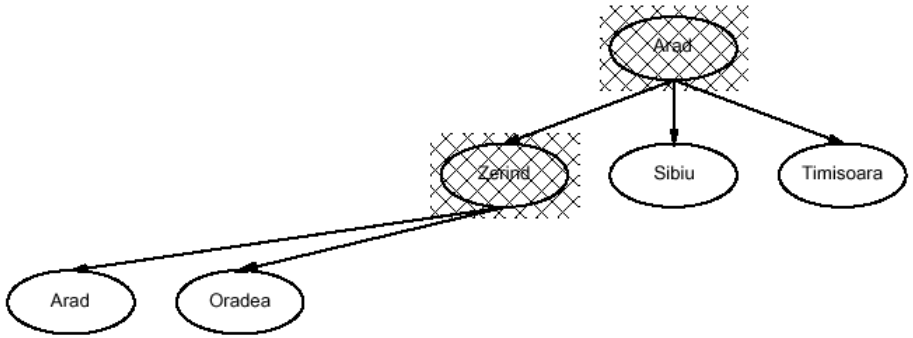
Iterative deepening search $l = 0$

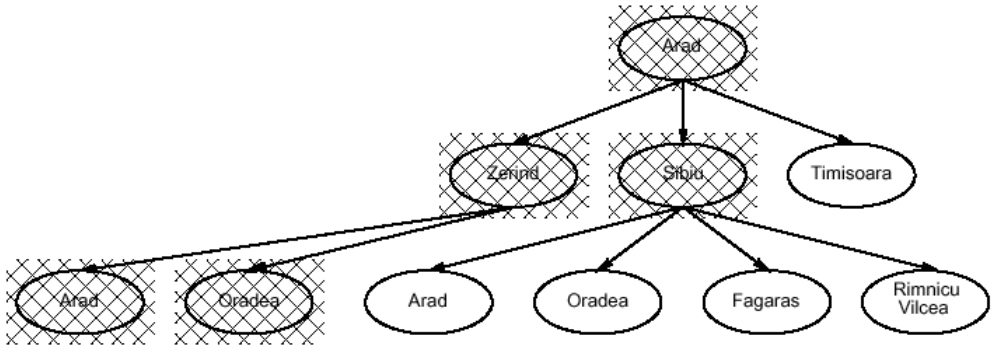
Iterative deepening search $l = 1$

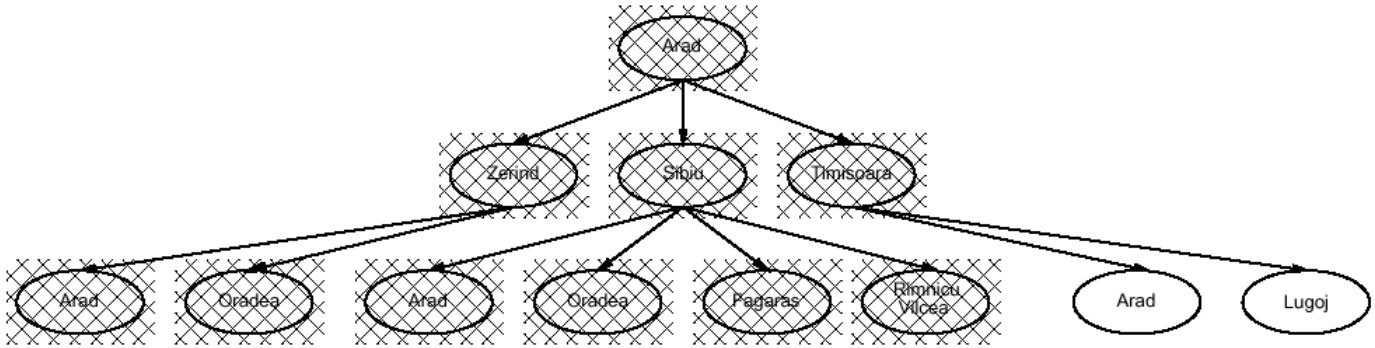

Arad
Zerind
Sibiu
Timisoara

Iterative deepening search $l = 2$


Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea




Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Pagaras
Rimnicu Vilcea
Arad
Lugoj

Iterative deepening complexity

- Iterative deepening search may seem wasteful because so many states are expanded multiple times.

- In practice, however, the overhead of these multiple expansions is small, because most of the nodes are towards leaves (bottom) of the search tree:

  thus, the nodes that are evaluated several times (towards top of tree) are in relatively small number.

Iterative deepening complexity

- In iterative deepening, nodes at bottom level are expanded once, level above twice, etc. up to root (expanded d+1 times) so total number of expansions is:

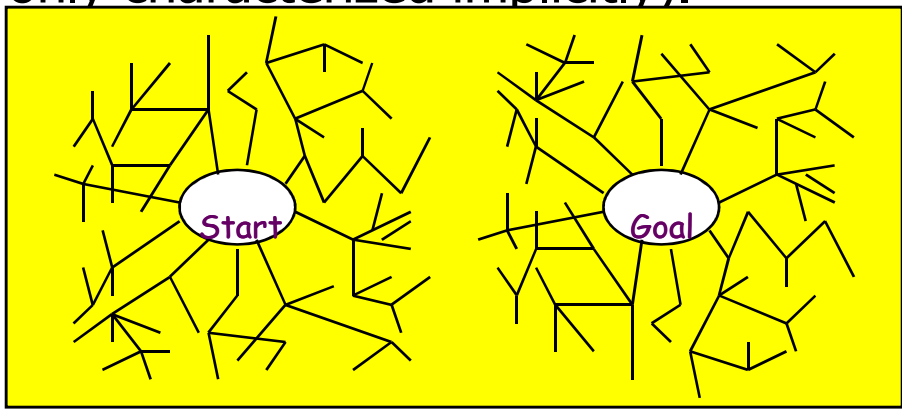
  $(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{(d-2)} + 2b^{(d-1)} + 1b^d = O(b^d)$

- In general, iterative deepening is preferred to depth-first or breadth-first when search space large and depth of solution not known.

Bidirectional search

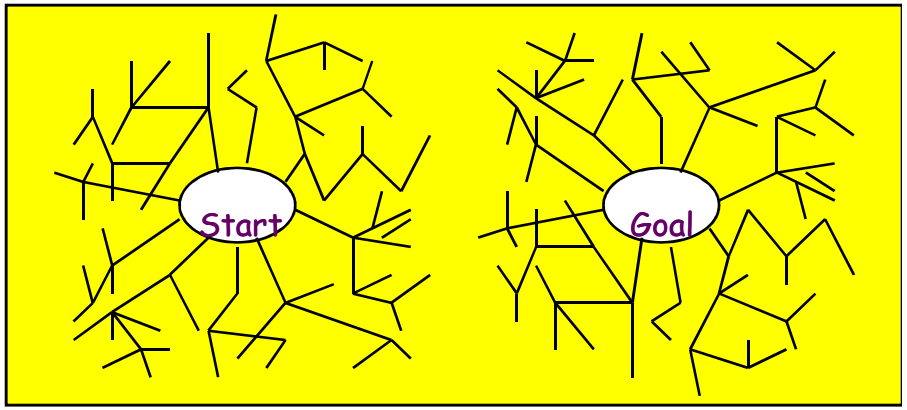
- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.

- Problem: how do we search backwards from goal??
  - predecessor of node n = all nodes that have n as successor
  - this may not always be easy to compute!
  - if several goal states, apply predecessor function to them just as we applied successor (only works well if goals are explicitly known; may be difficult if goals only characterized implicitly).



Bidirectional search

- Problem: how do we search backwards from goal?? (cont.)
  - …
  - for bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.
  - select a given search algorithm for each half.



Bidirectional search

1. QUEUE1 <-- path only containing the root;
   QUEUE2 <-- path only containing the goal;

2. WHILE both QUEUEs are not empty
   AND QUEUE1 and QUEUE2 do NOT share a state

   DO remove their first paths;
   create their new paths (to all children);
   reject their new paths with loops;
   add their new paths to back;

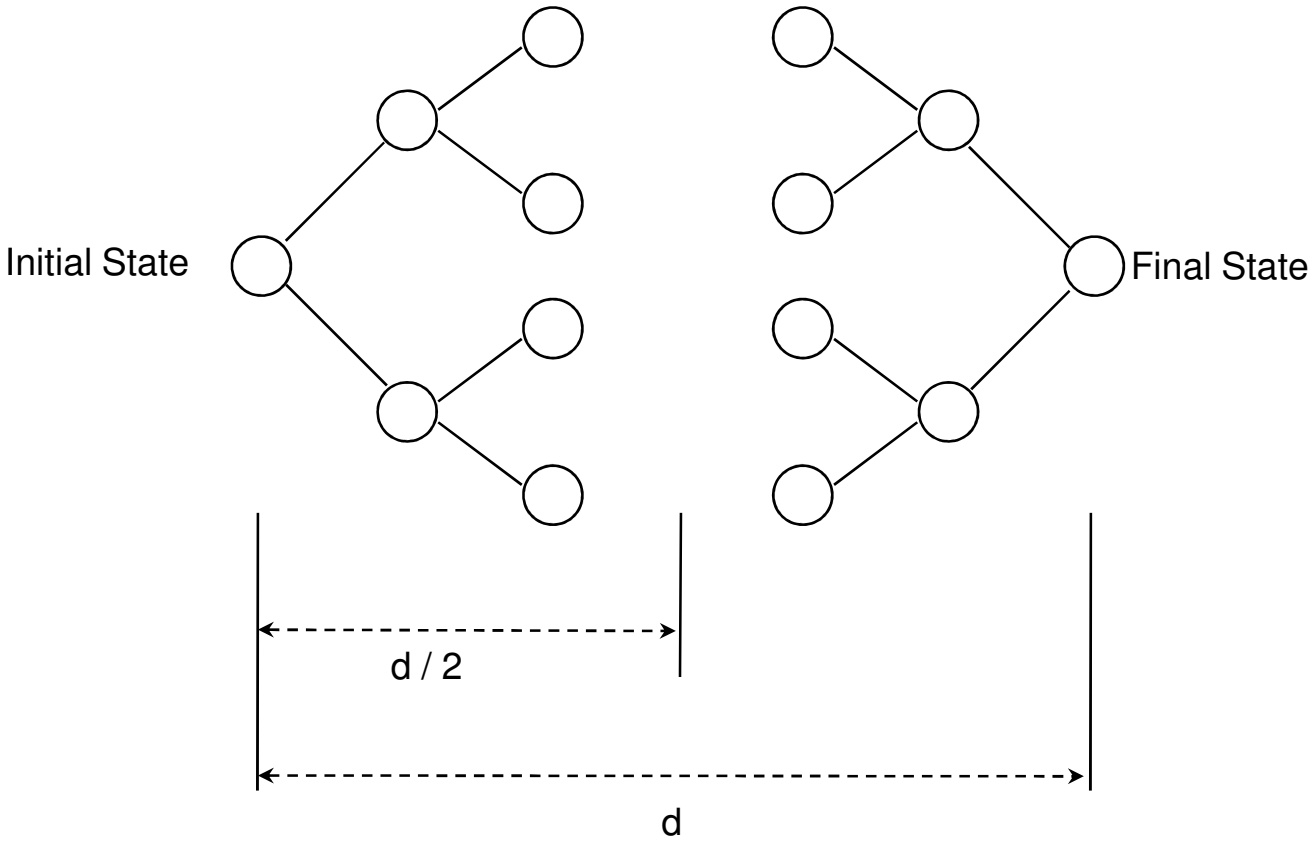
3. IF QUEUE1 and QUEUE2 share a state
   THEN success;
   ELSE failure;

Bidirectional search

- Completeness: Yes,
- Time complexity: $2*O(b^{d/2}) = O(b^{d/2})$
- Space complexity: $O(b^{m/2})$
- Optimality: Yes

- To avoid one by one comparison, we need a hash table of size $O(b^{m/2})$
- *If hash table is used, the cost of comparison is O(1)*

Bidirectional Search



Bidirectional search

- Bidirectional search merits:
  - Big difference for problems with branching factor b in both directions
    - A solution of length d will be found in $O(2b^{d/2}) = O(b^{d/2})$
    - For $b = 10$ and $d = 6$, only 2,222 nodes are needed instead of 1,111,111 for breadth-first search

Bidirectional search

- Bidirectional search issues
  - *Predecessors* of a node need to be generated
    - Difficult when operators are not reversible
  - What to do if there is no *explicit list of goal* states?
  - For each node: *check if it appeared in the other search*
    - Needs a hash table of $O(b^{d/2})$
  - What is the *best search strategy* for the two searches?

Comparing uninformed search strategies

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≥d	Yes	Yes

- *b* – max branching factor of the search tree
- *d* – depth of the least-cost solution
- *m* – max depth of the state-space (may be infinity)
- *l* – depth cutoff

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can be explored using computer algorithms.
- Once problem is formulated in abstract form, complexity analysis helps us picking out best algorithm to solve problem.
- Variety of uninformed search strategies; difference lies in method used to pick node that will be further expanded.
- Iterative deepening search only uses linear space and not much more time than other uniformed search strategies.