# CSCI 561 - Foundation for Artificial Intelligence

## 03 Search Algorithms
### Uninformed and Informed

Professor Wei-Min Shen
University of Southern California

# Outline

## Search Algorithms

**- Uninformed Search (this lecture)**

Search strategies: breadth-first, uniform-cost, depth-first, bi-directional, …

**- Informed Search (next lecture)**

Search strategies: best-first, A*

Heuristic functions

# TYPES OF SEARCH ALGORITHMS

# Remember: Implementation of search algorithms

```
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
```

**Queuing-Fn(*queue, elements*)** is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.

# Remember: Implementation of search algorithms

```
Function General-Search(problem, Queuing-Fn) returns a solution, or failure
    nodesQueue ← make-queue(make-node(initial-state[problem]))
    loop do
            if nodesQueue is empty then return failure
            node ← Remove-Front(nodesQueue)
            if Goal-Test[problem] applied to State(node) succeeds then return node
            nodesQueue ← Queuing-Fn(nodesQueue, Expand(node, Operators[problem]))
    end
```

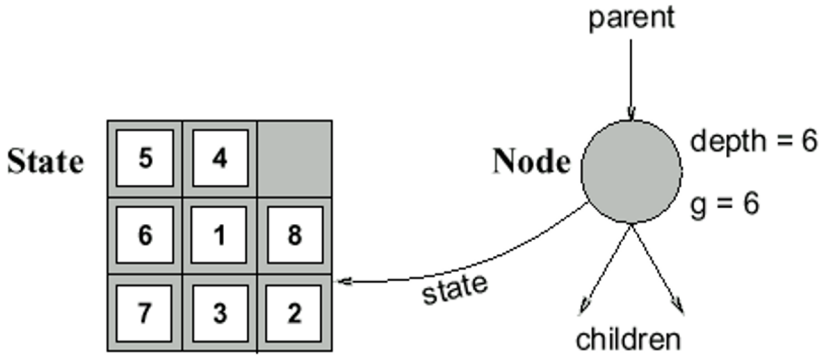
**Breadth-first search:** Enqueue expanded (children) nodes to the **back** of the queue (FIFO order)

**Depth-first search:** Enqueue expanded (children) nodes to the **front** of the queue (LIFO order)

**Uniform cost search:** Enqueue expanded (children) nodes so that queue is **ordered by the path (past) cost of the nodes** (priority queue order).

# Encapsulating *state* information in *nodes*

A *state* is a (representation of) a physical configuration
A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$
*States* do not have parents, children, depth, or path cost!




The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

# Evaluation of search strategies

- A search strategy is defined by picking the order of node expansion.

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take as function of num. of nodes?
  - **Space complexity:** how much memory does it require?
  - **Optimality:** does it guarantee the least-cost solution?

- Time and space complexity are measured in terms of:
  - $b$ – max branching factor of the search tree
  - $d$ – depth of the least-cost solution
  - $m$ – max depth of the search tree (may be infinity)

# Note: Approximations

- In our complexity analysis, we do not take the built-in loop-detection into account.

- The results only 'formally' apply to the variants of our algorithms WITHOUT loop-checks.
- Studying the effect of the loop-checking on the complexity is hard:
  - overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree.

- Also: our analysis DOES NOT take the length (space) of representing paths into account !!

http://www.cs.kuleuven.ac.be/~dannyd/FAI/

## Uninformed search strategies

Use only information available in the problem formulation

- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

# BREADTH-FIRST SEARCH

# Breadth-first search

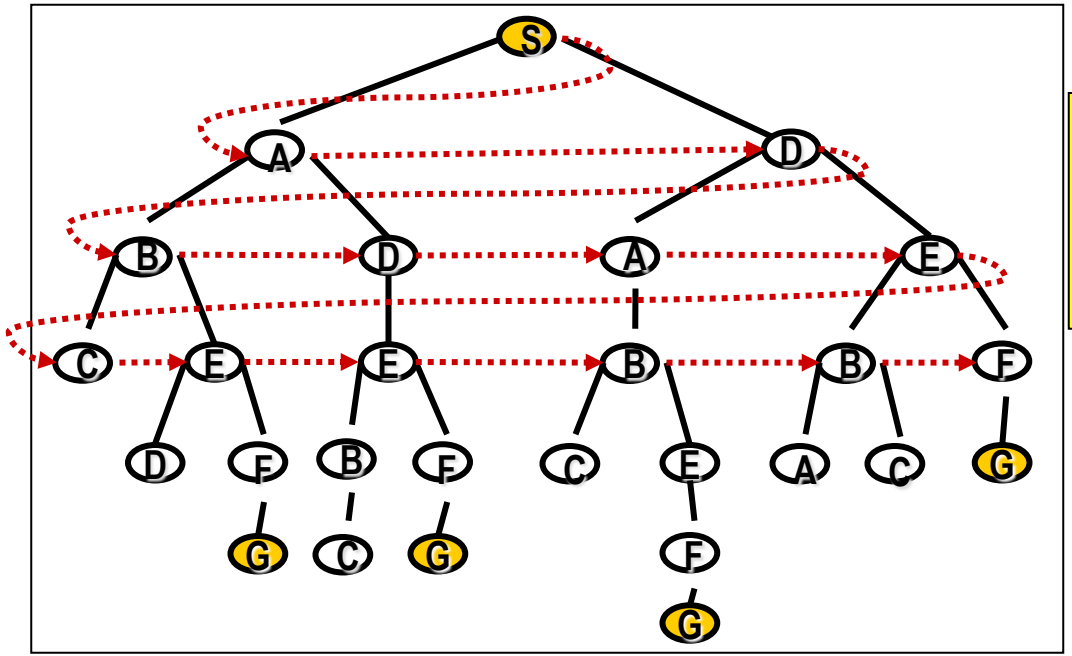
Expand shallowest unexpanded node

"First-in, first-out"

Implementation:

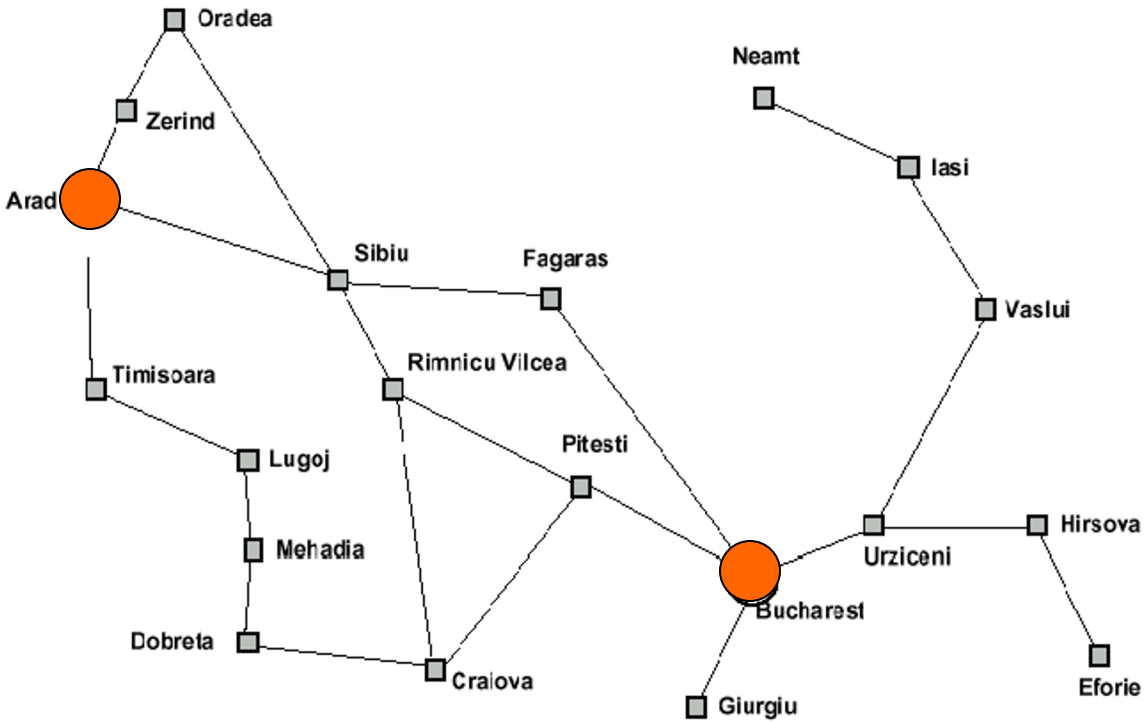
QUEUEINGFN = put successors at end of queue

# Breadth-first search

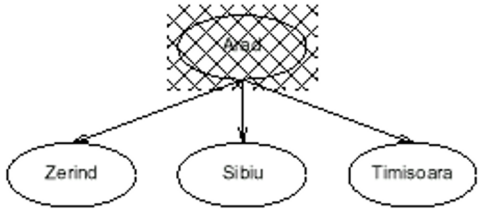



Move downwards, level by level, until goal is reached.

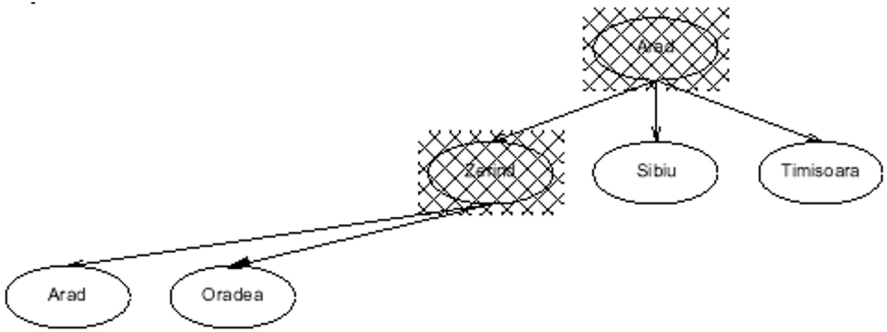
# Example: Traveling from Arad To Bucharest



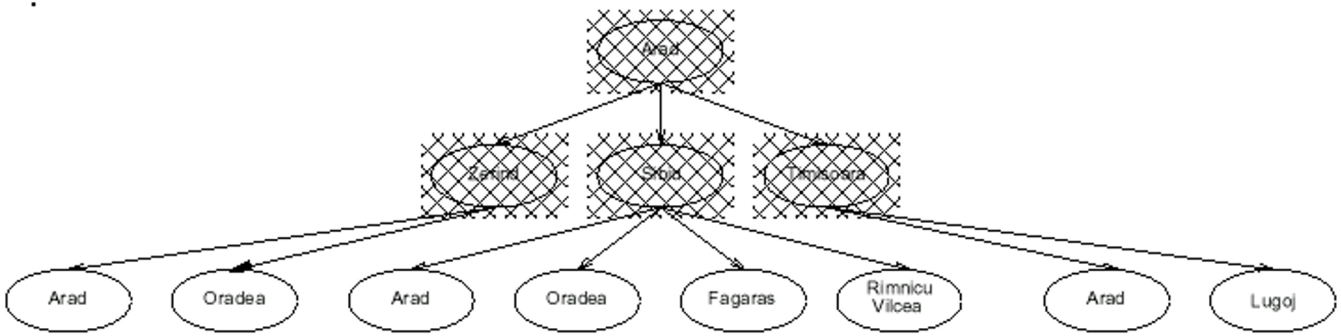
# Breadth-first search



# Breadth-first search



# Breadth-first search



# Properties of breadth-first search

- Completeness:
- Time complexity:
- Space complexity:
- Optimality:

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness:** does it always find a solution if one exists?
  - **Time complexity:** how long does it take as function of num. of nodes?
  - **Space complexity:** how much memory does it require?
  - **Optimality:** does it guarantee the least-cost solution?

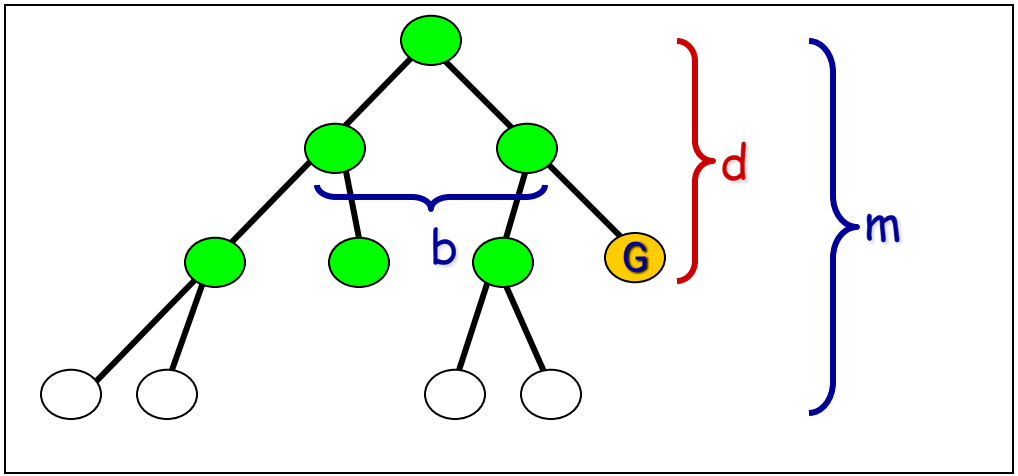
- Time and space complexity are measured in terms of:
  - $b$ – max branching factor of the search tree
  - $d$ – depth of the least-cost solution
  - $m$ – max depth of the search tree (may be infinity)

## Properties of breadth-first search

- Completeness: Yes, if *b* is finite
- Time complexity: $1+b+b^2+\ldots+b^d = O(b^d)$, i.e., exponential in *d*
- Space complexity: $O(b^d)$ (see following slides)
- Optimality: Yes (assuming cost = 1 per step)

# Time complexity of breadth-first search

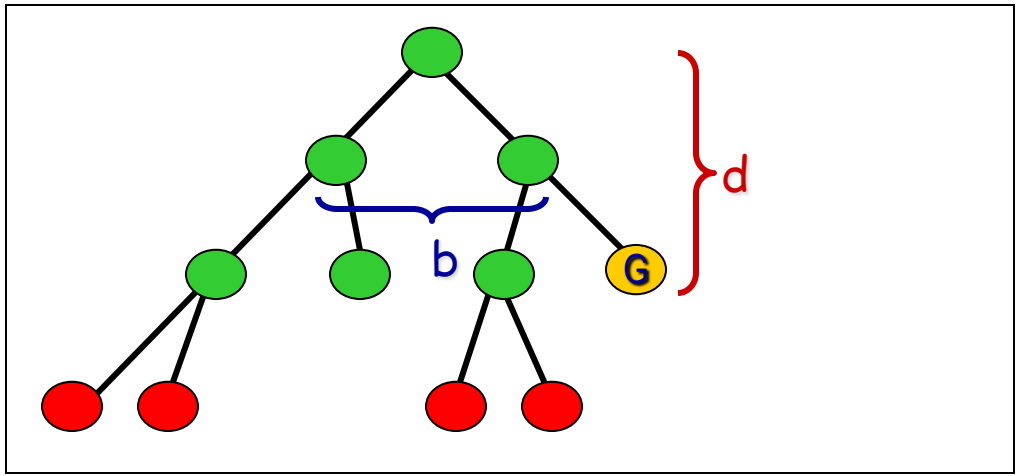
- If a goal node is found on depth d of the tree, all nodes up till that depth are created and examined (note: and the children of nodes at depth d are created and enqueued, but not yet examined).




- Thus: $O(b^d)$

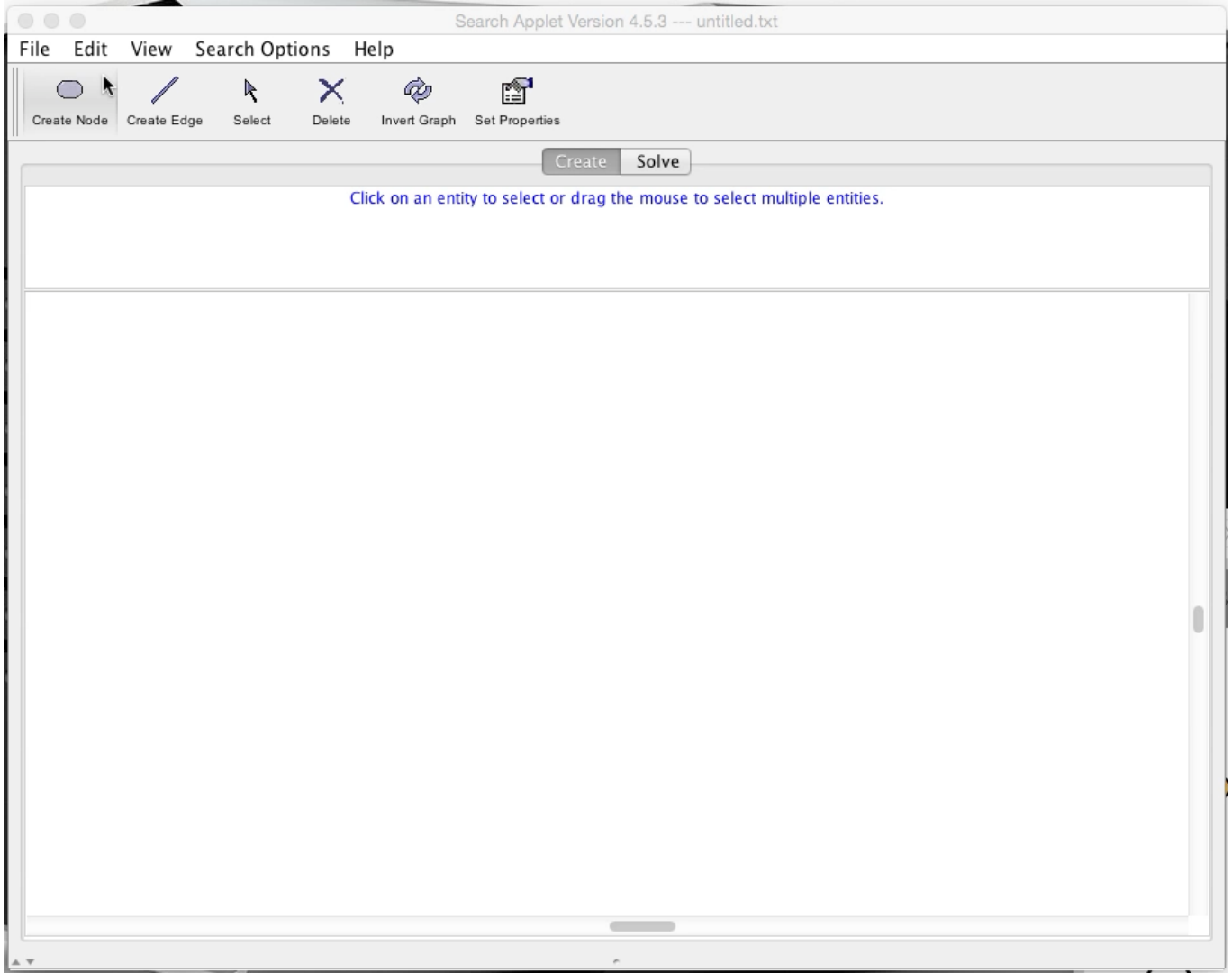
# Space complexity of breadth-first

- Largest number of nodes in QUEUE is reached on the level d+1 just beyond the goal node.




- QUEUE contains all  nodes. (Thus: 4) .
- In General: $b^{d+1} - b \sim b^{d}$

# Demo


Search Applet Version 4.5.3 --- untitled.txt
File
Edit
View
Search Options
Help
Create Node
Create Edge
Select
Delete
Invert Graph
Set Properties
Create
Solve
Click on an entity to select or drag the mouse to select multiple entities.

# UNIFORM-COST SEARCH

## Uniform-cost search

Expand least-cost unexpanded node

Implementation:

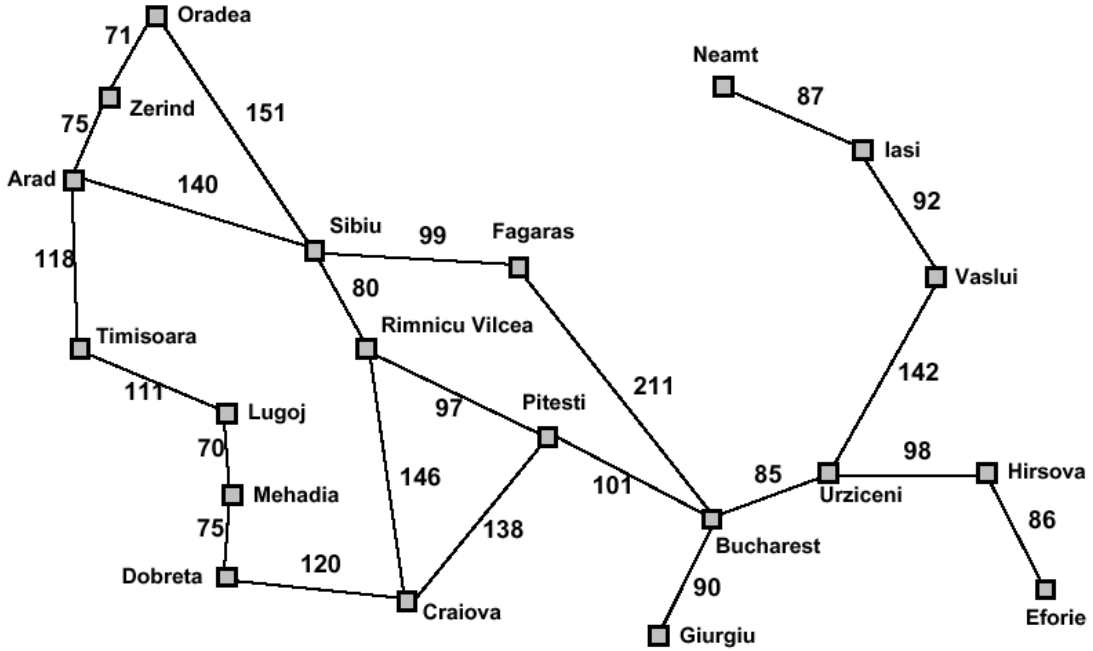
QUEUEINGFN = insert in order of increasing path cost


*So, the queueing function keeps the node list sorted by increasing path cost, and we expand the first unexpanded node (hence with smallest path cost)*

A refinement of the breadth-first strategy:

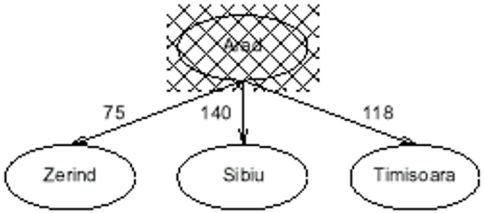
Breadth-first = uniform-cost with path cost = node depth

# Romania with step costs in km

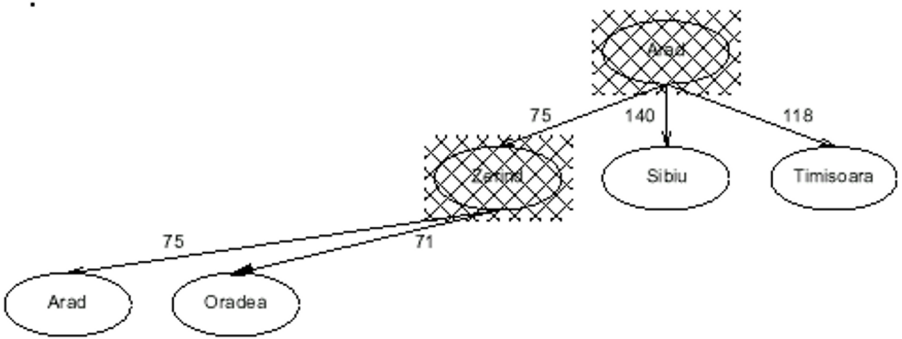



| Straight–line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea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

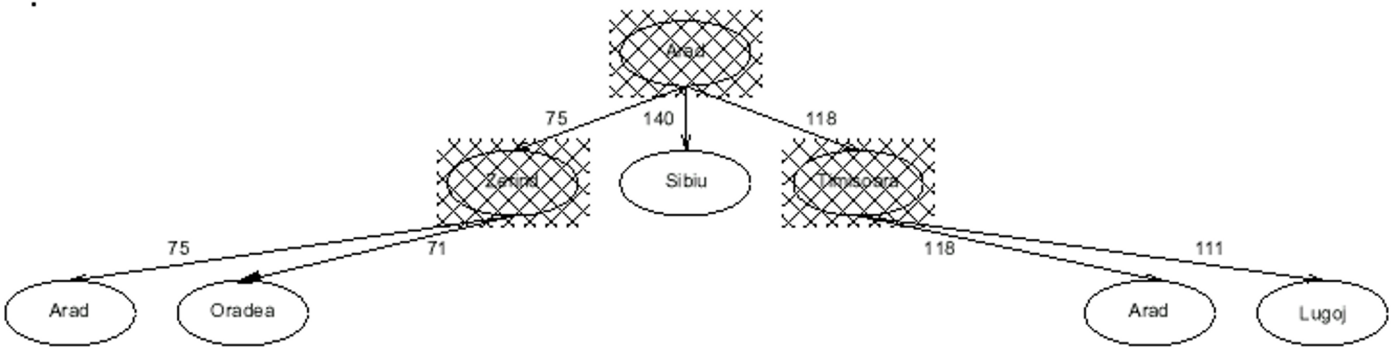
# Uniform-cost search



# Uniform-cost search



# Uniform-cost search



# Properties of uniform-cost search

- Completeness: Yes, if step cost $\geq \varepsilon > 0$
- Time complexity: # nodes with g $\leq$ cost of optimal solution, $\leq O(b^d)$
- Space complexity: # nodes with g $\leq$ cost of optimal solution, $\leq O(b^d)$
- Optimality: Yes, as long as path cost never decreases

g(n) is the path cost to node n

Remember:

b = branching factor

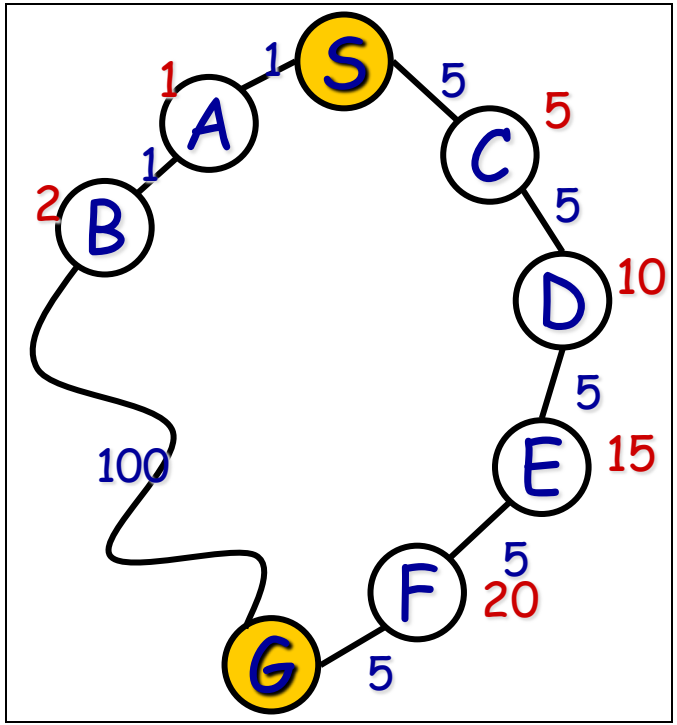
d = depth of least-cost solution

## Implementation of uniform-cost search

- Initialize Queue with root node (built from start state)
- Repeat until (Queue empty) or (first node has Goal state):
  - Remove first node from front of Queue
  - Expand node (find its children)
  - Reject those children that have already been considered, to avoid loops
  - Add remaining children to Queue, ***in a way that keeps entire queue sorted by increasing path cost***
- If Goal was reached, return success, otherwise failure

## Caution! Don’t termiate the search pre-maturally

- Uniform-cost search would not be optimal if it is terminated when ***any*** node in the queue has goal state.




- Uniform cost returns the path with cost 102 (if any goal node is considered a solution), while there is a path with cost 25.

# Note: Loop Detection

- In class, we saw that the search may fail or be sub-optimal if:

  - no loop detection: then algorithm runs into infinite cycles
    (A -> B -> A -> B -> …)

  - not queuing-up a node that has a state which we have already visited: may yield suboptimal solution

  - simply avoiding to go back to our parent: looks promising, but we have not proven that it works

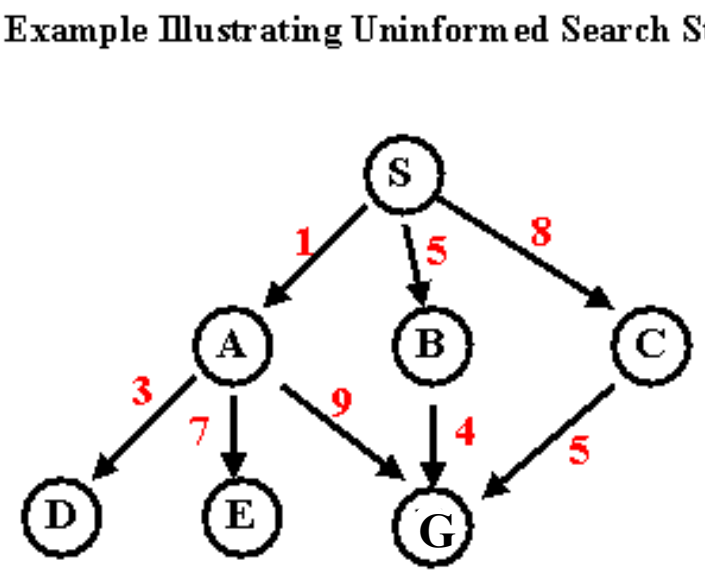
Solution? do not enqueue a node if its state matches the state of any of its parents (assuming path costs>0).

Indeed, if path costs > 0, it will always cost us more to consider a node with that state again than it had already cost us the first time.

**Is that enough??**

# Example

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/



# Breadth-First Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

## Breadth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-END)

| exp. node | nodes list |
|---|---|
| | { S } |
| S | { A B C } |
| A | { B C D E G } |
| B | { C D E G G' } |
| C | { D E G G' G" } |
| D | { E G G' G" } |
| E | { G G' G" } |
| G | { G' G" } |

Solution path found is S A G <-- this G also has cost 10

Number of nodes expanded (including goal node) = 7

# Uniform-Cost Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

## Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

| exp. node | nodes list |
|---|---|
| | { S } |
| S | { A(1) B(5) C(8) } |
| A | { D(4) B(5) C(8) E(8) G(10) } (NB, we don't return G) |
| D | { B(5) C(8) E(8) G(10) } |
| B | { C(8) E(8) G(9) G(10) } |
| C | { E(8) G(9) G(10) G(13) } |
| E | { G(9) G(10) G(13) } |
| G | { } |

Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

# Note: Queueing in Uniform-Cost Search

In the previous example, it is wasteful (but not incorrect) to queue-up three nodes with G state, if our goal if to find the least-cost solution:

Although they represent different paths, we know for sure that the one with smallest path cost (9 in the example) will yield a solution with smaller total path cost than the others.

So we can refine the queueing function by:

- queue-up node if

1) its state does not match the state of any parent // it is new

and 2) path cost smaller than path cost of any // it is better unexpanded node with same state in the queue (and in this case, replace old node with same state by our new node)

**Is that it??**

# A Clean Robust Algorithm

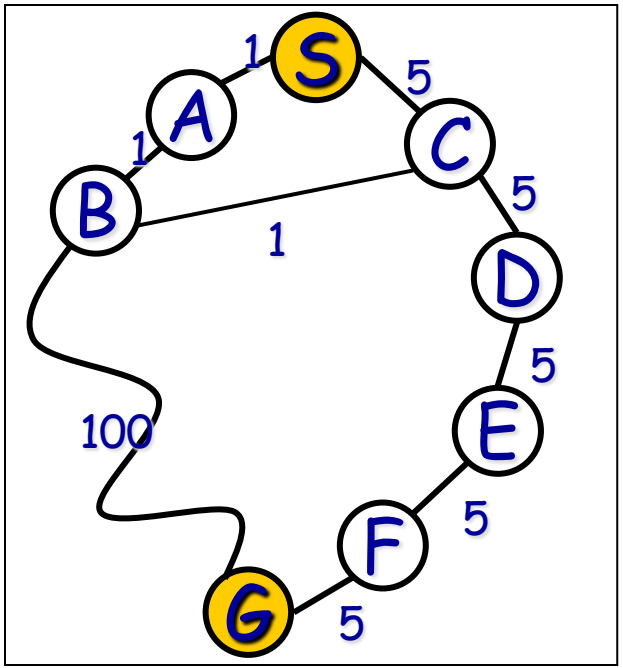
```
Function UniformCost-Search(problem, Queuing-Fn) returns a solution, or failure
    open ← make-queue(make-node(initial-state[problem]))
    closed ← [empty]
    loop do
            if open is empty then return failure
            currnode ← Remove-Front(open)
            if Goal-Test[problem] applied 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

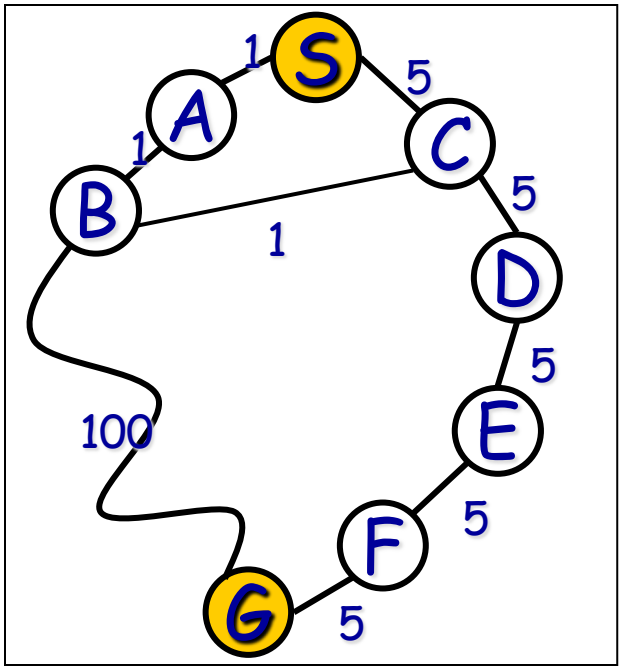
```
[... see previous slide ...]
        children ← Expand(currnode, Operators[problem])
        while children not empty
                child ← Remove-Front(children)
                if no node in open or closed has child's state
                        open ← Queuing-Fn(open, child)
             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

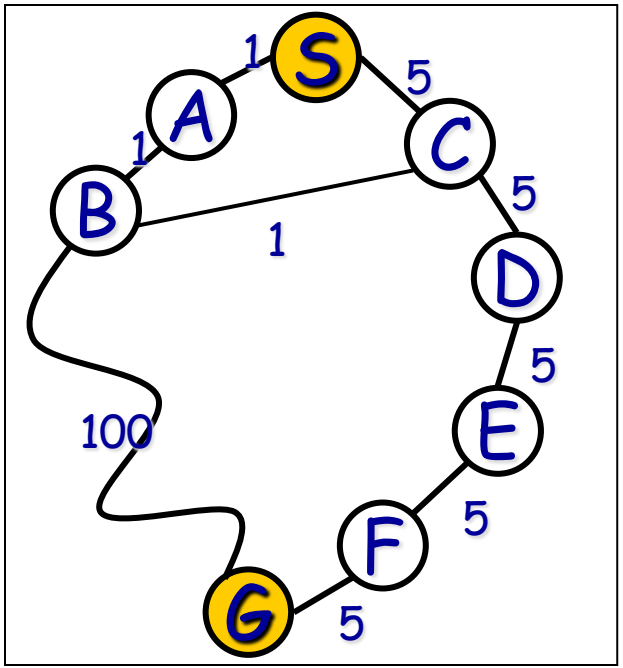



| # | State | Depth | Cost | Parent |
|---|---|---|---|---|
| 1 | S | 0 | 0 | - |
| 2 | A | 1 | 1 | 1 |
| 3 | C | 1 | 5 | 1 |

Black = open queue
Grey = closed queue

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

# Example

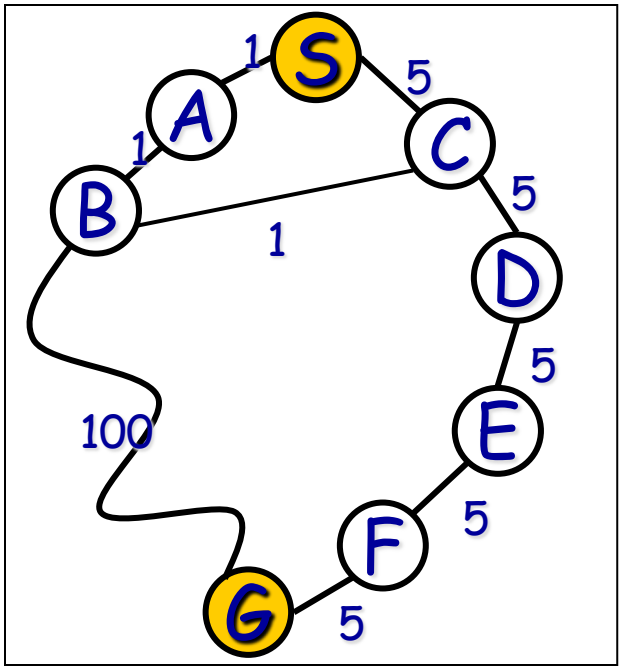



| # | State | Depth | Cost | Parent |
|---|---|---|---|---|
| 1 | S | 0 | 0 | - |
| 2 | A | 1 | 1 | 1 |
| 4 | B | 2 | 2 | 2 |
| 3 | C | 1 | 5 | 1 |

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.

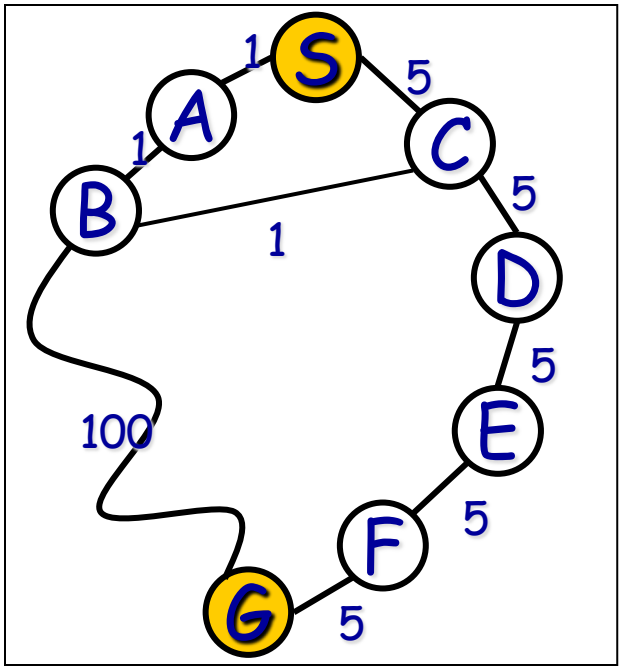
# Example




| # | 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 |

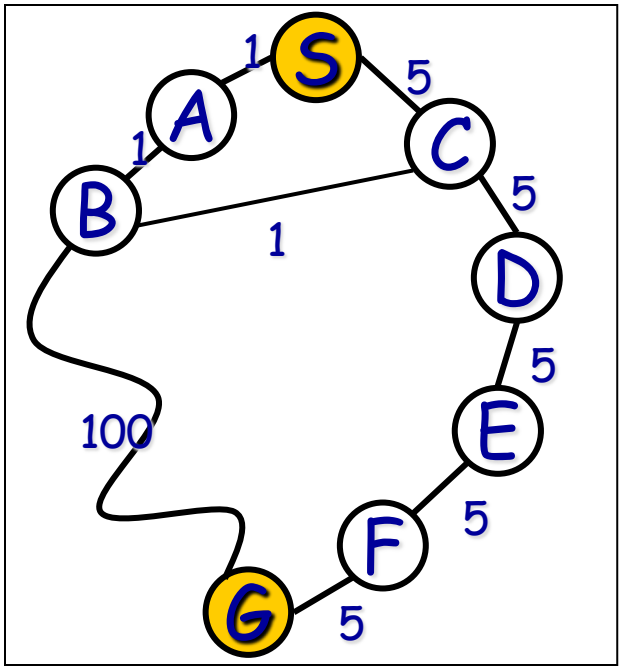
Node 4 has a successor with state C and
Cost smaller than node #3 in *open* that
Also had state C; so we update *open*
To reflect the shortest path.

# Example



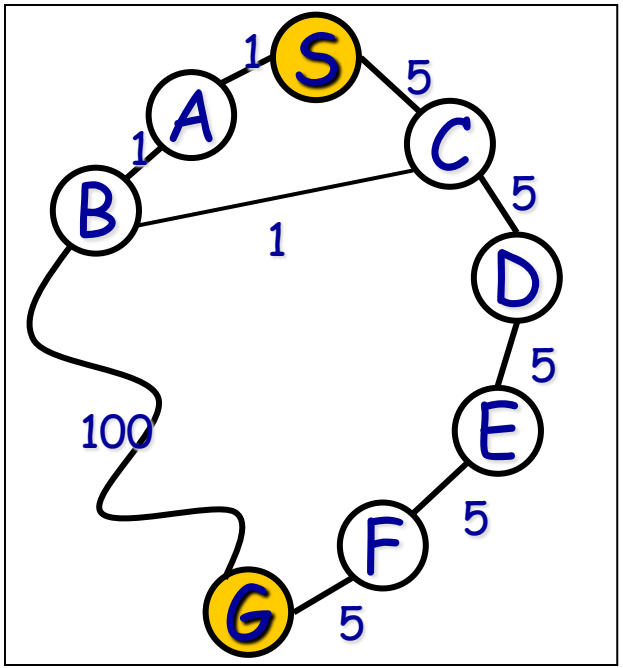

| # | State | Depth | Cost | Parent |
|---|---|---|---|---|
| 1 | S | 0 | 0 | - |
| 2 | A | 1 | 1 | 1 |
| 4 | B | 2 | 2 | 2 |
| 5 | C | 3 | 3 | 4 |
| 7 | D | 4 | 8 | 5 |
| 6 | G | 3 | 102 | 4 |

# Example



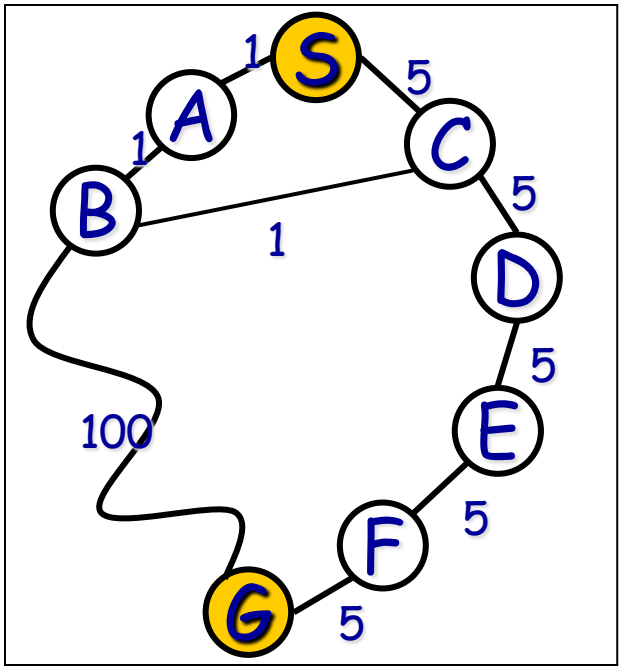

| # | State | Depth | Cost | Parent |
|---|---|---|---|---|
| 1 | S | 0 | 0 | - |
| 2 | A | 1 | 1 | 1 |
| 4 | B | 2 | 2 | 2 |
| 5 | C | 3 | 3 | 4 |
| 7 | D | 4 | 8 | 5 |
| 8 | E | 5 | 13 | 7 |
| 6 | G | 3 | 102 | 4 |

# 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 |

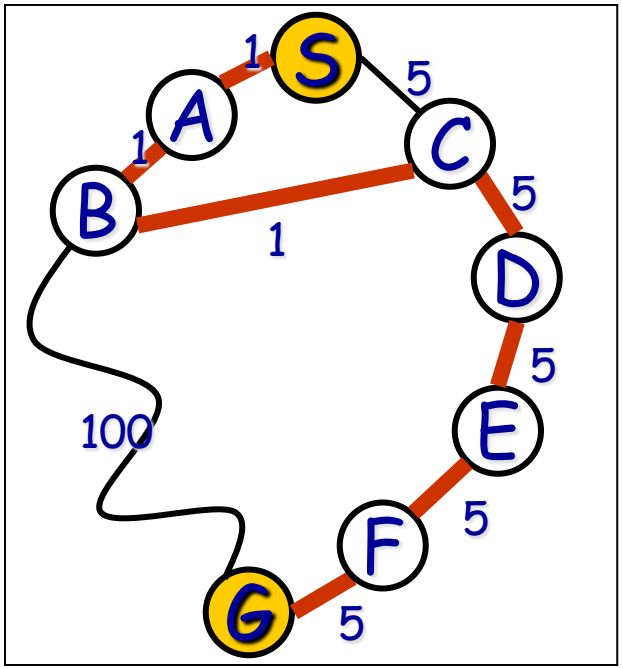
# Example




| # | 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 |

The node with state G and cost 102 has been removed from the open queue and replaced by cheaper node with state G and code 23 which was pushed into the open queue.

# 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 |

Goal reached

# DEPTH-FIRST SEARCH

# Depth-first search

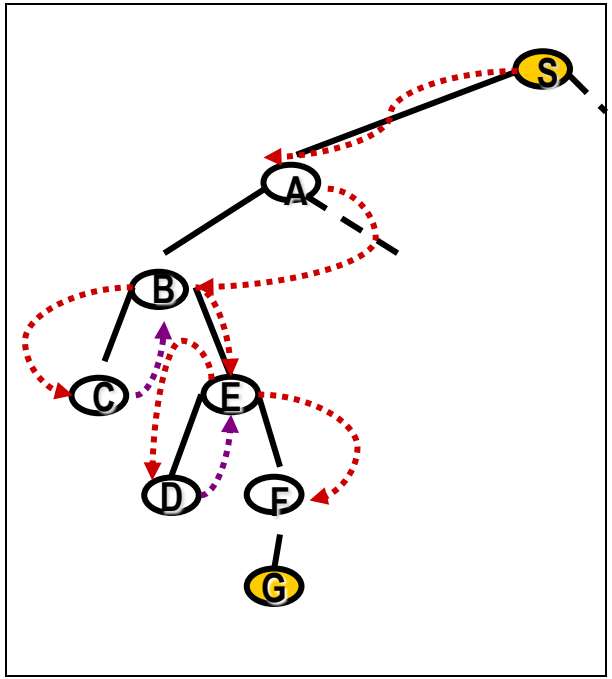
Expand deepest unexpanded node

"last-in, first-out"

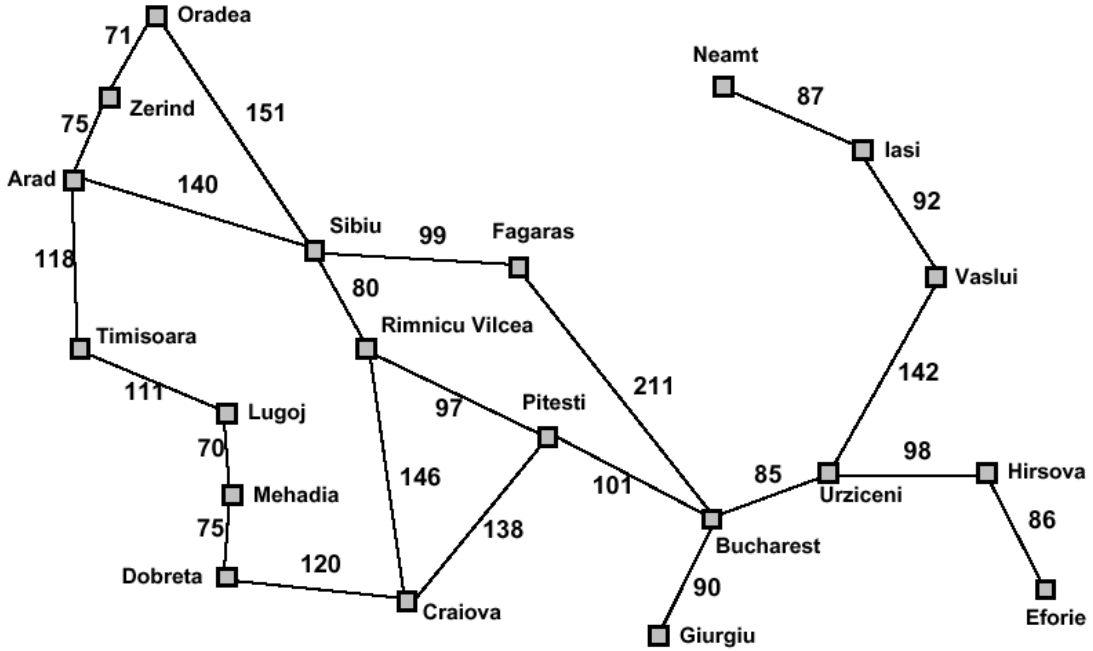
Implementation:

QUEUEINGFN = insert successors at front of queue

# Depth First Search

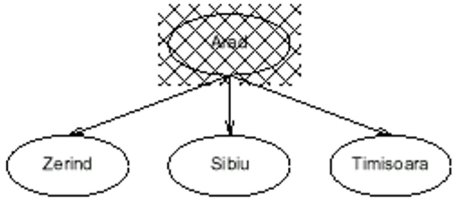


# Romania with step costs in km

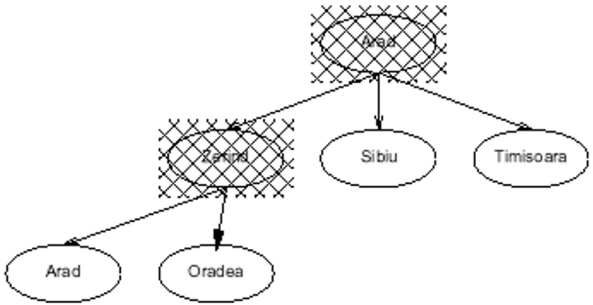



| Straight–line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea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

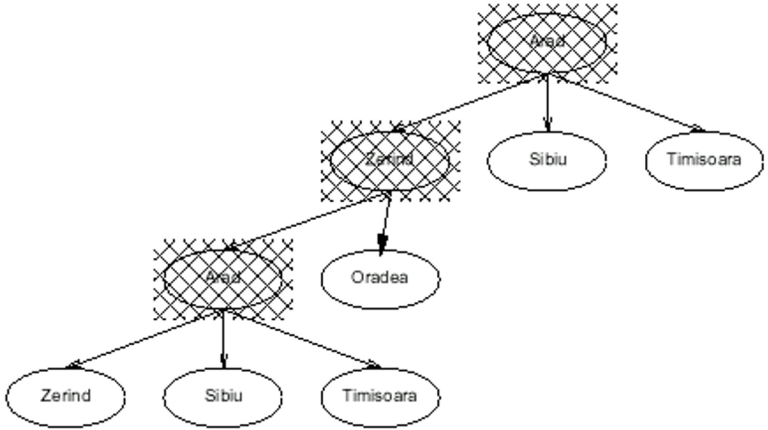
# Depth-first search



# 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

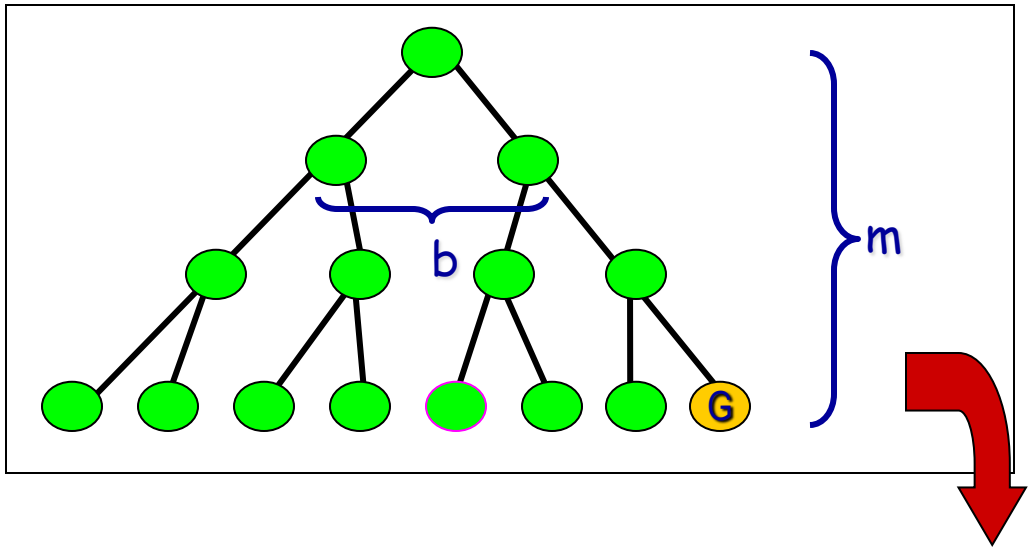
Remember:

b = branching factor

m = max depth of search tree

# Time complexity of depth-first: details

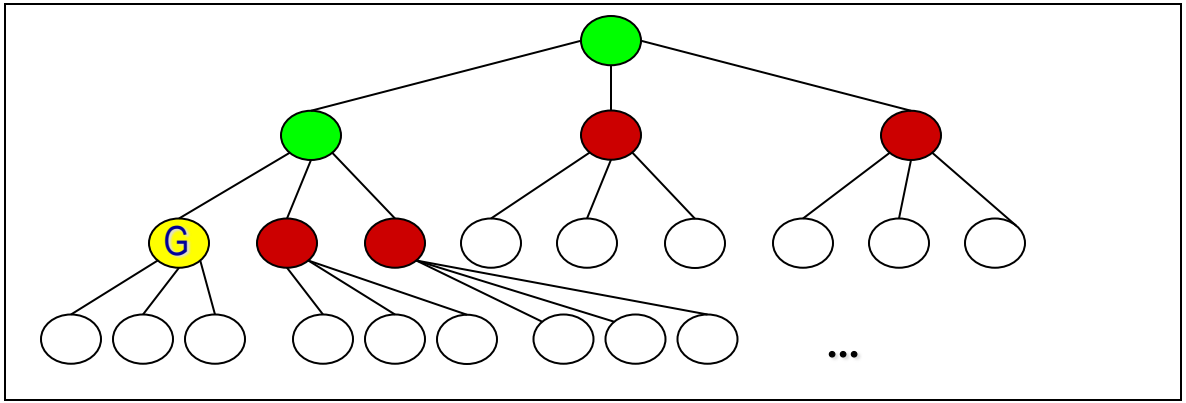
- In the worst case:
  - the (only) goal node may be on the right-most branch,




- 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 = 2,  b = 3 :




- QUEUE contains all  nodes.  Thus: 4.
- In General:  ((b-1) * m)
- 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.

**Optimal**: it does not guarantee to find the least-cost solution

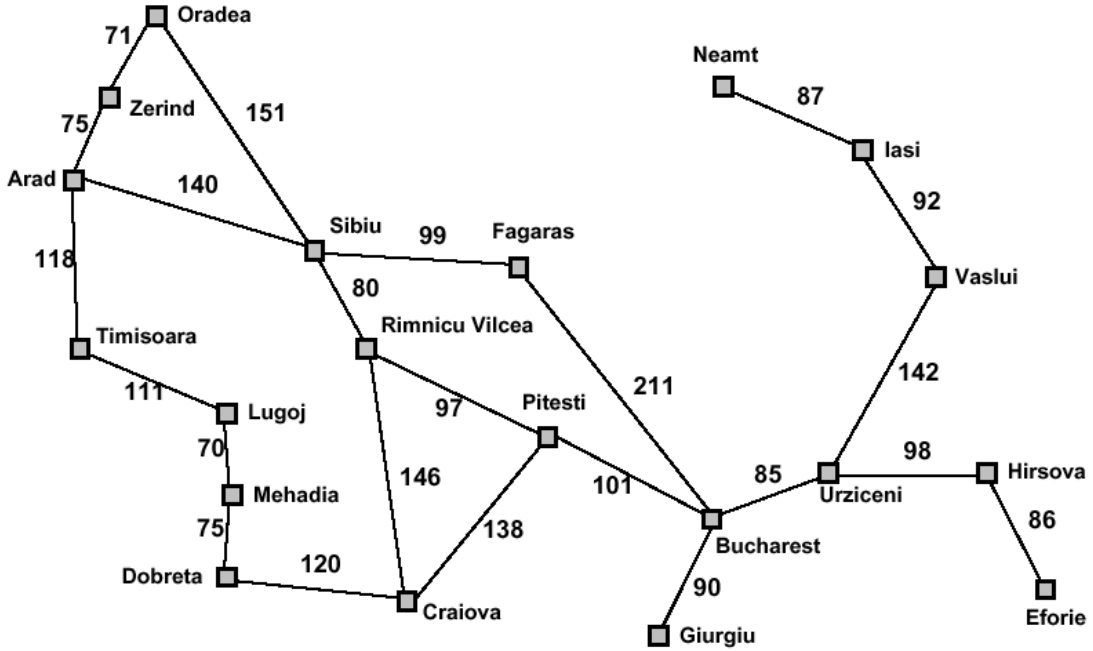
## Iterative deepening search

```
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 |

Adobe Acrobat - [chapter03.pdf]
File Edit Document Tools View Window Help

# Iterative deepening search $l = 0$



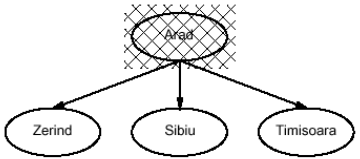

153% 53 of 63 11 x 8.5 in
Start CS 564 laurent talks CS 561a Aima Adobe Acr... Microsoft Pow... Telnet - pollux... books Adobe Photos... 8:22 PM

Adobe Acrobat - [chapter03.pdf]
File Edit Document Tools View Window Help

# Iterative deepening search $l = 1$




153% 54 of 63 11 x 8.5 in
Start CS 564 laurent talks CS 561a Aima Adobe Acr... Microsoft Pow... Telnet - pollux... books Adobe Photos... 8:24 PM

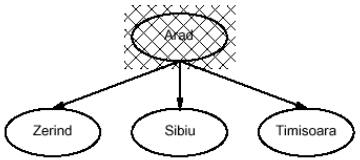

Arad
Zerind
Sibiu
Timisoara

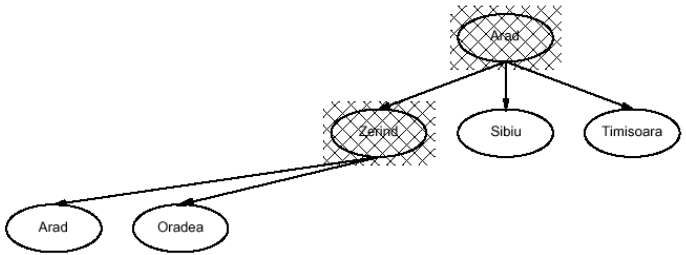
Adobe Acrobat - [chapter03.pdf]
File Edit Document Tools View Window Help

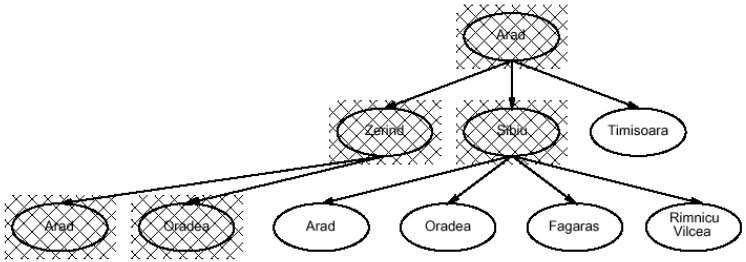
# Iterative deepening search $l = 2$

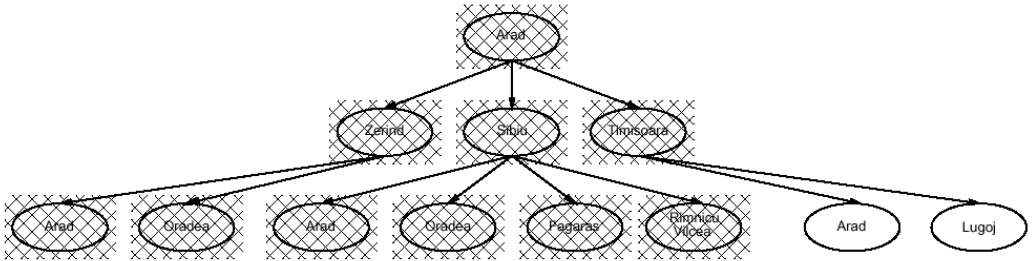



153% 56 of 63 11 x 8.5 in
Start CS 564 laurent talks CS 561a Aima Adobe Acr... Microsoft Pow... Telnet - pollux... books Adobe Photos... 8:25 PM


Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Rimnicu
Vilcea


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:

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

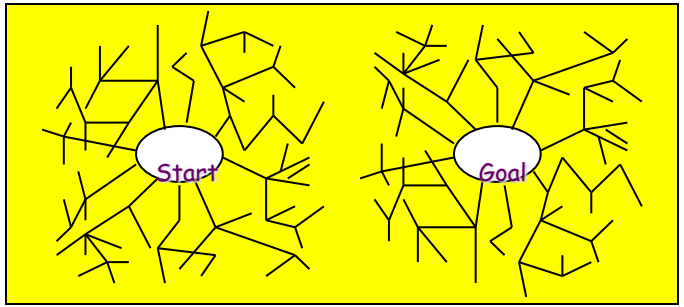
- In general, iterative deepening is preferred to depth-first or breadth-first when search space large and depth of solution not known.

# BI-DIRECTIONAL SEARCH

# Bidirectional search

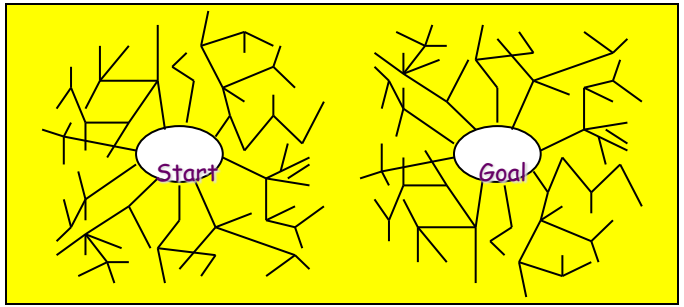
- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.

- 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;

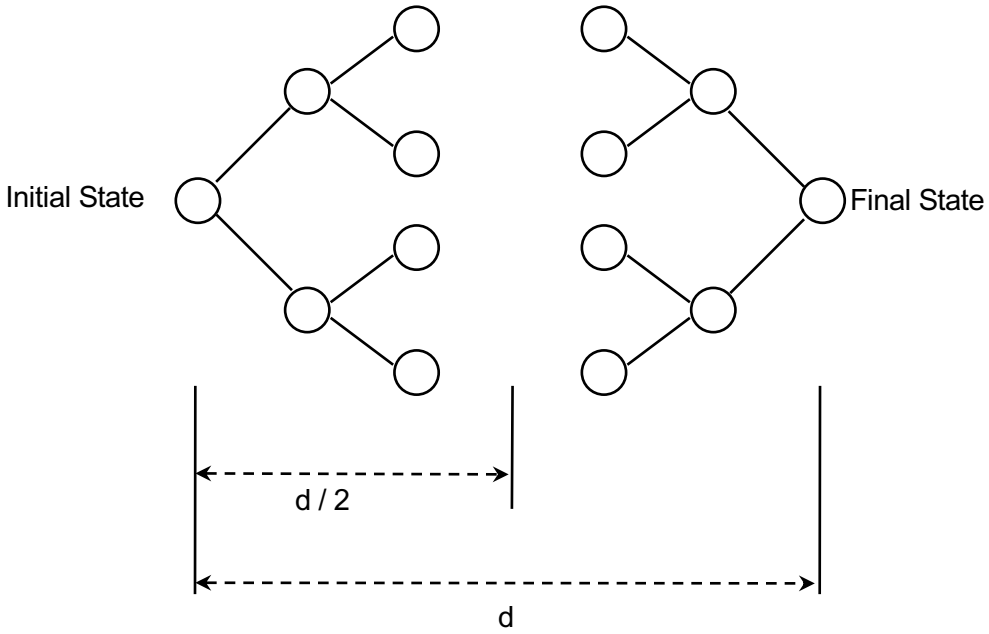
3. IF QUEUE1 and QUEUE2 share a state
   THEN success;
   ELSE failure;

## 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.