Artificial Intelligence

CS4365 --- Fall 2022

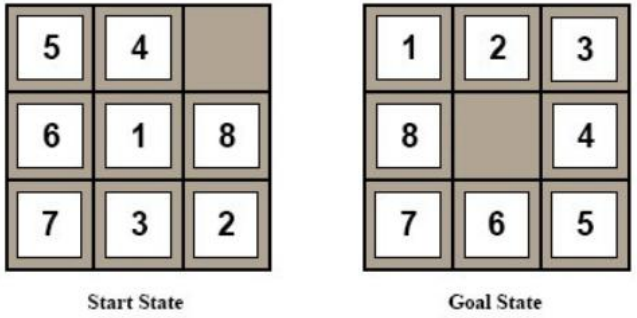
Uninformed Search

Instructor: Yunhui Guo

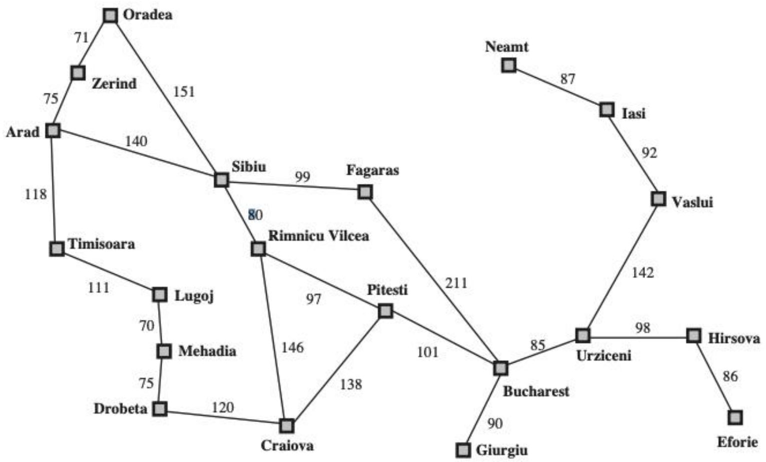
Problem Solving as Search

- Search is needed to solve many real-world problems

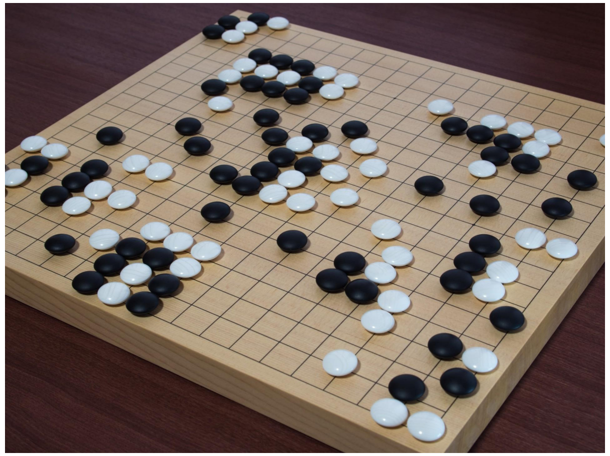
8-puzzle problem



Planning



Go

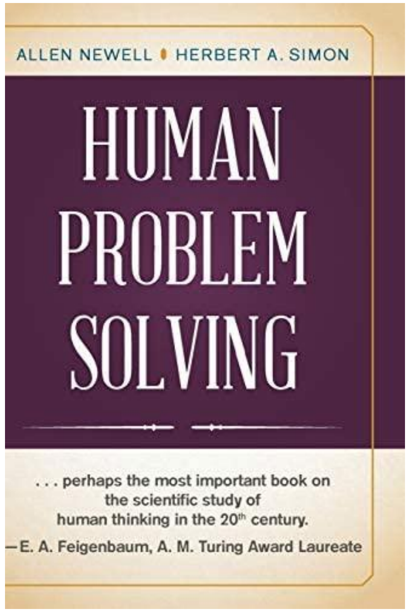


Problem Solving as Search

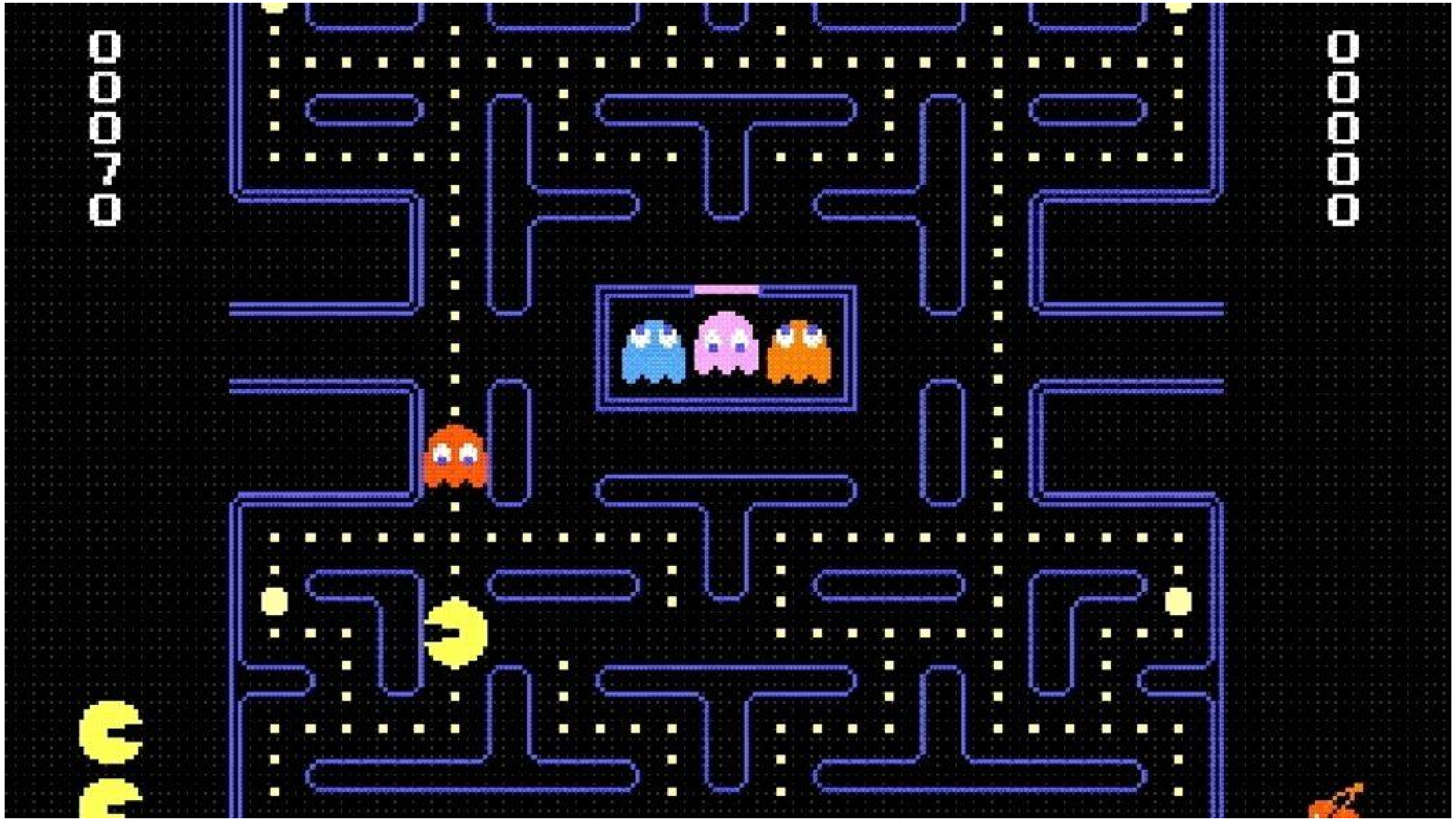
- Search is a central topic in AI
  - Originated with Newell and Simon's work on problem solving.
    Famous book:
    "Human Problem Solving" (1972)

  - Automated reasoning is a natural search task

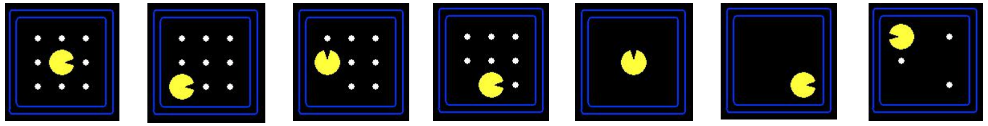
  - More recently: Given that almost all AI formalisms (planning, learning, etc.) are NP-complete or worse, some form of search is generally unavoidable (no "smarter" algorithm available).



Pac-Man

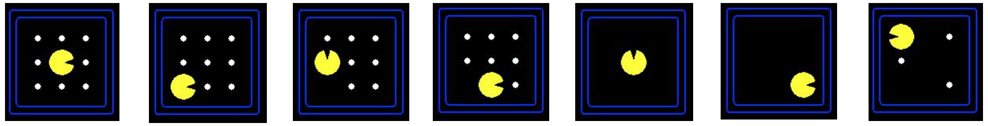


Define a Search Problem

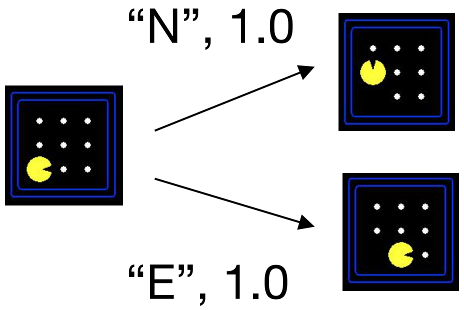
- A search problem consists of:
  - State space: 
  - A successor function:
    (action + cost)
  - A start state and a goal test

Define a Search Problem

- A search problem consists of:
  - State space:



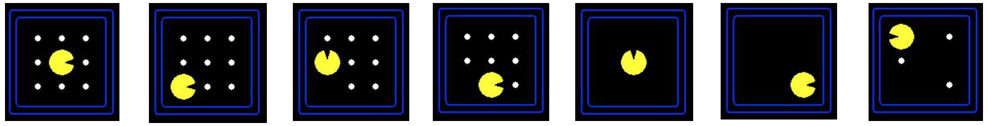
  - A successor function:
(action + cost)



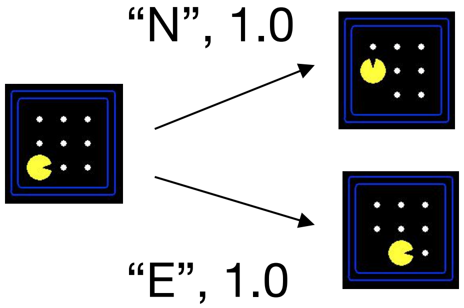
  - A start state and a goal test

Define a Search Problem

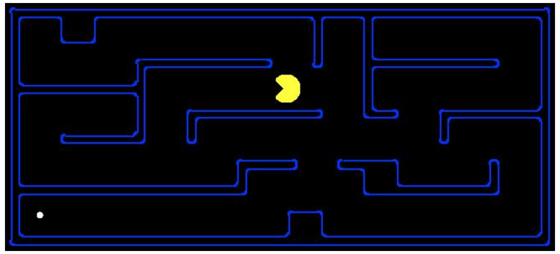
- A search problem consists of:
  - State space:



  - A successor function: (action + cost)



  - A start state and a goal test

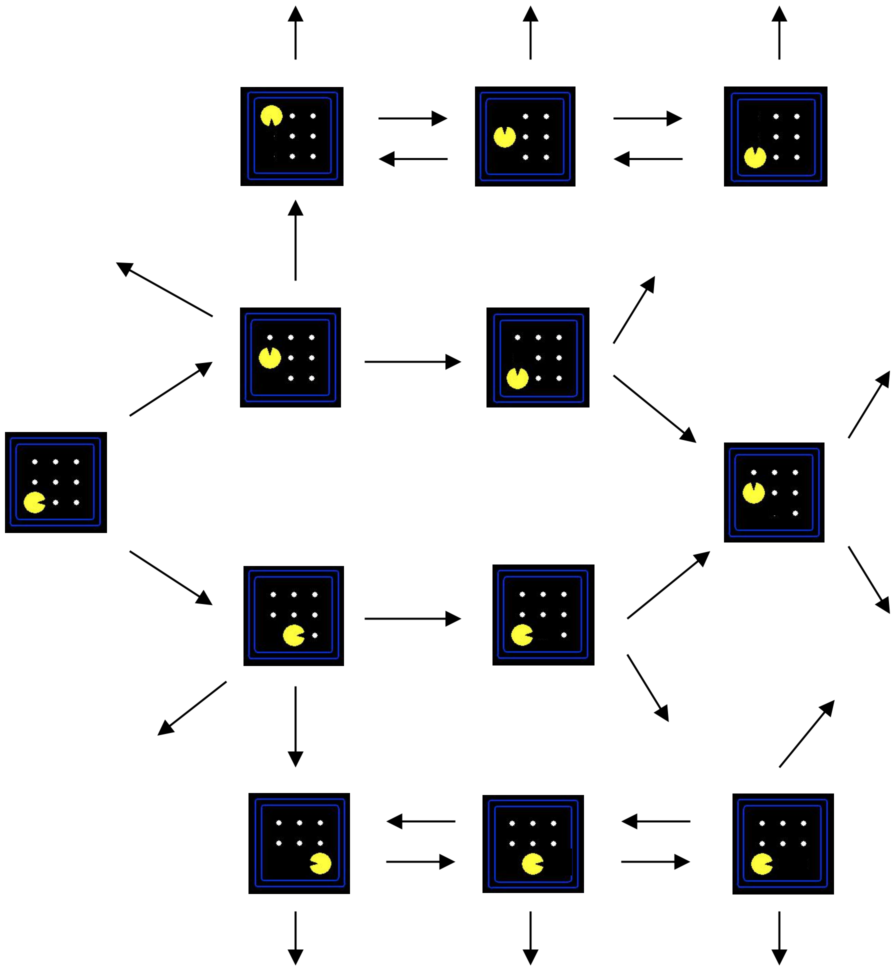


Define a Search Problem

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

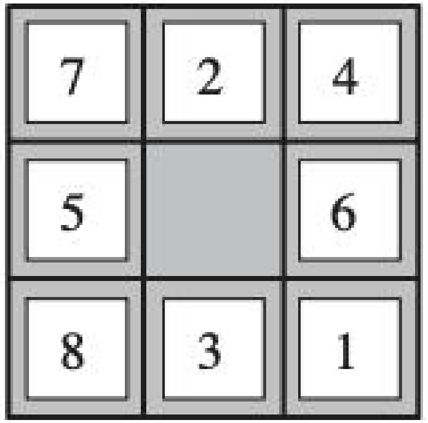
- Path cost – function that assigns a cost to a path; relevant if more than one path leads to the goal, and we want the shortest path.

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

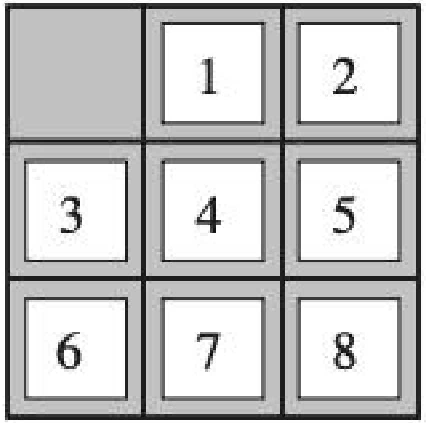


Example: The 8-Puzzle

- State space: ?
- Initial state: ?
- Goal test: ?
- Successor function: ?
- Path cost: ?



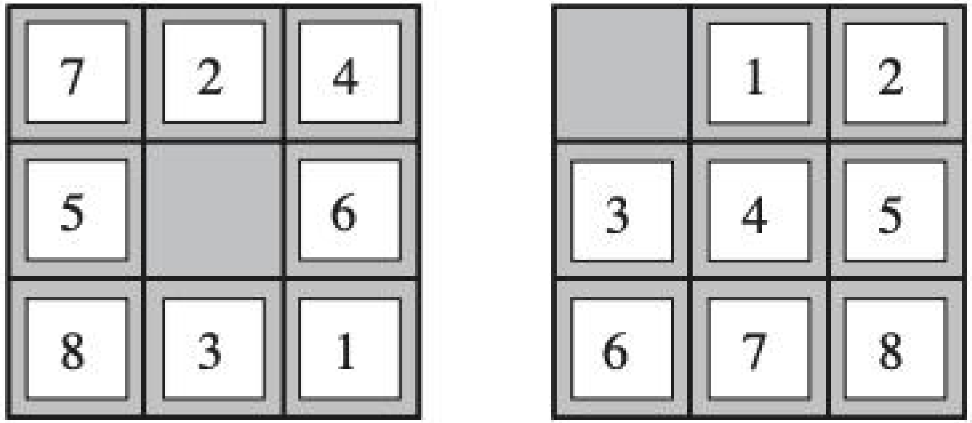
Start State



Goal State

Example: The 8-Puzzle

- State space:
  - The location of each tile
- Initial state:
  - Any state can be the inital state
- Goal test:
  - Whether the state matches the goal state
- Successor function:
  - The movement of the blank space (Left, Right, Up or Down)
- Path cost:
  - Each step costs 1



Start State

Goal State

Example: 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

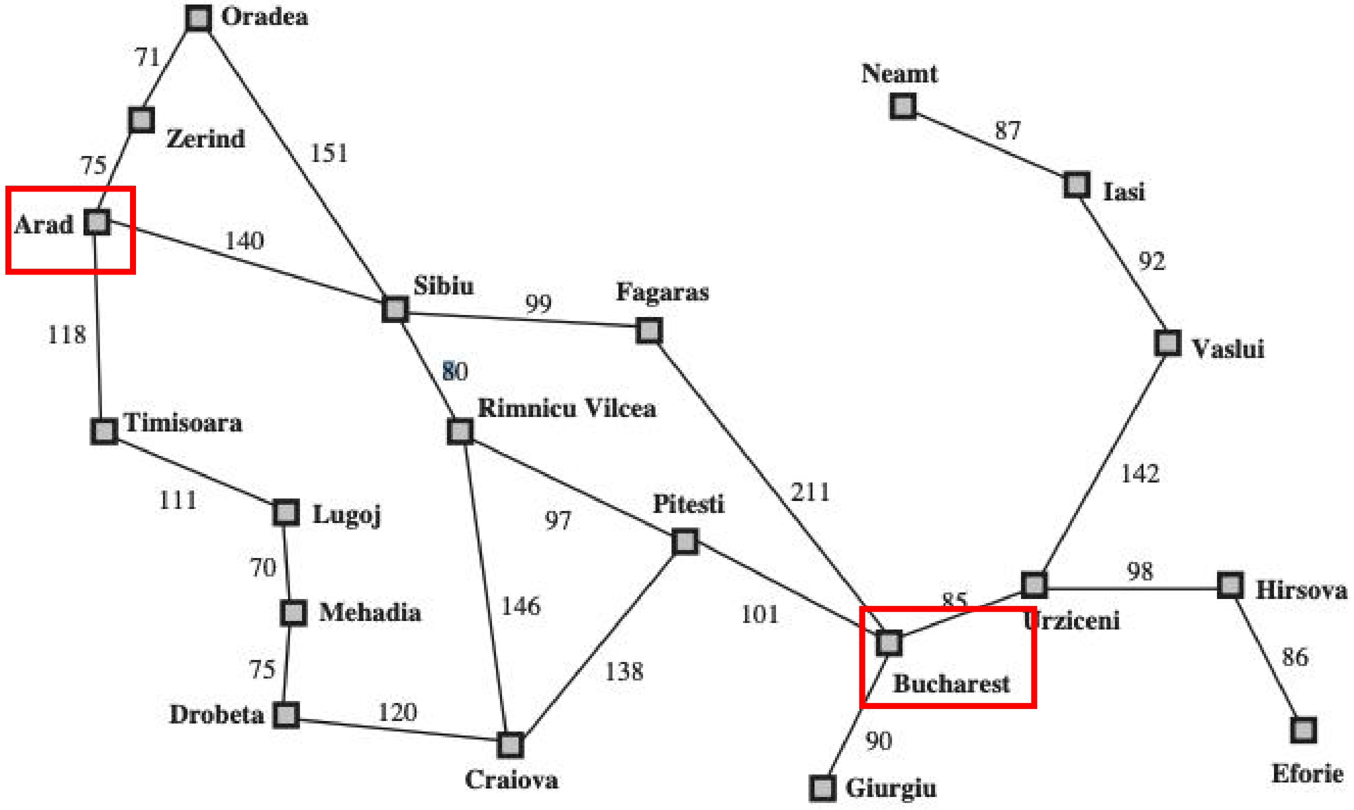
Cryptarithmetic, cont.

- State space: an 8-tuple indicating a (partial) assignment of digits to letters.
- Goal test: all letters have been assigned digits and sum is correct
- Successor function: represents the act of assigning digits to letters
- Path cost: all solutions are equally valid; step cost = 0

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

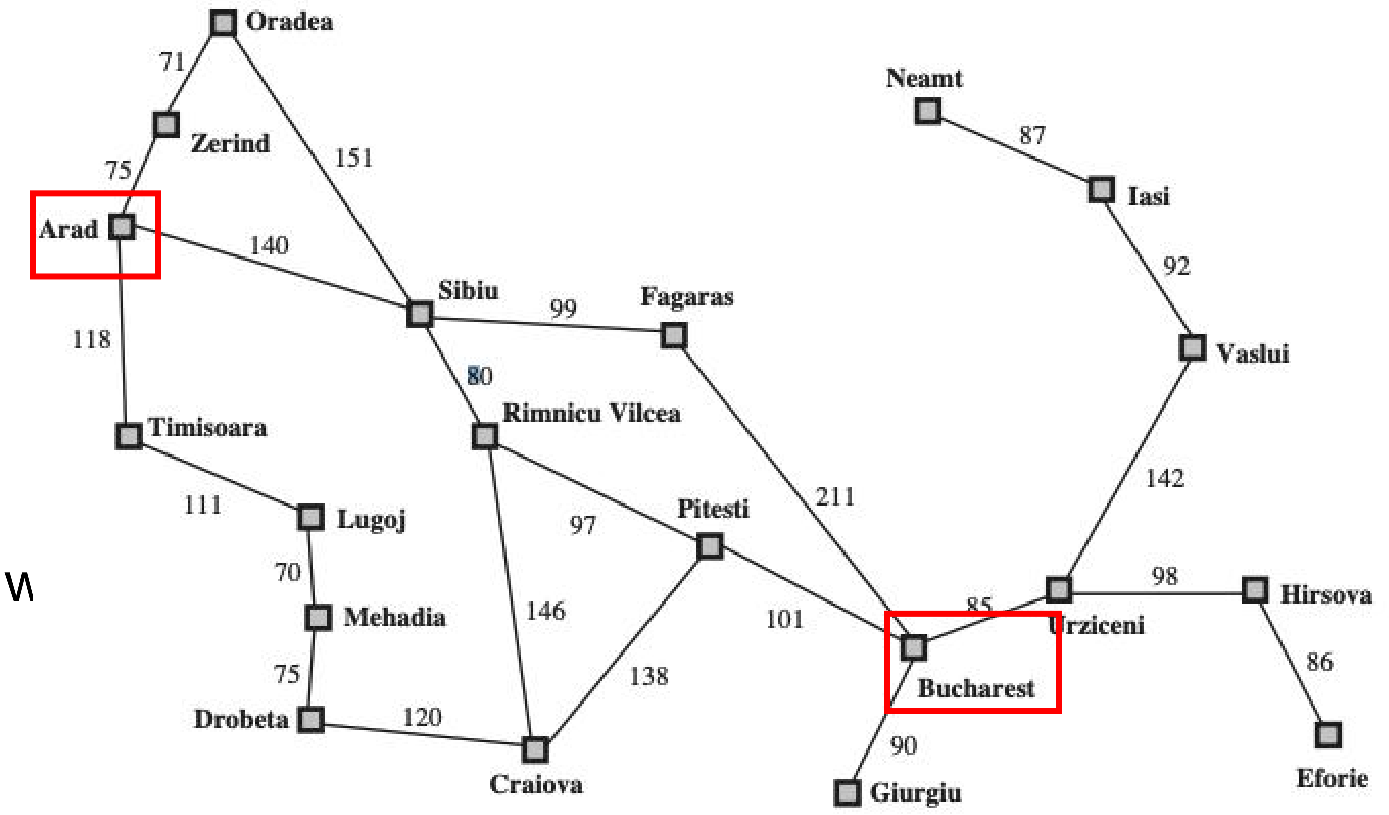
Example: Traveling in Romania

- State space: ?
- Initial state: ?
- Goal test: ?
- Successor function: ?
- Path cost: ?



Example: Traveling in Romania

- State space:
  - Cities
- Initial state:
  - Arad
- Goal test:
  - Is state == Bucharest
- Successor function:
  - Roads: go to adjacent cities w
  cost = distance
- Path cost:
  - The cost of a path



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

- Expanding: apply each legal action to the current state

- The leaf nodes available for expansion is called the frontier

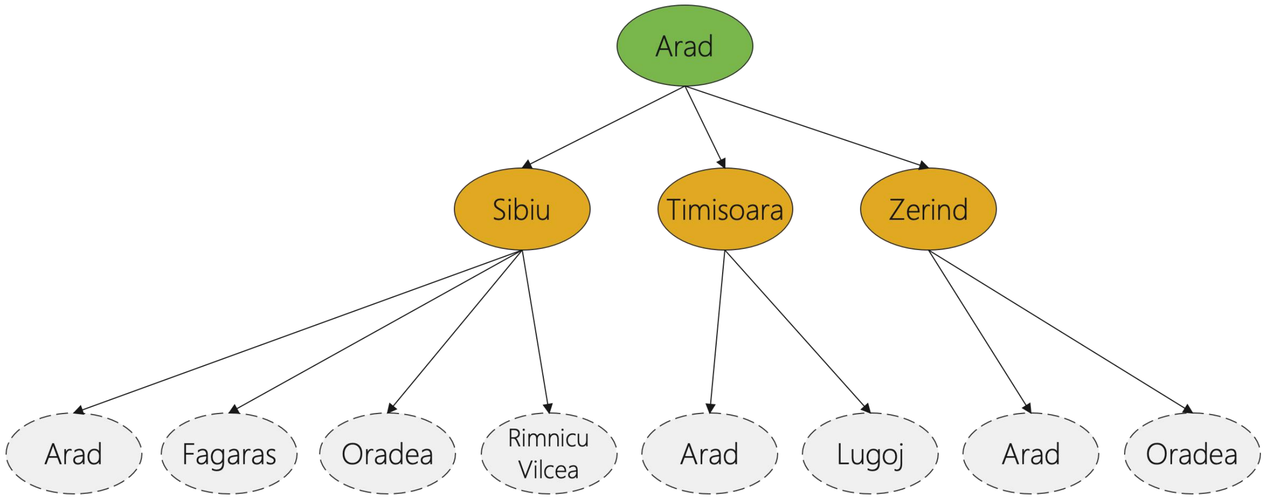
Search procedure defines a search tree

root node — initial state

Children of a node — successor states

Leaves of tree (frontier) — states not yet expaned

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



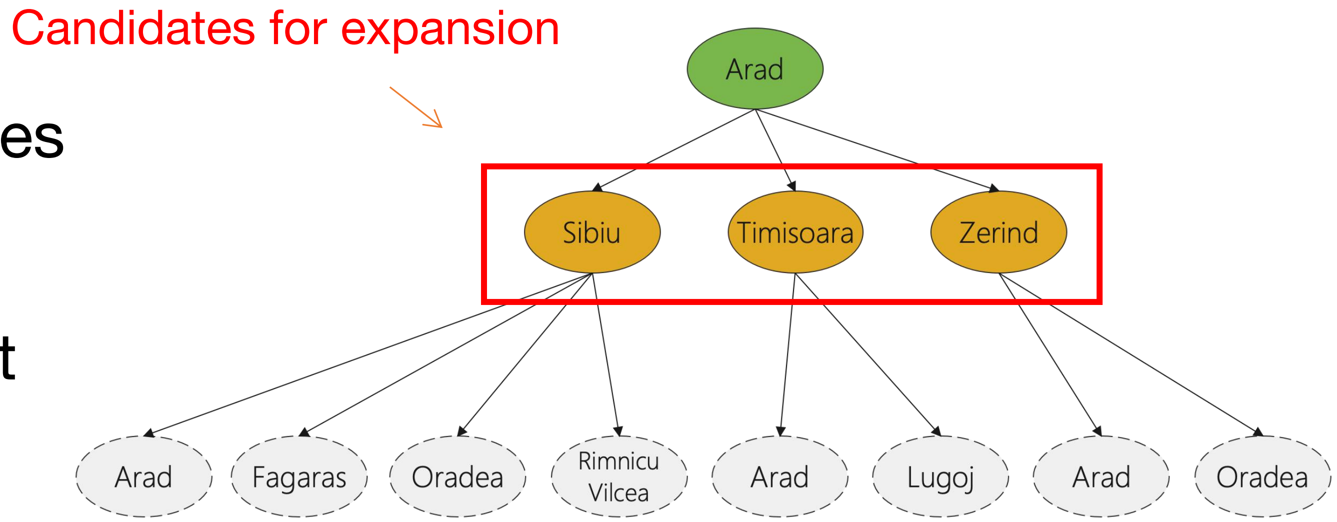
Search procedure defines a search tree

root node — initial state

Children of a node — successor states

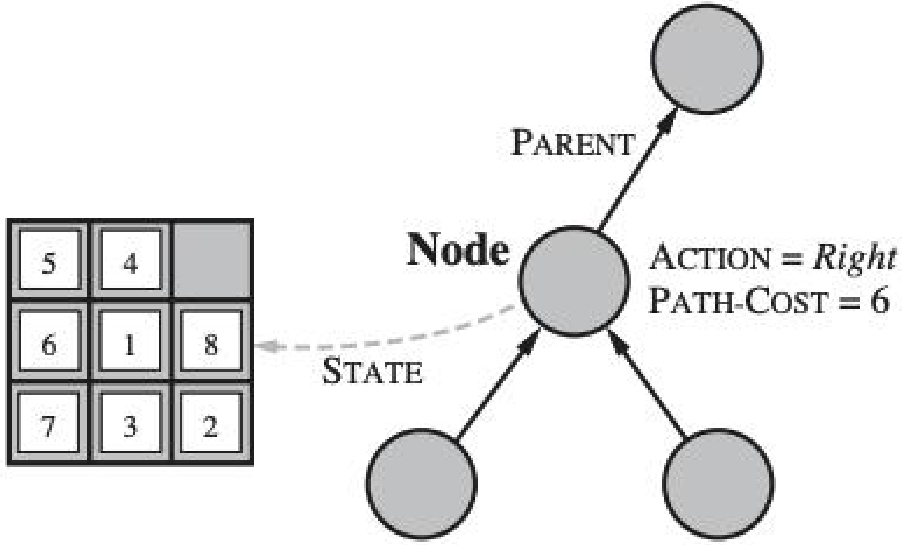
Leaves of tree (frontier) — states not yet expaned

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



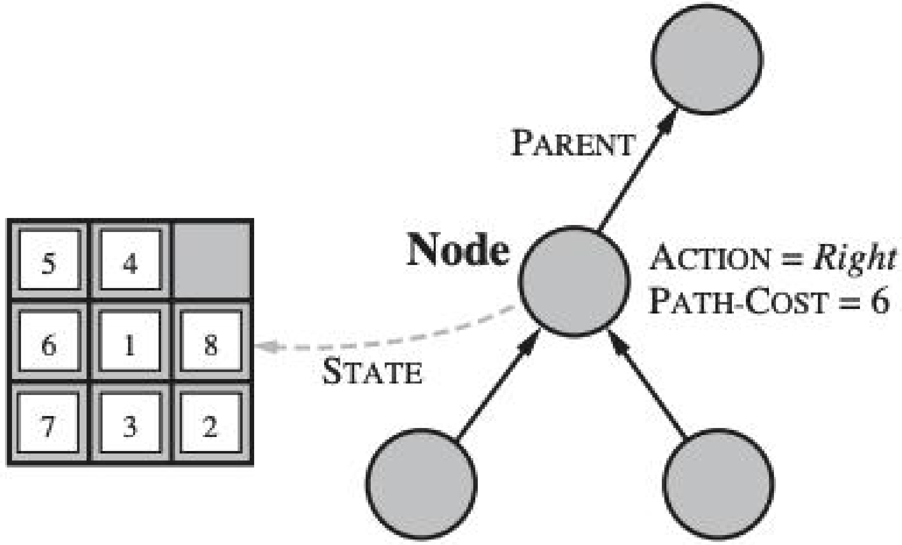
Node Data Structure

- Node data structure is used to keep track of the search tree



Node Data Structure

- Node data structure is used to keep track of the search tree



- Nodes vs. States
  - A node is a bookkeeping data structure used to represent the search tree
  - A state corresponds to a configuration of the world.

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?

Generic Tree-Search Algorithm

Add initial state to the frontier

Loop

node = remove-frontier( ) -- and save in order to return as part of path to goal

if goal-test(node) = true return path to goal

S = successors(node)

Add S to frontier

until frontier is empty

return failure

Generic Tree-Search Algorithm

Add initial state to the frontier

Loop

node = remove-frontier( ) -- and save in order to return as part of path to goal

if goal-test(node) = true return path to goal

S = successors(node)

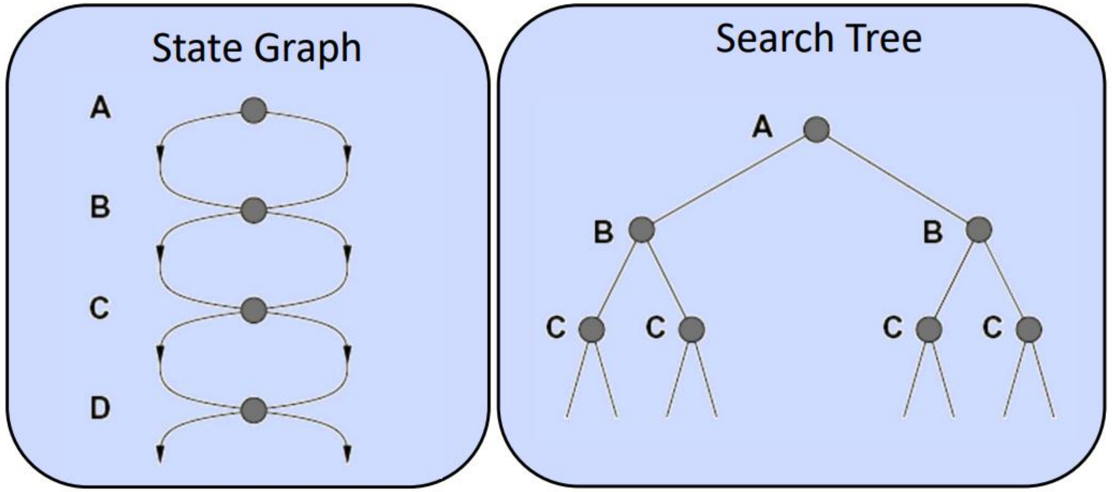
Add S to frontier

until frontier is empty

return failure

Tree Search vs. Graph Search

- Tree search alllows a state to be expanded more than once

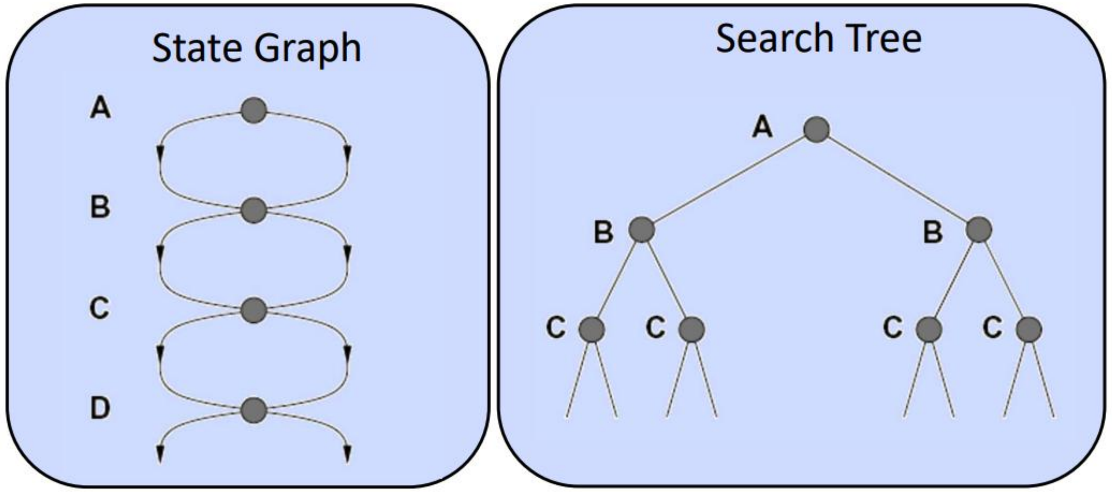


- Failure to detect repeated states can cause more work

- Advantage:

Tree Search vs. Graph Search

- Tree search alllows a state to be expanded more than once



- Failure to detect repeated states can cause more work

- Advantage: memory-efficent

Graph Search

- Idea: never expand a state twice

Graph Search

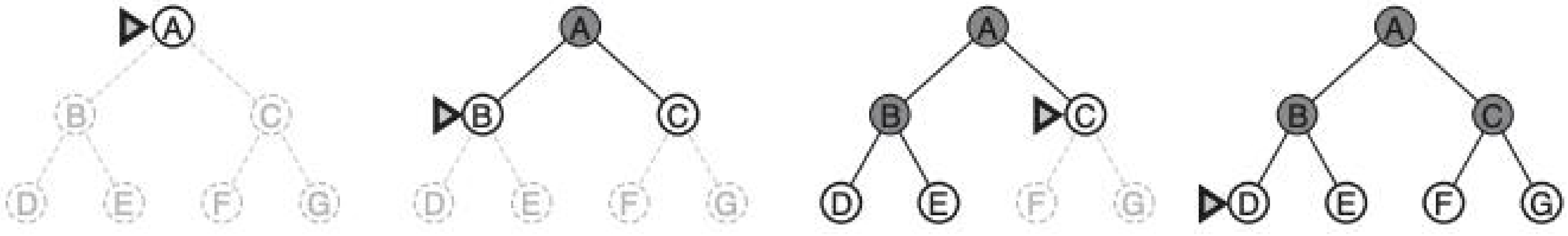
- Idea: never expand a state twice

- How to implement:

  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

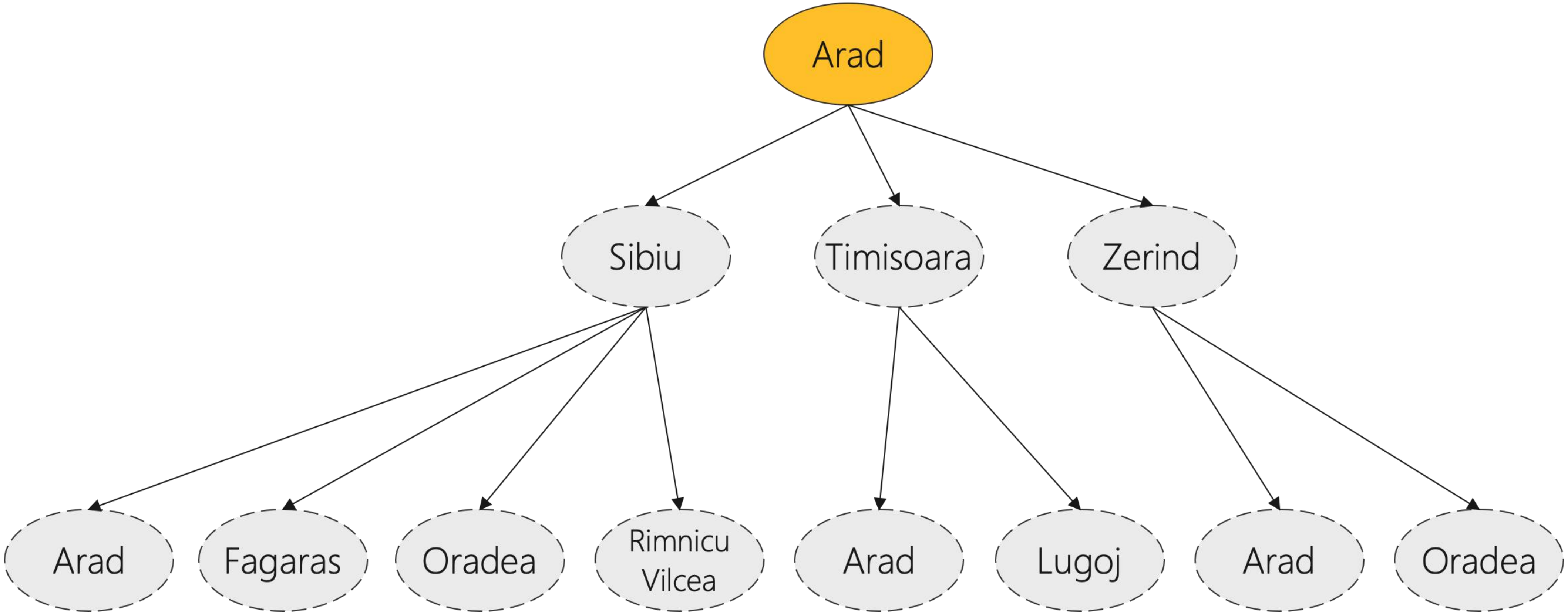
Uninformed Search: BFS

- Use the first-in, first out or FIFO queue to store the frontier

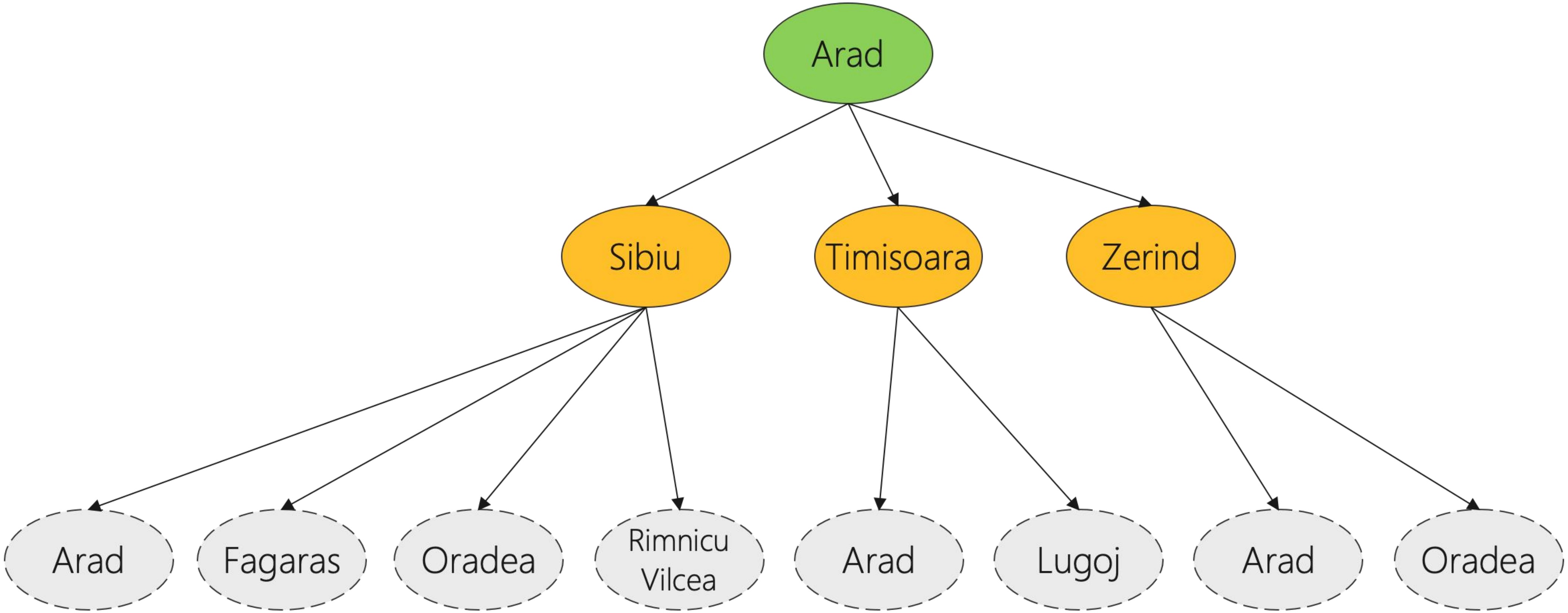


- Consider paths of length 1, then of length 2, then of length 3, then of length 4, ...

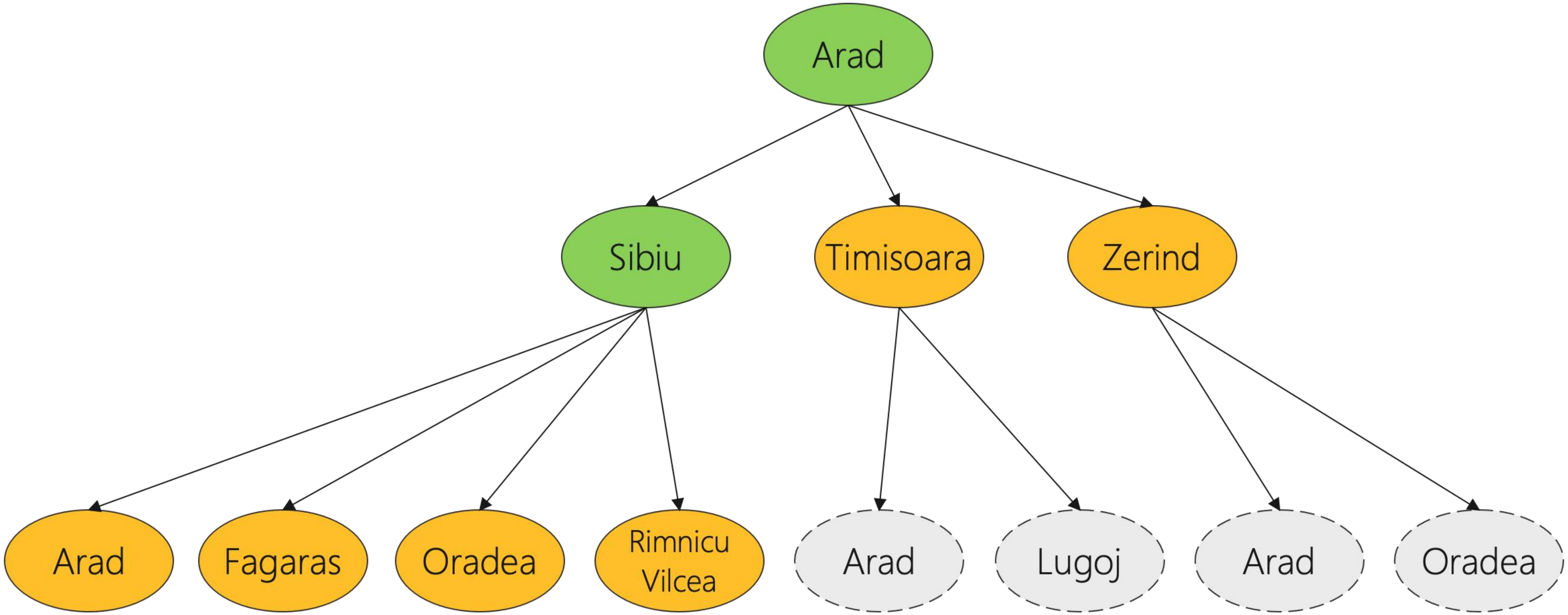
BFS: Example


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

BFS: Example

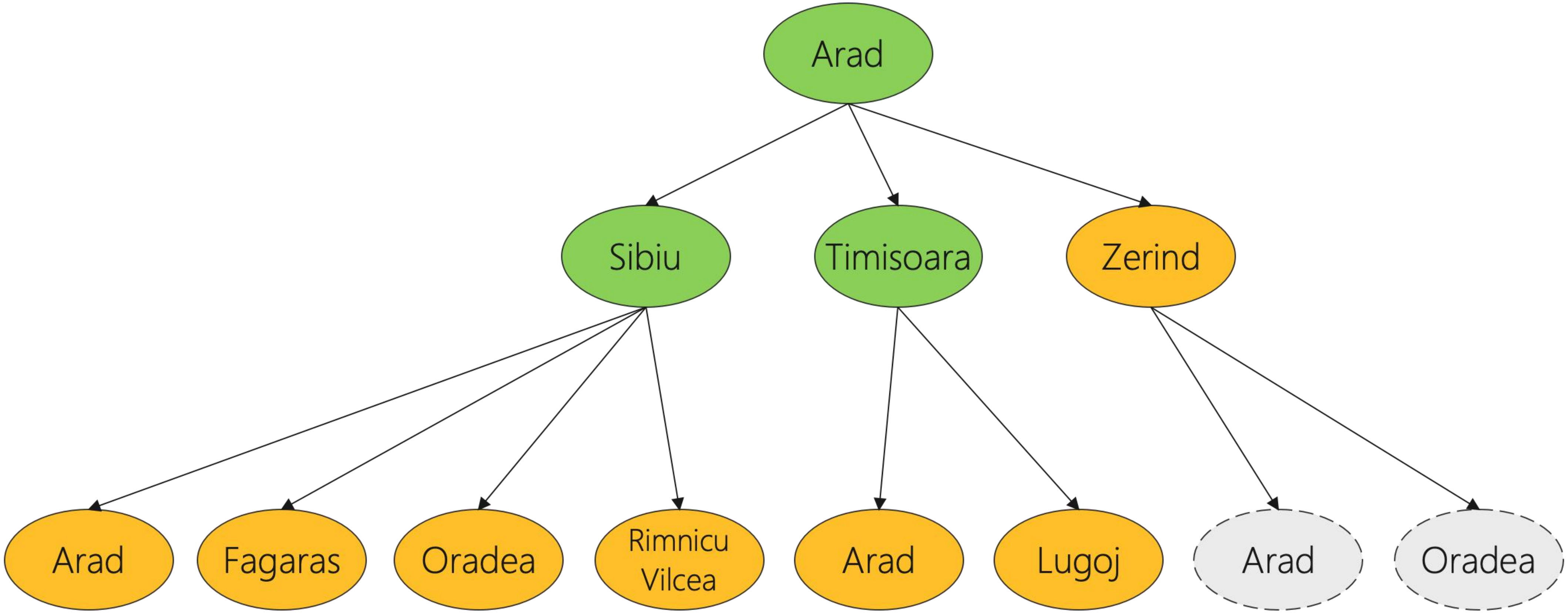

Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu
Vilcea
Arad
Lugoj
Arad
Oradea

BFS: Example

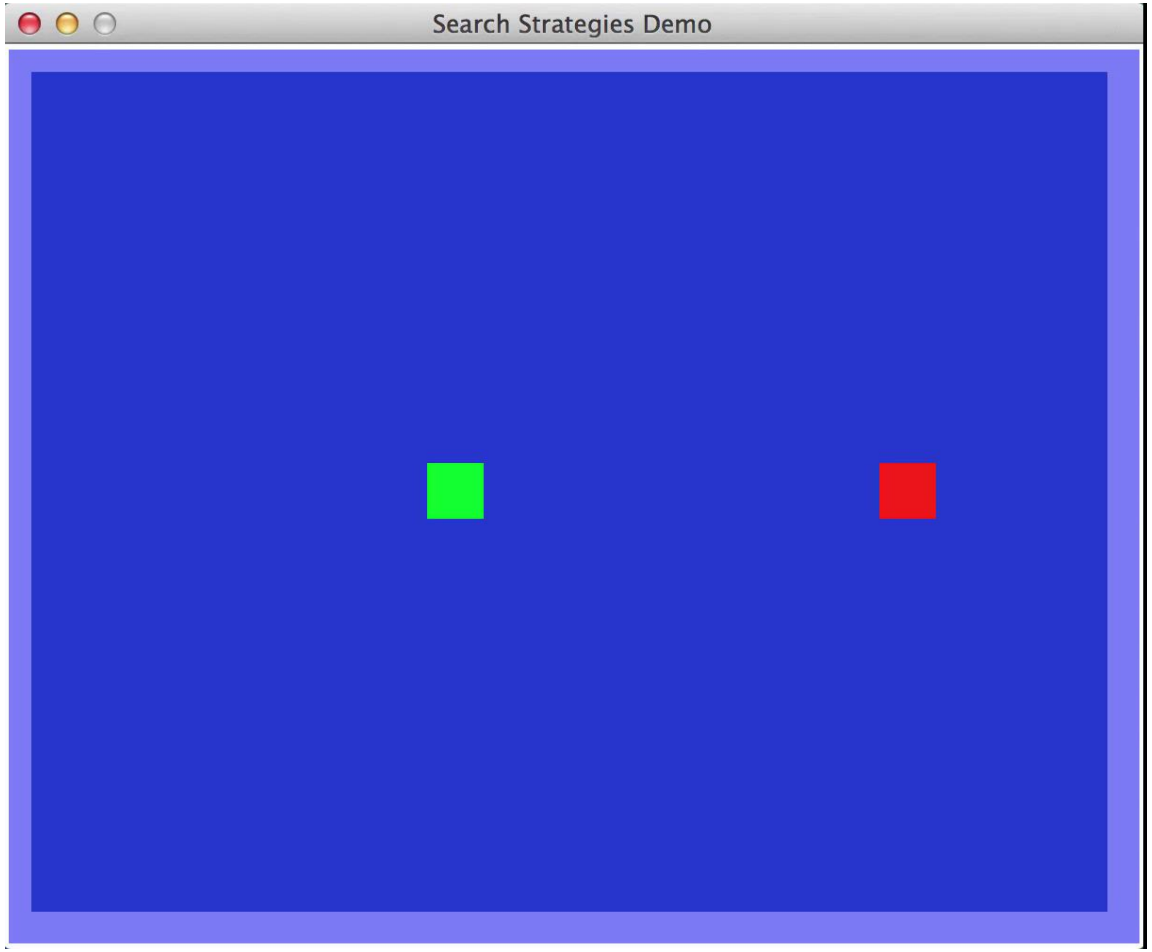


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

BFS: Example


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

BFS: Example



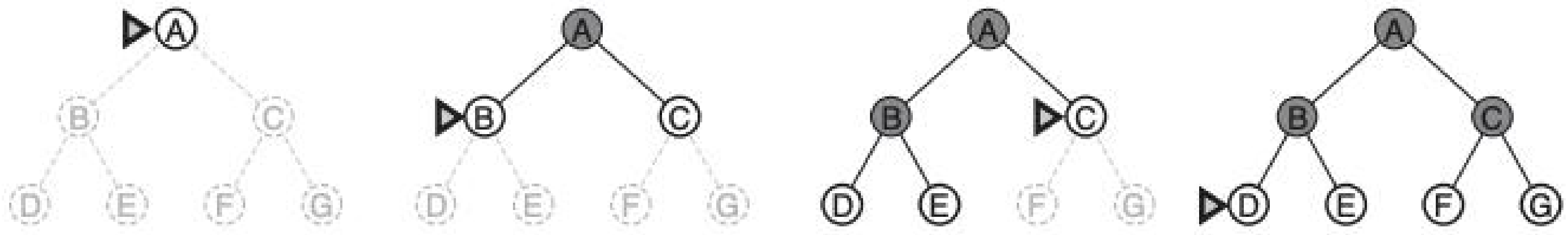
Time and Memory Requirements for BFS

- Let b = branching factor -> maximum number of successors of any node
- d = solution depth -> the shallowest goal node
- Then the maximum number of nodes generated is:

$$b + b^2 + \ldots + b^d = O(b^d)$$

- For graph search,

$O(b^{d-1})$ in the closed set and $O(b^d)$ in the frontier



Time and Memory Requirements for BFS

- branching factor = 10
- 1 million nodes / second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and Memory Requirements for BFS

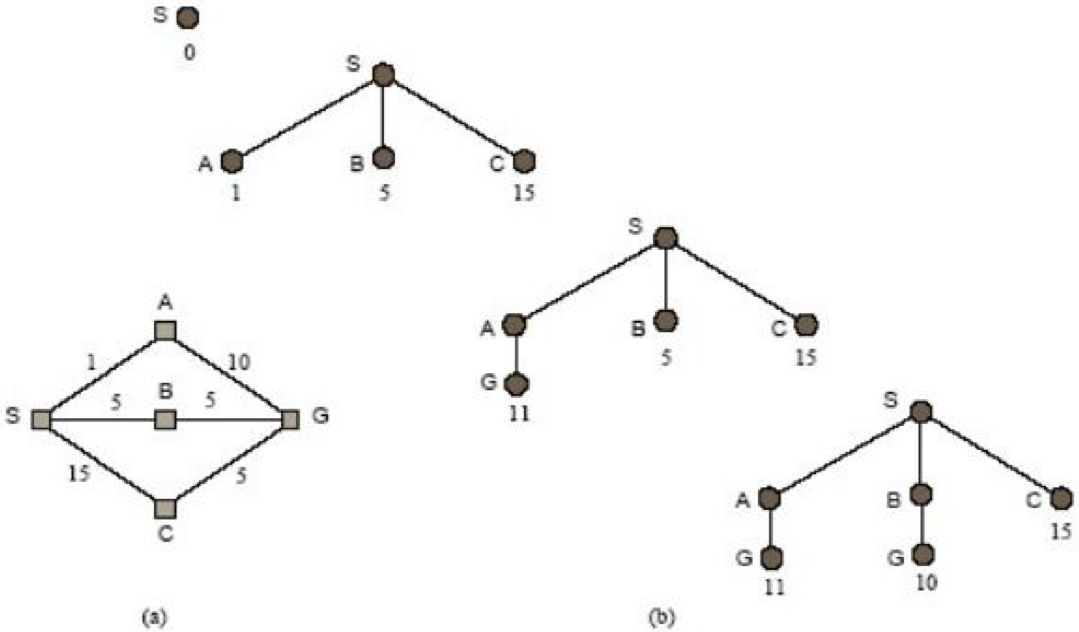
- branching factor = 10
- 1 million nodes / second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

125000 * 8GB

Uniform-Cost Search

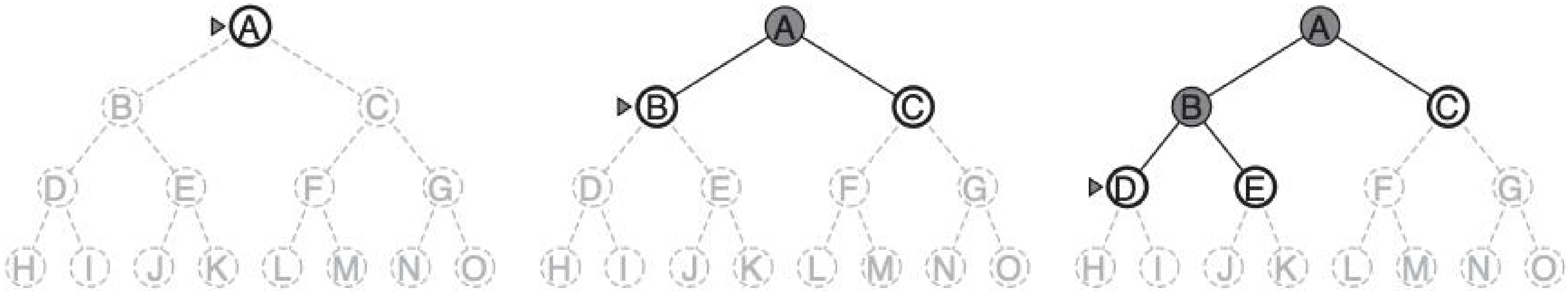
- Use BFS, but always expand the lowest-cost node on the frontier as measured by path cost g(n)



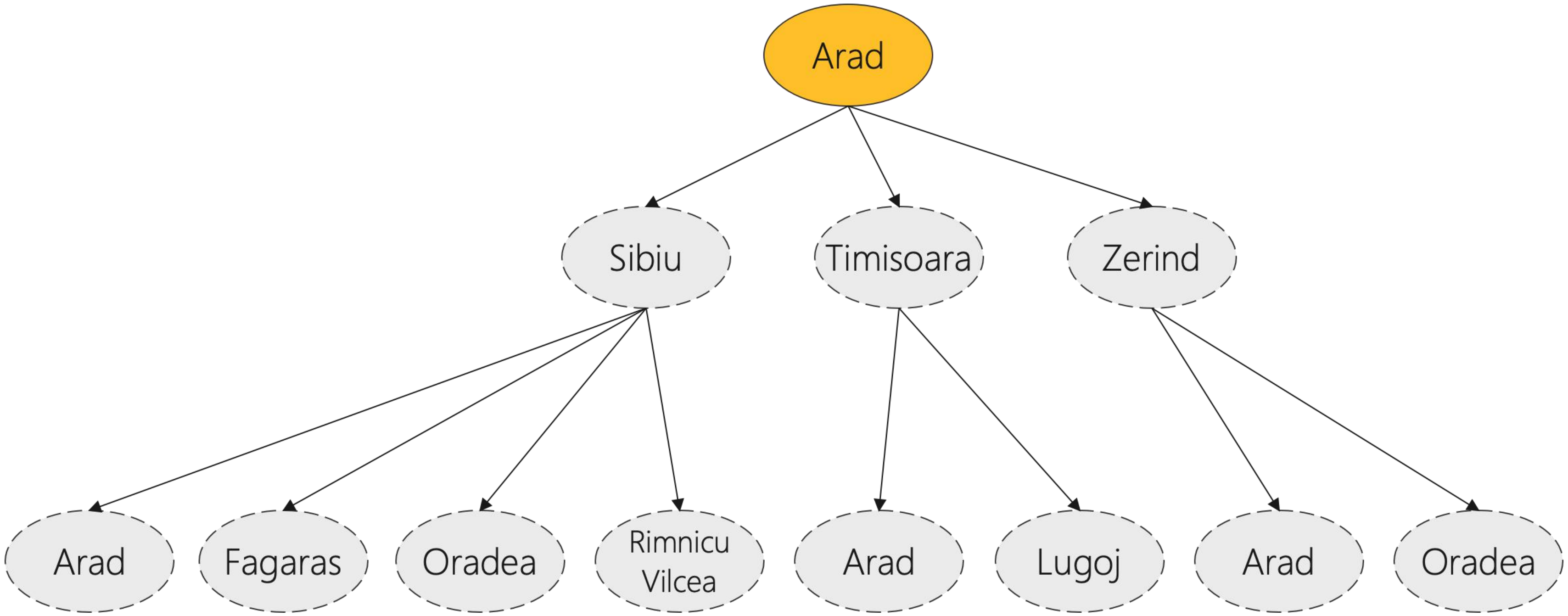
- g(Successor(n)) > g(n) is a necessary conditon for completeness and a sufficent condition for optimality

Uninformed search: DFS

- Use the last-in, first out or LIFO queue to store the frontier

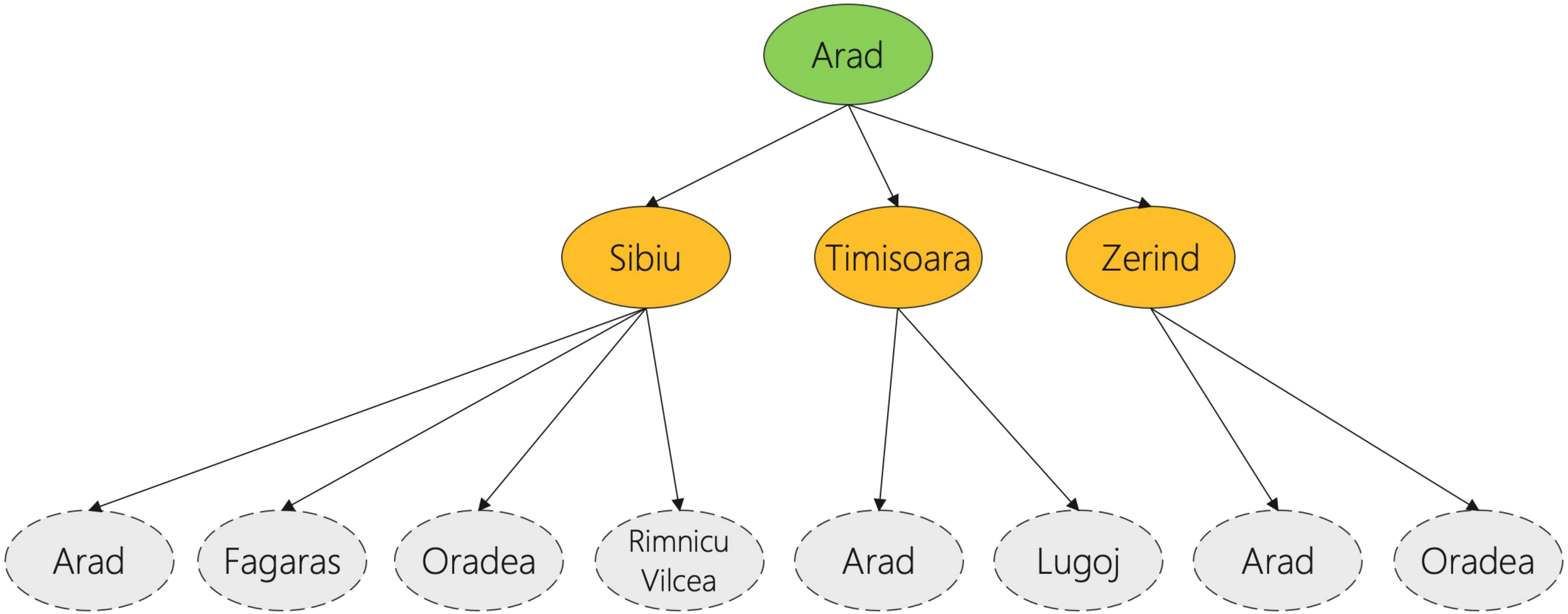


DFS: Example



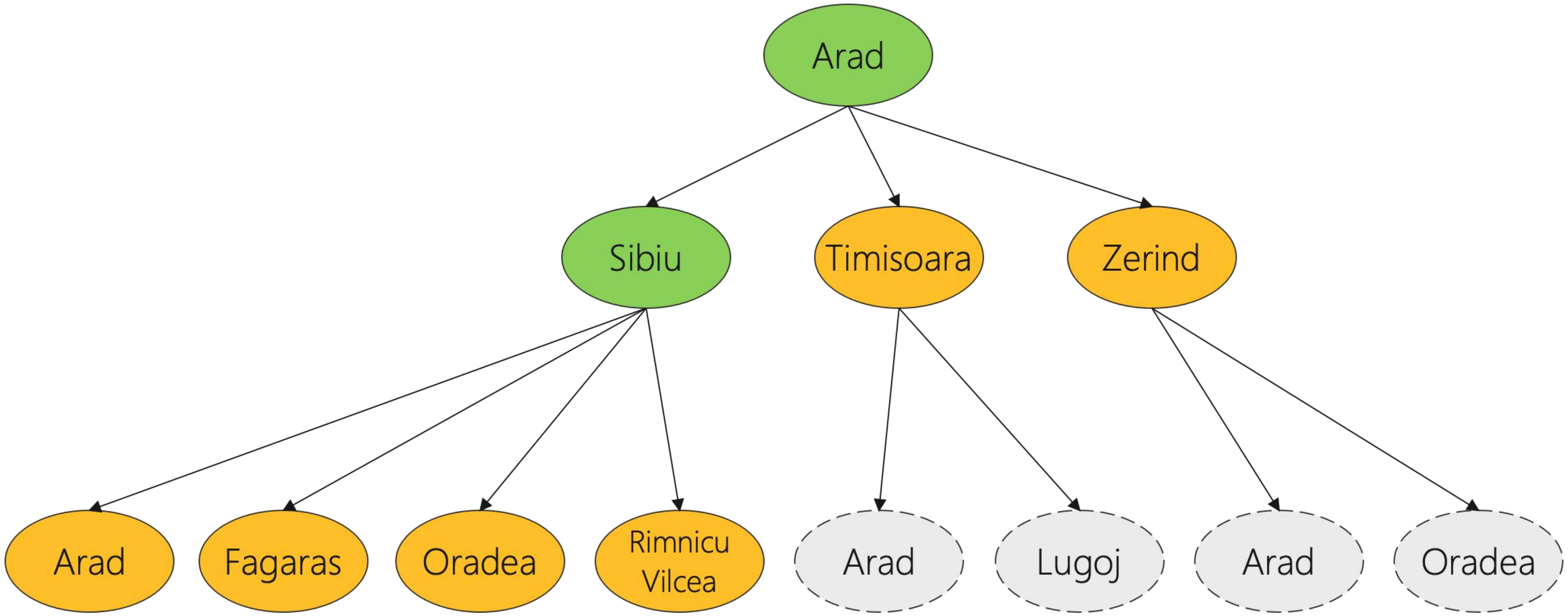
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

DFS: Example

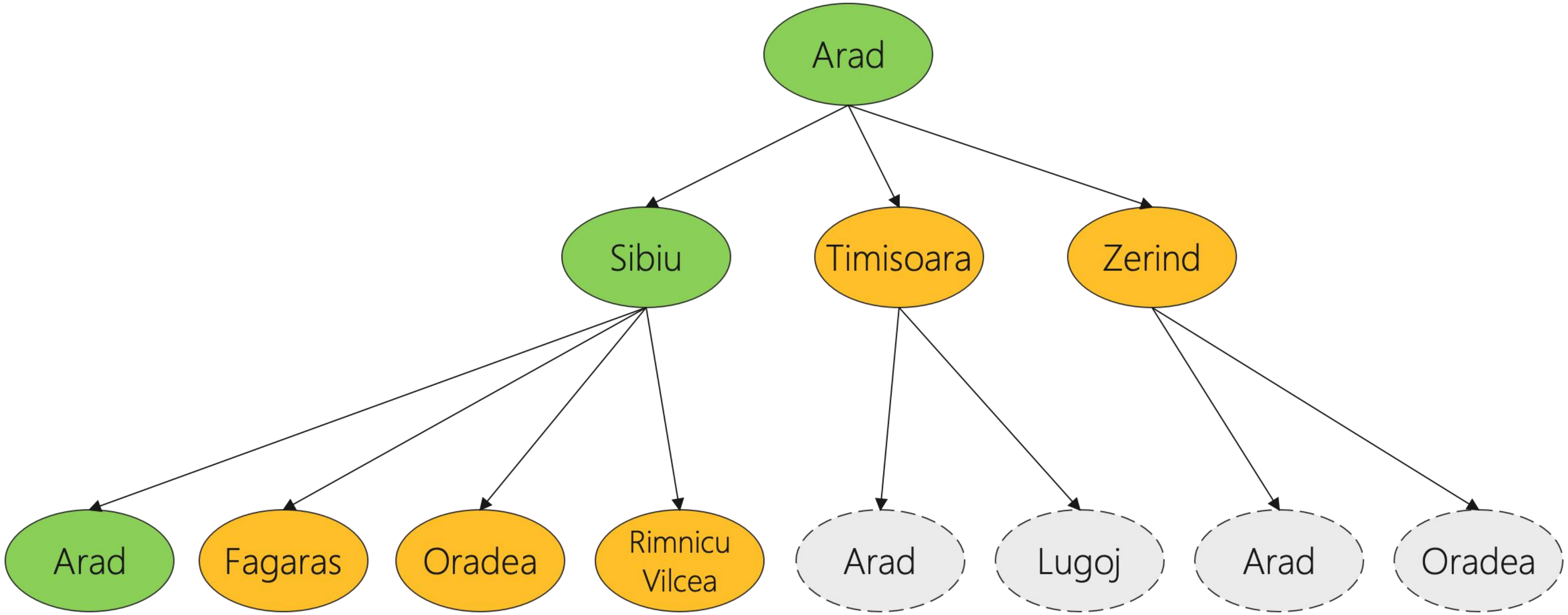


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

DFS: Example

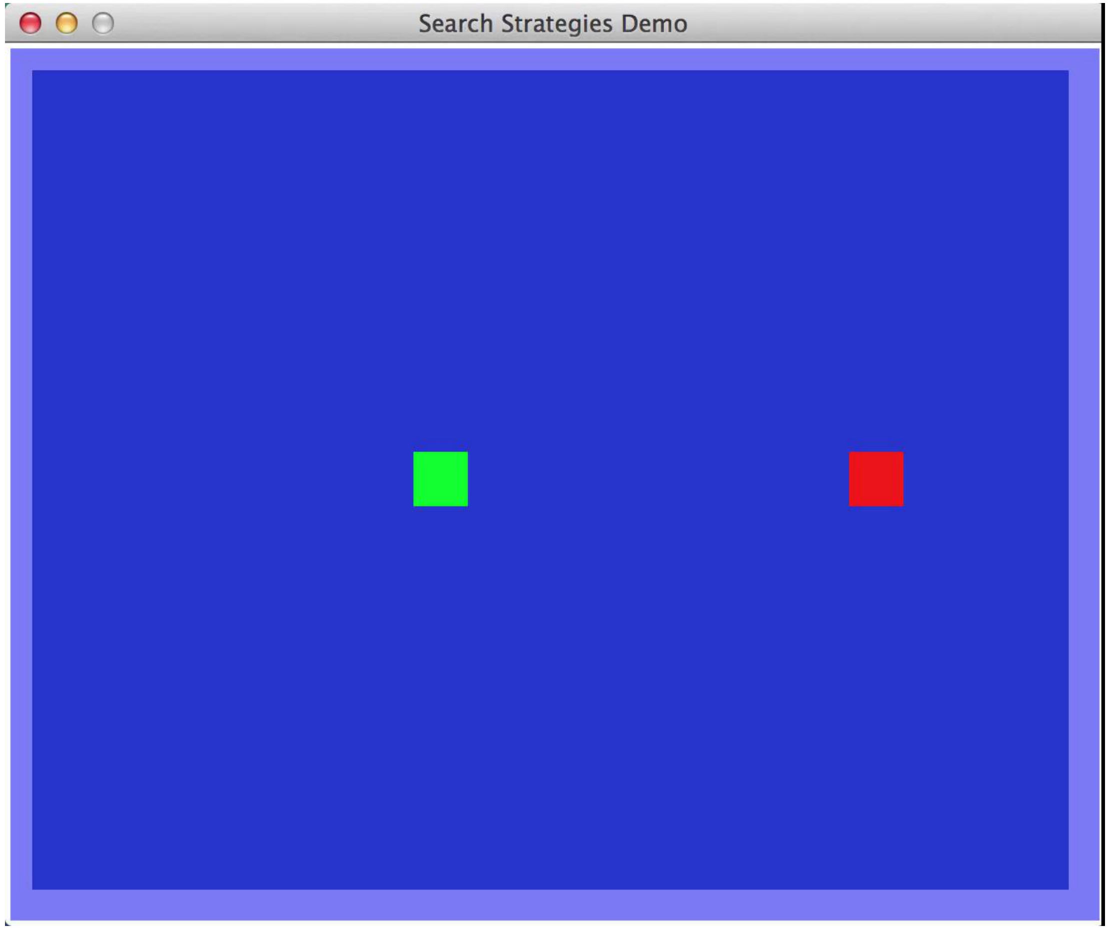

Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

DFS: Example



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

DFS: Example



Time and Memory Requirements for DFS

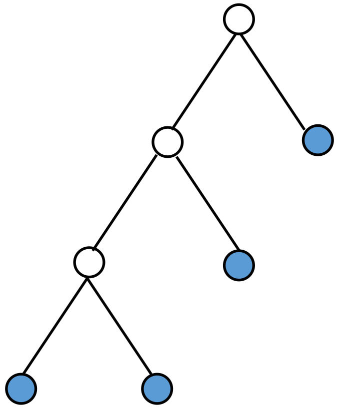
- Let b = branching factor -> maximum number of successors of any node
- m = maximum depth of any node

Time and Memory Requirements for DFS

- Let b = branching factor -> maximum number of successors of any node
- m = maximum depth of any node
- Time: for tree search, then the maximum number of nodes generated is: $O(b^m)$
- Space: for tree search, only need to store ? nodes

Time and Memory Requirements for DFS

- Let b = branching factor -> maximum number of successors of any node
- m = maximum depth of any node
- Time: for tree search, then the maximum number of nodes generated is: $O(b^m)$
- Space: for tree search, only need to store O(bm) nodes
  - at depth l < d we have b-1 nodes
  - at depth d we have b nodes
  - total = (m-1)*(b-1) + b = O(bm)



DFS vs. BFS

	Complete?	Optimal?	Time	Space
BFS	YES	YES	$O(b^d)$	$O(b^d)$
DFS	finte depth	NO	$O(b^m)$	$O(bm)$

d: depth of the shallowest solution   m: maximum depth

- Takeaways:
  - If the solution is not far from the root: BFS might be faster
  - If the search tree is very deep: DFS may never find solution
  - If the search tree is wide: BFS might take too much memory
  - If d is known: DFS is preferred
  - If optimal solution is needed: BFS is preferred

Iterative Deepening [Korf 1985]

- Problem of DFS:

Iterative Deepening [Korf 1985]

- Problem of DFS: cannot avoid infinite loops

Iterative Deepening [Korf 1985]

- Problem of DFS: cannot avoid infinite loops

- Idea:

  Use an artificial depth cutoff, c.

  If search to depth c succeeds, we are done. if not, increase c by 1 and start over.

  Each iteration searches using DFS.

Iterative Deepening [Korf 1985]

- Problem of DFS: cannot avoid infinite loops
- Idea:

  Use an artificial depth cutoff, c.

If search to depth c succeeds, we are done. if not, increase c by 1 and start over.

Each iteration searches using DFS. Why?

Iterative Deepening [Korf 1985]

- Problem of DFS: cannot avoid infinite loops

- Idea:

  Use an artificial depth cutoff, c.

If search to depth c succeeds, we are done. if not, increase c by 1 and start over.

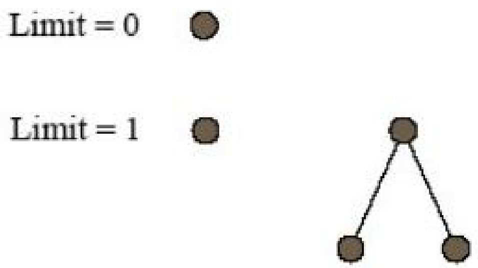
Each iteration searches using DFS.

Combine the benefit of the DFS and BFS

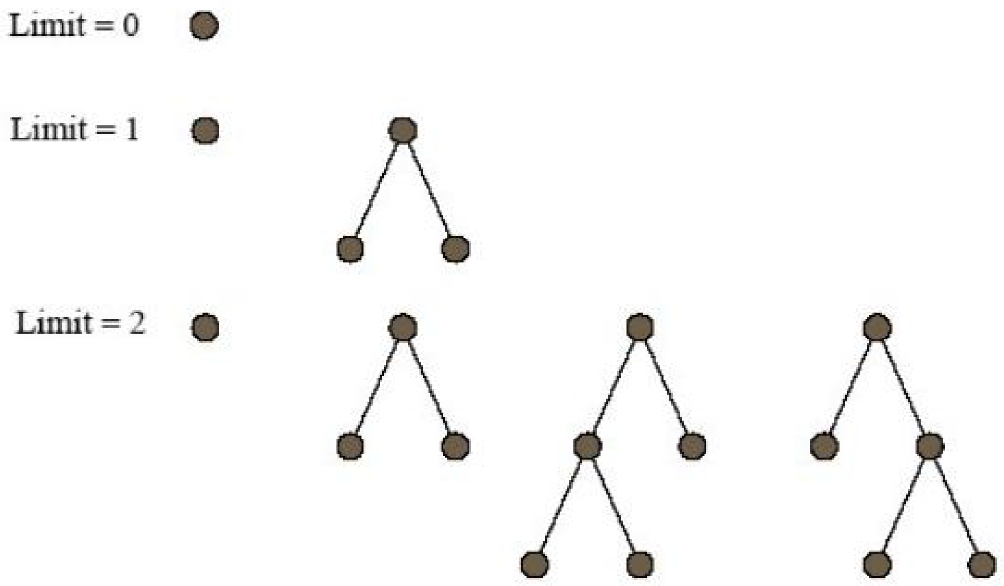
Iterative Deepening

Limit = 0 

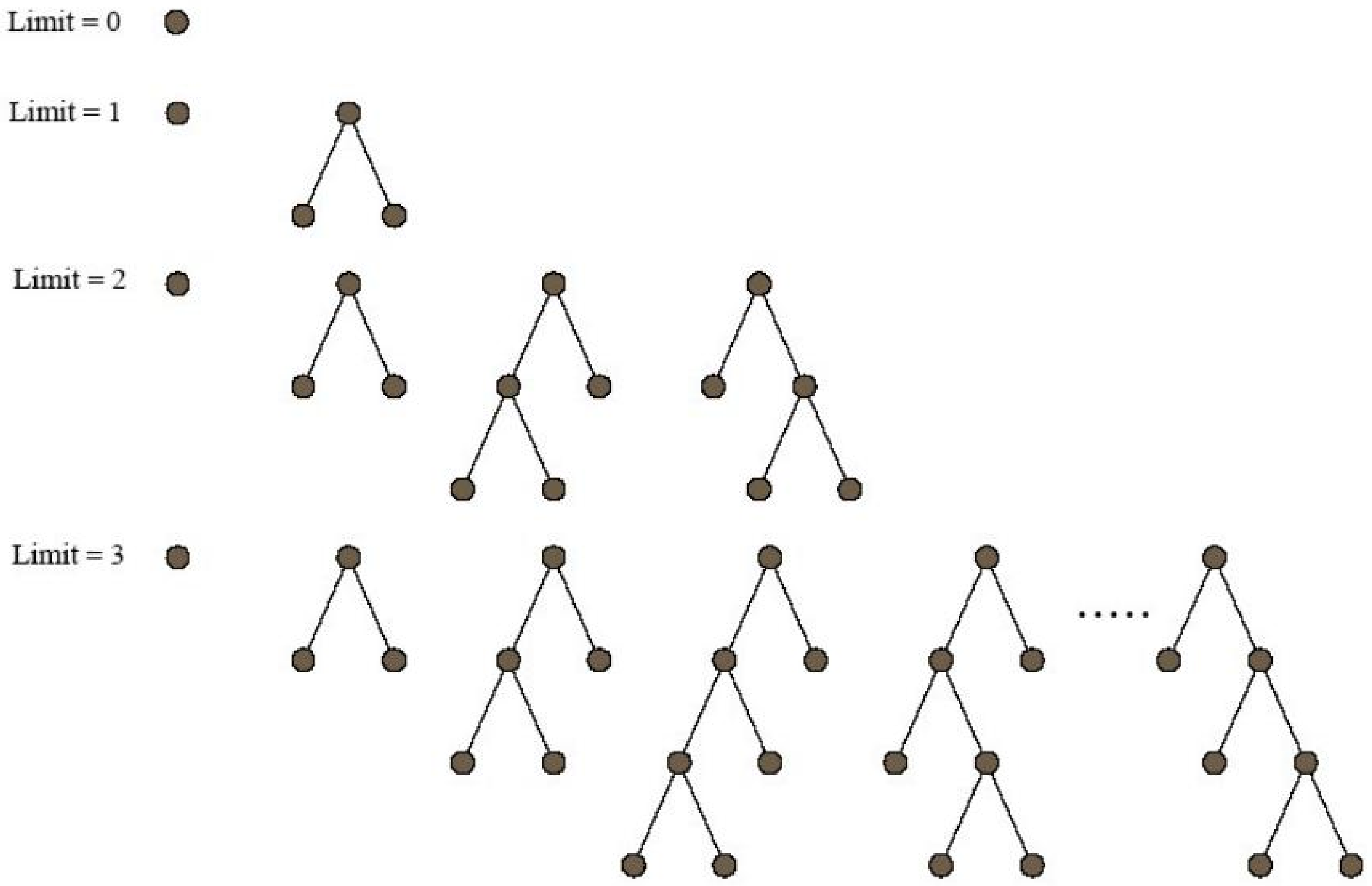
Iterative Deepening



Iterative Deepening



Iterative Deepening



Iterative Deepening

- Space requirements: same as DFS.

Iterative Deepening

- Space requirements: same as DFS.

- Time requirements: would seem very expensive! But not much different from single BFS or DFS to depth d

Iterative Deepening

- Space requirements: same as DFS.
- Time requirements: would seem very expensive! But not much different from single BFS or DFS to depth d
- Reason: Amost 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 the final layer!
- So, repeated runs are on much smaller trees (i.e., expoentially smaller).

Iterative Deepening

Examples:

$b = 10$, $d = 5$, the number of nodes generated in a BFS:

$b + b^2 + ... + b^d = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110$

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.

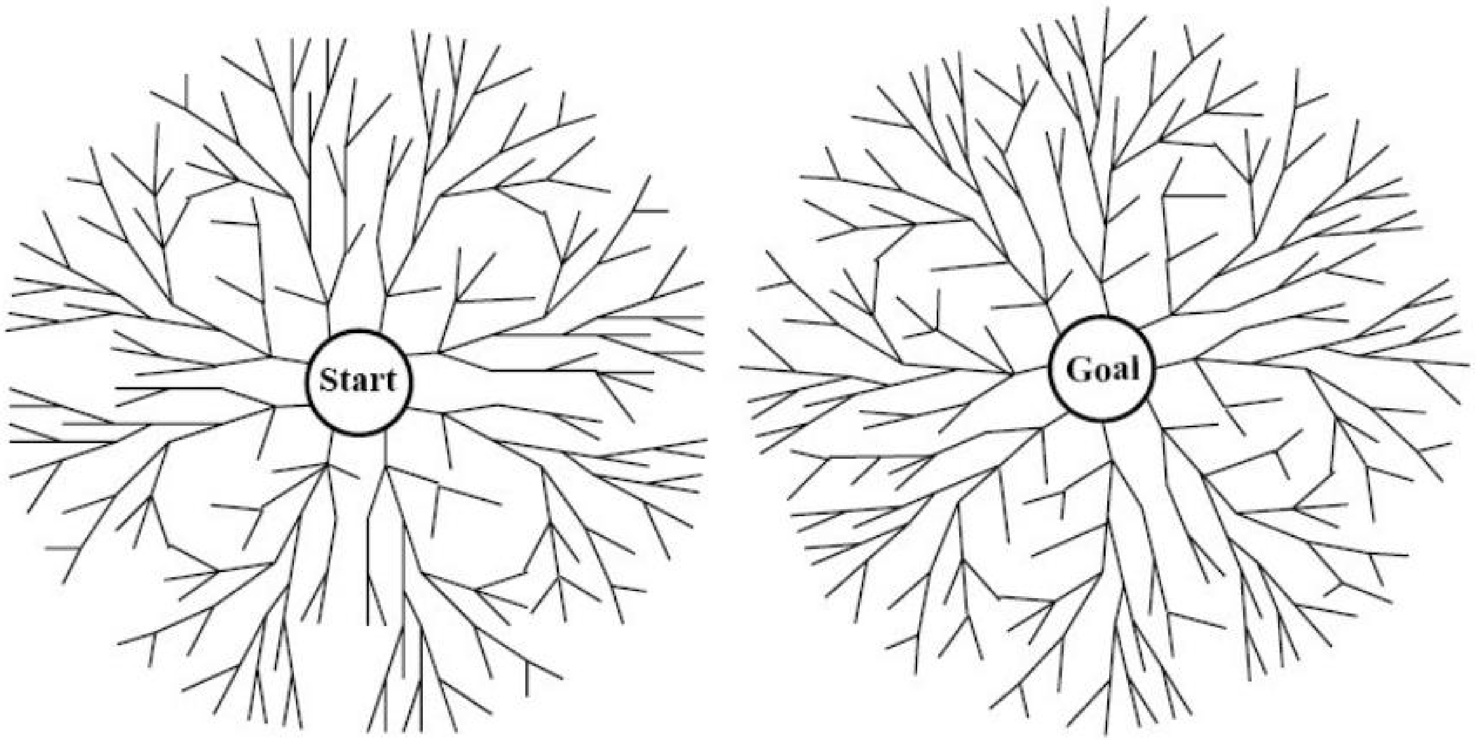
Cost of Iterative Deepening

- Space: O(bd) as in DFS, time: O(b^d)

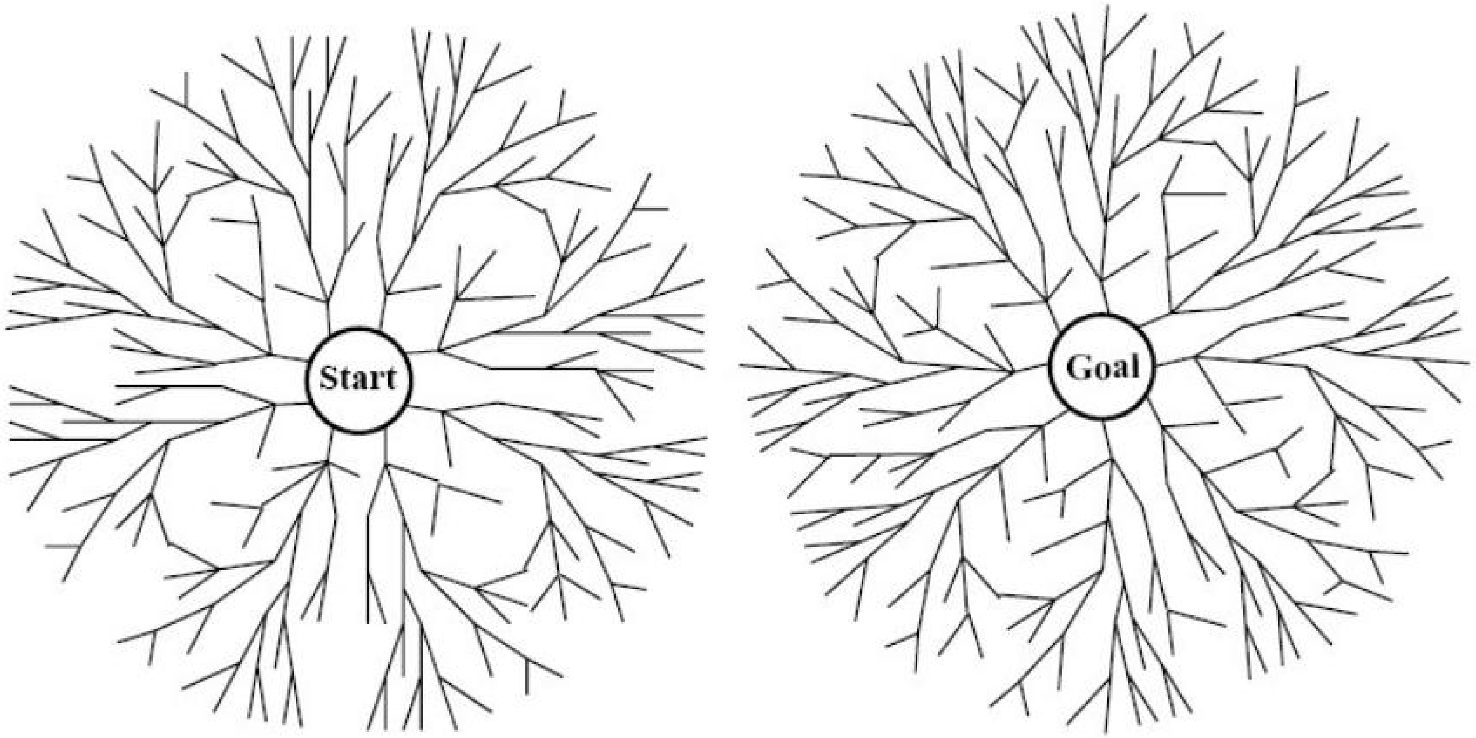
- Asymptotic ratio of the number of nodes generated by BFS and IDS:
  - (b+1)/(b-1)

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

Bidirectional Search

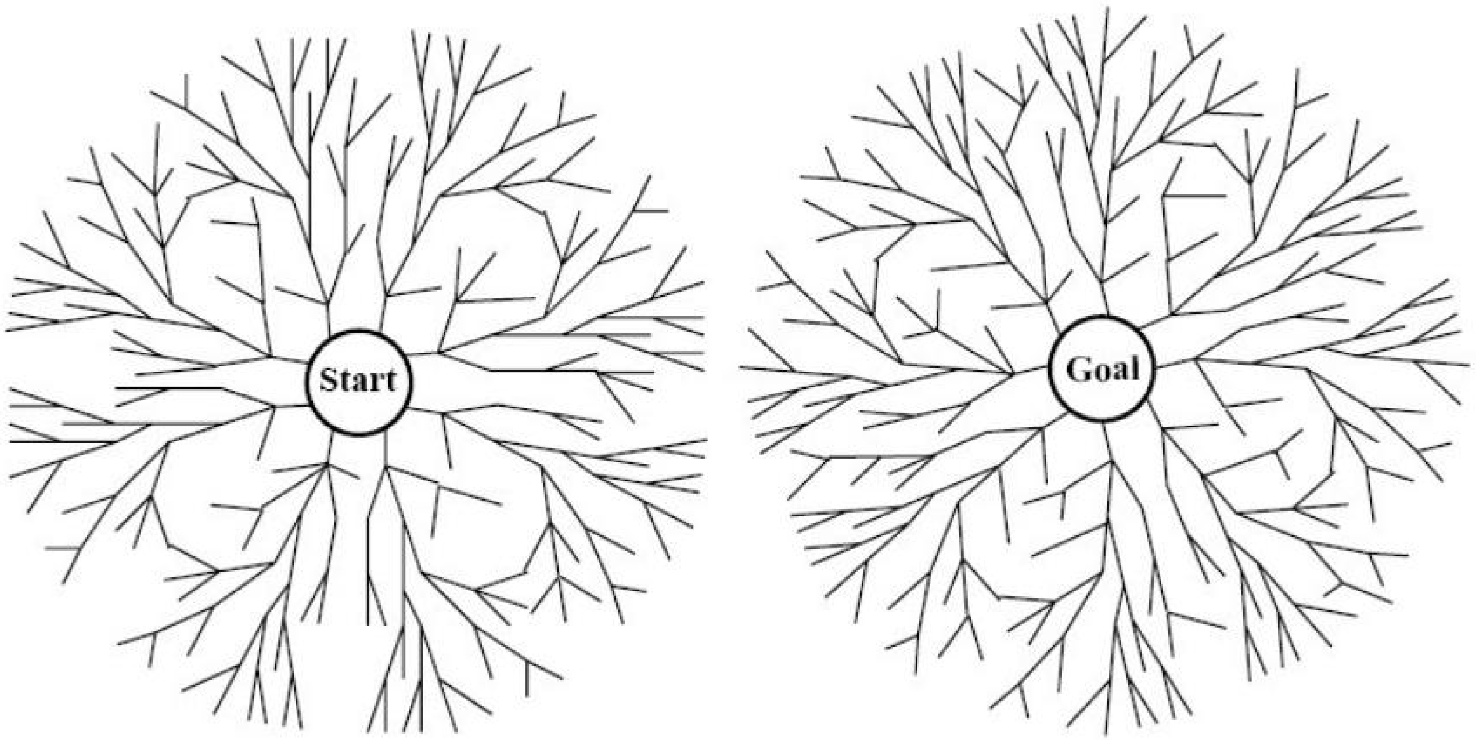


Bidirectional Search



- When is bidirectional search applicable?
  - Generate predecessors is easy

Bidirectional Search



- When is bidirectional search applicable?
  - Generate predecessors is easy
  - Goal state is clearly specified. ("no queen attacks another queen")

Bidirectional Search

- Search forward from the start state and backward from the goal state simultanesouly and stop when the two searches meet the middle

Bidirectional Search

- Search forward from the start state and backward from the goal state simultanesouly 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.

Bidirectional Search

- Search forward from the start state and backward from the goal state simultanesouly 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 1,111,110 nodes and bidirectional search needs only 2,220.

Limitations

- What are the problems of all the methods?

Limitations

- What are the problems of all the methods? Slow!

Limitations

- What are the problems of all the methods? Slow!

- The search is blind in the sense that the information of the goal state is not used

Limitations

- What are the problems of all the methods? Slow!
- The search is blind in the sense that the information of the goal state is not used
- Informed search:
  - with the guidance of the goal state

Goals of This Lecture

- Understand state-space search

- Understand BFS, DFS, UCS and IDS

- Know the advantages and disadvantages of each search strategy

Reading: Sections 3.1 - 3.4, R&N.