CS 1571 Introduction to AI
Lecture 3

Uninformed search methods

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Announcements

- **Homework 1 is out**
  - Access through the course web page
    http://www.cs.pitt.edu/~milos/courses/cs1571/
  - Two things to download:
    - Problem statement
    - C/C++ programs you will need for the assignment
- **Due date**: September 9, 2003 before the lecture
- **Submission:**
  - **Reports**: on the paper at the lecture
  - **Programs**: electronic submissions
    Submission guidelines:
    http://www.cs.pitt.edu/~milos/courses/cs1571/program-submissions.html

Problem-solving as search

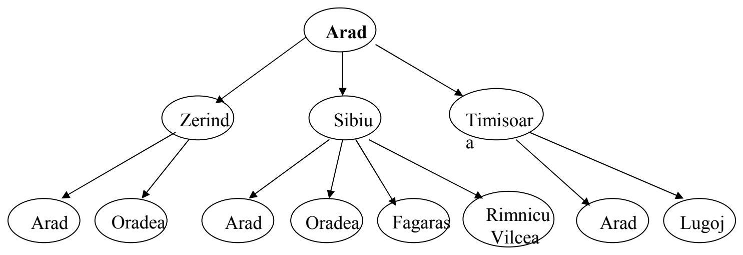
- Many search problems in practice can be converted to graph search problems
- **A graph search problem can be described in terms of:**
  - **A set of states** representing different world situations
  - **Initial state**
  - **Goal condition**
  - **Operators** defining valid moves between states
- **Two types of search:**
  - **Configuration search:** solution is **a state** satisfying the goal condition
  - **Path search:** solution **is a path** to a goal state
- **Optimal solution** = a solution with the optimal value
  - E.g. shortest path between the two cities, or
  - a desired n-queen configuration

Searching for the solution

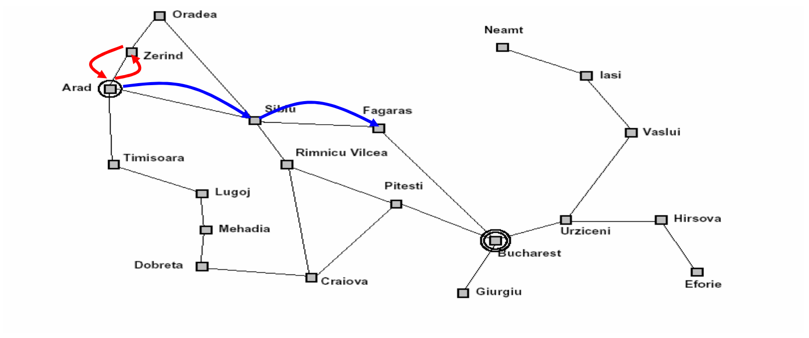
Search: exploration of the state space through successive application of operators from the initial state and goal testing

Search process can be though of as a process of building a **search tree**, with nodes corresponding to explored states

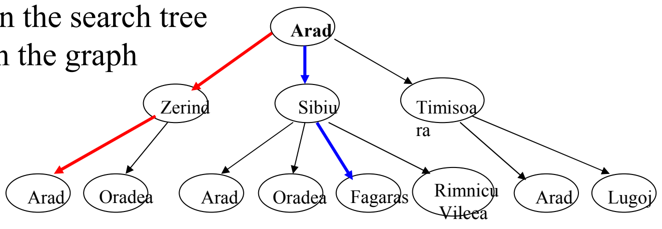
Search tree: a data structure that represents all paths from the initial state that has been explored during the search so far



Search tree

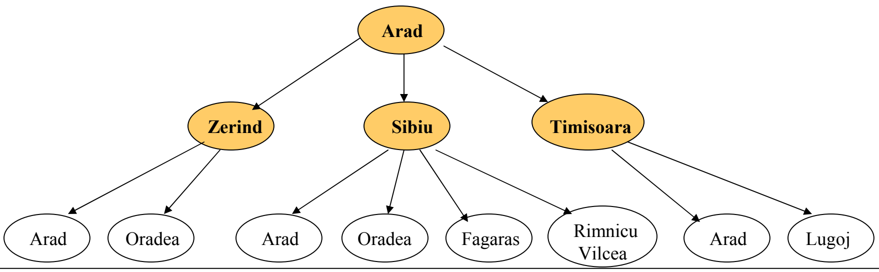


A branch in the search tree
= path in the graph



General search algorithm

```
General-search (problem, strategy)
 initialize the search tree with the initial state of problem
 loop
    if there are no candidate states to explore return failure
    choose a leaf node of the tree to expand next according to strategy
    if the node satisfies the goal condition return the solution
    expand the node and add all of its successors to the tree
 end loop
```



General search algorithm

```
General-search (problem, strategy)
 initialize the search tree with the initial state of problem
 loop
    if there are no candidate states to explore return failure
    choose a leaf node of the tree to expand next according to the strategy
    if the node satisfies the goal condition return the solution
    expand the node and add all of its successors to the tree
 end loop
```

- Search methods can differ in how they explore the space, that is how they **choose** the node to expand next

Implementation of search

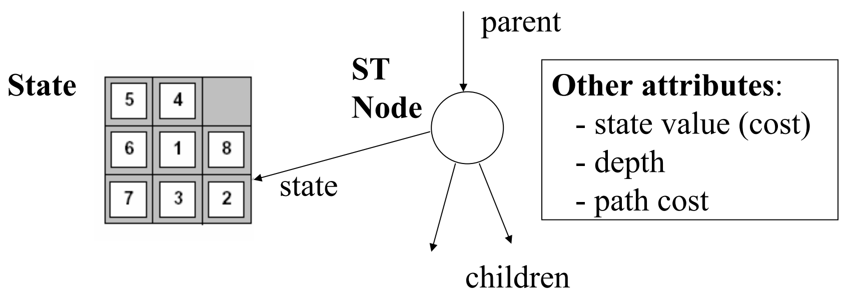
- Search methods can be implemented using the **queue** structure

```
General search (problem, Queuing-fn)
   nodes ← Make-queue(Make-node(Initial-state(problem)))
   loop
      if nodes is empty then return failure
      node ← Remove-node(nodes)
      if Goal-test(problem) applied to State(node) is satisfied then return node
      nodes ← Queuing-fn(nodes, Expand(node, Operators(node)))
   end loop
```

- Candidates are added to *nodes* representing the queue structure

Implementation of the search tree structure

- A **search tree node** is a data-structure constituting part of a search tree



- Expand node function – applies Operators to the state represented by the search tree node.

Uninformed search methods

- Many different ways to explore the state space (build a tree)

Uninformed search methods:

– use only information available in the problem definition

- **Breadth first search**
- **Depth first search**
- **Iterative deepening**
- **Bi-directional search**

For the minimum cost path problem:

- **Uniform cost search**

Search methods

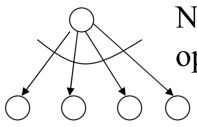
Properties of search methods :

- **Completeness.**
  - Does the method find the solution if it exists?
- **Optimality.**
  - Is the solution returned by the algorithm optimal? Does it give a minimum length path?
- **Space and time complexity.**
  - How much time it takes to find the solution?
  - How much memory is needed to do this?

Parameters to measure complexities.

- **Space and time complexity.**
  - **Complexities** are measured in terms of parameters:
    - *b* – maximum branching factor
    - *d* – depth of the optimal solution
    - *m* – maximum depth of the state space

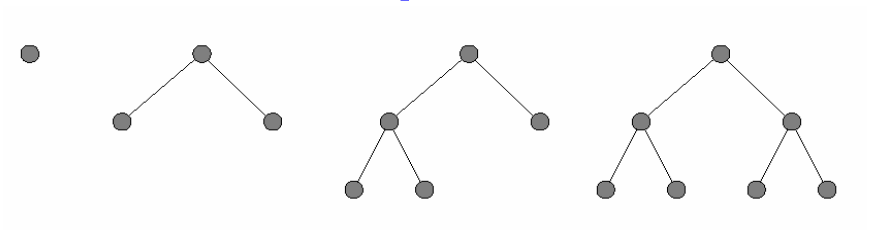
Branching factor

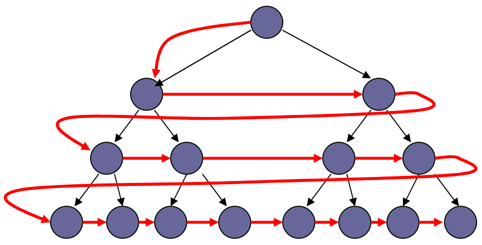


Number of operators

Breadth first search (BFS)

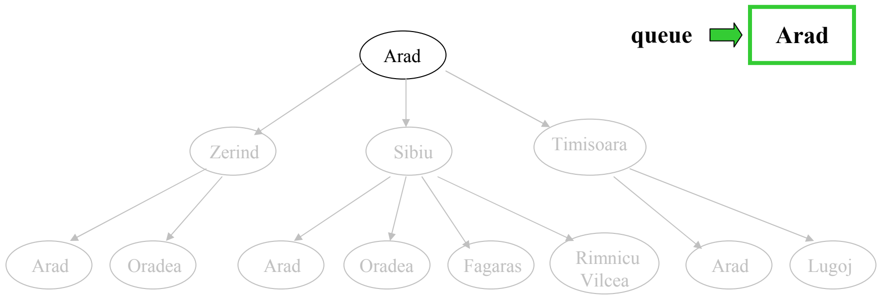
- The shallowest node is expanded first





Breadth-first search

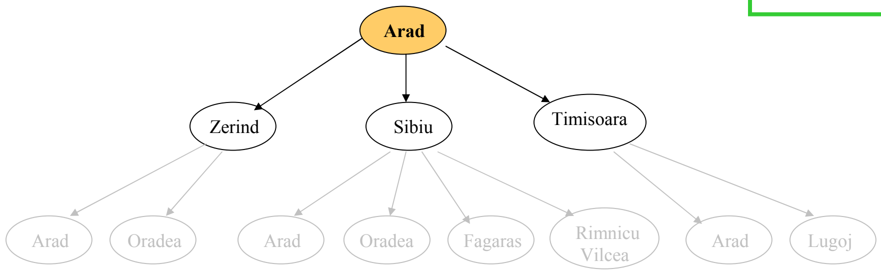
- Expand the shallowest node first
- Implementation: put successors to the end of the queue (FIFO)



Breadth-first search

queue

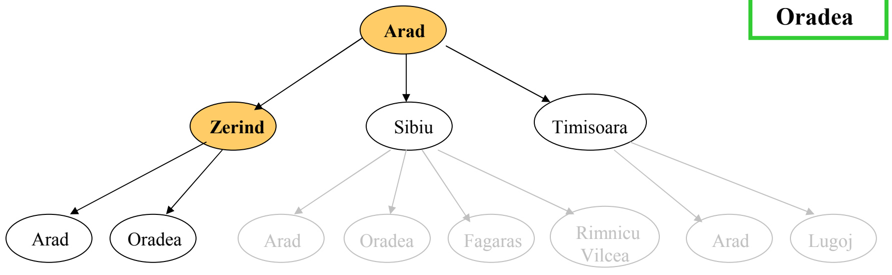
Zerind
Sibiu
Timisoara



Breadth-first search

queue

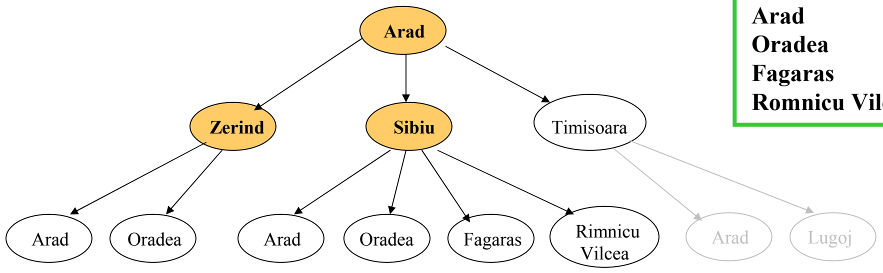
Sibiu
Timisoara
Arad
Oradea



Breadth-first search

queue

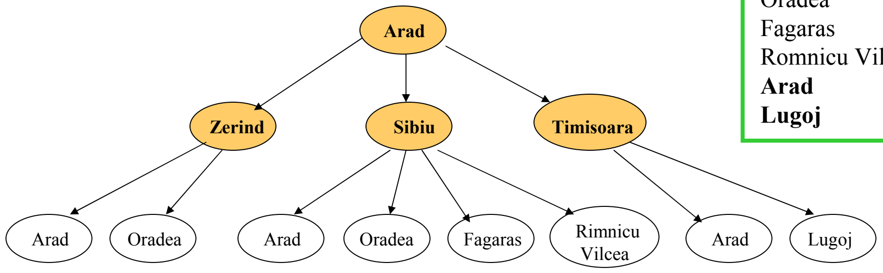
Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Romnicu Vilcea



Breadth-first search

queue

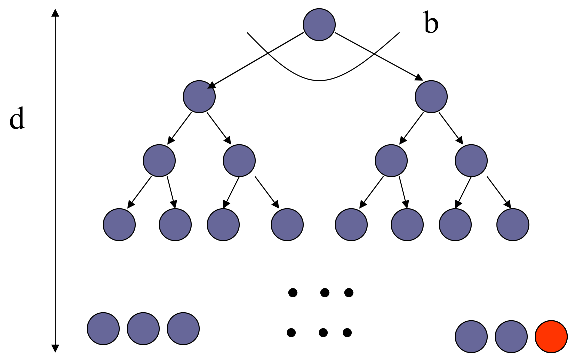
Arad
Oradea
Arad
Oradea
Fagaras
Romnicu Vilcea
Arad
Lugoj



Properties of breadth-first search

- **Completeness:** **Yes.** The solution is reached if it exists.
- **Optimality:** **Yes**, for the shortest path.
- **Time complexity: ?**
- **Memory (space) complexity: ?**

BFS – time complexity

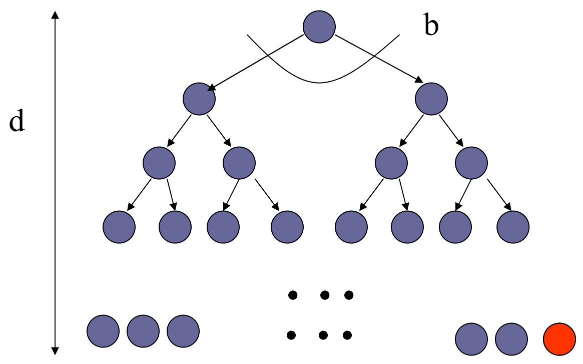


depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	$2^d= b^d$

Total nodes: $O(b^d)$

BFS –memory complexity

Count nodes kept in the tree structure and/or in the queue



depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	$2^d=b^d$

Total nodes: $O(b^d)$

Properties of breadth-first search

- **Completeness:** **Yes.** The solution is reached if it exists.
- **Optimality:** **Yes**, for the shortest path.
- **Time complexity:**

$$1 + b + b^2 + \ldots + b^d = O(b^d)$$

exponential in the depth of the solution *d*

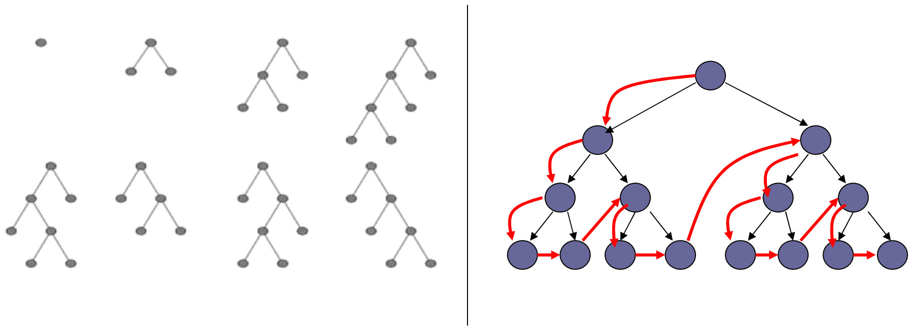
- **Memory (space) complexity:**

$$O(b^d)$$

every node of the tree is kept in the memory

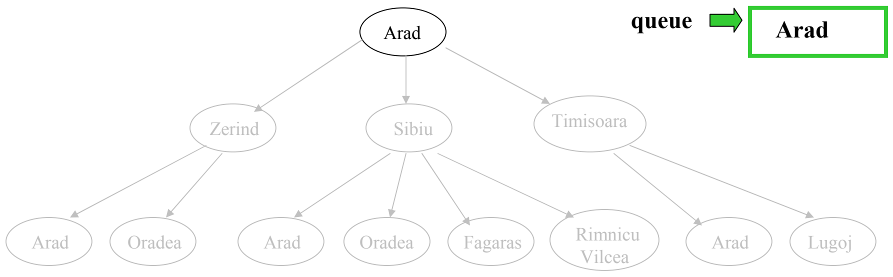
Depth-first search (DFS)

- **The deepest node is expanded first**
- Backtrack when the path cannot be further expanded

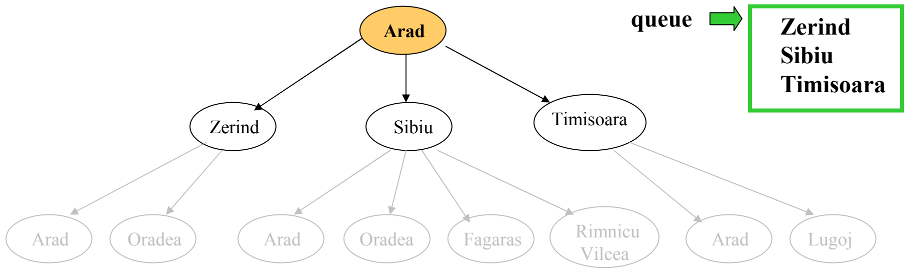


Depth-first search

- **The deepest node is expanded first**
- Implementation: put successors to the beginning of the queue



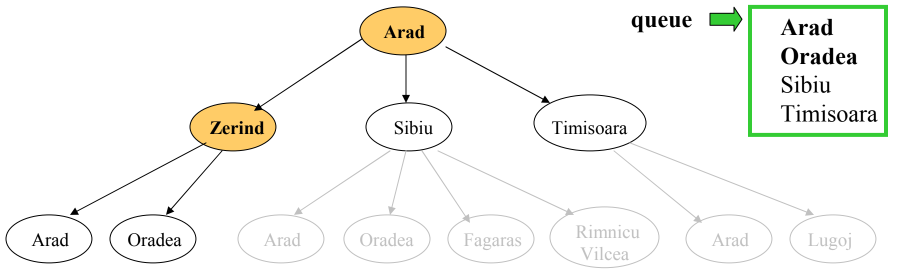
Depth-first search



queue

Zerind
Sibiu
Timisoara

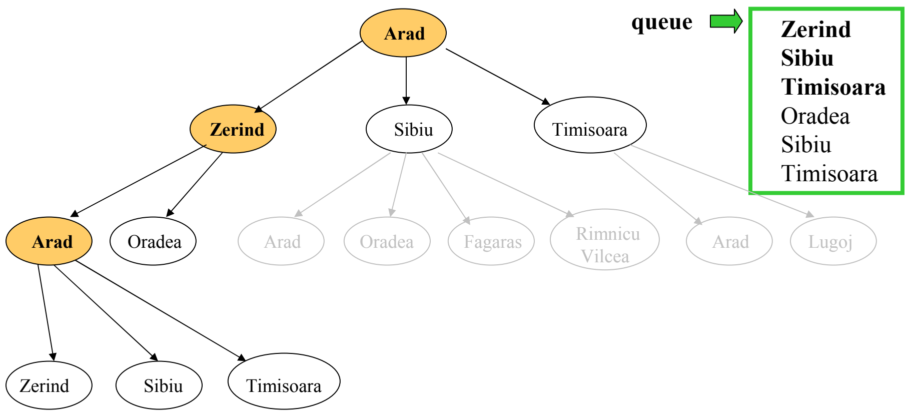
Depth-first search



queue

Arad
Oradea
Sibiu
Timisoara

Depth-first search



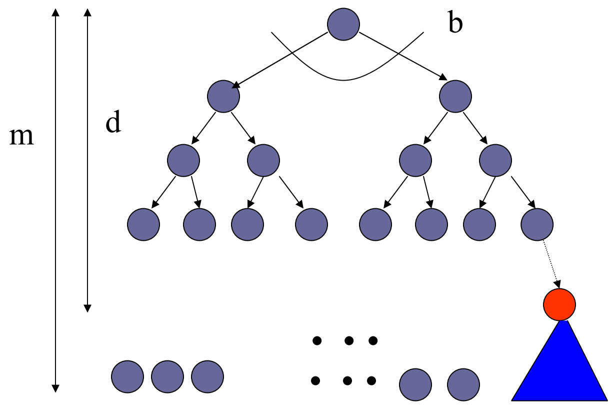
Zerind
Sibiu
Timisoara
Oradea
Sibiu
Timisoara

Note: Arad – Zerind – Arad cycle

Properties of depth-first search

- **Completeness:** No. Infinite loops can occur.
- **Optimality:** No. Solution found first may not be the shortest possible.
- **Time complexity: ?**
- **Memory (space) complexity: ?**

DFS – time complexity

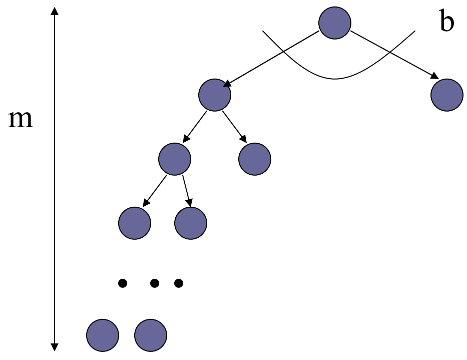


depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
•	• •
d	2^d
•	• •
m	2^m-2^{m-d}

Total nodes: $O(b^m)$

DFS – memory complexity

Count nodes kept in the tree structure and/or in the queue



depth	number of nodes
0	1
1	2 = b
2	2
3	2
• • •	
m	2

Total nodes: $O(bm)$

Properties of depth-first search

- **Completeness:** **No.** Infinite loops can occur.
- **Optimality:** **No.** Solution found first may not be the shortest possible.
- **Time complexity:**

  $O(b^m)$

  exponential in the maximum depth of the search tree *m*
- **Memory (space) complexity:**

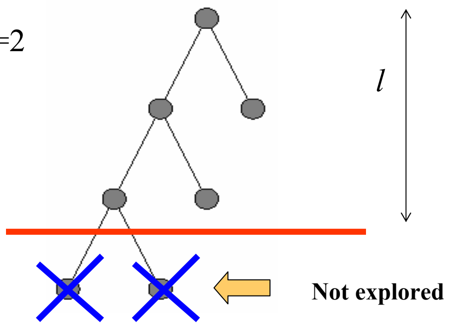
  $O(bm)$

the tree size we need to keep is linear in the maximum depth of the search tree *m*

Limited-depth depth first search

- The limit *(l)* on the depth of the depth-first exploration

Limit l=2



- **Time complexity:** $O(b^l)$
- **Memory complexity**: $O(bl)$

l - is the given limit

Limited depth depth-first search

- Avoids pitfalls of depth first search
- Cutoff on the maximum depth of the tree
- If we know that the solution length is within a limit the solution the algorithm gives is complete
- How to design the limit?
  - 20 cities in the travel problem
  - We need to consider only paths of length < 20
- Without the known limit the search may fail to find the solution
- **Time complexity:** $O(b^l)$ l - is the limit
- **Memory complexity**: $O(bl)$

Iterative deepening algorithm (IDA)

- Based on the idea of the limited-depth search, but
- It resolves the difficulty of knowing the depth limit ahead of time.

Idea: try all depth limits in an increasing order.

That is, search first with the depth limit $l=0$, then $l=1$, $l=2$, and so on until the solution is reached

Iterative deepening combines advantages of the depth-first and breadth-first search with only moderate computational overhead

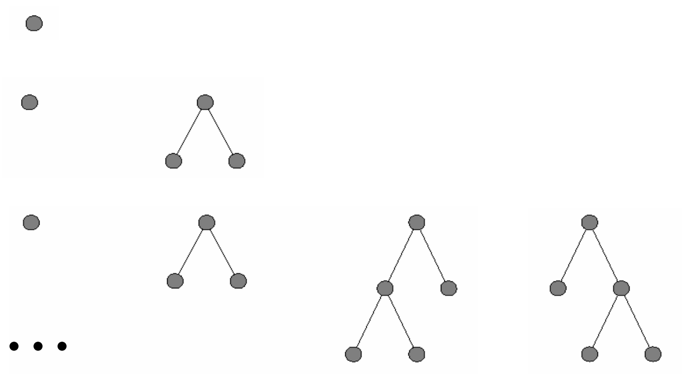
Iterative deepening algorithm (IDA)

- Progressively increases the limit of the limited-depth depth-first search

Limit 0

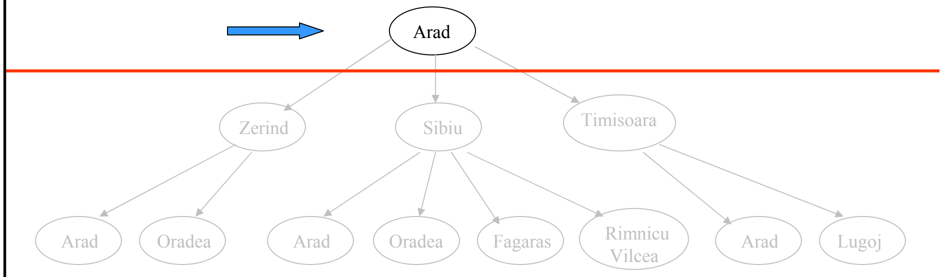
Limit 1

Limit 2



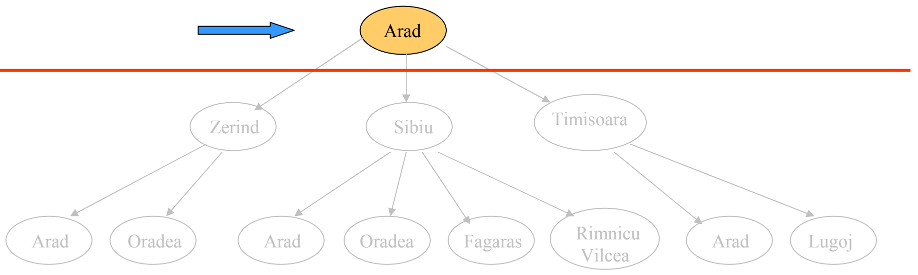
Iterative deepening

Cutoff depth = 0



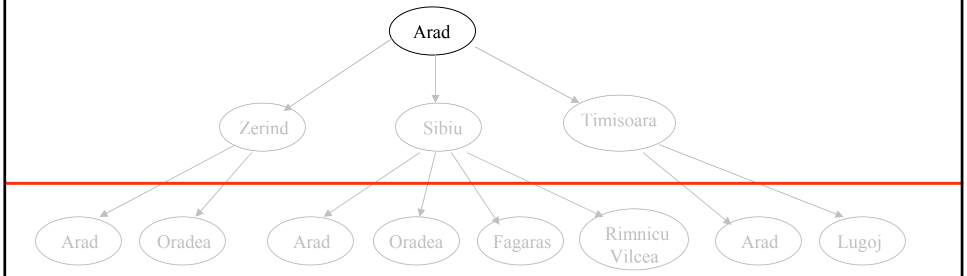
Iterative deepening

Cutoff depth = 0



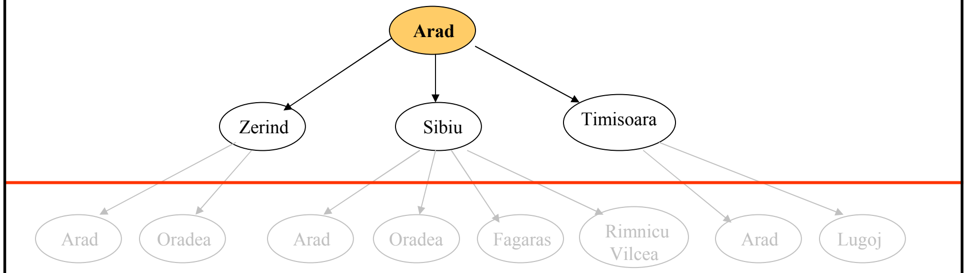
Iterative deepening

Cutoff depth = 1



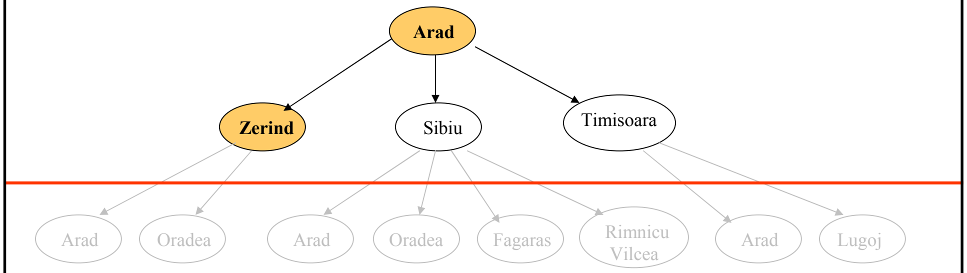
Iterative deepening

Cutoff depth = 1



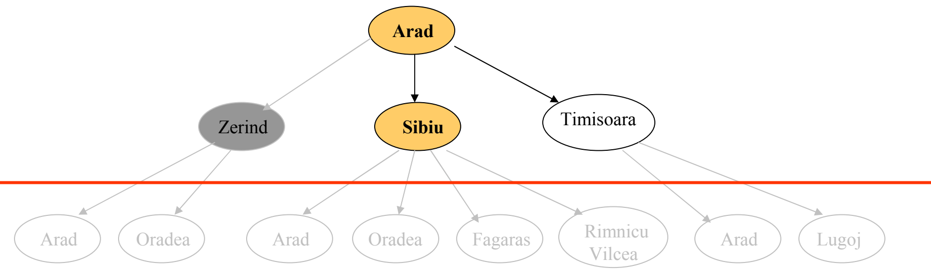
Iterative deepening

Cutoff depth = 1



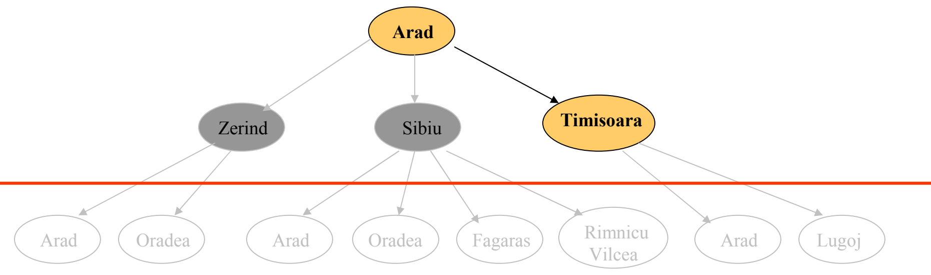
Iterative deepening

Cutoff depth = 1



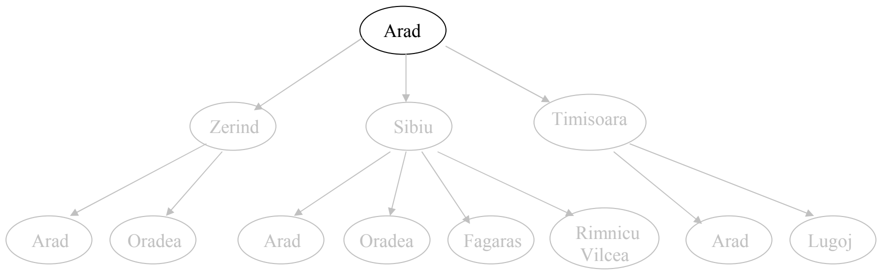
Iterative deepening

Cutoff depth = 1



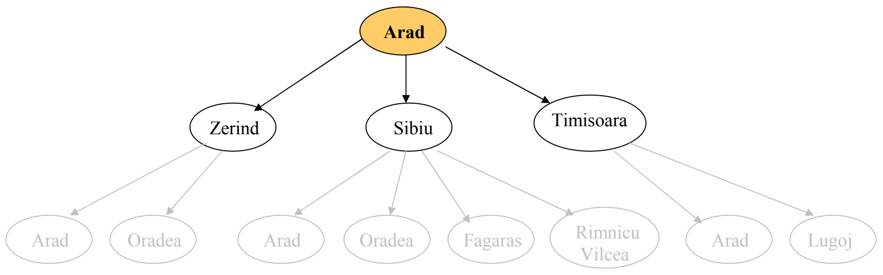
Iterative deepening

Cutoff depth = 2



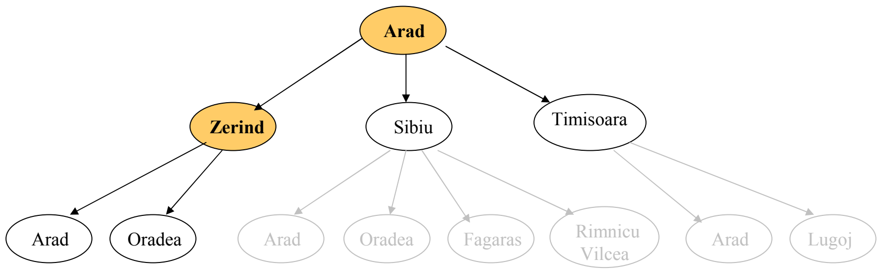
Iterative deepening

Cutoff depth = 2



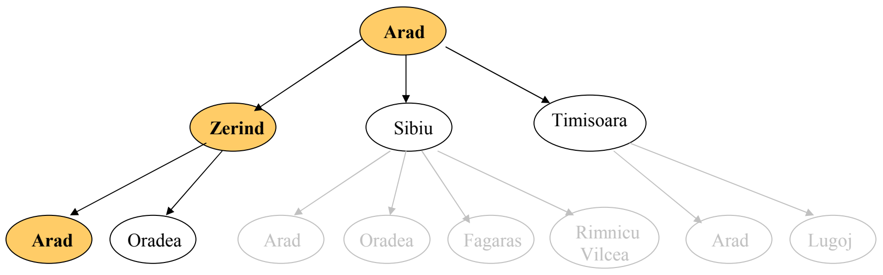
Iterative deepening

Cutoff depth = 2



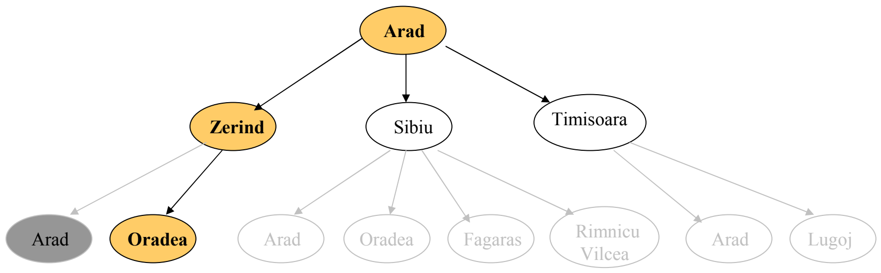
Iterative deepening

Cutoff depth = 2



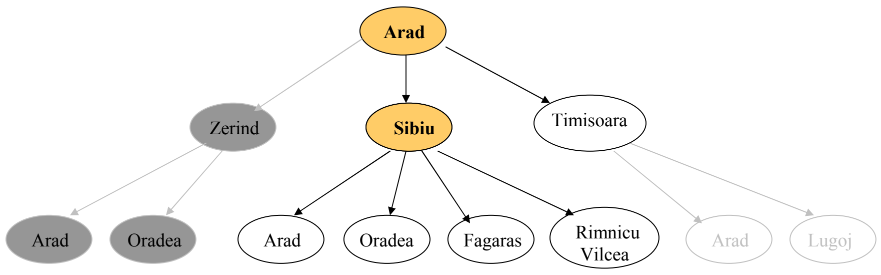
Iterative deepening

Cutoff depth = 2



Iterative deepening

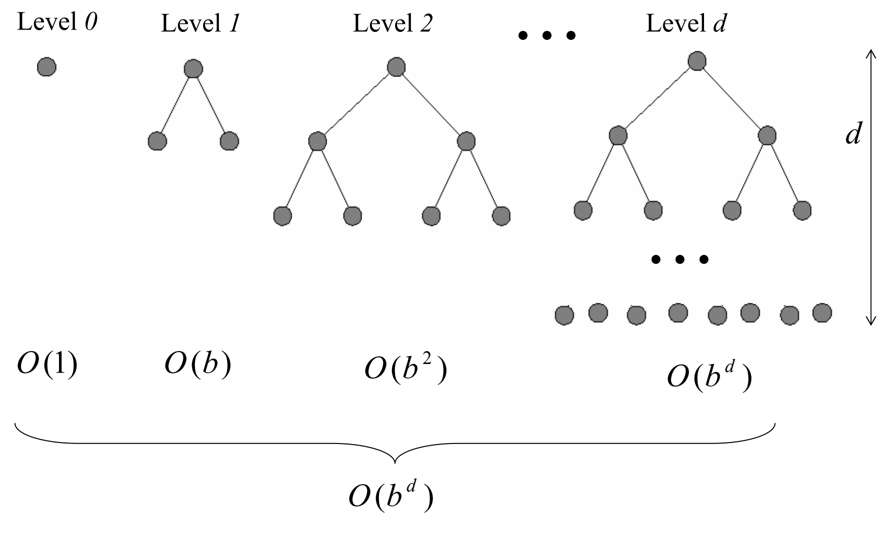
Cutoff depth = 2



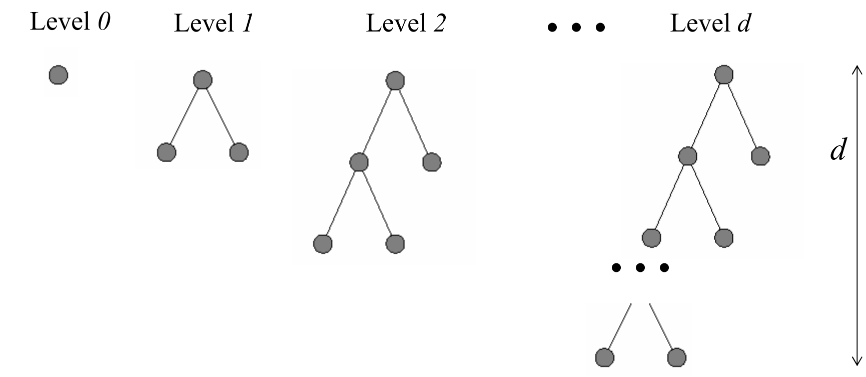
Properties of IDA

- **Completeness:** **Yes.** The solution is reached if it exists.
  (the same as BFS when limit is always increased by 1)
- **Optimality:** **Yes**, for the shortest path.
  (the same as BFS)
- **Time complexity:**
  ?
- **Memory (space) complexity:**
  ?

IDA – time complexity



IDA – memory complexity



$$\underbrace{O(1) \quad O(b) \quad O(2b) \quad O(db)}_{O(db)}$$

Properties of IDA

- **Completeness:** **Yes.** The solution is reached if it exists. (the same as BFS)
- **Optimality:** **Yes**, for the shortest path. (the same as BFS)
- **Time complexity:**

  $$O(1)+O(b^1)+O(b^2)+\ldots+O(b^d)=O(b^d)$$

  exponential in the depth of the solution *d*

  worse than BFS, but asymptotically the same
- **Memory (space) complexity:**

  $$O(db)$$

  much better than BFS

Elimination of state repeats

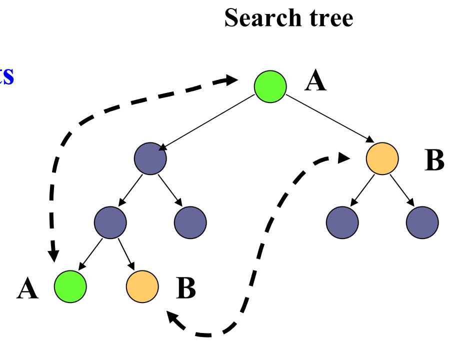
While searching the state space for the solution we can encounter the same state many times.

Question: Is it necessary to keep and expand all copies of states in the search tree?

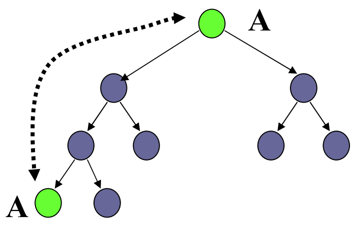
Two possible cases:

(A) Cyclic state repeats

(B) Non-cyclic state repeats



Elimination of cycles

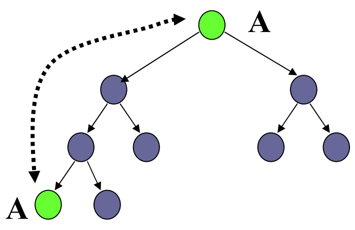


Case A: Corresponds to the path with a cycle

Can the branch (path) in which the same state is visited twice ever be a part of the optimal (shortest) path between the initial state and the goal?

???

Elimination of cycles

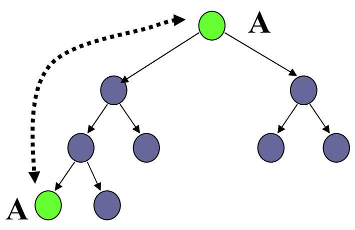


Case A: Corresponds to the path with a cycle

Can the branch (path) in which the same state is visited twice ever be a part of the optimal (shortest) path between the initial state and the goal? **No !!**

Branches representing cycles cannot be the part of the shortest solution and can be eliminated.

Elimination of cycles

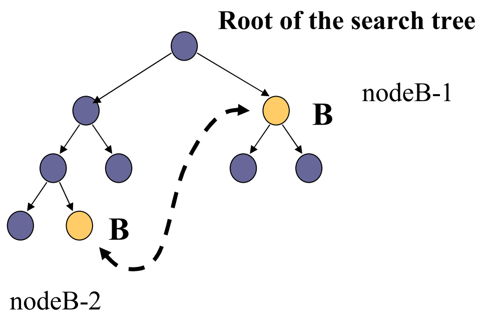


How to check for cyclic state repeats:

Check ancestors in the tree structure.

Do not expand the node with the state that is the same as the state in one of its ancestors.

Elimination of non-cyclic state repeats

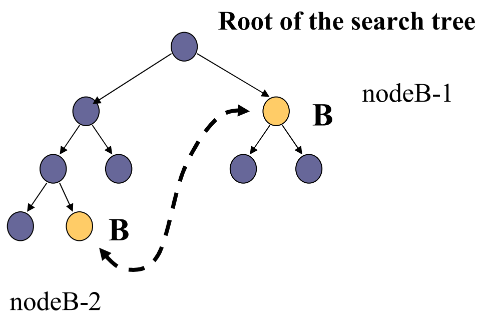


Case B: nodes with the same state are not on the same path from the initial state

Is one of the nodes nodeB-1, nodeB-2 better or preferable?

?

Elimination of non-cyclic state repeats

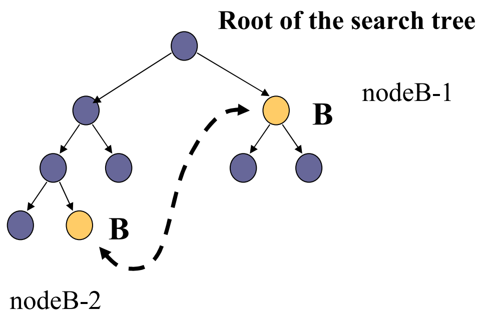


Case B: nodes with the same state are not on the same path from the initial state

Is one of the nodes nodeB-1, nodeB-2 better or preferable?

Yes. nodeB-1 represents the shorter path between the initial state and B

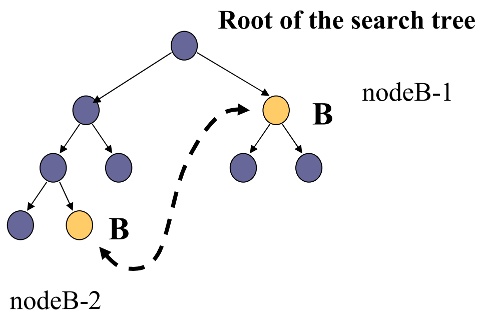
Elimination of non-cyclic state repeats



Since we are happy with the optimal solution nodeB-2 can be eliminated. It does not affect the optimality of the solution.

Problem: Nodes can be encountered in different order during different search strategies.

Elimination of non-cyclic state repeats with BFS



Breadth FS is well behaved with regard to non-cyclic state repeats: nodeB-1 is always expanded before nodeB-2

- Order of expansion determines the correct elimination strategy
- we can safely eliminate the node that is associated with the state that has been expanded before

Elimination of state repeats for the BFS

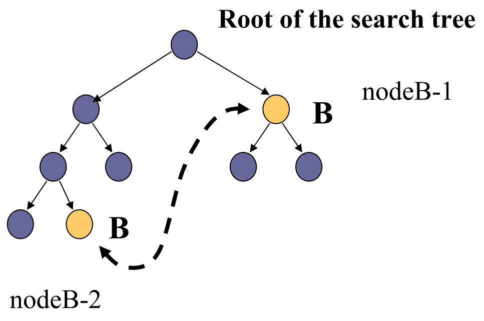
For **the breadth-first search (BFS)**

- we can safely eliminate all second, third, fourth, etc. occurrences of the same state
- this rule covers both cyclic and non-cyclic repeats !!!

Implementation of all state repeat elimination through **marking**:

- All expanded states are marked
- All marked states are stored in a special data structure (a hash table)
- Checking if the node has ever been expanded corresponds to the mark structure lookup

Elimination of non-cyclic state repeats with DFS



Depth FS: nodeB-2 is expanded before nodeB-1

- The order of node expansion does not imply correct elimination strategy
- we need to remember the length of the path between nodes to safely eliminate them

Elimination of all state redundancies

- **General strategy:** A node is redundant if there is another node with exactly the same state and a shorter path from the initial state
  - Works for any search method
  - Uses additional path length information

Implementation: marking with the minimum path value:

- The new node is redundant and can be eliminated if
  - it is in the hash table (it is marked), and
  - its path is longer or equal to the value stored.
- Otherwise the new node cannot be eliminated and it is entered together with its value into the hash table. (if the state was in the hash table the new path value is better and needs to be overwritten.)