CS 2710 Foundations of AI
Lecture 3

Uninformed search methods

Milos Hauskrecht

milos@pitt.edu

5329 Sennott Square

Announcements

Homework assignment 1 is out

- Due on Thursday, September 3, 2020 at 4:15pm before the lecture
- **Report and programming part**:
  - Programming part involves Puzzle 8 problem.

Homework submission:

- Electronic via Canvas
- Separate file upload for
  - Reports (pdf file)
  - Programs (zip file)

Announcements

TA assigned to the course:

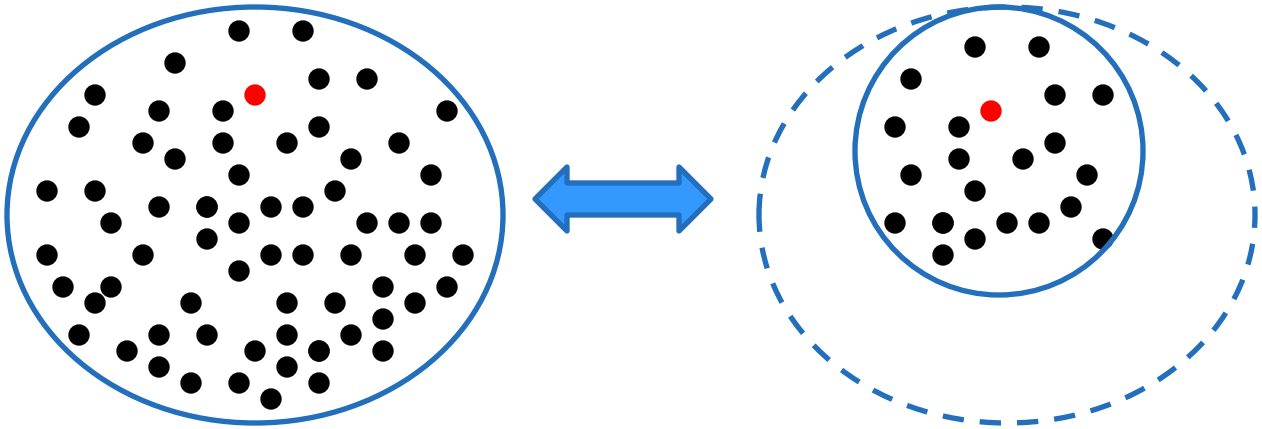
Yoones Rezaei

Email: yor10@pitt.edu

Office hours: TBA tomorrow

Search

- **Search (process)**
  - The process of exploration of the search space
- **The efficiency of the search depends on:**
  - **The search space and its size**
  - Method used to explore (traverse) the search space
  - Condition to test the satisfaction of the search objective (what it takes to determine I found the desired goal object



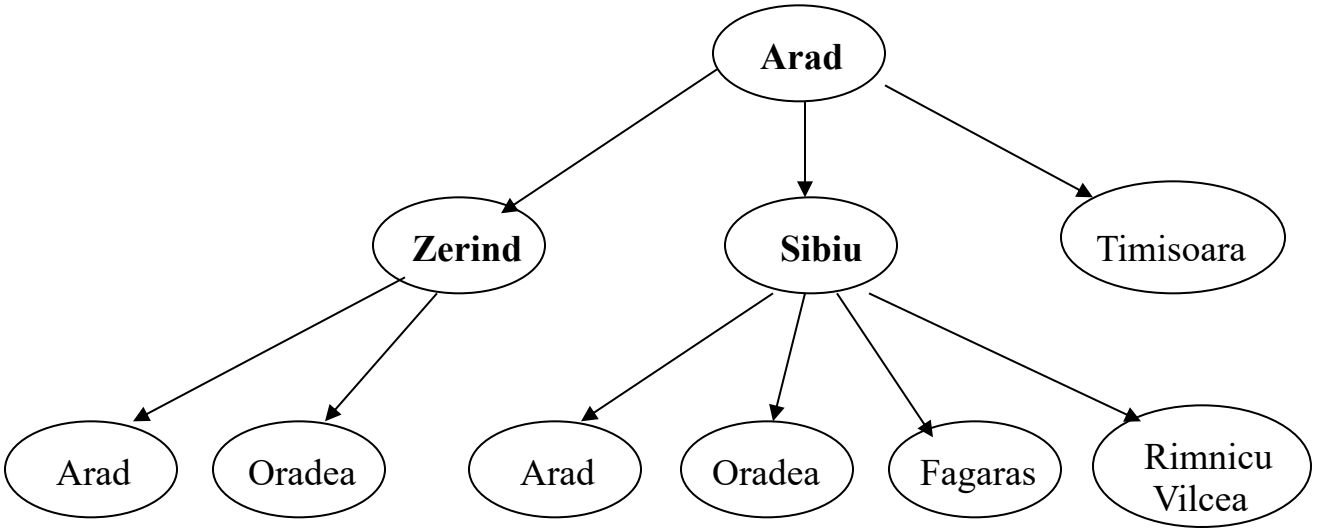
Search

- **Search (process)**
  - The process of exploration of the search space
- **The efficiency of the search depends on:**
  - The search space and its size
  - **Method used to explore (traverse) the search space**
  - Condition to test the satisfaction of the search objective (what it takes to determine I found the desired goal object

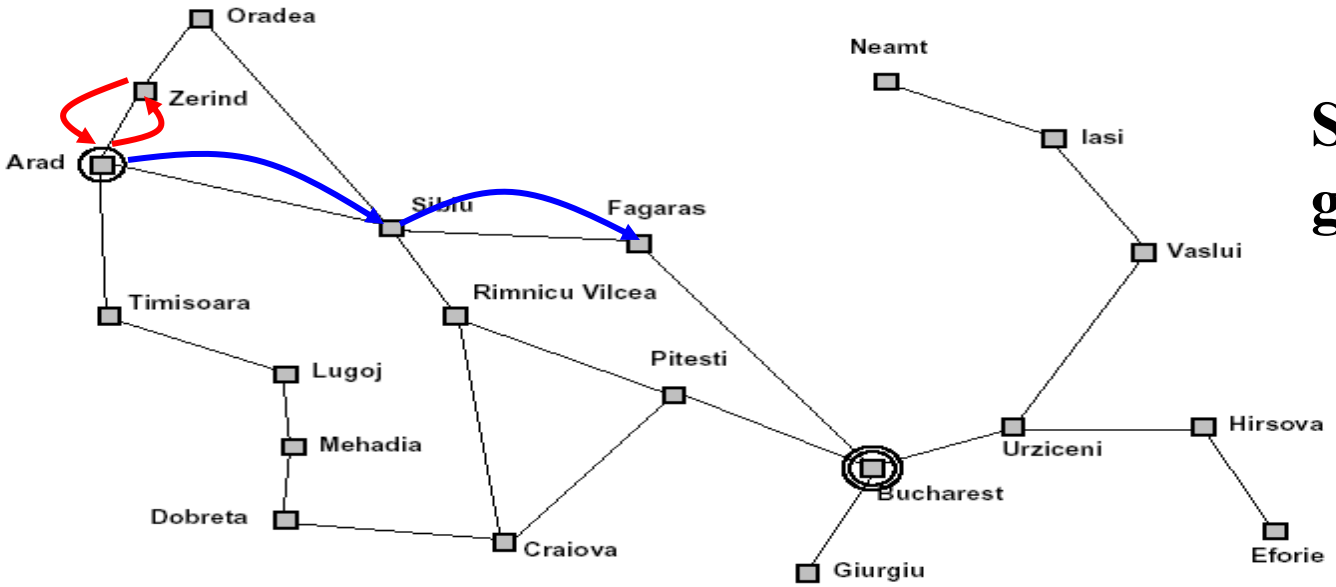
Search process

Exploration of the state space through successive application of operators from the initial state

- **Search tree = structure representing the exploration trace**
  - Is built on-line during the search process
  - Branches correspond to explored paths, and leaf nodes to the exploration fringe

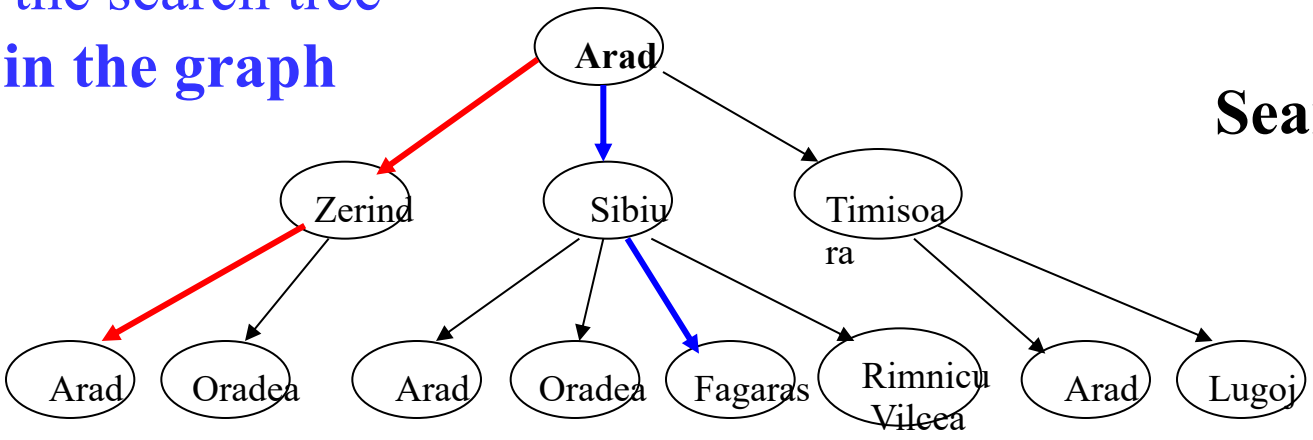


Search tree



State space graph

A branch in the search tree
= **a path in the graph**



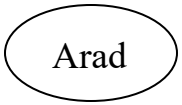
Search tree

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 next **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 next **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

Node chosen to be expanded next

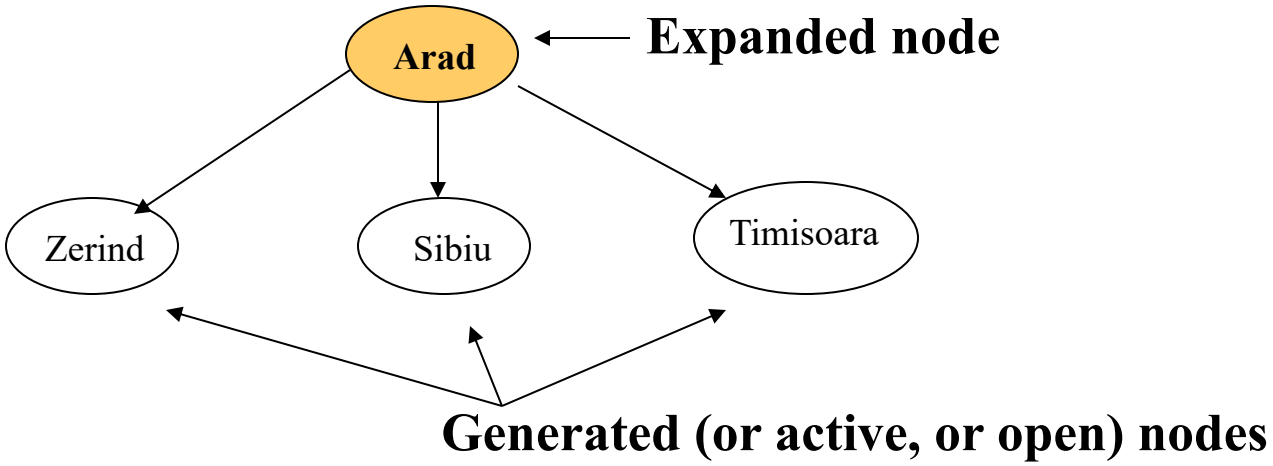
General search algorithm

General-search (*problem, strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **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

Check if the node satisfied the goal

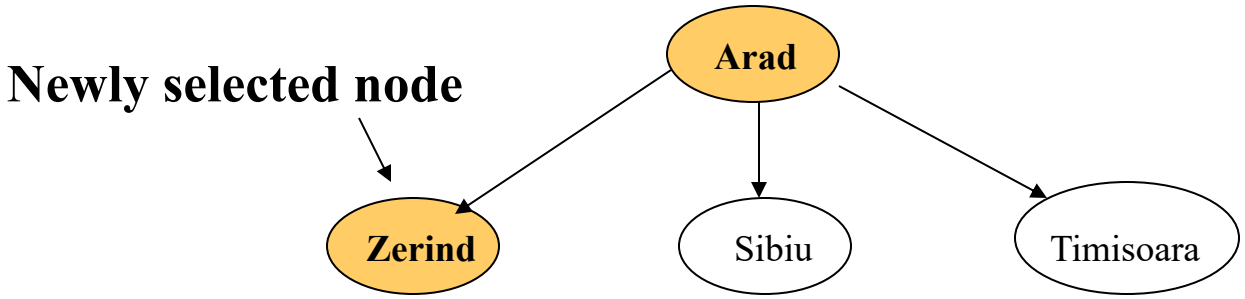
General search algorithm

General-search (*problem, strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **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



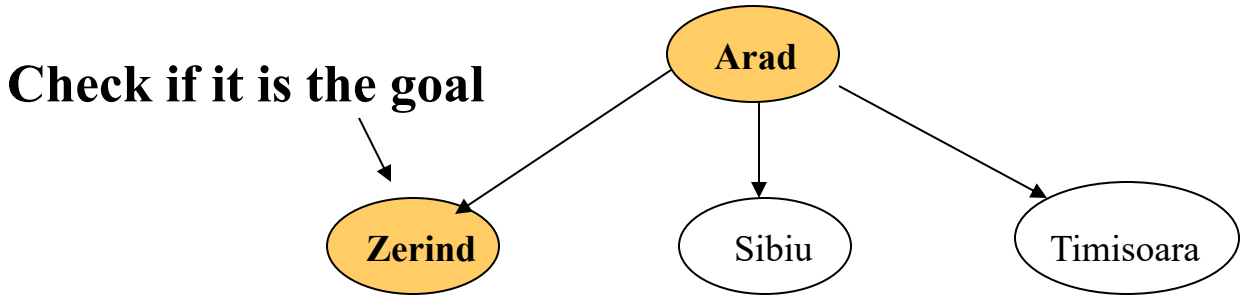
General search algorithm

General-search (*problem, strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **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



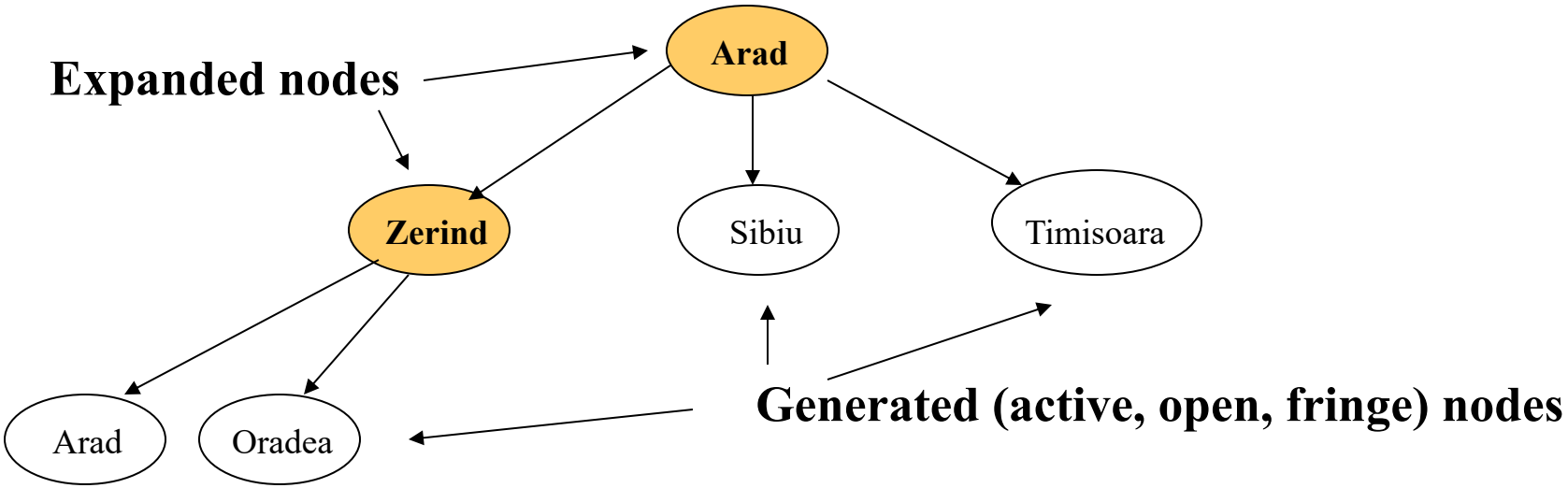
General search algorithm

General-search (*problem, strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **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



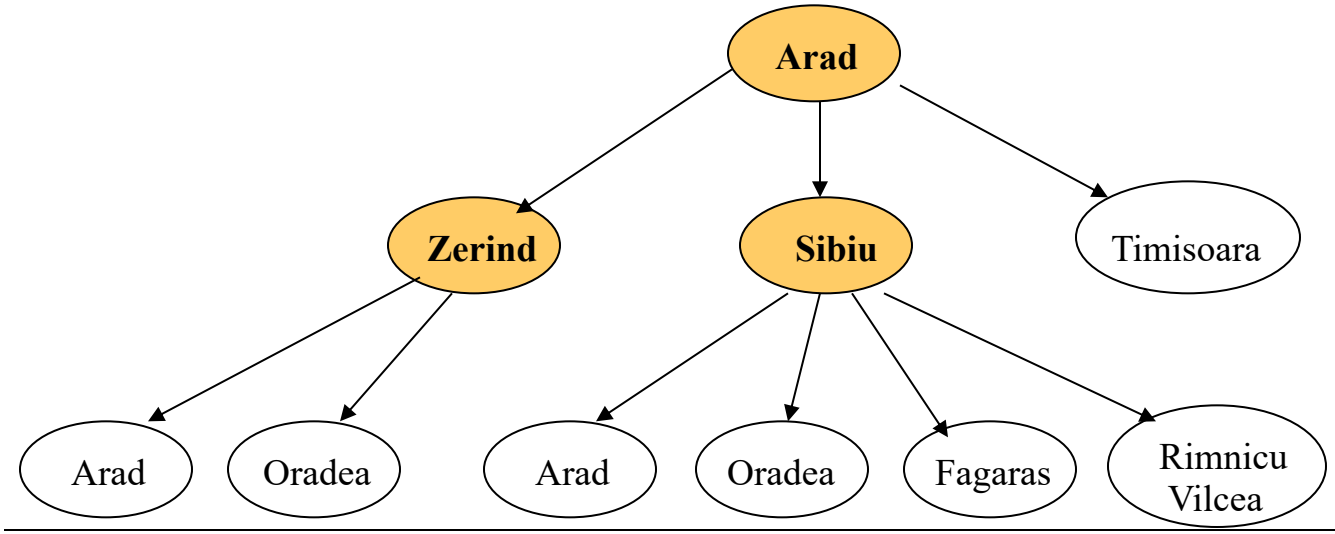
General search algorithm

General-search (*problem, strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **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



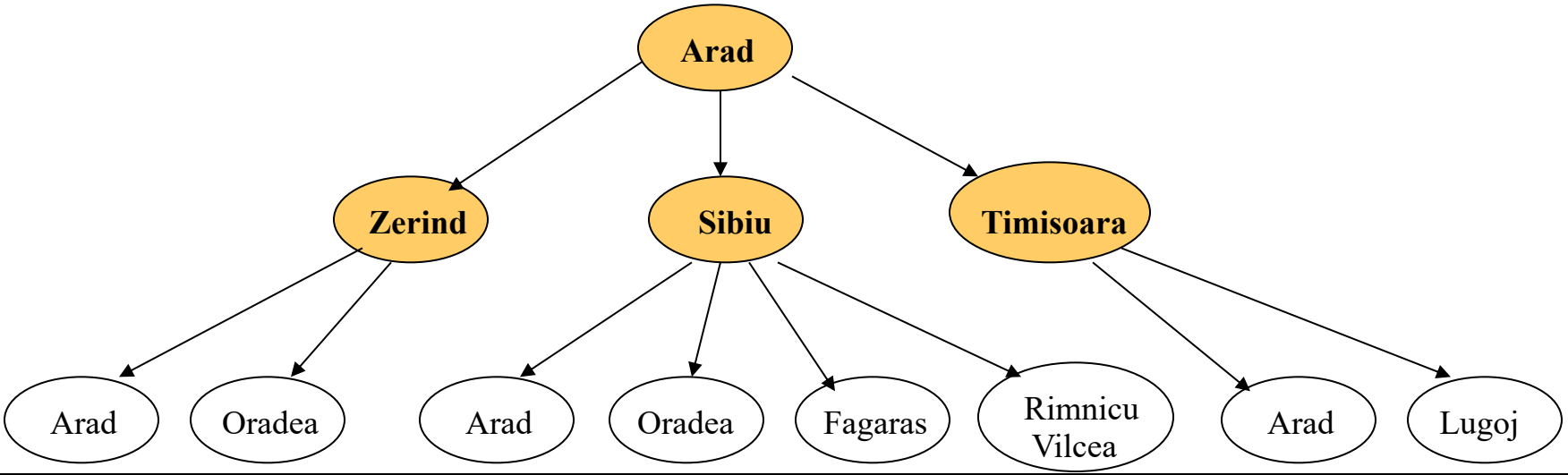
General search algorithm

General-search (*problem, strategy*)
initialize the search tree with the initial state of *problem*
loop
if there are no candidate states to explore next **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 next **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 next **return** failure
choose a leaf node of the tree to expand next **according to a *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 differ in how they explore the space, that is how they choose the node to expand next !!!!!**

Implementation of search

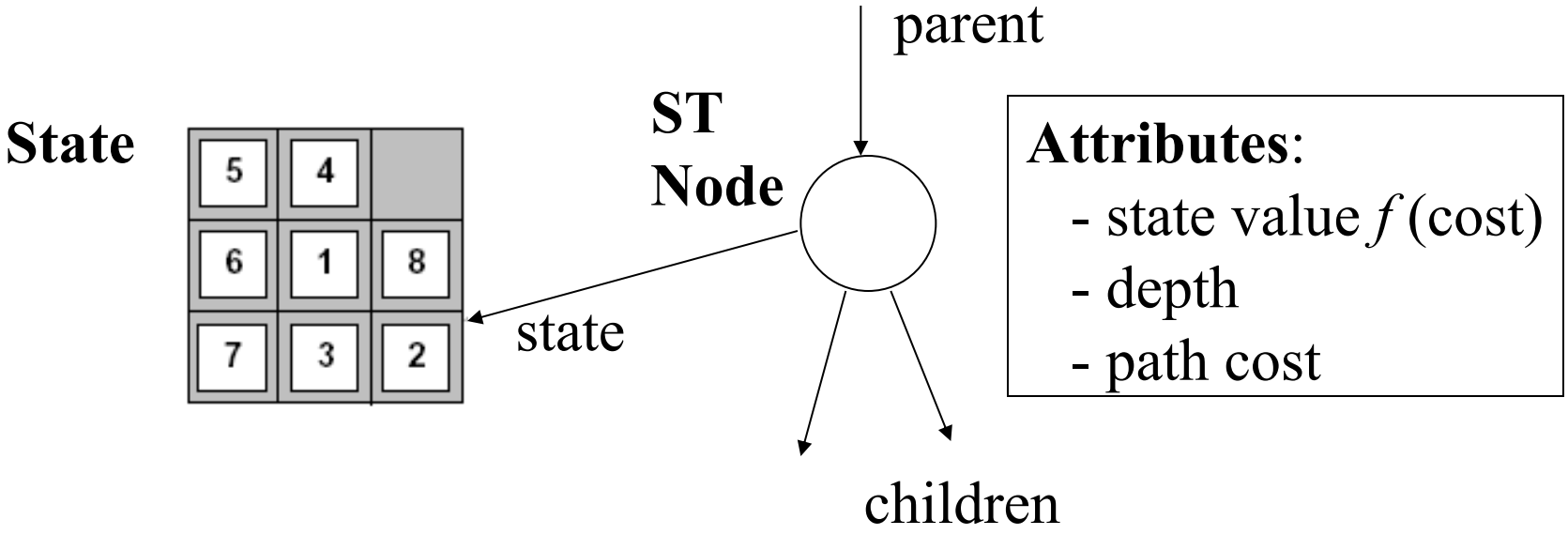
- Search methods can be implemented using the **queue** structure and a **queuing function *f***

```
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 (*search tree nodes*) are added to the queue structure
- **Queuing function *f*** determines what node will be selected next

Implementation of search

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



- **Expand function** – applies Operators to the state represented by the search tree *node*. Together with queuing-function *f* it fills the attributes.

Uninformed search methods

- Search techniques that rely only on the 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?

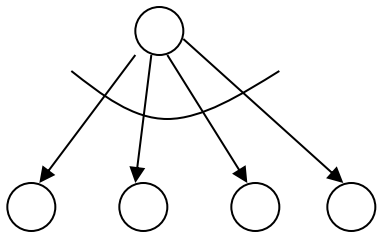
- **Optimality.**
  - Is the solution returned by the algorithm optimal? Does it give a minimum length path?

- **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.**
  - **Complexity** is measured in terms of the following tree parameters:
    - b – **maximum branching factor**
    - d – **depth of the optimal solution**
    - m – **maximum depth (size of the state space)**

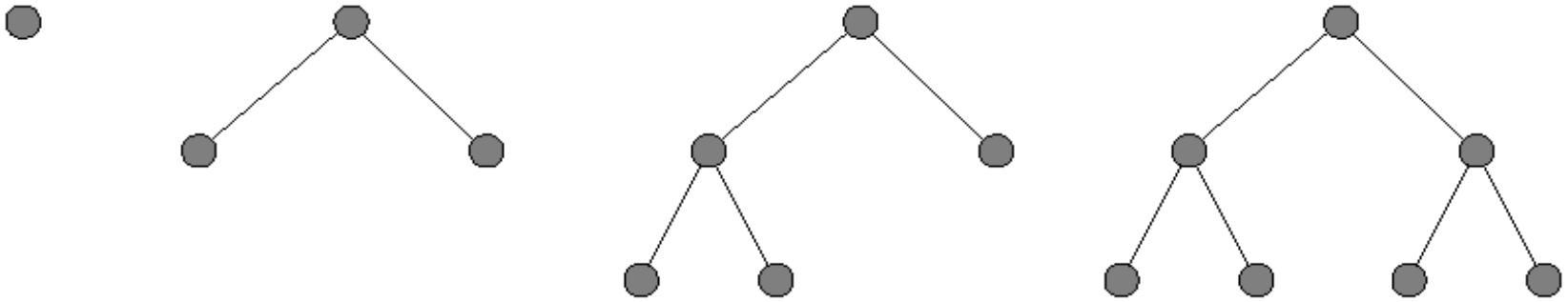
Branching factor

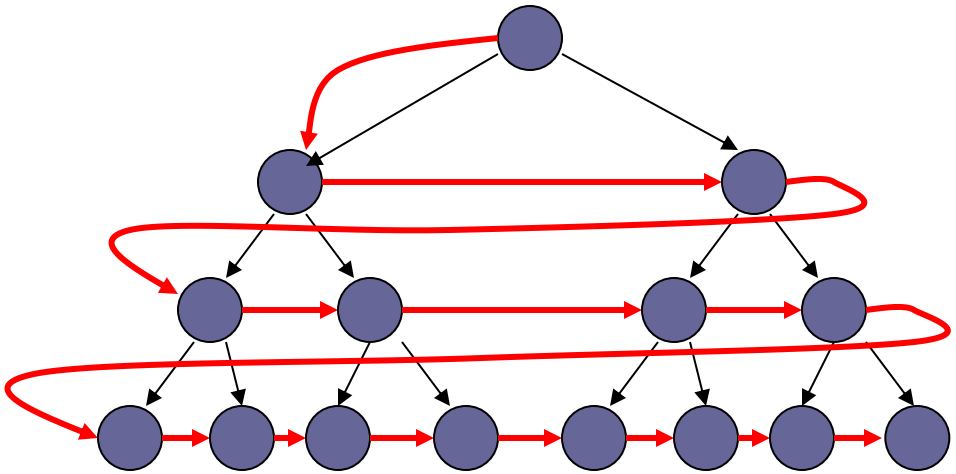


The number of applicable operators

Breadth first search (BFS)

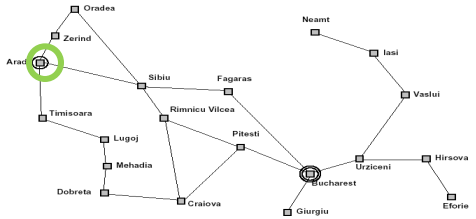
- The shallowest node is expanded first



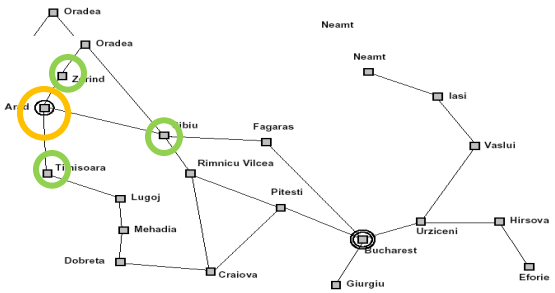


Breadth-first search

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

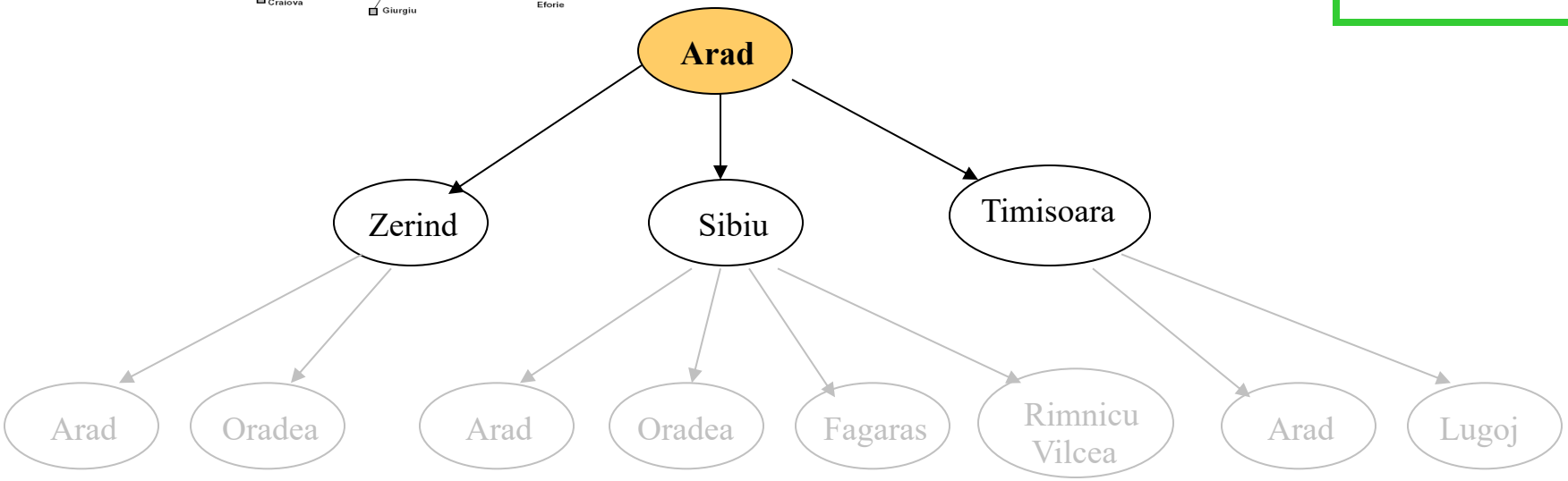


Breadth-first search

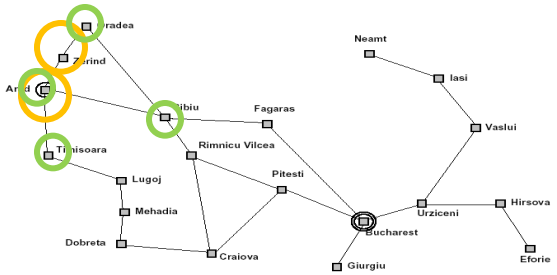


queue ➡

Zerind
Sibiu
Timisoara

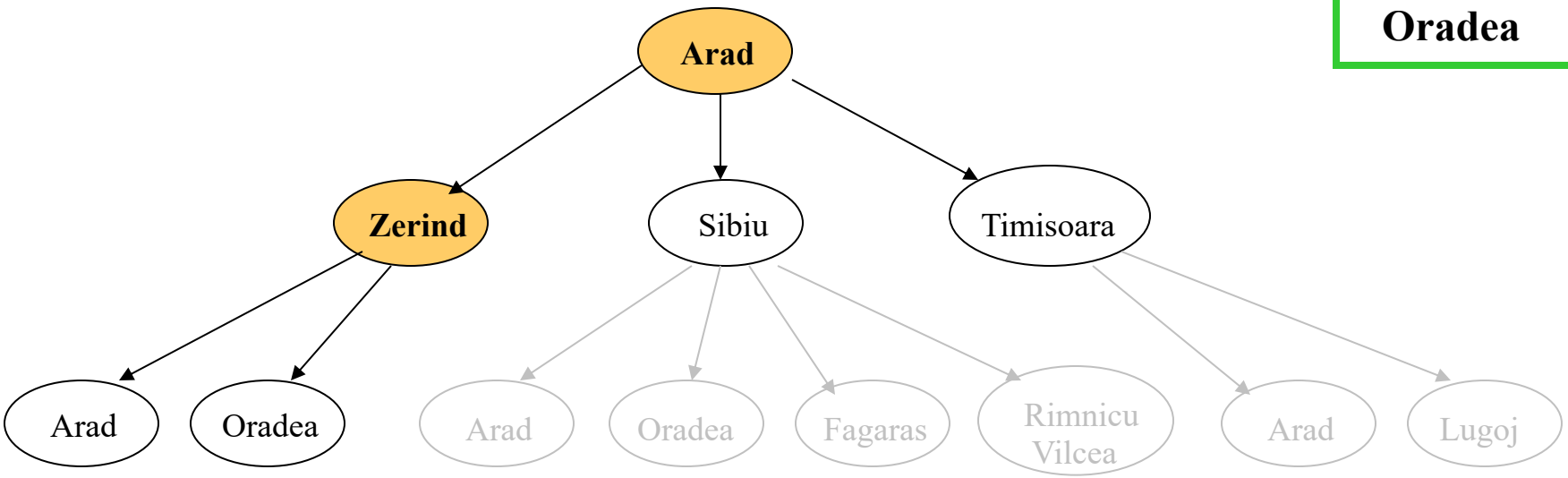


Breadth-first search

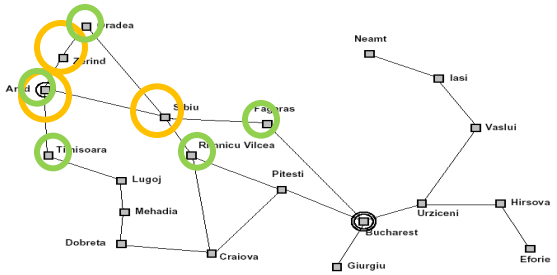


queue ➡

Sibiu
Timisoara
Arad
Oradea

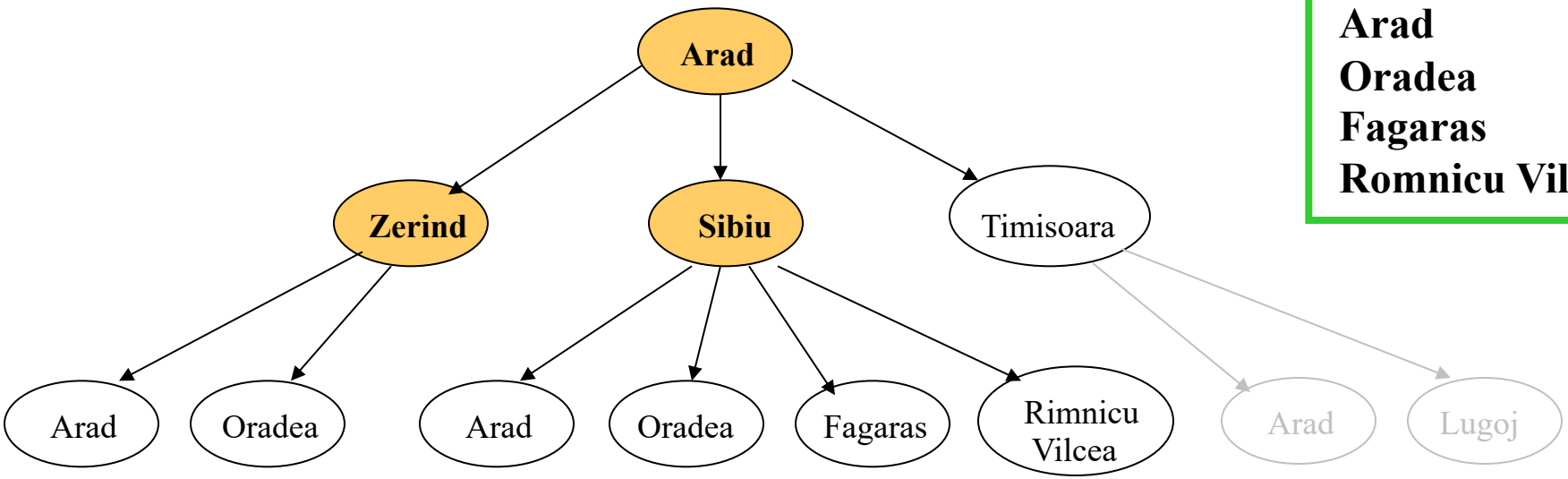


Breadth-first search

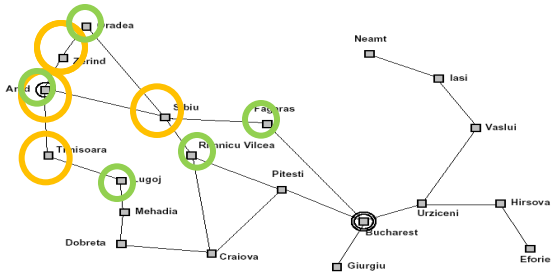


queue ➡

Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Romnicu Vilcea

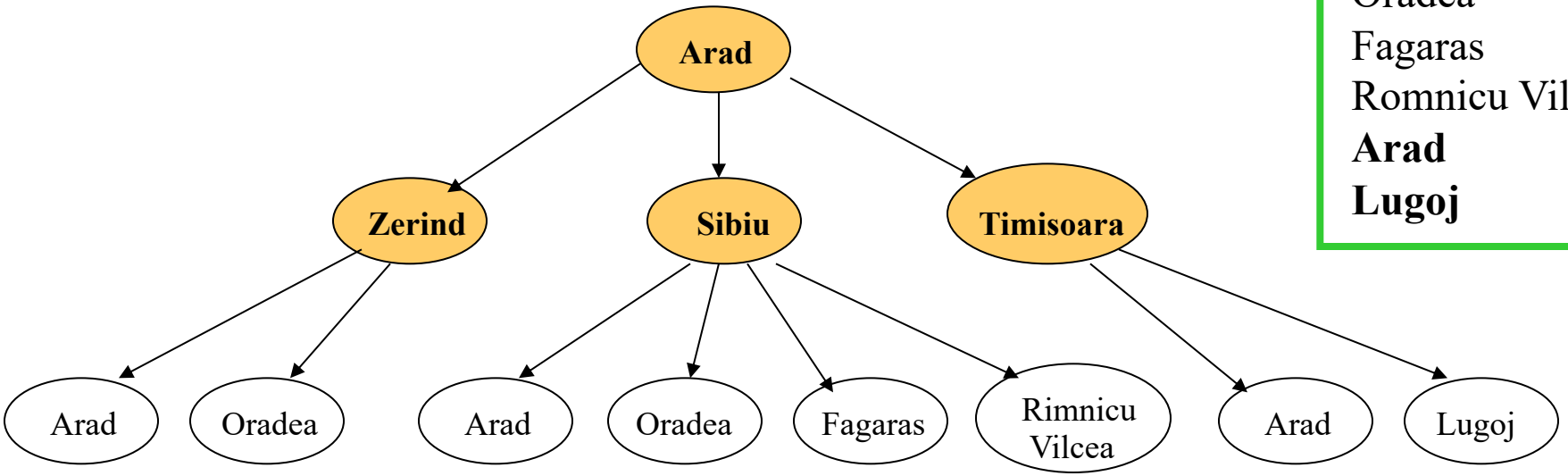


Breadth-first search



queue

Arad
Oradea
Arad
Oradea
Fagaras
Romnicu Vilcea
Arad
Lugoj



Properties of breadth-first search

- **Completeness:** 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?

- **Time complexity: ?**
- **Memory (space) complexity: ?**

  - **For complexity use:**
    - b – maximum branching factor
    - d – depth of the optimal solution
    - m – maximum depth of the search tree

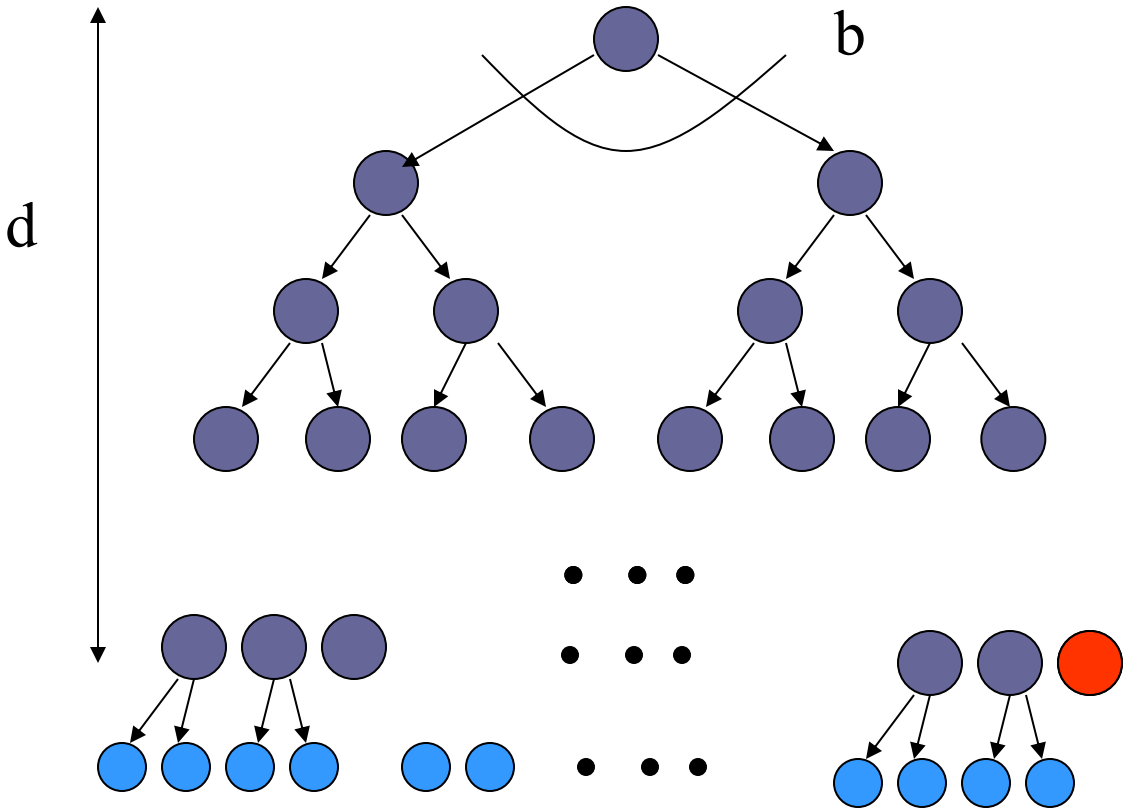
Properties of breadth-first search

- **Completeness: Yes.**
- **Optimality: ?**
- **Time complexity: ?**
- **Memory (space) complexity: ?**

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: ?**

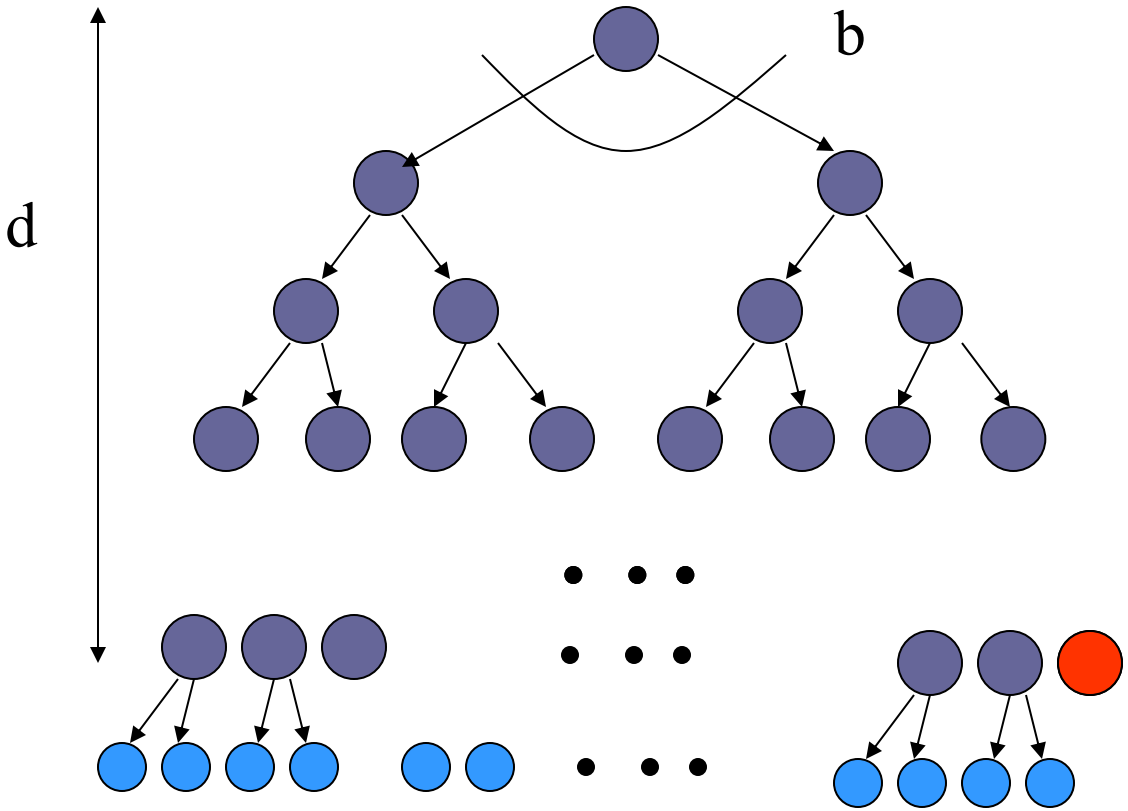
BFS – time complexity



depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Total nodes: **?**

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)
d+1	2^{d+1} (b^{d+1})

Expanded nodes: $O(b^d)$

Total nodes: $O(b^{d+1})$

Properties of breadth-first search

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

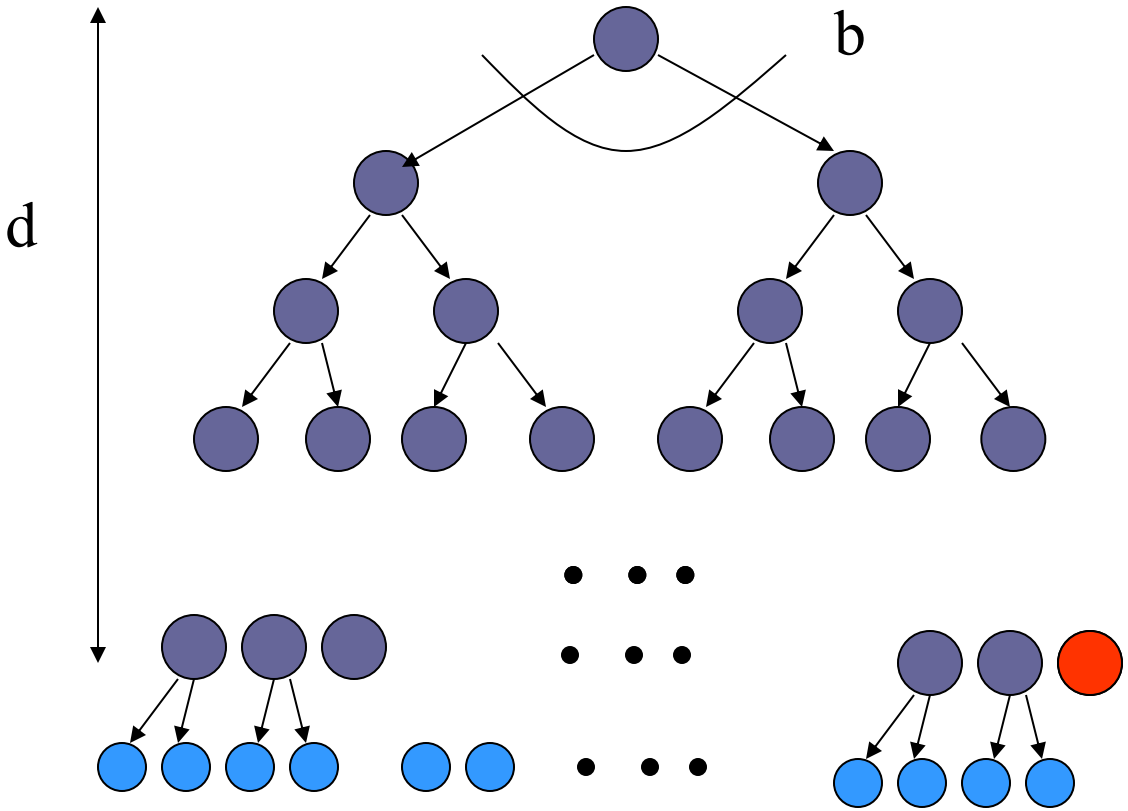
$$1 + b + b^2 + \ldots + b^d = O(b^d)$$

exponential in the depth of the solution *d*

- **Memory (space) complexity: ?**

BFS – memory complexity

- Count nodes kept in the tree structure or in the queue

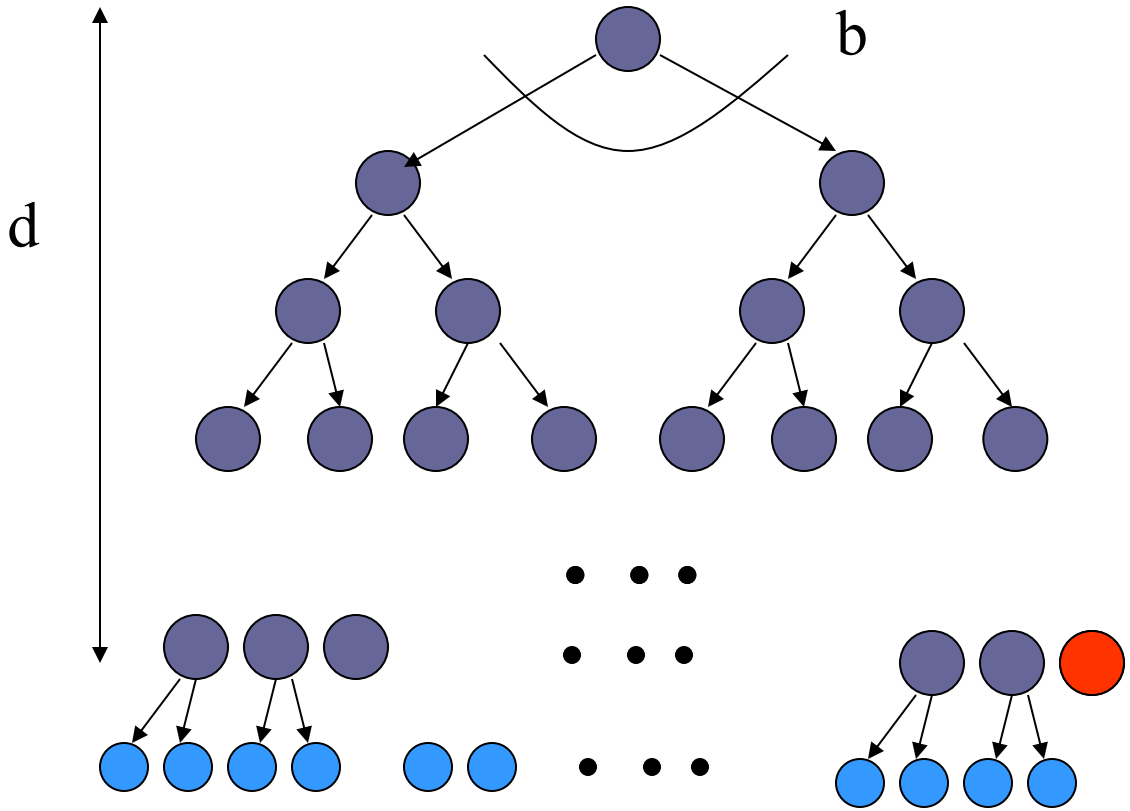


depth	number of nodes
0	1
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Total nodes: **?**

BFS – memory complexity

- Count nodes kept in the tree structure 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)
d+1	2^{d+1} (b^{d+1})

Expanded nodes: $O(b^d)$

Total nodes: $O(b^{d+1})$

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*

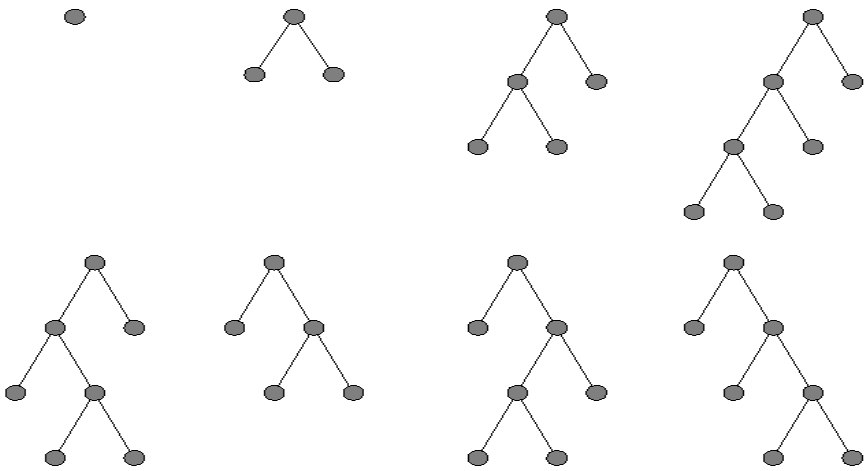
- **Memory (space) complexity:**

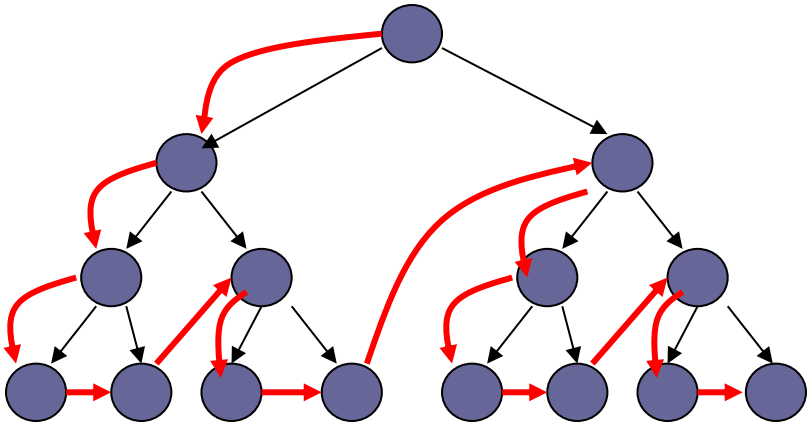
$$O(b^d)$$

nodes are kept in the memory

Depth-first search (DFS)

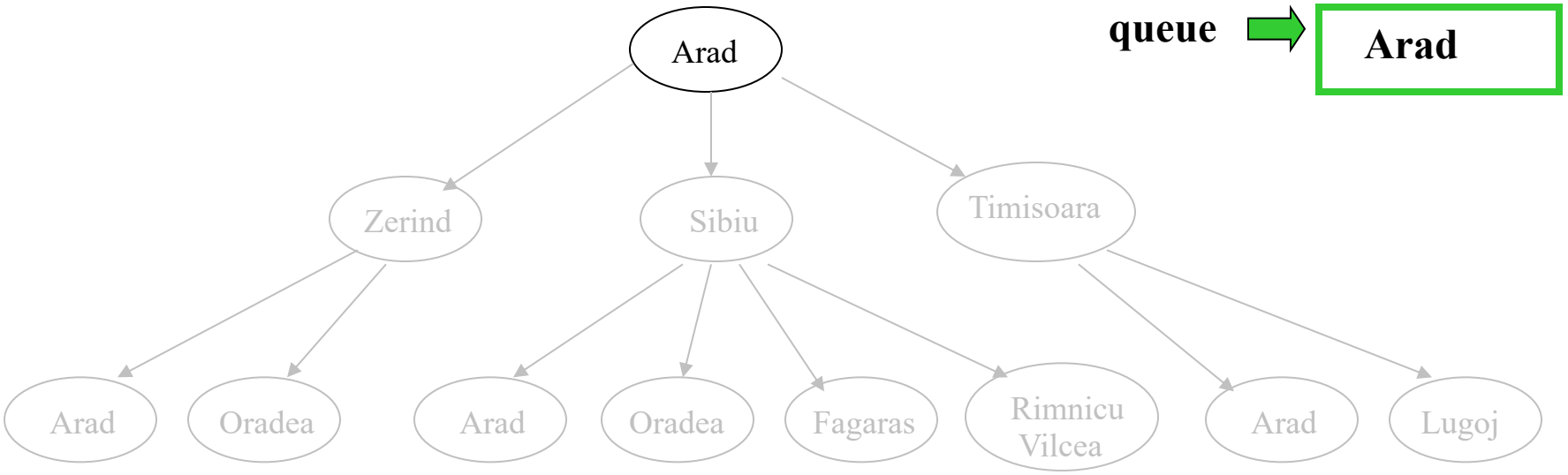
- **The deepest node is expanded first**
- Backtrack when the path cannot be further expanded



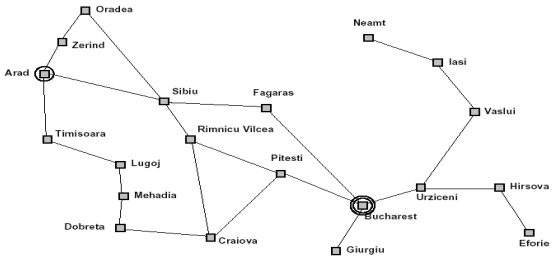


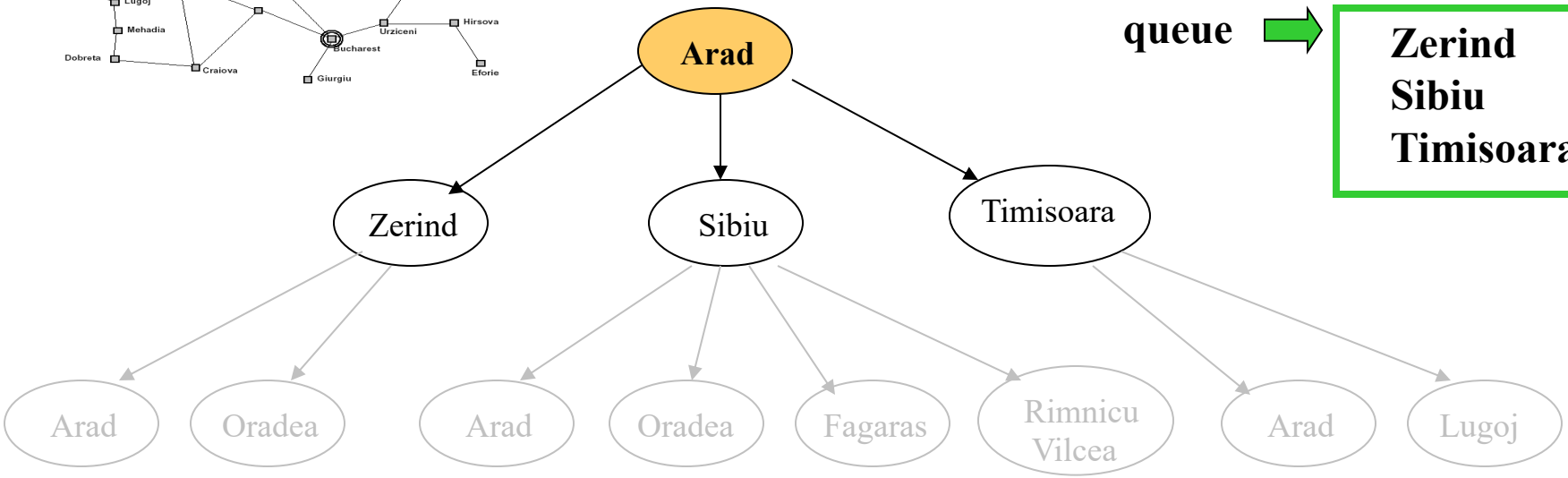
Depth-first search

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



Depth-first search

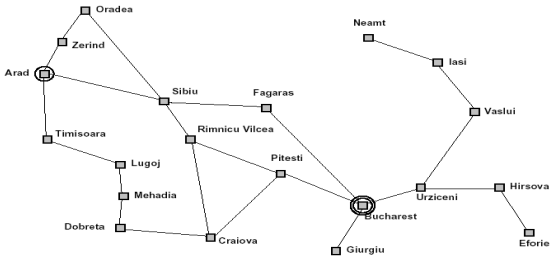


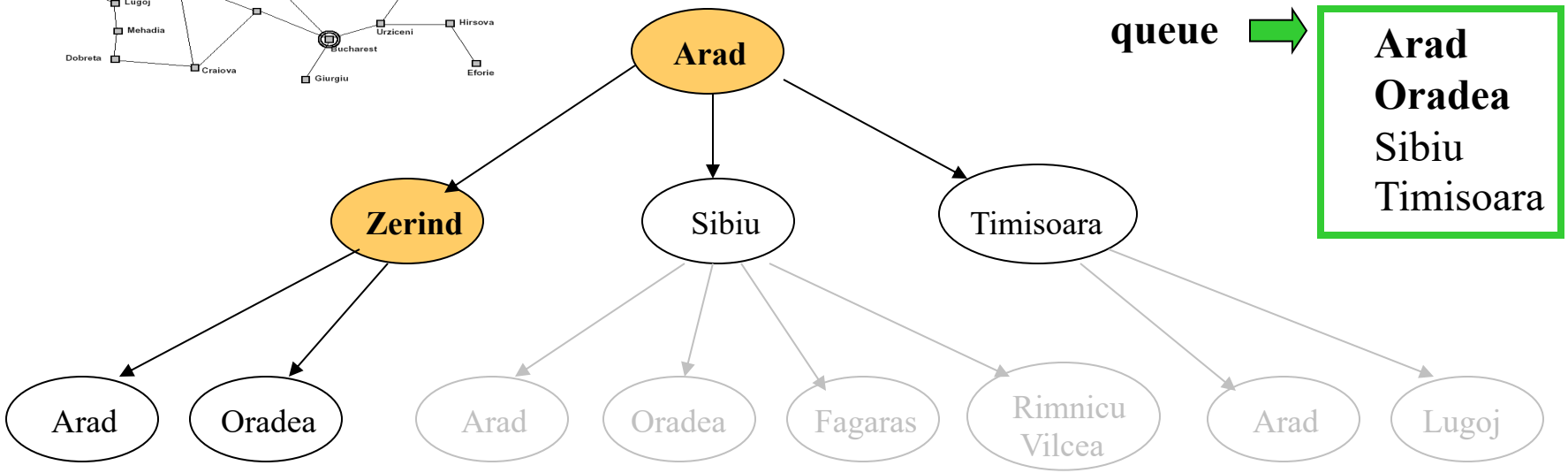


queue

Zerind
Sibiu
Timisoara

Depth-first search

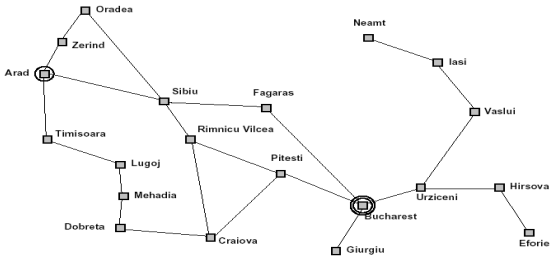


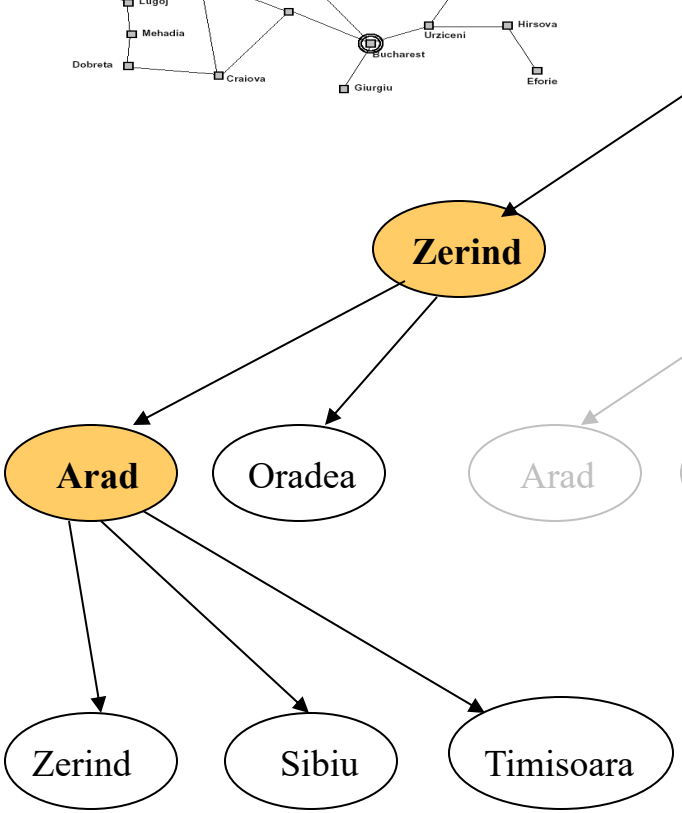


queue

Arad
Oradea
Sibiu
Timisoara

Depth-first search





queue

Zerind
Sibiu
Timisoara
Oradea
Sibiu
Timisoara

Note: Arad – Zerind – Arad cycle

Properties of depth-first search

- **Completeness: Does it always find the solution if it exists?**
- **Optimality: ?**
- **Time complexity: ?**
- **Memory (space) complexity: ?**

Properties of depth-first search

- **Completeness:** **No.** If infinite loops can occur.
  - **Solution 1:** set the maximum depth limit *m*
  - **Solution 2:** prevent occurrence of cycles
- **Optimality: does it find the minimum length path ?**

- **Time complexity: ?**

- **Memory (space) complexity: ?**

Properties of depth-first search

- **Completeness:** **No,** if we permit infinite loops. .
  - **Solution 1:** set the maximum depth limit *m*
  - **Solution 2:** prevent occurrence of cycles
- **Optimality: does it find the minimum length path ?**

- **Time complexity: ?**

- **Memory (space) complexity: ?**

Properties of depth-first search

- **Completeness:** **No.** If we permit infinite loops.
  - **Solution 1:** set the maximum depth limit *m*
  - **Solution 2:** prevent occurrence of cycles
- **Optimality:** **No.** Solution found first may not be the shortest possible.

- **Time complexity: ?**

- **Memory (space) complexity: ?**

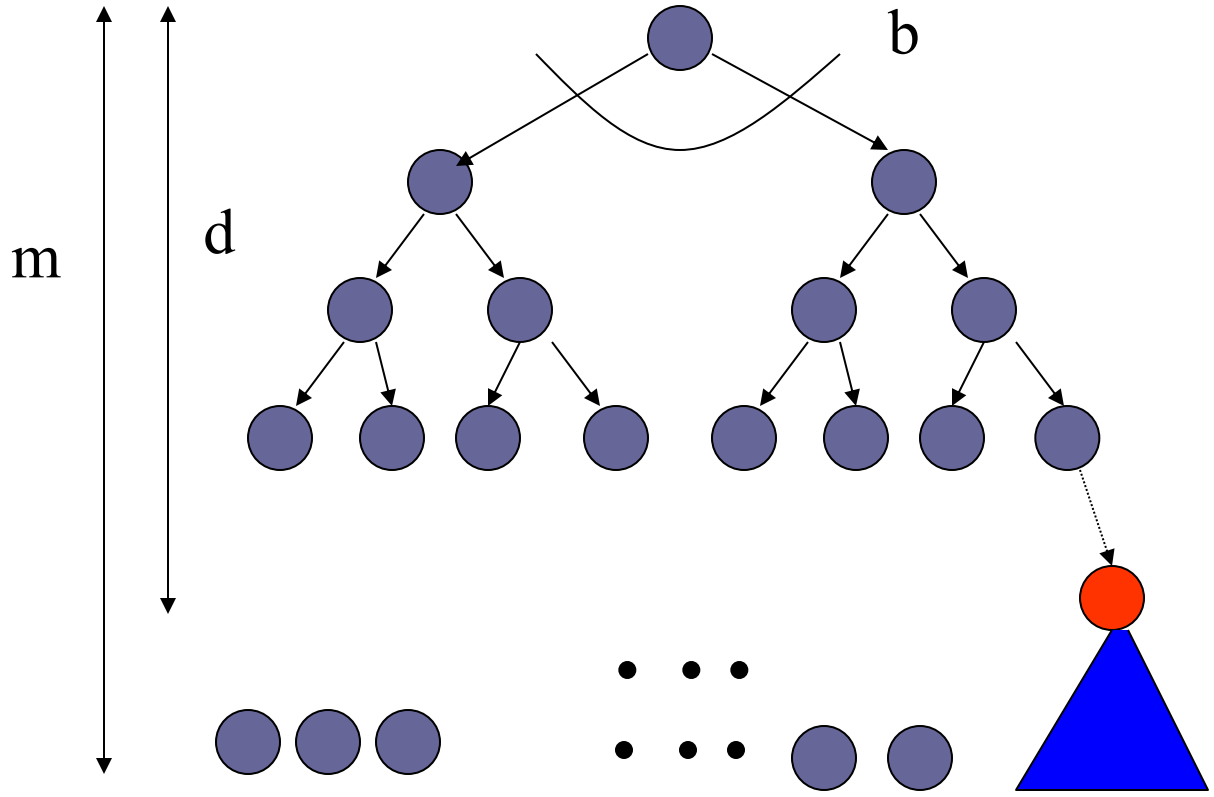
Properties of depth-first search

- **Completeness:** **No.** If we permit infinite loops.
  - **Solution 1:** set the maximum depth limit *m*
  - **Solution 2:** prevent occurrence of cycles
- **Optimality:** **No.** Solution found first may not be the shortest possible.

- **Time complexity: assume a finite maximum tree depth *m***

- **Memory (space) complexity: ?**

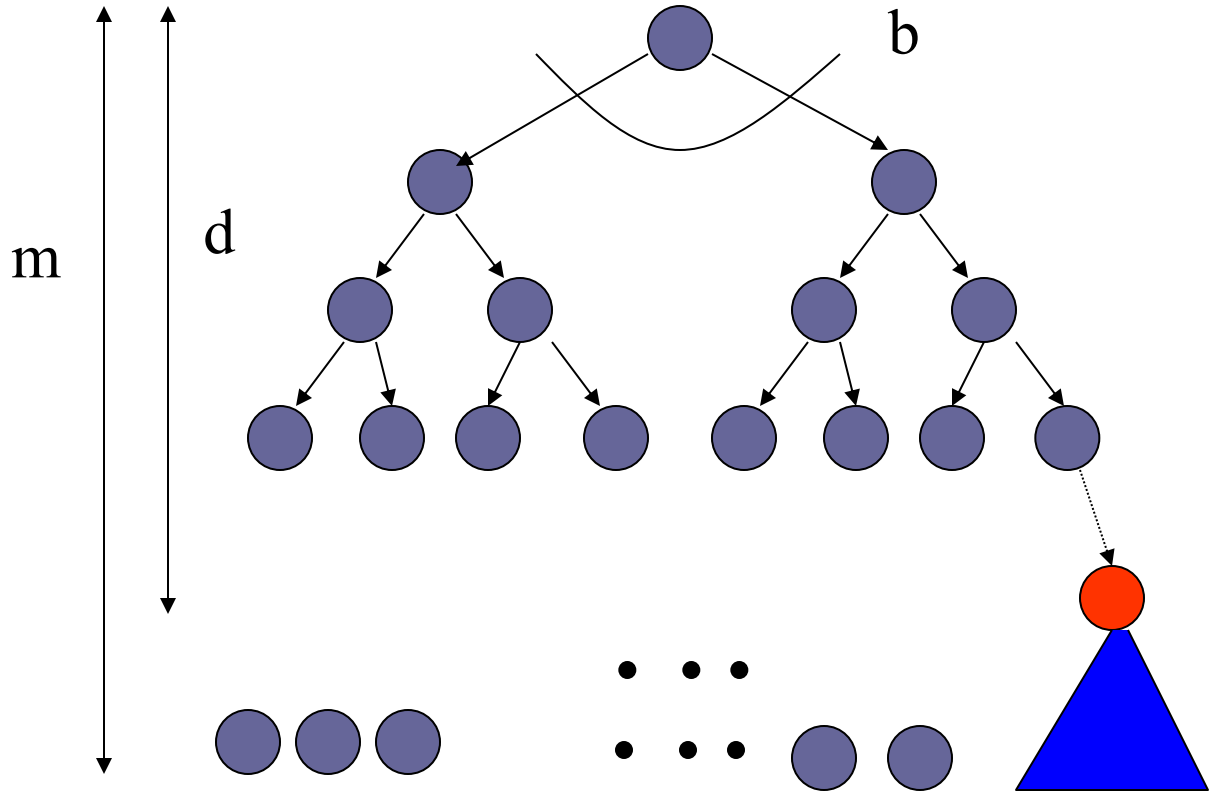
DFS – time complexity



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}

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}

Complexity: $O(b^m)$

Properties of depth-first search

- **Completeness:** **No.** If we permit infinite loops.
  - **Solution 1:** set the maximum depth limit m
  - **Solution 2:** prevent occurrence of cycles
- **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*

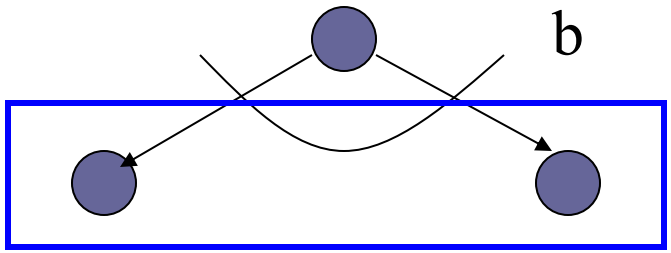
- **Memory (space) complexity: ?**

DFS – memory complexity

b

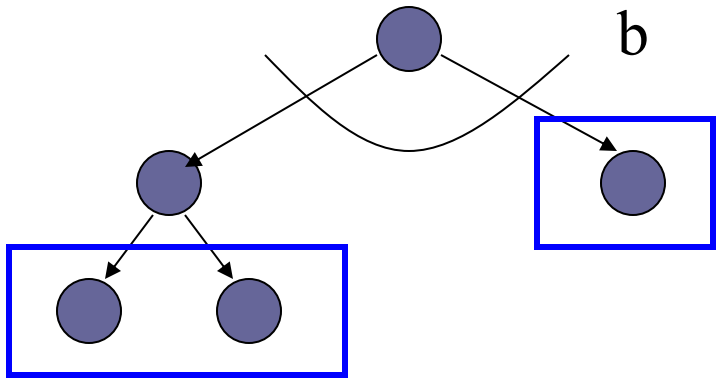
depth	number of nodes kept
0	1

DFS – memory complexity



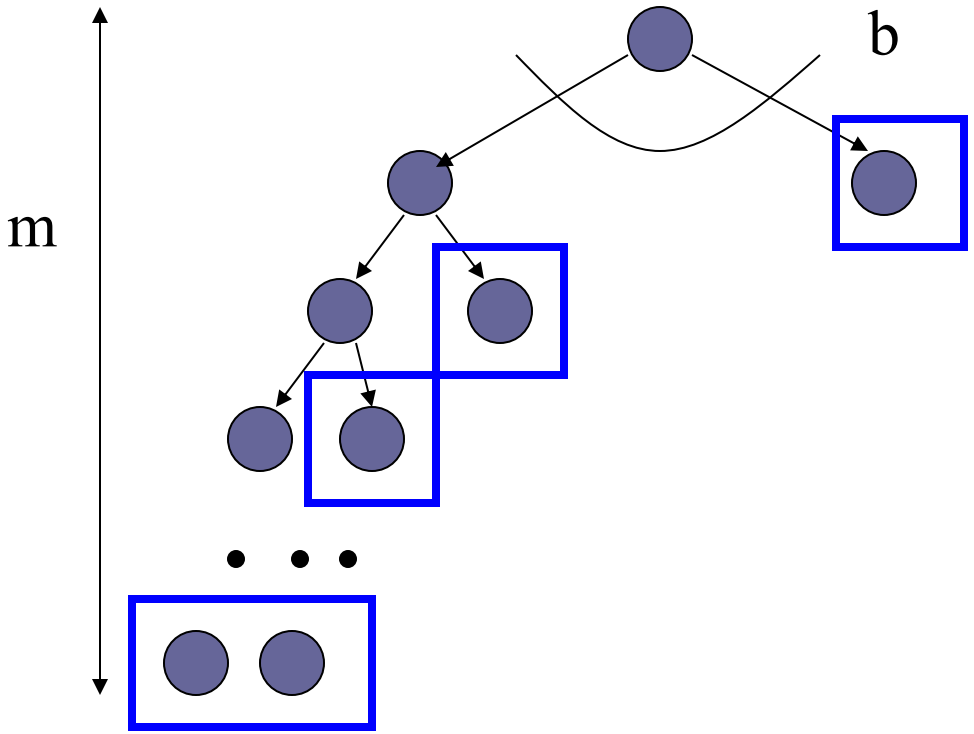
depth	number of nodes kept
0	0
1	2 = b

DFS – memory complexity



depth	number of nodes kept
0	0
1	1 = (b-1)
2	2 = b

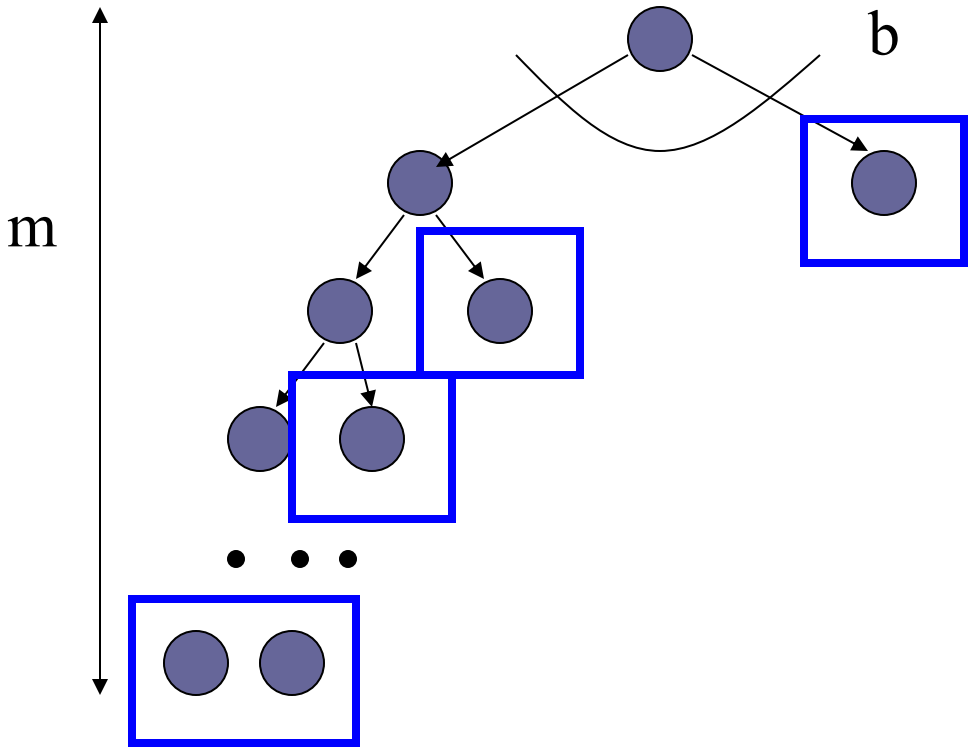
DFS – memory complexity



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

Complexity:

DFS – memory complexity

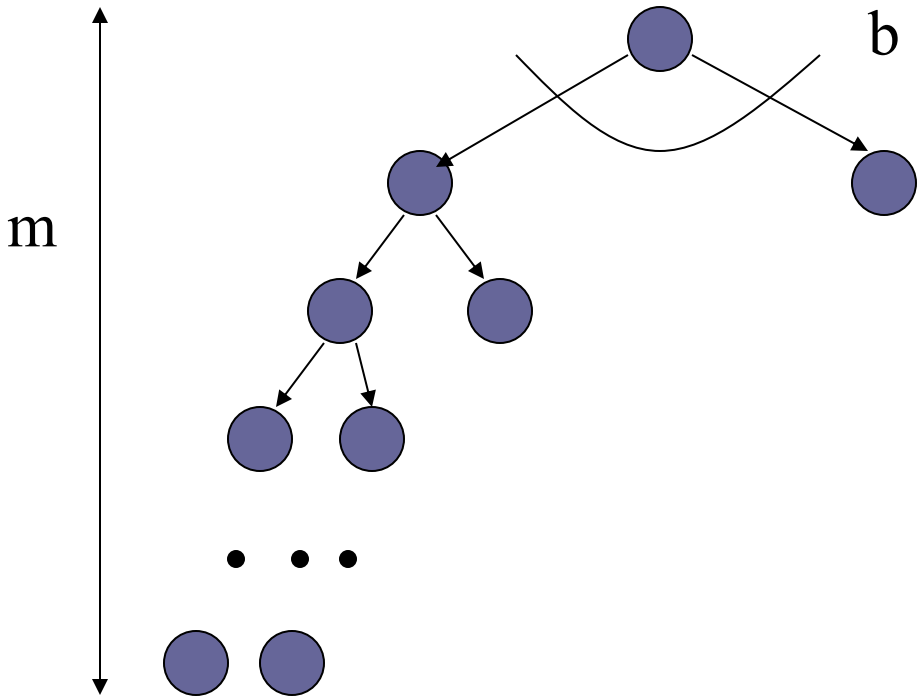


depth	number of nodes kept
0	0
1	1=(b-1)
2	1= (b-1)
3	1 =(b-1)
• • •	
m	2=b

Complexity: $O(bm)$

DFS – memory complexity

Count nodes kept in the tree structure or 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.** If we permit infinite loops.
  - **Solution 1:** set the maximum depth limit *m*
  - **Solution 2:** prevent occurrence of cycles
- **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: ?**

$$O(bm)$$

linear in the maximum depth of the search tree *m*

Setting the maximum depth of the depth-first search

- Setting the maximum depth of the search tree avoids pitfalls of the depth first search

- **Problem:** **How to pick the maximum depth?**

- **Assume:** we have a traveler problem with 20 cities
- How to pick the maximum tree depth?

Setting the maximum depth of the depth-first search

- Setting the maximum depth of the search tree avoids pitfalls of depth first search

- **Problem: How to pick the maximum depth?**

- **Assume:** we have a traveler problem with 20 cities
  - How to pick the maximum tree depth?
  - **We need to consider only paths of length < 20**

- Limited depth DFS
- **Time complexity:** $O(b^m)$
- **Memory complexity**: $O(bm)$

Elimination of state repeats

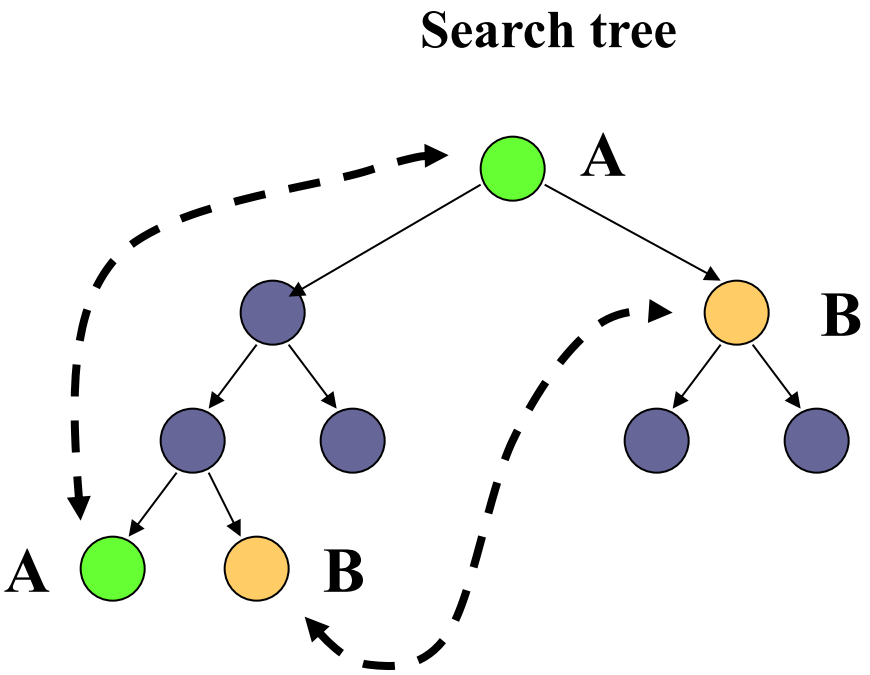
While searching the state space for the solution we can encounter the same state many times. Recall more tree nodes can point to the same state (e.g. city).

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

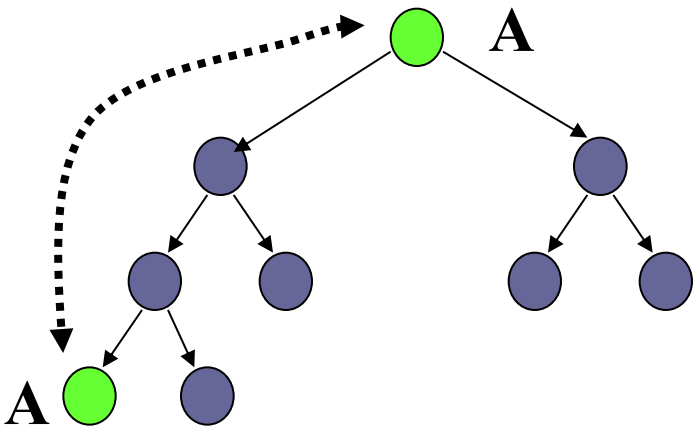
Two possible cases:

(A) Cyclic state repeats

(B) Non-cyclic state repeats



Elimination of cycles

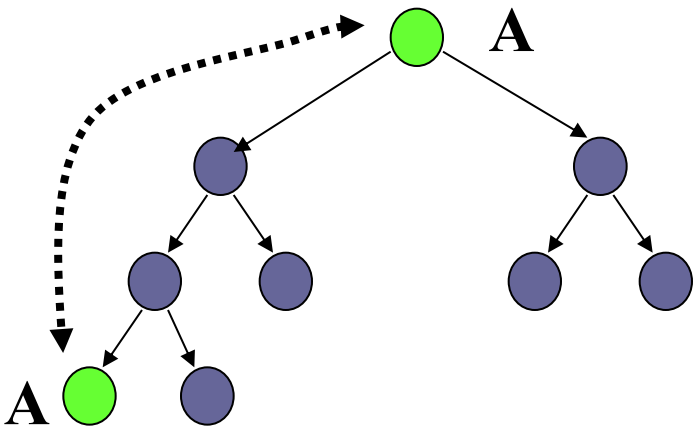


Case A: Corresponds to the path with a cycle

Question: 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

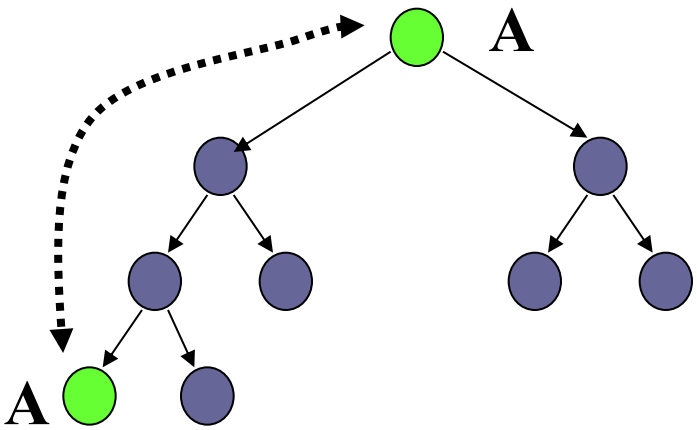


Case A: Corresponds to the path with a cycle

Question: 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

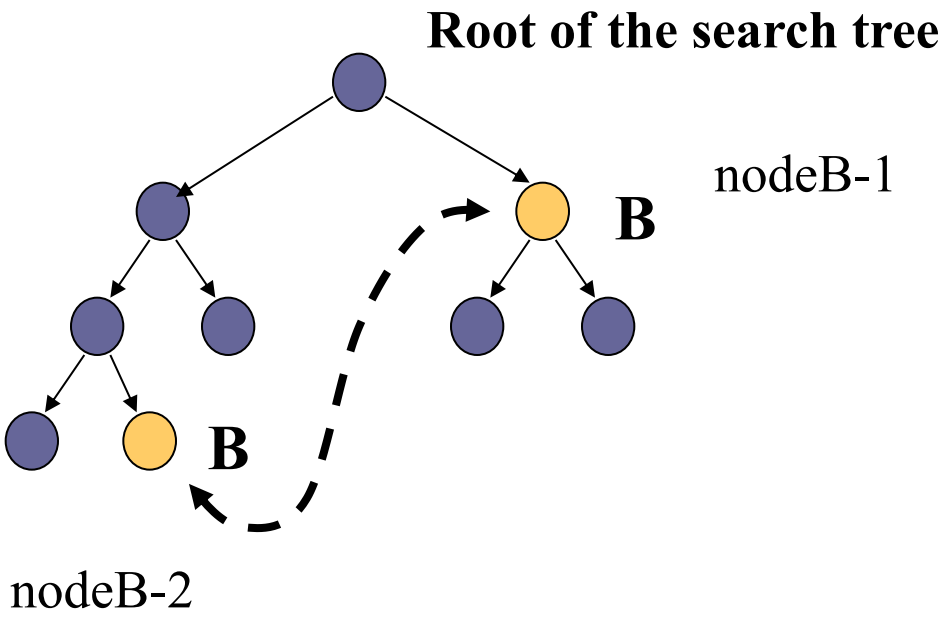


How to check for cyclic state repeats:

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

- Check ancestors in the tree structure
- **Caveat:** we need to keep the tree traverse it up.

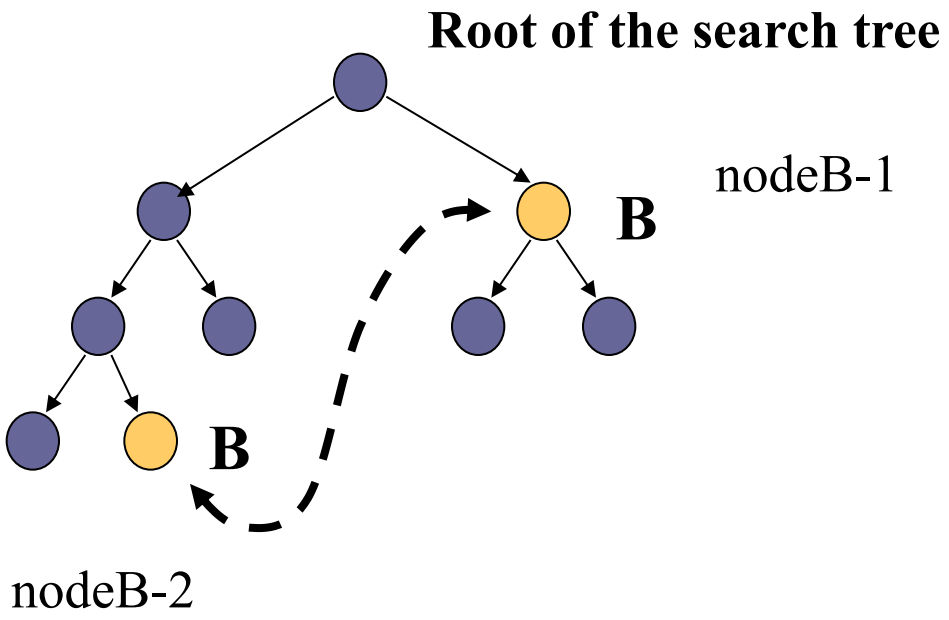
Elimination of non-cyclic state repeats



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

Question: Is one of the two nodes: nodeB-1, or nodeB-2 better and preferable?

Elimination of non-cyclic state repeats

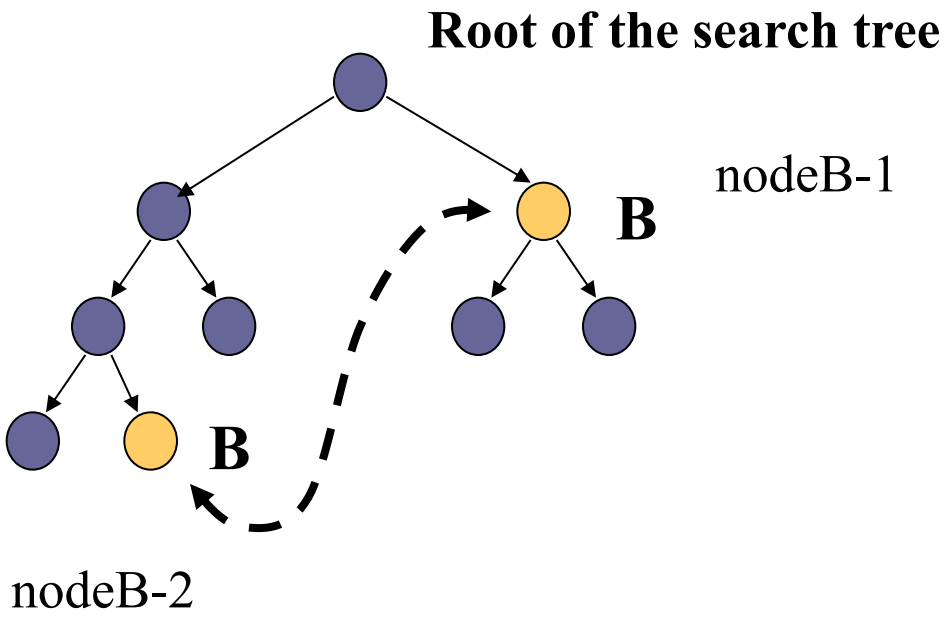


Case B: nodes with the same state are not on the same path from the initial state

Question: Is one of the two nodes: nodeB-1, or nodeB-2 better and preferable?

Yes. nodeB-1 represents a shorter path from the initial state to B

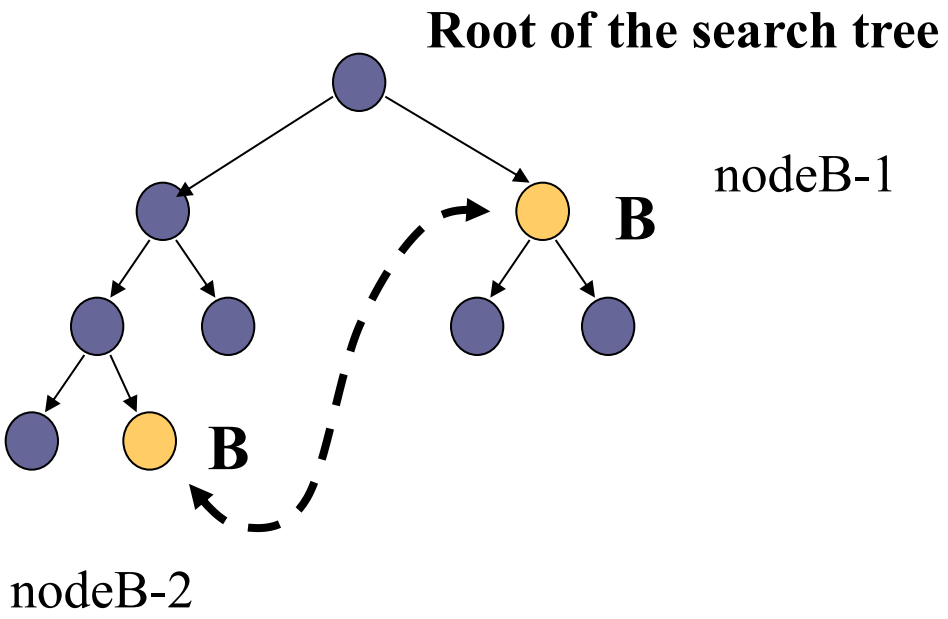
Elimination of non-cyclic state repeats



Conclusion: **Since we are happy with the optimal solution nodeB-2 can be safely 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)**

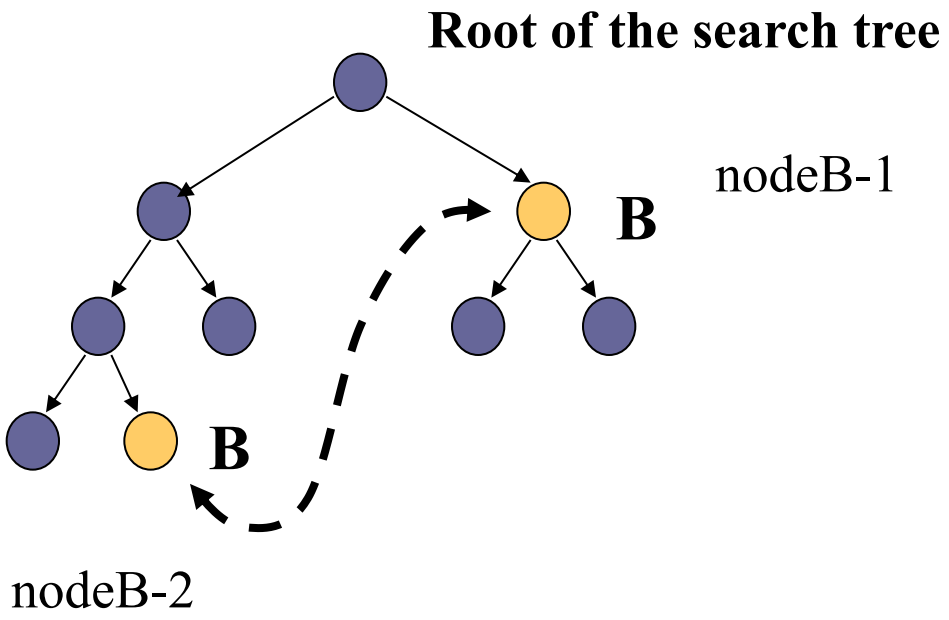
- we can safely eliminate all second, third, fourth, etc. occurrences of the same state
- this rule covers both cyclic and non-cyclic repeats !!!

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

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

Use hash table to implement marking

Elimination of non-cyclic state repeats with DFS



Depth FS: nodeB-2 can be 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: hash table 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.)