

# COMP219: Artificial Intelligence

## Overview

### Lecture 8: Combining Search Strategies and Speeding Up

- Last time
  - Basic problem solving techniques:
    - Breadth-first search
      - complete but expensive
    - Depth-first search
      - cheap but incomplete
- Today
  - Variations and combinations
    - Limited depth search
    - Iterative deepening search
  - Speeding up techniques
    - Avoiding repetitive states
    - Bi-directional search
- Learning outcome covered today:  
Identify, contrast and apply to simple examples the major search techniques that have been developed for problem-solving in AI

1

## Depth Limited Search



- Depth first search has some desirable properties:  
space complexity
- But if wrong branch expanded (with no solution on it),  
then it may not terminate
- Idea: introduce a **depth limit** on branches to be expanded
- Don't expand a branch below this depth
- Most useful if you know the maximum depth of the  
solution

2

## Depth Limited Search

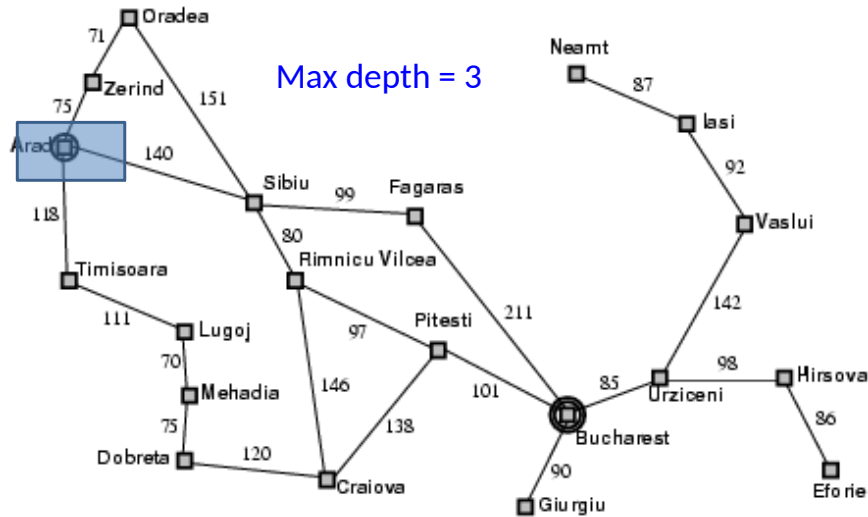
```
depth limit = max depth to search to;
agenda = [initial state];
if initial state is goal state then
    return solution
else
    while agenda not empty do
        take node from front of agenda;
        if depth(node) < depth limit then
            {
                new nodes = apply operations to node;
                add new nodes to front of agenda;
                if goal state in new nodes then
                    return solution;
            }
}
```

3

4

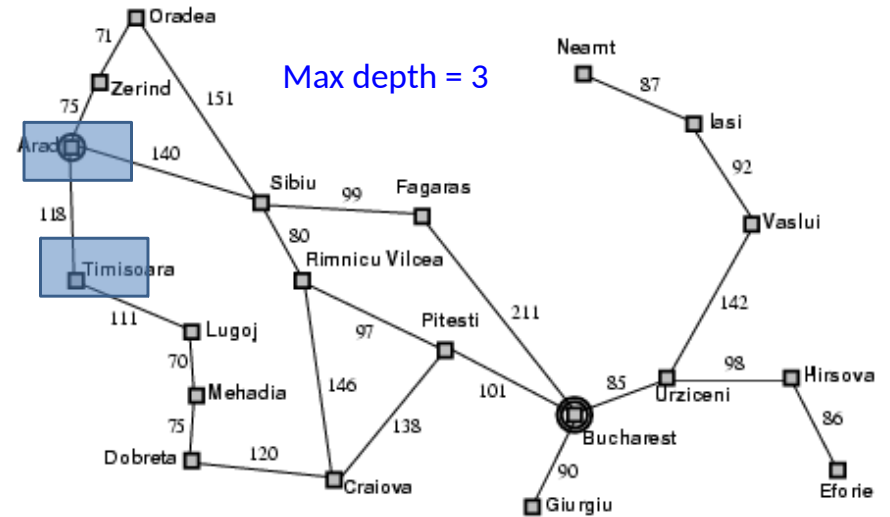
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



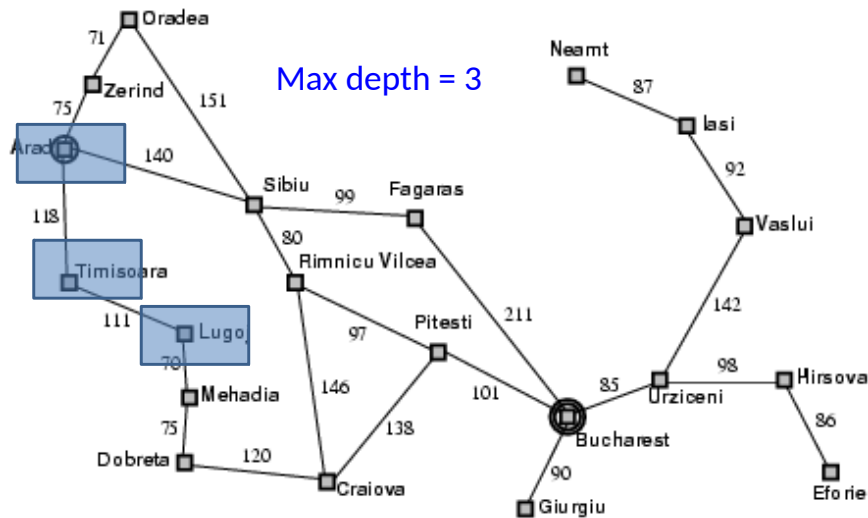
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



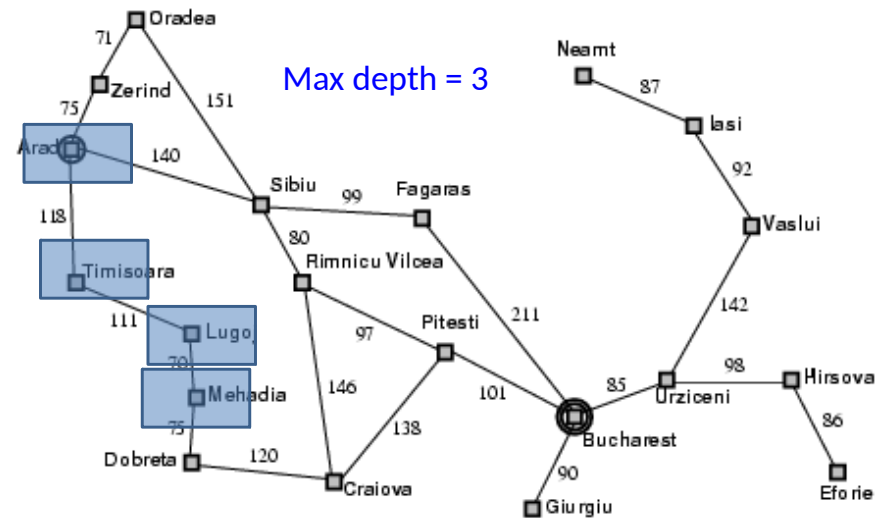
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



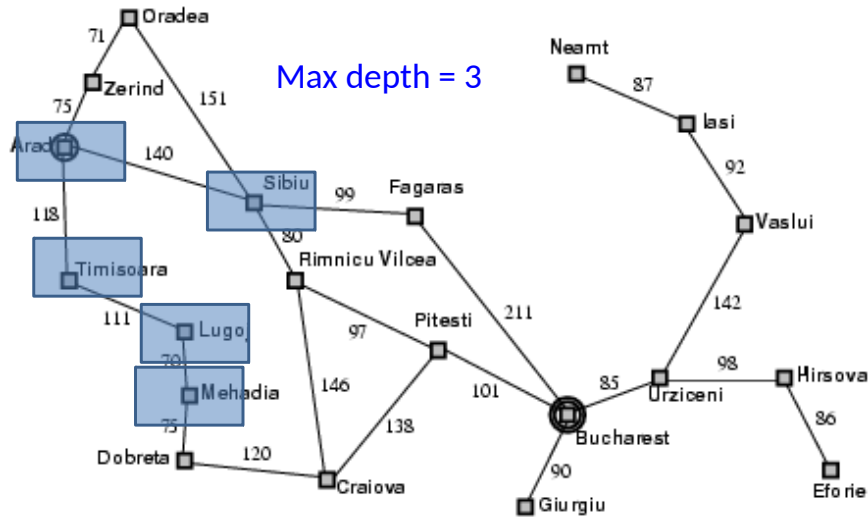
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



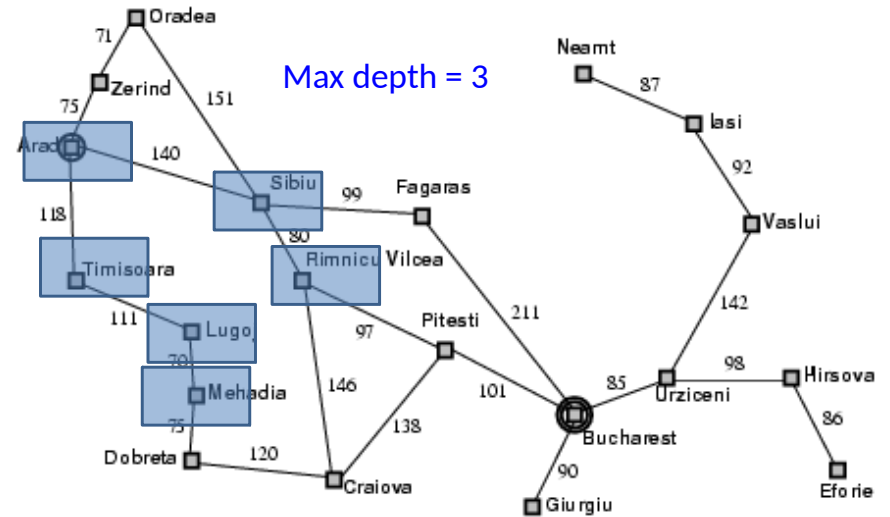
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



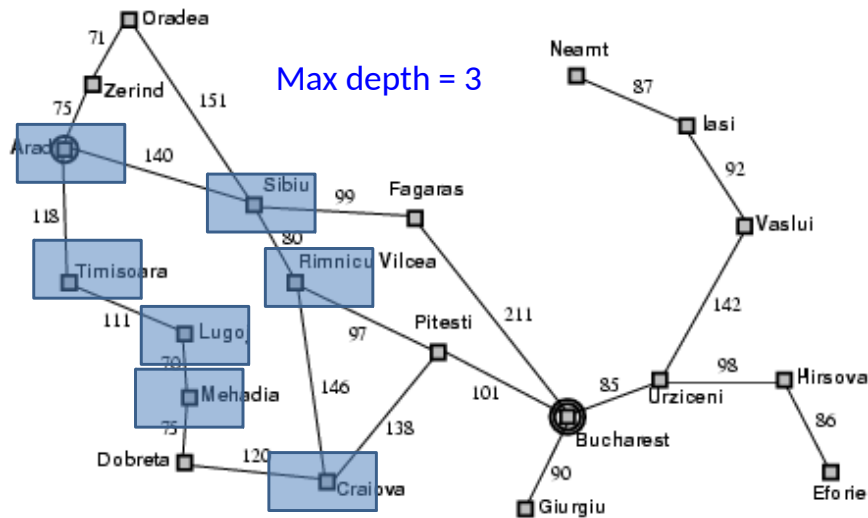
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



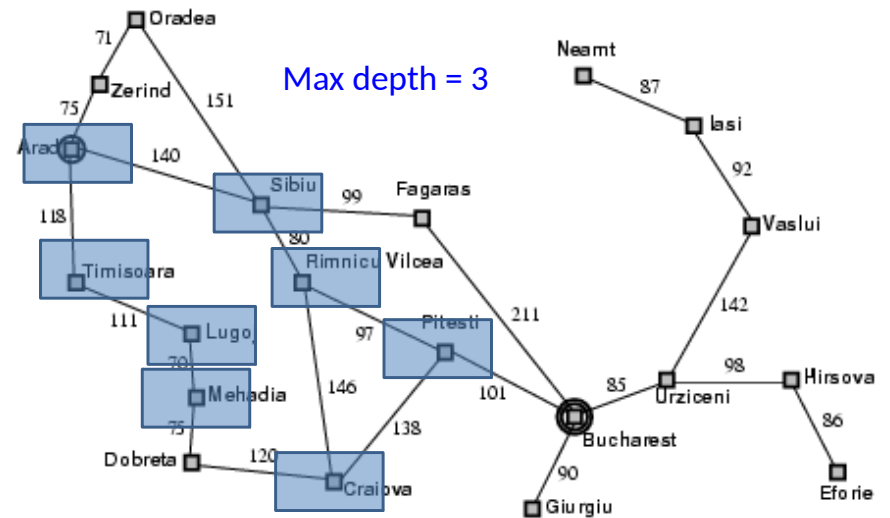
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



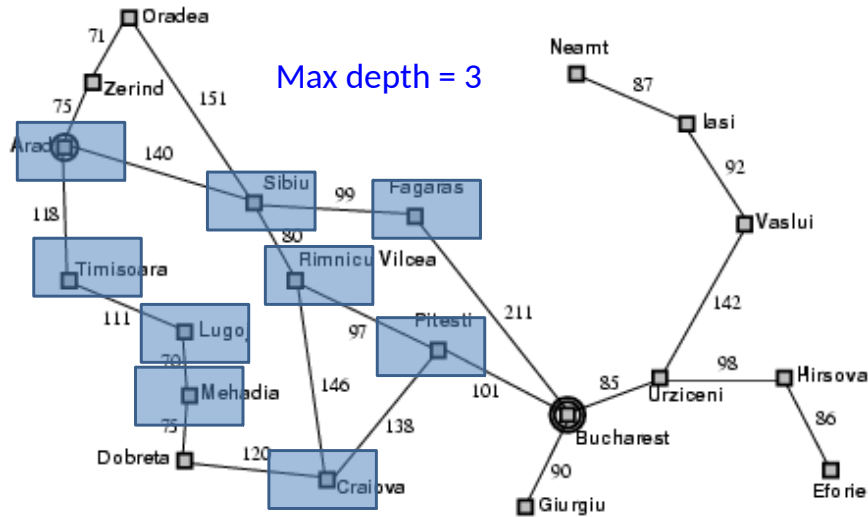
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
In fact, any city can reach any other in at most 9 steps.



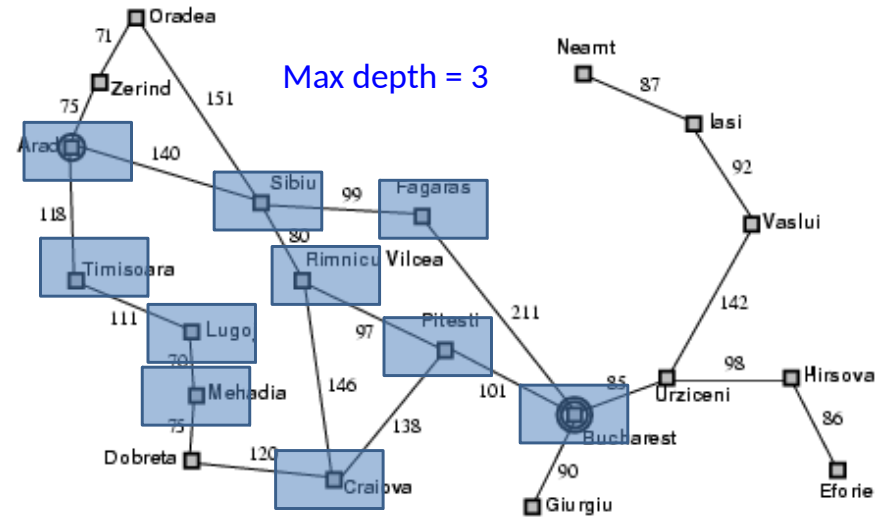
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
 In fact, any city can reach any other in at most 9 steps.



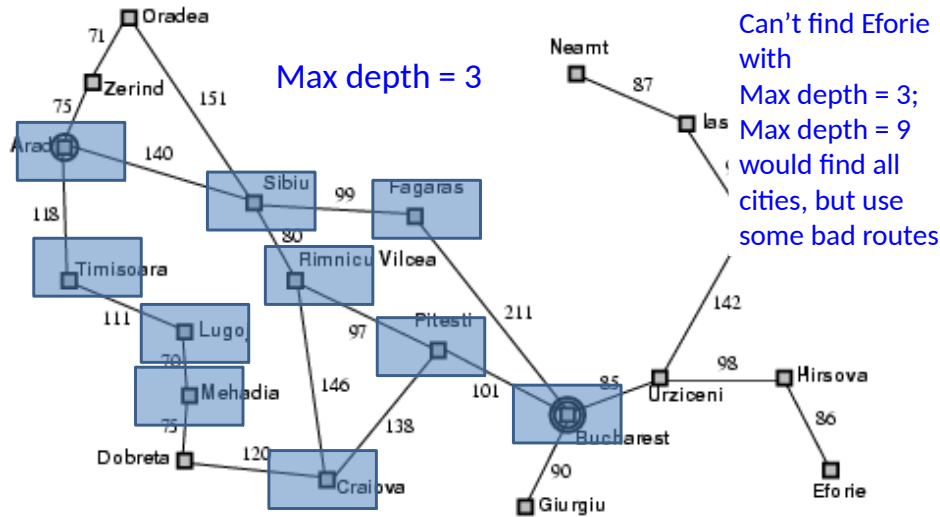
# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
 In fact, any city can reach any other in at most 9 steps.



# Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.  
 In fact, any city can reach any other in at most 9 steps.



# Depth Limited Search

- Will always terminate.
- Will find solution if there is one in the depth bound.
- But, Goldilocks principle:
  - Too small a depth bound misses solutions ('incomplete')
  - Too large a depth bound may find poor solutions when there are better ones.

# Iterative Deepening

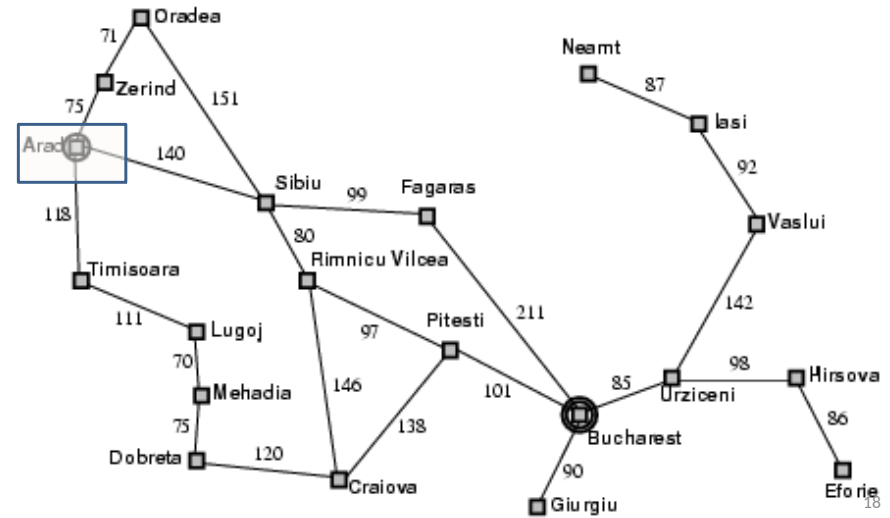


- Iterative deepening
  - addresses problem of choosing depth bound
  - is complete and finds best solution.
- Basic idea is:
  - do d.l.s. for depth  $n = 0$ ; if solution found, return it;
  - otherwise do d.l.s. for depth  $n = n + 1$ ; if solution found, return it, etc;
  - So we repeat d.l.s. for all depths until solution found.
- Useful if the search space is large and the maximum depth of the solution is not known.

17

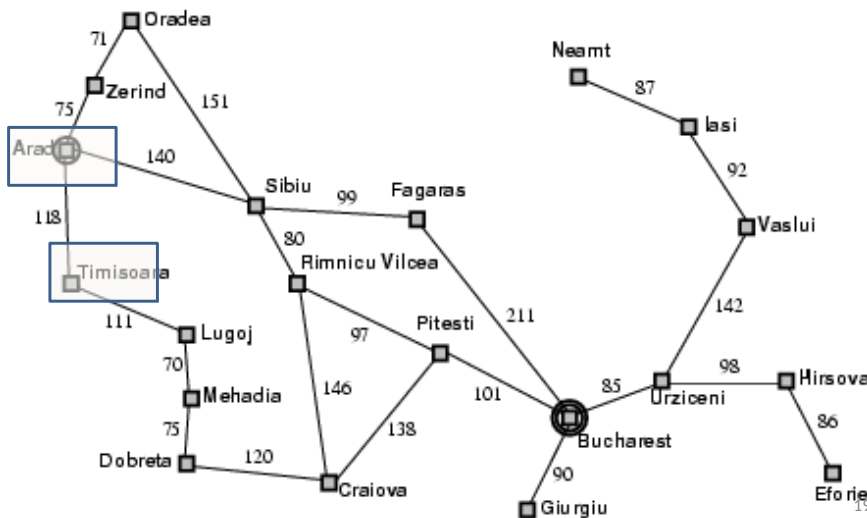
# Example: Romania Problem

D=1



# Example: Romania Problem

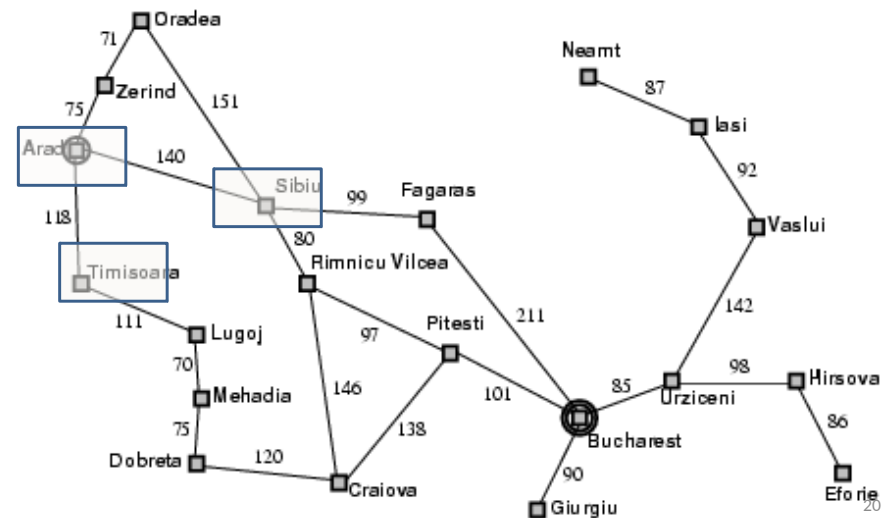
D=1



19

# Example: Romania Problem

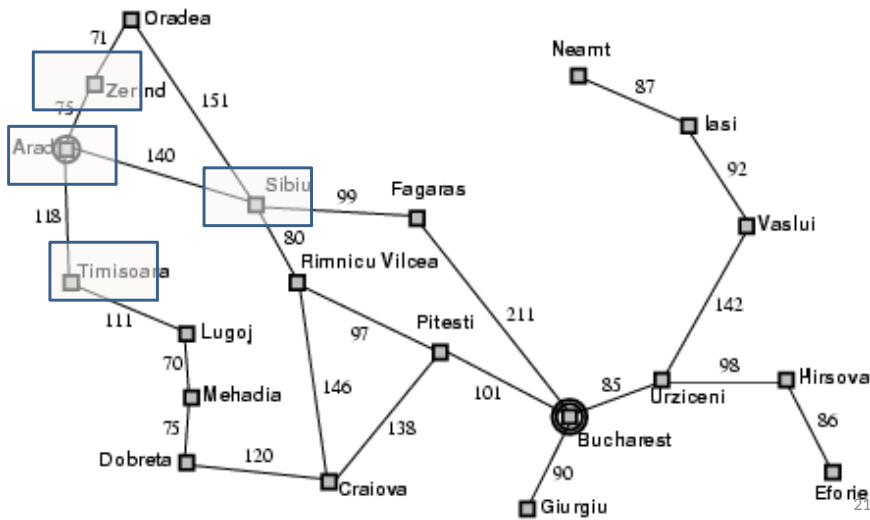
D=1



20

# Example: Romania Problem

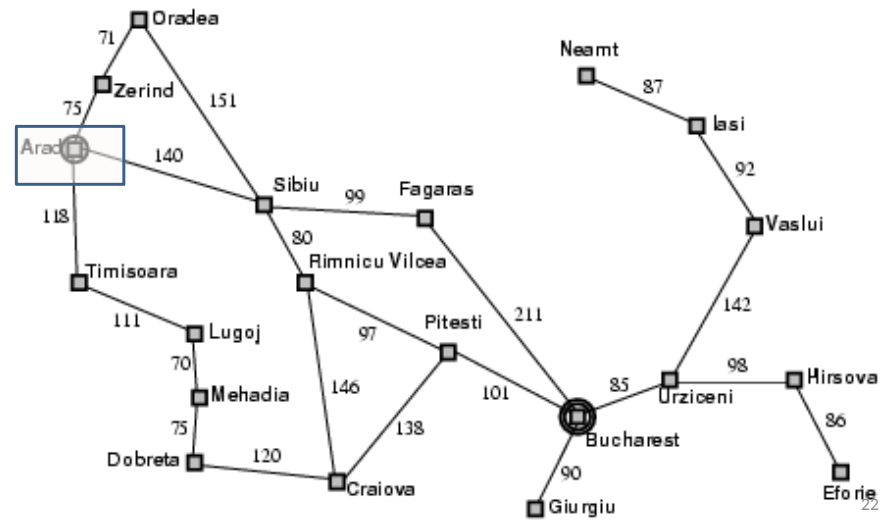
D = 1



# Example: Romania Problem

D = 1

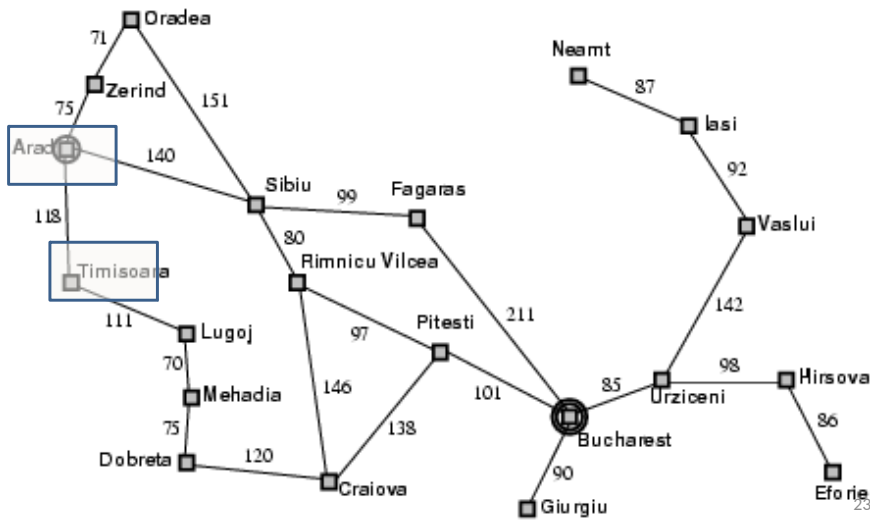
D = 2



# Example: Romania Problem

D = 1

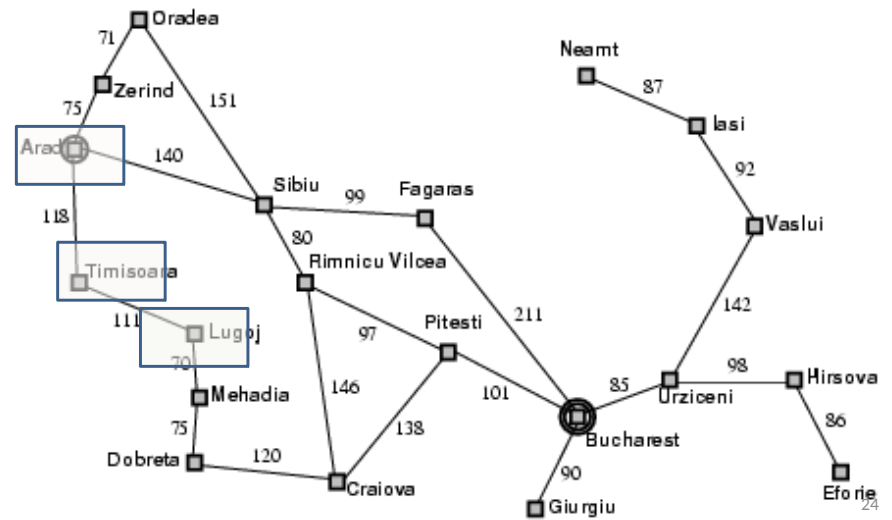
D = 2



# Example: Romania Problem

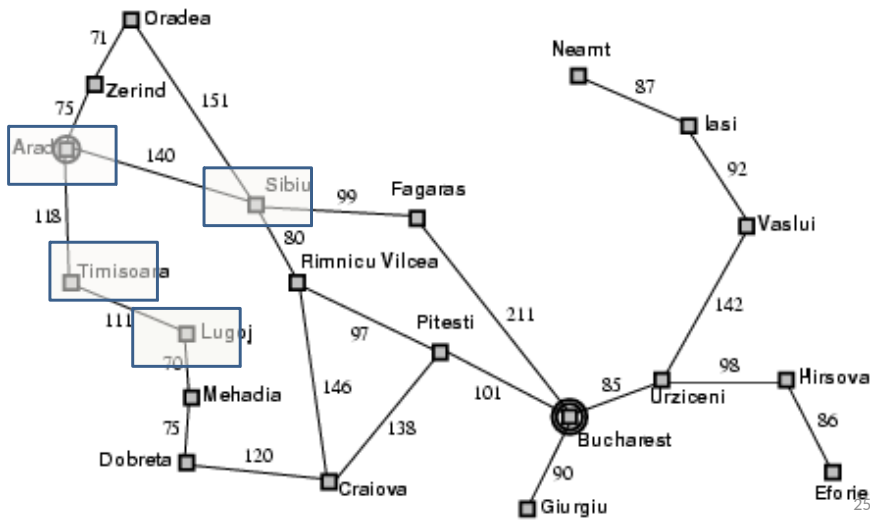
D = 1

D = 2



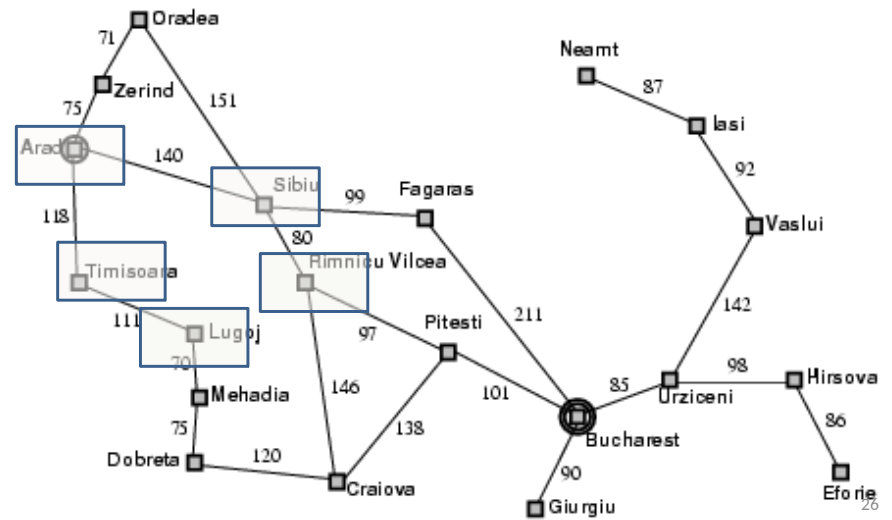
# Example: Romania Problem

D=1 D=2



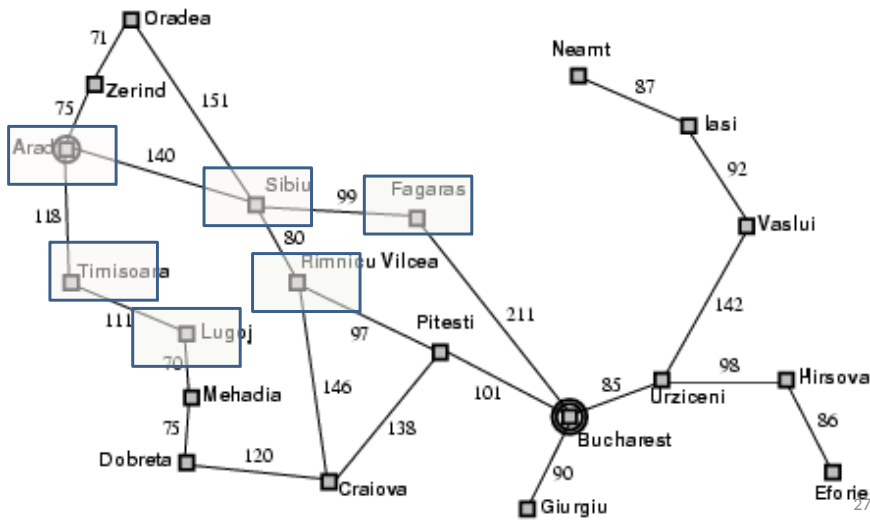
# Example: Romania Problem

D=1 D=2



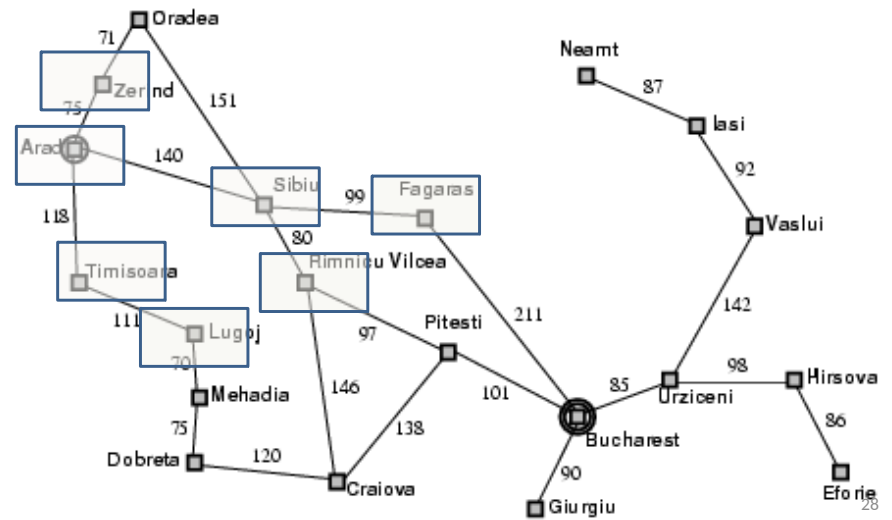
# Example: Romania Problem

D=1 D=2



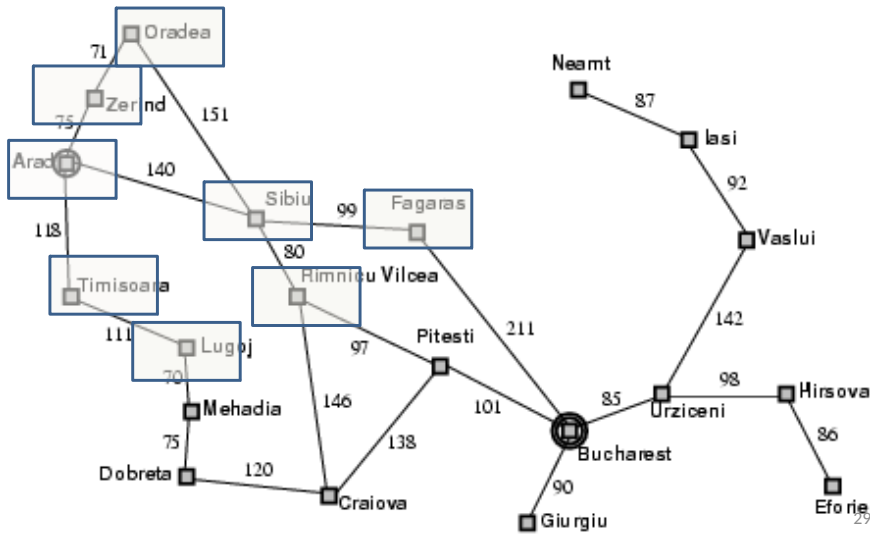
# Example: Romania Problem

D=1 D=2



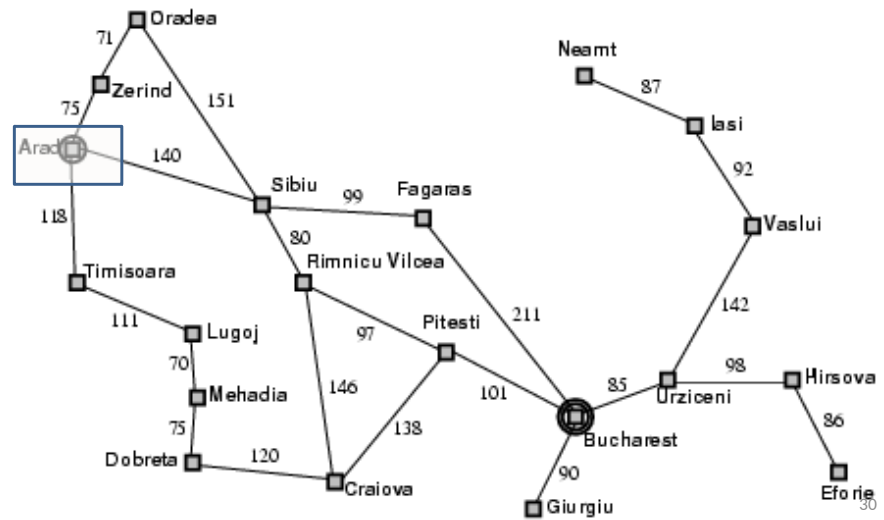
# Example: Romania Problem

D=1   D=2



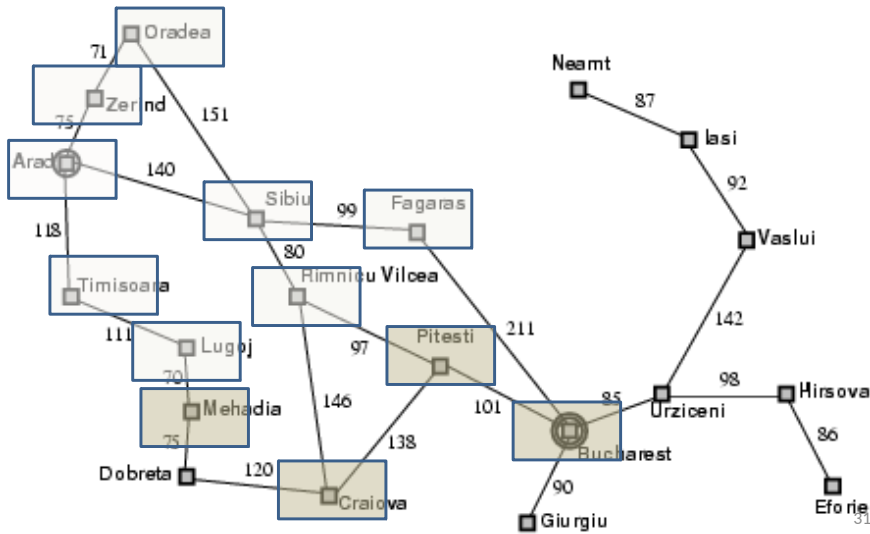
# Example: Romania Problem

D=1   D=2   D=3



# Example: Romania Problem

D=1   D=2   D=3



# General Algorithm for Iterative Deepening

```

depth_limit = 0;
while(true)      /* infinite loop */
{
    result = depth_limited_search(
                max_depth = depth limit,
                agenda = initial node);
    if result contains goal then
        return result;
    depth limit = depth limit + 1;
}
    
```

- Calls d.l.s. as subroutine.



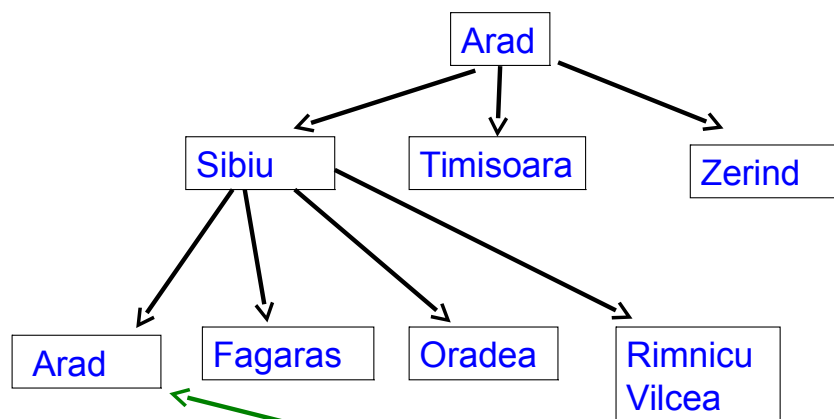
# IDS Properties

- Note that in iterative deepening, we re-generate nodes *on the fly*.
- Each time we do a call on depth limited search for depth  $d$ , we need to **regenerate the tree** to depth  $d - 1$ .
- Trade off **time** for **memory**.
- In general we might take a little more time, but we save a lot of memory.
  - Example: Suppose  $b = 10$  and  $d = 5$ .
  - Breadth first search would require examining 111,110 nodes, with memory requirement of 100,000 nodes.
  - Iterative deepening for same problem: 123,450 nodes to be searched, with memory requirement of only 50 nodes.
  - Takes **11%** longer in this case, but savings on memory are immense.

33

## Techniques for Speeding Up

### Repeated States - The Search Tree



Blind search may **repeat** nodes; if the search path contains cycles we may get into an infinite loop when doing depth first search

35

### Avoiding Repeated States



- There are three ways to deal with this (in order of increasing effectiveness **and** computational overhead):
  - do not return to the state you have just come from
  - do not create paths with cycles in them
  - do not generate any state that was ever generated before
- Note there is a **trade-off** between the **cost** of extra **search** and the **cost** of **checking** for repeated states

36

# The Impact of Branching



- In analyses branching is often assumed to be uniform
- But in practice this is often not so
- This can make a big difference to the search space

37

# Goal vs Data driven search



- We can choose to search from
  - the initial state to the goal (*data driven, forward*)
  - the goal to the initial state (*goal driven, backward*)
- The branching may be *very* different...
- *Goal* driven search is very often very much more efficient (*few paths reach the goal*)
- Often used in expert systems (*and Prolog*)

38

## Example - Blocks World

- Consider the blocks world:  
Blocks are laid out on a table and a robot can move any *clear* block to another *clear* block. Suppose the initial state and goal state are as follows:

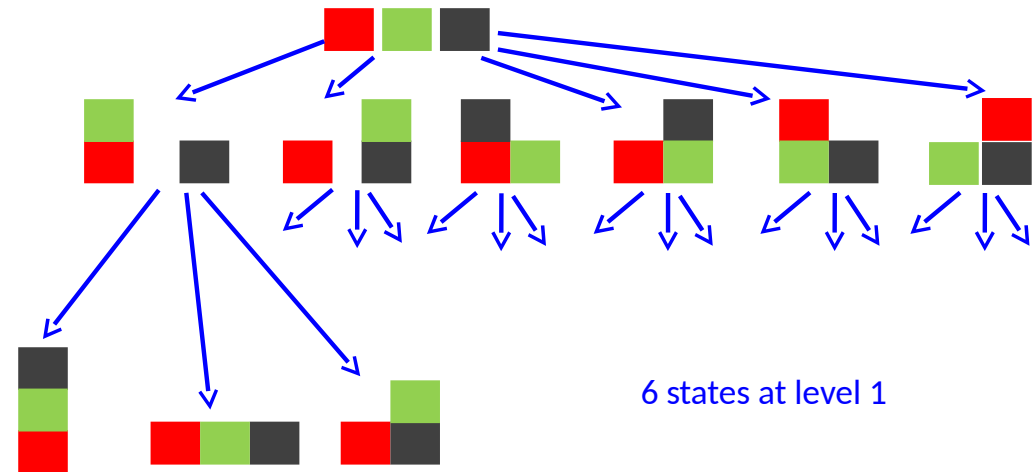
Initial



Goal



## Forward Search Space: Initial → Goal



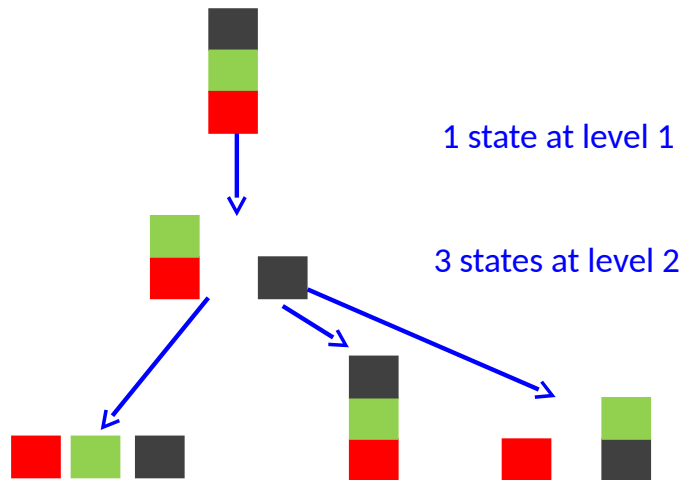
6 states at level 1

18 states at level 2

39

40

## Backward Search Space: Goal → Initial



41



## Bi-directional Search

- If we are unsure of the branching factor, then searching from both ends may be best

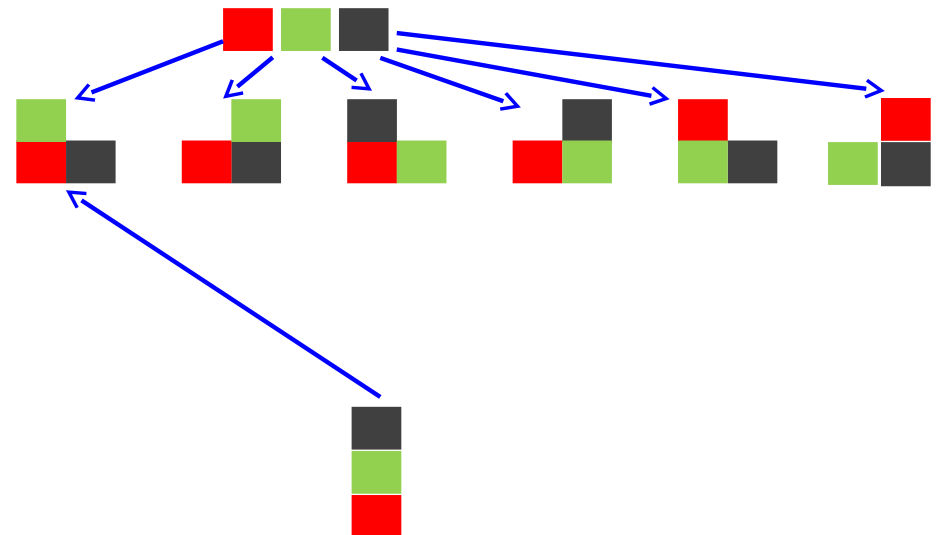
43

## About Backward Search

- Backward (goal driven) search can be more effective...  
...but may not always be applicable!
- need to be able to
  - generate predecessors (can be difficult, there could be many)
  - effectively describe the goal (“no queen attacks another queen”...?)

42

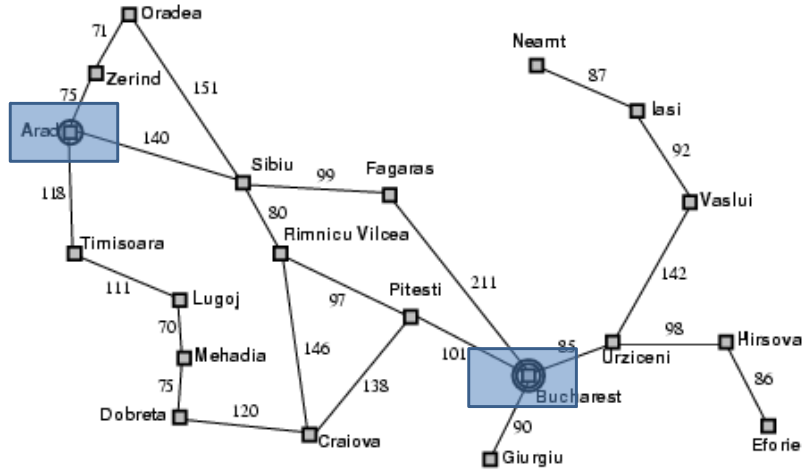
## Bidirectional Search



44

# Example: Romania

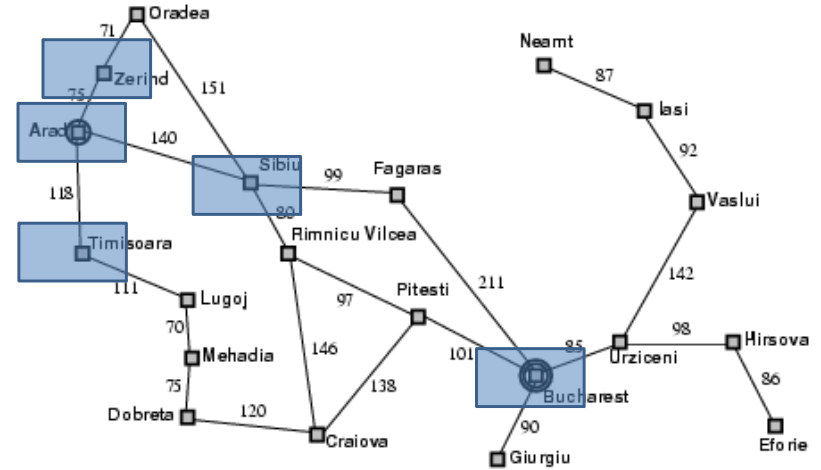
- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest



45

# Example: Romania

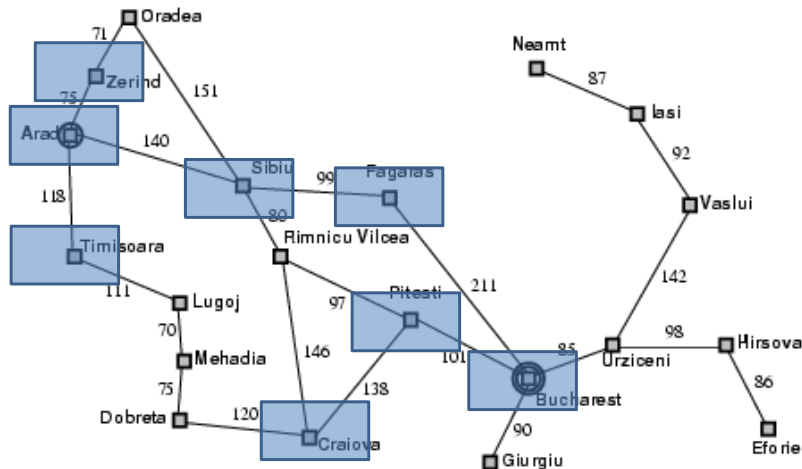
- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest



46

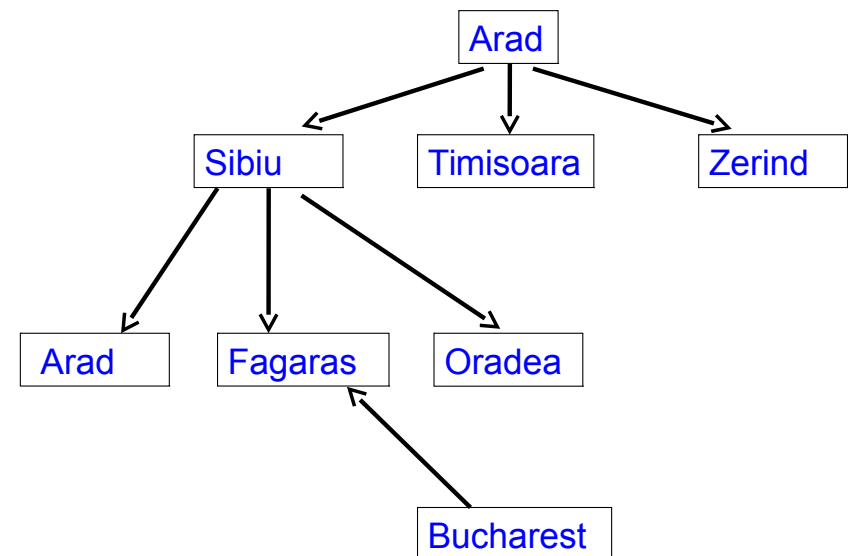
# Example: Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest



47

# Bi-directional Search



48

## Bi-directional Search: Good



- Much more **efficient**
- Rather than doing **one** search of  $b^d$  ...  
...we do **two**  $b^{d/2}$  searches
  - E.g., Suppose  $b = 10$ ,  $d = 6$ 
    - Breadth first search will examine  $10^6 = 1,000,000$  nodes
    - Bidirectional search will examine  $2 \times 10^3 = 2,000$  nodes
- Can combine different search strategies in different directions

49

## Bi-directional Search: Bad



- Depends on applicability of backward search
- Needs an efficient way to check whether each new node appears in the other search:
  - need to store nodes in frontier, so large memory requirements
  - For example, for
    - two bi-directional breadth-first searches,
    - branching factor  $b$ ,
    - depth of the solution  $d$ ,  
→ memory requirement of  $b^{d/2}$  for each search
  - For large  $d$ , is still impractical

50

## Summary

- More advanced problem-solving techniques:
  - Depth-limited search
  - Iterative deepening
  - Bi-directional search
  - Avoiding repeated states
- These improve on basic techniques like breadth-first and depth-first search
- However, they still aren't always powerful enough to give solutions for realistic problems
- Are there more improvements we can make...?
- **Next time**
  - Lists in Prolog

51