SEARCH

Overview

Aims of the this lecture:

- Introduce *problem solving*;
- Introduce *goal formulation*;
- Show how problems can be stated as *state space search*;
- Show the importance and role of *abstraction*;
- Introduce *undirected* and *heuristic* search:
  - breadth first, depth first search;
  - best first search, A*
- Define main performance measures for search.

Problem Solving Agents

- Lecture 1 introduced *rational agents* but didn't say much abot how we might construct them.
- Today we make a start on understanding how to do this.
- Consider agents as *problem solvers*:

  Systems that have *goals* and find *sequences of actions* that achieve these goals.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
   static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

   state ← UPDATE-STATE(state, percept)
   if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
   action ← RECOMMENDATION(seq, state)
   seq ← REMAINDER(seq, state)
   return action
```

- Key difficulties:
  - FORMULATE-GOAL(...)
  - FORMULATE-PROBLEM(...)
  - SEARCH(...)—
- It isn't easy to see how to tackle any of these.
- Here we will concentrate mainly on search but first we'll say a bit about goal formulation and problem formulation.

Goal Formulation

- Where do an agent's goals come from?
  - Agent is a *program* with a *specification*.
  - Specification is to maximise performance measure.
  - Should *adopt goal* if achievement of that goal will maximise this measure.
- But what does that mean in practice?

- As the book suggests, let’s imagine we (or any other agent) are in Arad, Romania:

- On a given day, we might do a number of things:
  - get a suntan;
  - go sightseeing;
  - improve our spoken Romanian;
  - enjoy the nightlife;
  - avoid a hangover; and so on
- But if we have a non-refundable ticket for a flight from Bucharest the next day, then we can eliminate most of these options, and adopt the goal of getting to Bucharest.
- Anything else will clearly have a lower value.

- Goals provide a *focus* and *filter* for decision-making:
  - *focus*: need to consider how to achieve them;
  - *filter*: need not consider actions that are incompatible with goals.
- Both of these help computationally.

Problem Formulation

- What is a problem?
- Formal definition is that a problem contains 5 components:
  - Initial state;
  - Actions;
  - Transition model;
  - Goal test; and
  - Path cost.
- Let's look at each of these in detail.

Initial state

- The state that the agent starts in.
- In the Romania example the initial state might be described as:

$$In(Arad)$$

- We could obviously include a lot more detail:

$$\begin{array}{r} In(Arad) \\ Temperature(high) \\ Suntan(acceptable) \\ Romanian(rudimentary) \end{array}$$

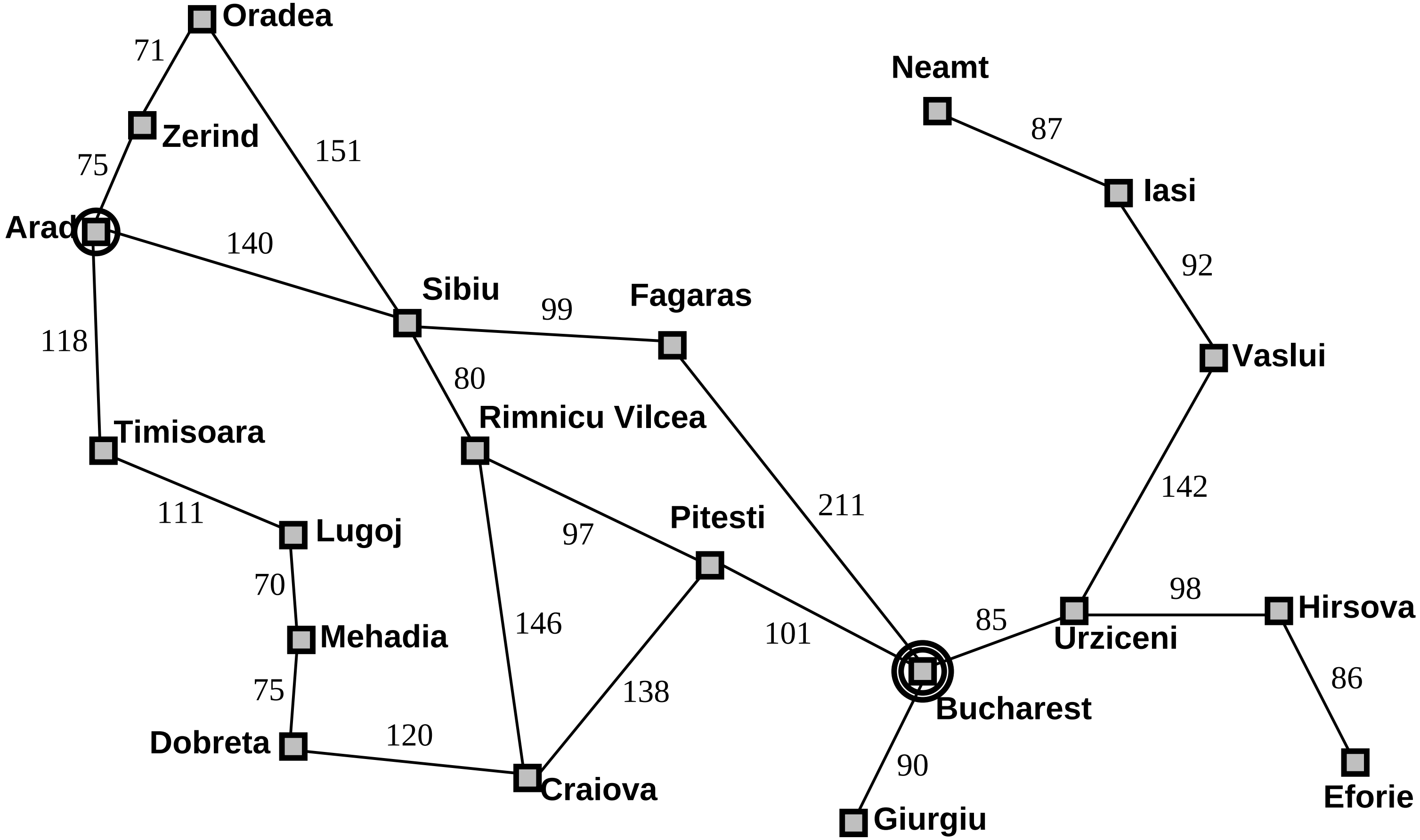
  and finding the corrected level of *abstraction* is important.
- Too much detail and (as we will see) the problem can be intractable.

Actions

- The actions that the agent can perform.
- These tend to be dependent on what state the agent is in.
- Given a particular state s, $\textsc{Actions}(s)$ is the set of actions that are *applicable*.
- In the Romania example, in the state $In(Arad)$, the relevant actions are:

$$\{Go(Sibiu), Go(Timosoara), Go(Zerind)\}$$

- Again, abstraction is important.


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
Urziceni
98
Hirsova
85
Bucharest
90
Giurgiu
86
Eforie

Transition model

- The transition model describes what each action does.
- Formally we have a function $\textsc{Result}(s, a)$ which defines the state the agent gets to when it executes action a in state s.

  We will call the state we get to a *successor state*.
- In the Romania example:

$$\textsc{Result}(In(Arad), Go(Zerind)) = In(Zerind)$$

- For now we will deal with deterministic environments, so that a state only has a single successor.

- The combination of initial state, actions, and transitions define what we call the *state space*.
- This is the set of all states that we can get to from the intitial state.
- The state space can be pictured as a directed graph in which nodes are states and links are actions.
- In the Romania example, the map can be thought of as a picture of the state space.
- A *path* in a state space is a sequence of actions and states.
- A path through the state space from initial state to goal state is a *plan* to get to the goal.

Goal test

- Determines whether a given state is the goal state.
- In the Romania example:

$$\{In(Bucharest)\}$$

is the goal.

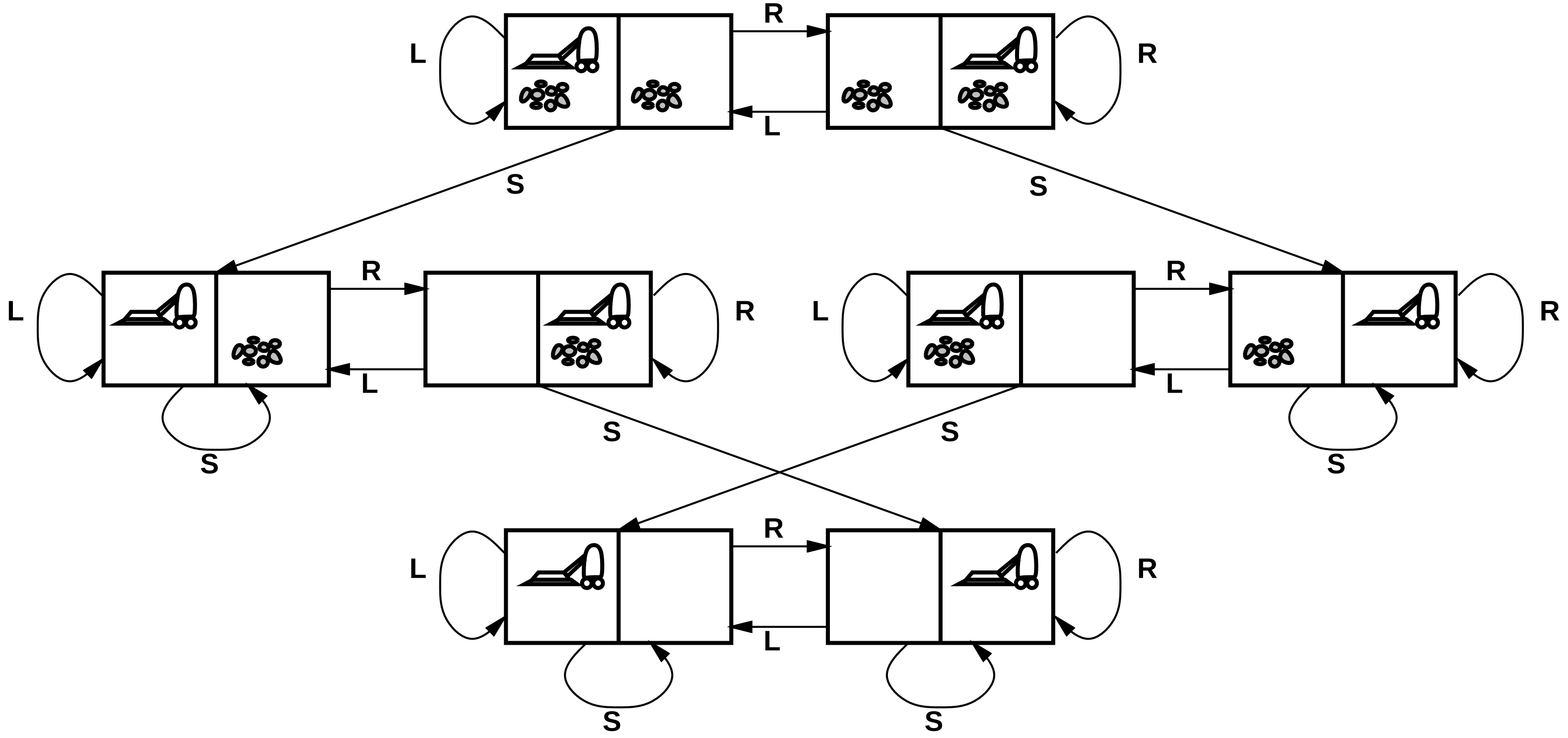
Path cost

- Function that assigns a numeric cost to each path.
- What we use as a path cost depends on the problem we are solving.
- In the Romania example it makes sense to use distance as a cost function since the agent is in a hurry.
- A more leisurely agent might want to use the price of taking the bus on each leg as the cost function.
- We will often assume that the path cost can be computed as the sum of the costs along a path.
- The *step cost* of taking action a in state s to reach state s' is written as $c(s, a, s')$.

Problem

- Together these elements define a problem.
- A *solution* is an action sequence (plan) that leads from the initial state to the goal.
- The quality of a solution is measured by the path cost.
- The *optimal* solution is the one with the lowest path cost.

Example problem: Vacuum world



- States: There are two locations, each of which may contain dirt, and the agent can be in either.

  That leads to 8 possible states.

  We might consider any of these to be the initial state.
- Actions: *Left*, *Right*, *Suck*.
- Transition model: The actions work as their names suggest, except that *Left* and *Right* have no effect in (respectively) the leftmost and rightmost positions.

  Suck has no effect in a clean square.
- Goal test: Checks if both squares are clean.
- Path cost: Each step costs 1.

Example problem: 8 puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- States: Each state specifies the location of each tile and the blank. Any of these can be the initial state.
- Actions: Simplest way to specify actions is to say what happens to the blank — *Left*, *Right*, *Up* and *Down*.

  Not all of these will b applicable in all locations of the blank.
- Transition model: Gives the resulting state of each action. For example *Left* in the initial state above switches the 5 and the blank.
- Goal test: Checks if the goal configuration has been reached.
- Path cost: Each step costs 1.

Problem Solving as Search

- As with the Romania example, we can think of the state-space of a problem as a graph.
- Systematically generate a *search tree*
- The tree is built by taking the initial state and identifying some states that can be obtained by applying a single operator.
- These new states become the *children* of the initial state in the tree.
- These new states are then examined to see if they are the goal state.
- If not, the process is repeated on the new states.
- We can formalise this description by giving an algorithm for it.

function TREE-SEARCH(*problem*, *strategy*) **returns** a solution, or failure

initialize the search tree using the initial state of *problem*

loop do

if there are no candidates for expansion **then return** failure

choose a leaf node for expansion according to *strategy*

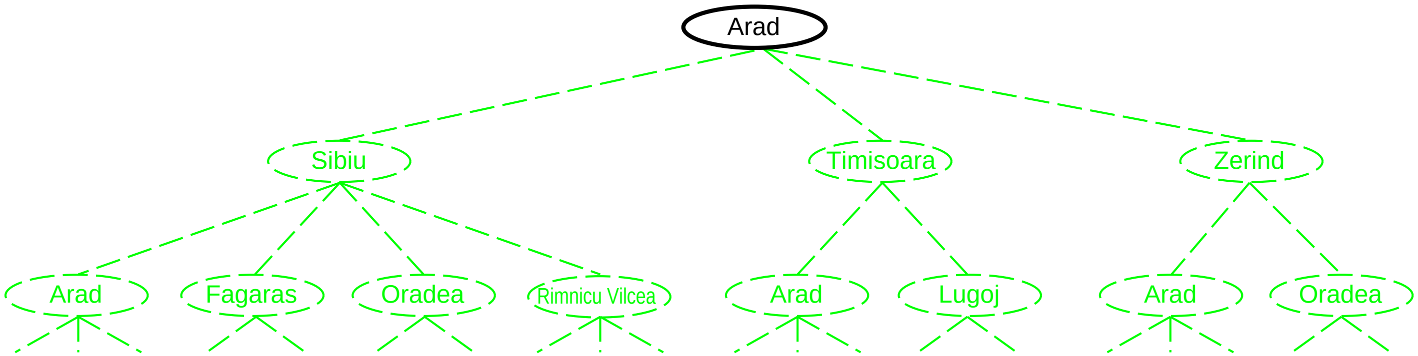
if the node contains a goal state **then return** the corresponding solution

else expand the node and add the resulting nodes to the search tree

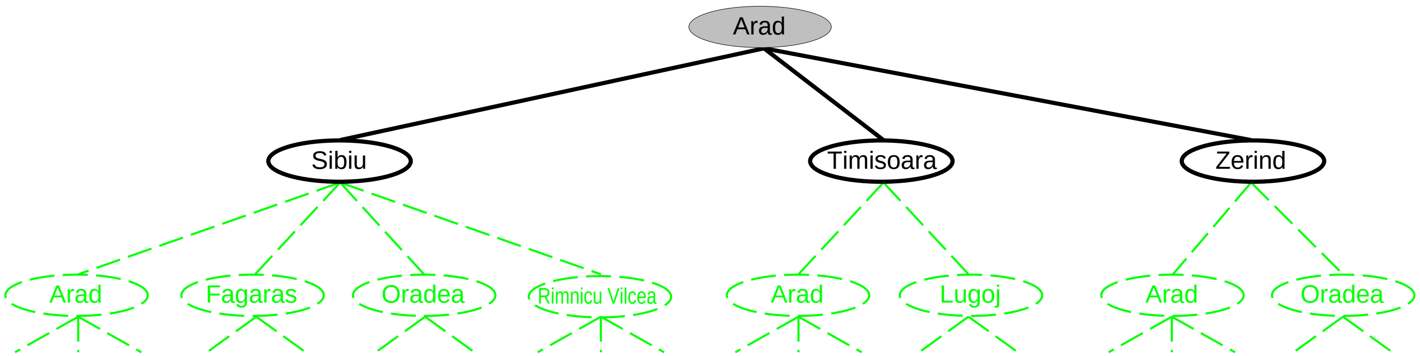
end

- Note that we call “candidates for expansion” both *fringe* and *frontier*.

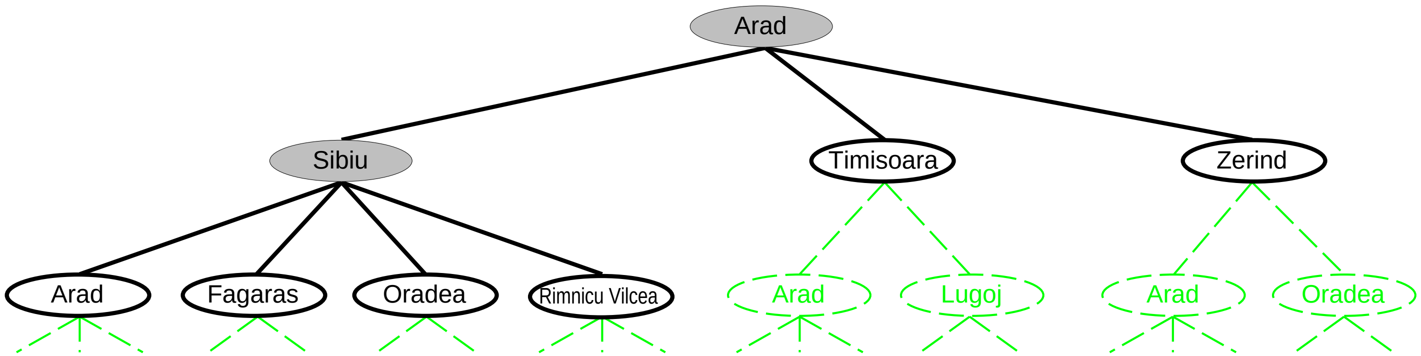
- Initial state



- Successor states of the initial state.



- Successors of the sucessors



- Note how Arad reappears

- Note the difference between *state space* and *search tree*.
- State space is every possible state and the relationships between them.
  - It is static.
- Search tree the set of states the agent has looked at (is looking at) and some of the relationships between them.
  - It is dynamic.
- Now, about those states that pop up more than once.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
         add STATE[node] to closed
         fringe ← INSERTALL(EXPAND(node, problem), fringe)
   end
```

Search strategies

- Question: How to pick states for expansion?
- A range of possibilities:
  - Breadth-first
  - Depth-first
  - Iterative deepening
  - Best-first
  - A*

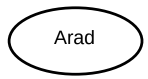
Breadth First Search

- Start by *expanding* initial state — gives tree of depth 1.
- Then expand *all* nodes that resulted from previous step — gives tree of depth 2.
- Then expand *all* nodes that resulted from previous step, and so on.
- Expand nodes at depth n before level $n + 1$.

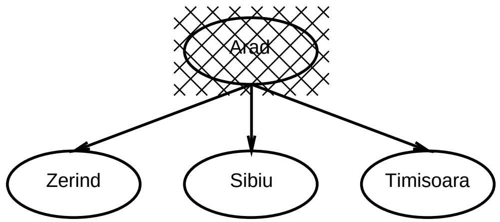
function BREADTH-FIRST-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

```
closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
        add STATE[node] to closed
        fringe ← ADDTOBACK(EXPAND(node, problem), fringe)
end
```

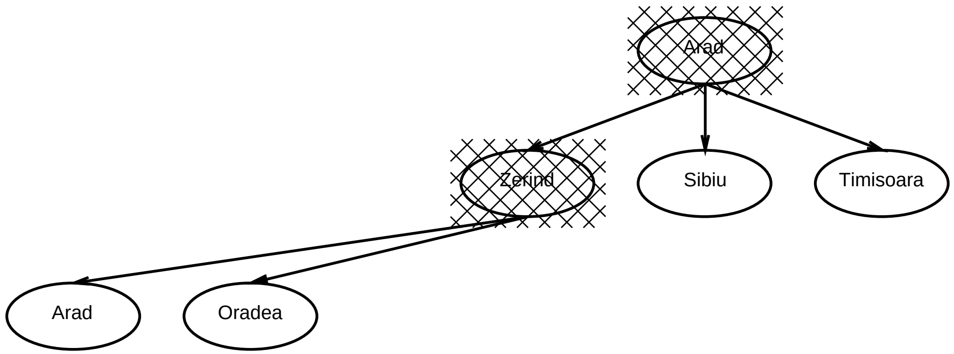
- Add the node representing the initial state into the fringe.



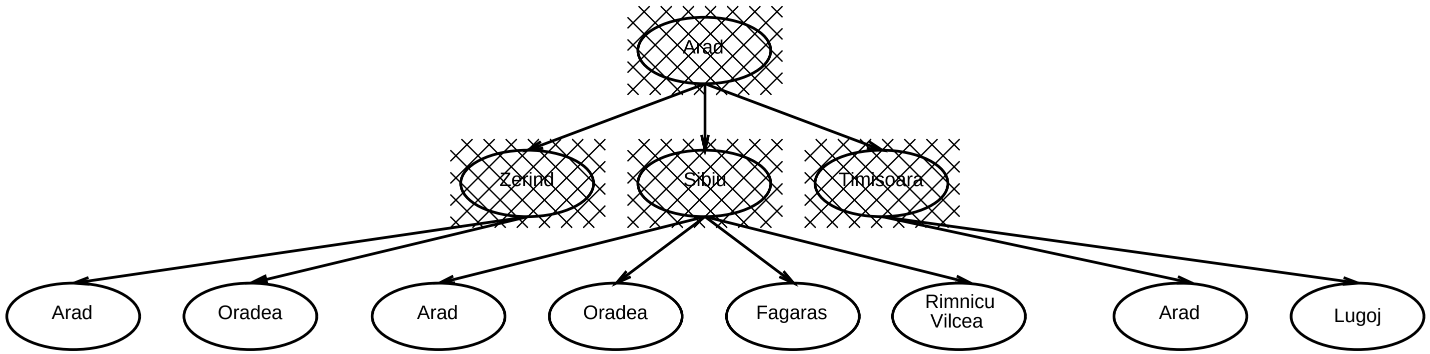
- Remove the first node in the fringe and add its children
- The queue is FIFO.



- Remove the first node in the fringe and add its children — they are added to the back of the queue.



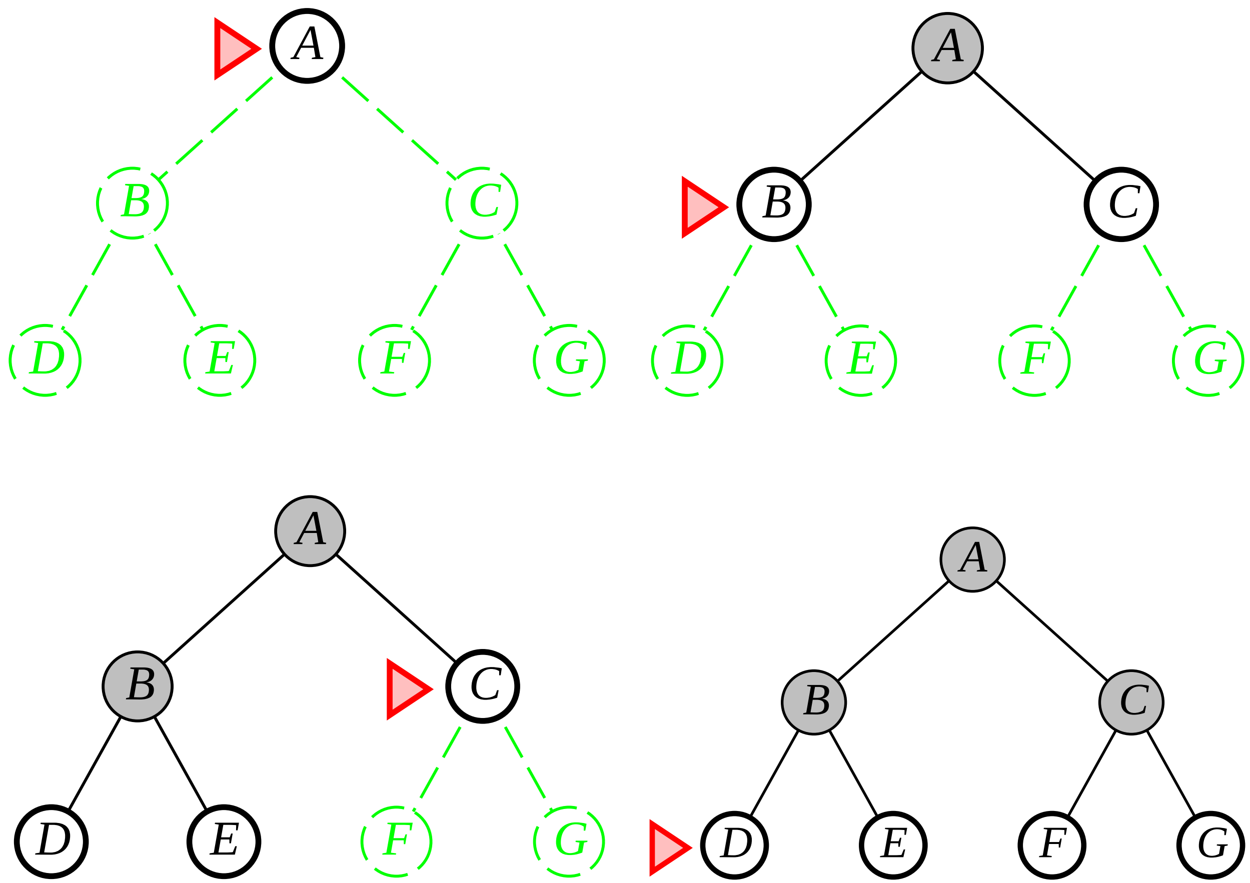
- Repeat twice more.



- Advantage: *guaranteed* to reach a solution if one exists.
- If all solutions occur at depth n, then this is good approach.
- Disadvantage: time taken to reach solution!
- Let b be *branching factor* — average number of operations that may be performed from any level.
- If solution occurs at depth d, then we will look at

$$1 + b + b^2 + \cdots + b^d$$

nodes before reaching solution — *exponential*.


A
B
C
D
E
F
G
A
B
C
D
E
F
G
A
B
C
D
E
F
G
A
B
C
D
E
F
G

- Time for breadth first search, $b = 10$, 1 million nodes per second, each node needs 1000 bytes of storage.

Depth	Nodes	Time	Memory
2	110	.11 msec	107 kilobytes
4	11,110	11 msecs	10.6 megabytes
6	10^6	1.1 secs	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
20	10^{20}	350 years	10 exabytes

- *Combinatorial explosion!*

Performance Measures for Search

- *Completeness*:
  Is the search technique *guaranteed* to find a solution if one exists?
- *Time complexity*:
  How many computations are required to find solution?
- *Space complexity*:
  How much memory space is required?
- *Optimality*:
  How good is a solution going to be w.r.t. the path cost function.

- Time and space complexity are measured in terms of:
  - b —maximum branching factor of the search tree.
  - d —depth of the least-cost solution.
  - m —maximum depth of the state space (may be ∞)

- How does breadth-first search measure up?

Uniform-cost search

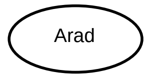
- Expand least-cost unexpanded node.
- We think of this as having an *evaluation function*:

$$g(n)$$

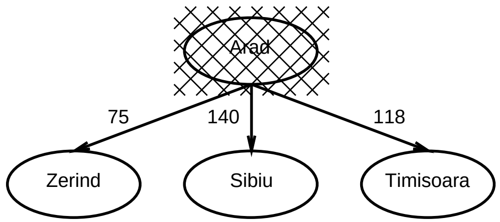
  which returns the path cost to a node n.
- *fringe* = queue ordered by evaluation function, lowest first
- Equivalent to breadth-first if step costs all equal
- Complete and optimal.
- Time and space complexity are as bad as for breadth-first search.

```
function UNIFORM-COST-SEARCH(problem, fringe) returns a solution, or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
         add STATE[node] to closed
         fringe ← INSERTALL(EXPAND(node, problem), fringe)
         fringe ← SORTBYGVALUE(fringe)
   end
```

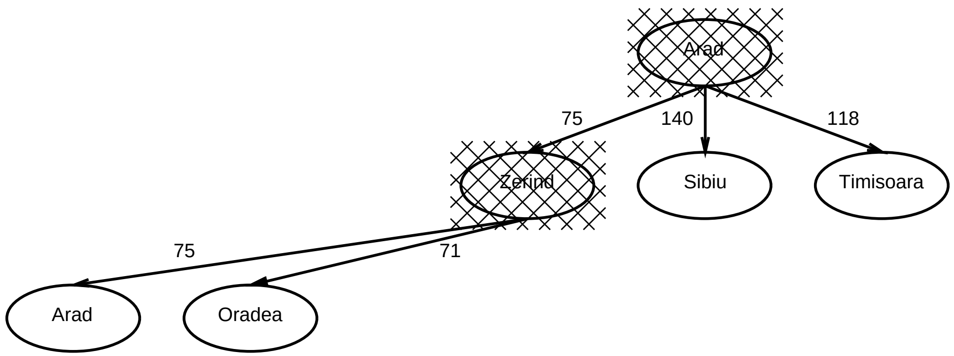
- Add the node representing the initial state into the fringe.



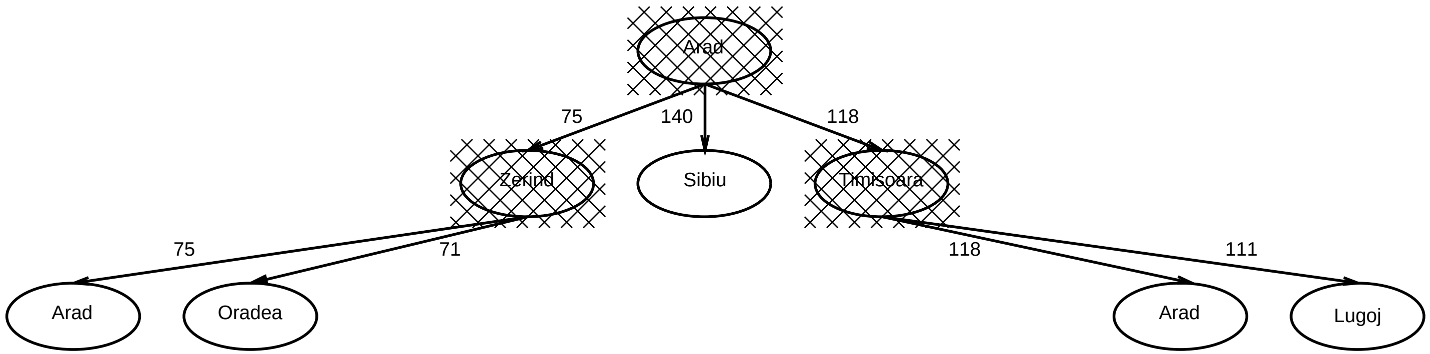
- Remove the first node in the fringe and add its children
- The queue is ordered with the cheapest first.



- Remove the first node in the fringe and add its children — they are added in priority order.



- Repeat.



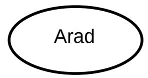
- What will be the next node to be expanded?

Depth First Search

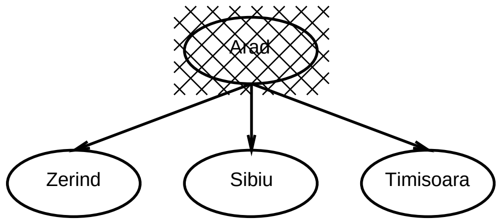
- Start by expanding initial state.
- Pick one of nodes resulting from 1st step, and expand it.
- Pick one of nodes resulting from 2nd step, and expand it, and so on.
- Always expand *deepest* node — make *fringe* a LIFO queue.
- Follow one “branch” of search tree.

```
function DEPTH-FIRST-SEARCH(problem, fringe) returns a solution, or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST(problem, STATE[node]) then return node
       if STATE[node] is not in closed then
           add STATE[node] to closed
           fringe ← ADDTOFRONT(EXPAND(node, problem), fringe)
   end
```

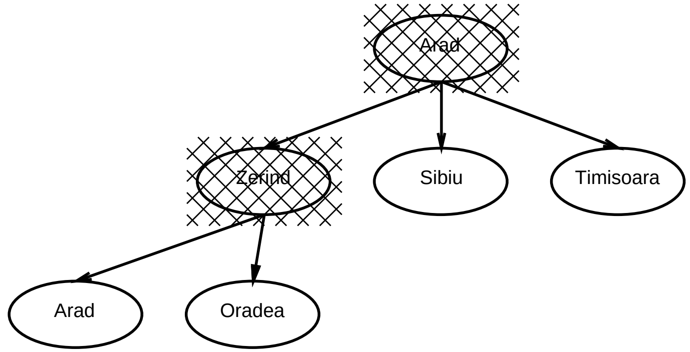
- Depth-first search on the Romania example — we start with the initial state in the frontier.



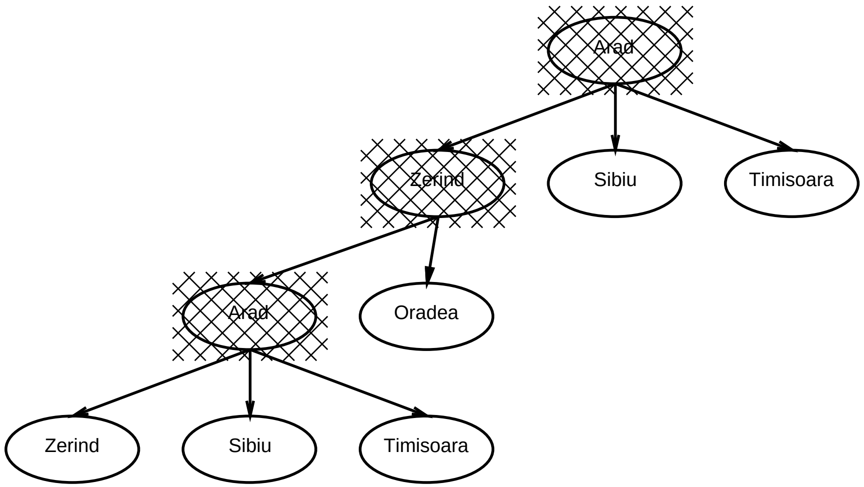
- Now we delete that node, and add its children.



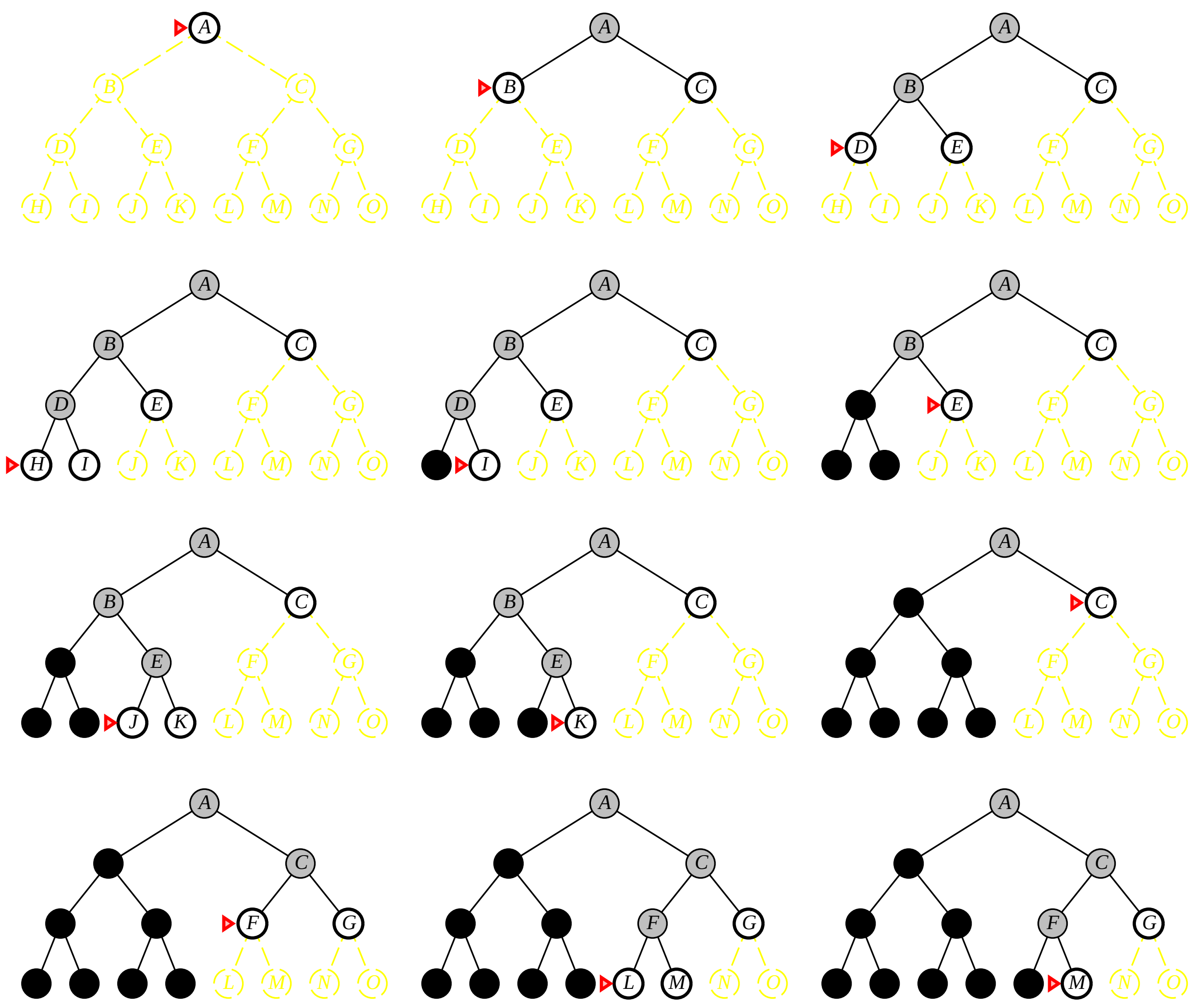
- Now pick a child and add its children



- and repeat.


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Zerind
Sibiu
Timisoara

- Depth first search is *not* guaranteed to find a solution if one exists.
- However, if it *does* find one, amount of time taken is much less than breadth first search.
- *Memory requirement* is much less than breadth first search.
- Solution found is *not* guaranteed to be the best.



Algorithmic Improvements

- Are then any *algorithmic* improvements we can make to basic search algorithms that will improve overall performance?
- Try to get *optimality* and *completeness* of breadth 1st search with *space efficiency* of depth 1st.
- Not too much to be done about time complexity :-(

Depth-limited Search

- Depth first search has some desirable properties — space complexity.
- But if wrong branch is expanded (with no solution on it), then it won't terminate.
- Idea: introduce a *depth limit* on branches to be expanded.
- Don't expand a branch below this depth.
- Obviously this can be a source of incompleteness,

  BUT knowledge of the problem can help to set a sensible limit.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
     result ← RECURSIVE-DLS(successor, problem, limit)
     if result = cutoff then cutoff-occurred? ← true
     else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Iterative Deepening

- Unfortunately, if we choose a max depth for DLS such that shortest solution is longer, DLS is not complete.
- Iterative deepening an ingenious *complete* version of it.
- Basic idea is:
  - do DLS for depth 1; if solution found, return it;
  - otherwise do DLS for depth n; if solution found, return it;
  - otherwise, ...
- So we *repeat* DLS for all depths until solution found.

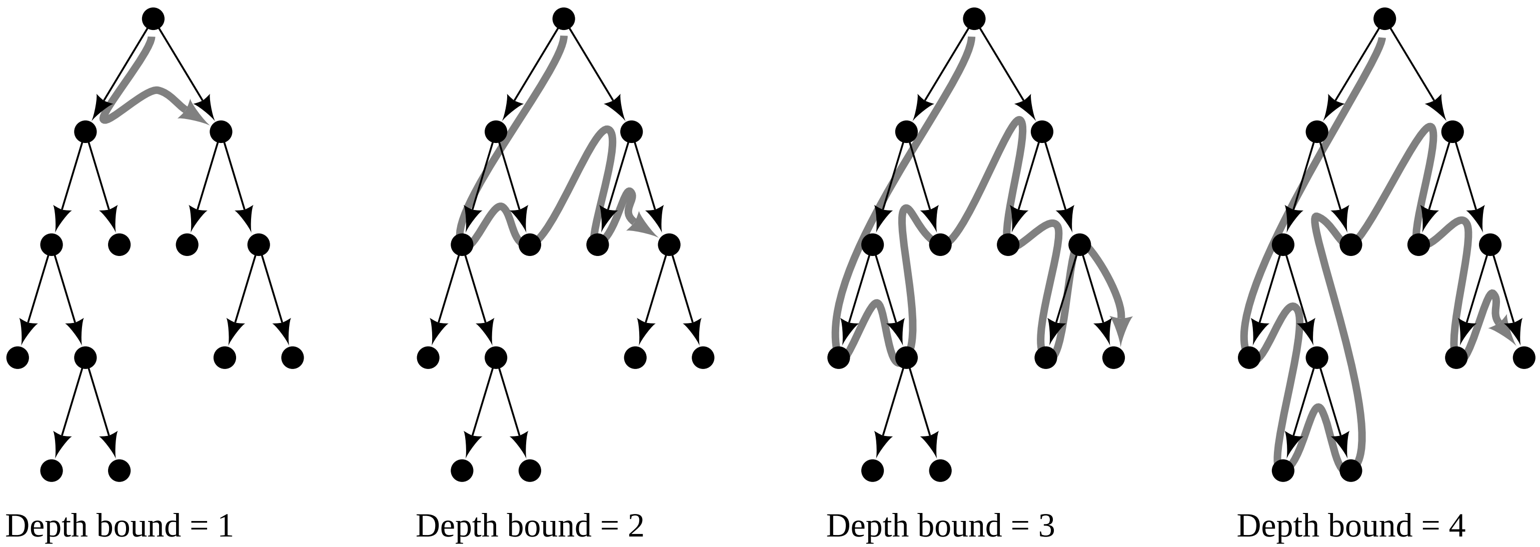
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution

inputs: *problem*, a problem

for *depth* $\leftarrow$ 0 **to** ∞ **do**
  result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
  if *result* $\neq$ cutoff **then return** *result*
end

- Calls DLS as subroutine.

- The search covers the whole state space down to the depth limit.



- The order it searches the nodes changes for each depth limit.

- Note that in iterative deepening, we *re-generate nodes on the fly.* Each time we do call on depth limited search for depth d, we need to regenerate the tree to depth $d-1$.
- Isn't this inefficient?
- Tradeoff *time* for *memory*.
- In general we might take a *little* more time, but we save a *lot* of memory.
- Now for breadth-first search to level d:

$$\begin{aligned} N_{bf} &= 1 + b + b^2 + \ldots b^d \\ &= \frac{b^{d+1} - 1}{b - 1} \end{aligned}$$

- In contrast a complete depth-limited search to level j:

$$N_{df}^{j} = \frac{b^{j+1}-1}{b-1}$$

- (This is just a breadth-first search to depth j.)
- In the worst case, then we have to do this to depth d, so expanding:

$$\begin{aligned} N_{id} &= \sum_{j=0}^{d} \frac{b^{j+1}-1}{b-1} \\ &\vdots \\ &= \frac{b^{d+2}-2b-bd+d+1}{(b-1)^2} \end{aligned}$$

- For large d:

$$\frac{N_{id}}{N_{bf}} = \frac{b}{b-1}$$

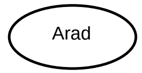
- So for high branching and relatively deep goals we do a small amount more work.
- Example: Suppose $b = 10$ and $d = 5$.

  Breadth first search would require examining $111,111$ nodes, with memory requirement of $100,000$ nodes.

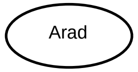
  Iterative deepening for same problem: $123,456$ nodes to be searched, with memory requirement only 50 nodes.

  Takes 11% longer in this case.

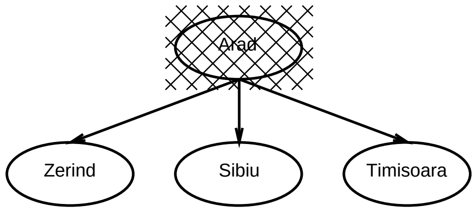
- On the Romania example we start with the initial state, expand one node, and fail to find the goal.



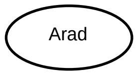
- For the next iteration we start over



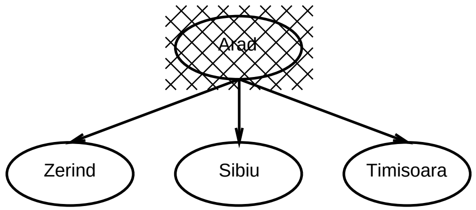
- This time we push down another level before failing.



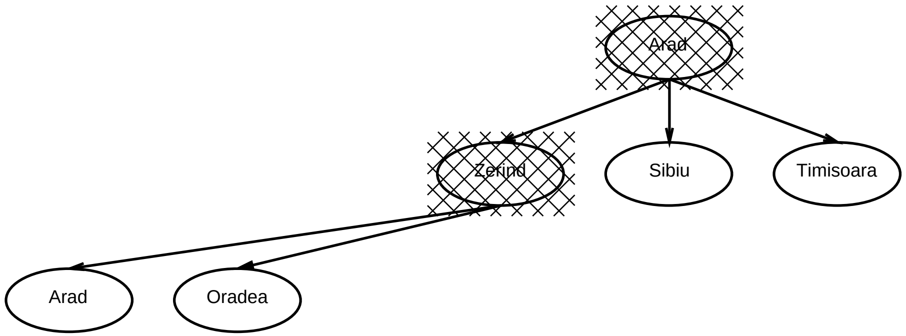
- Then we start a third time



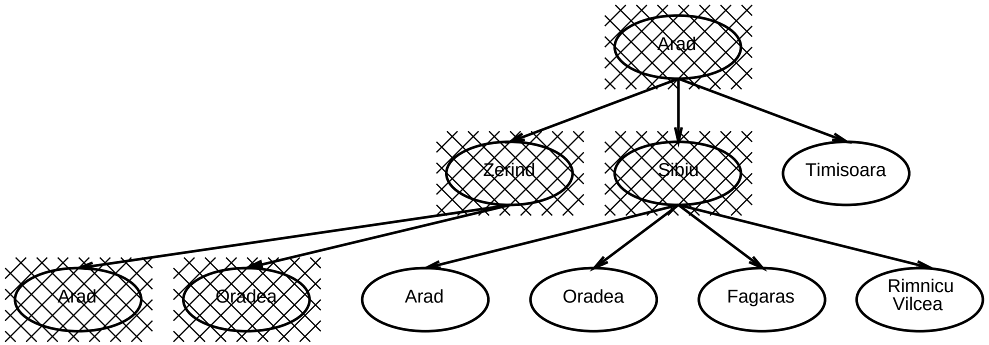
- And when we get here, we push down another level



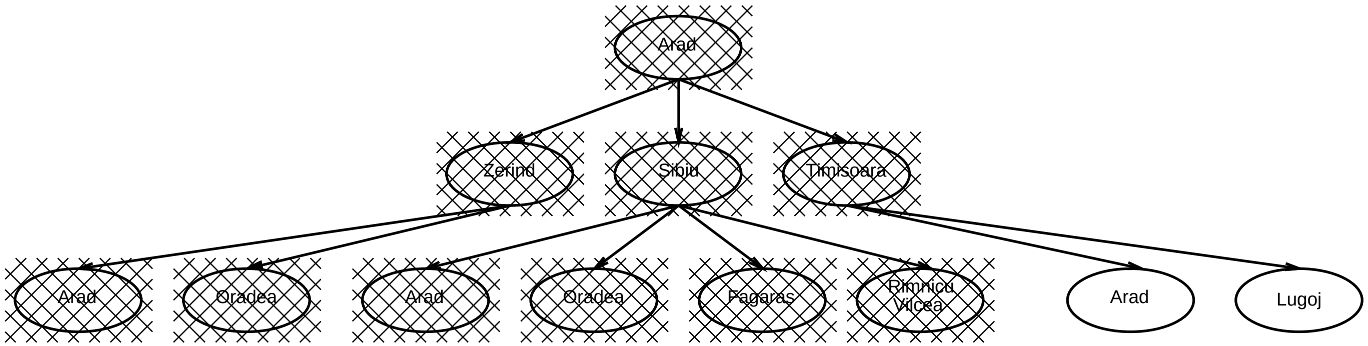
- Expanding the first child node on the second level.



- When that fails to produce a solution, we expand the second node on the second level.



- And finally the third node on that level.



Heuristic search

- We now turn to informed search — where the search uses problem specific information to guide the search.
- Whatever search technique we use, *exponential time complexity*.
- We want to search less, by having an idea where the goal is.
- Simplest form of problem specific knowledge is *heuristic*.
- Usual implementation in search is via an *evaluation function* which indicates desirability of a given node.

$$f(n)$$

- We are already familiar with this idea from uniform cost search where

$$f(n) = g(n)$$

Greedy Search

- Most heuristics estimate cost of *cheapest path* from node to solution.
- We have a *heuristic function*,

$$h : Nodes \rightarrow R$$

  which estimates the distance from the node to the goal.
- Example: In the Romania example, heuristic might be straight line distance from node to Bucharest.
- Heuristic is said to be *admissible* if it *never overestimates* cheapest solution.

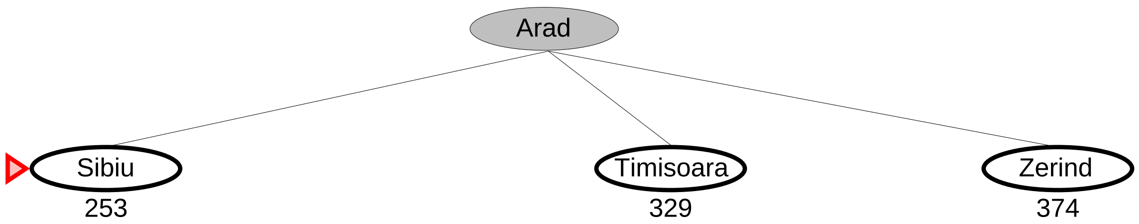
  Admissible = optimistic.
- Greedy search involves expanding node with cheapest expected cost to solution.

function GREEDY-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

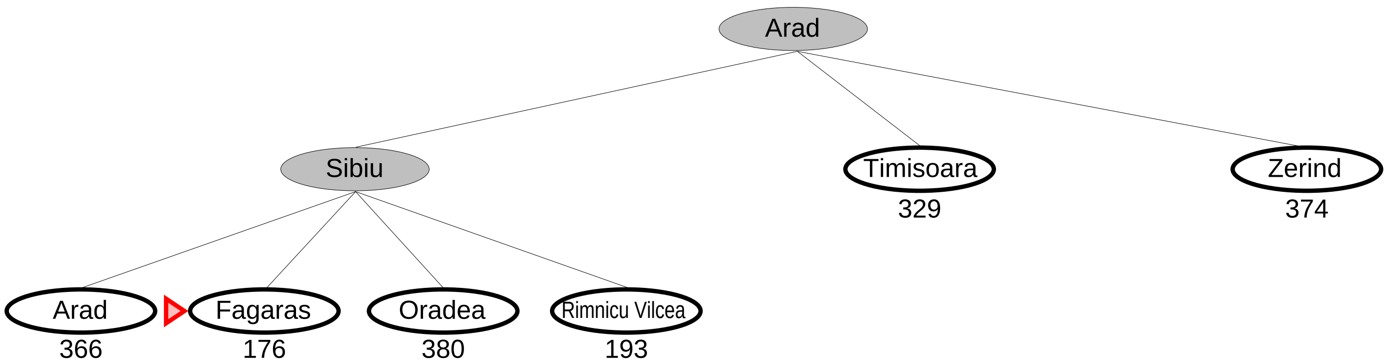
```
closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
        add STATE[node] to closed
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
        fringe ← SORTBYHVALUE(fringe)
end
```

- As ever we start with the initial node. Note the heuristic value.

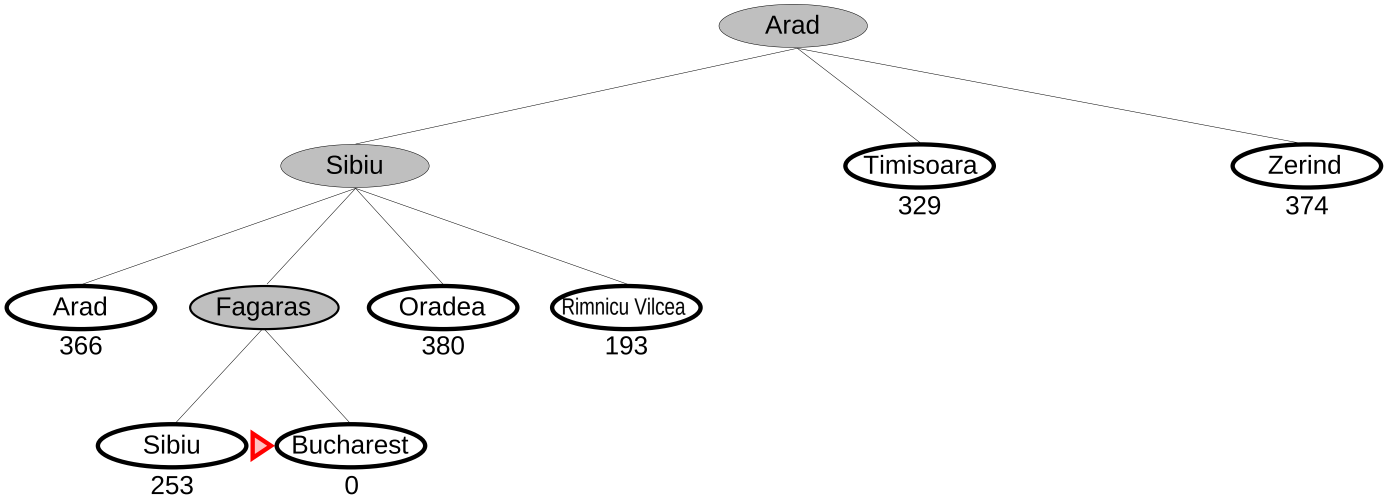
- When then expand the child node with the lowest heuristic value



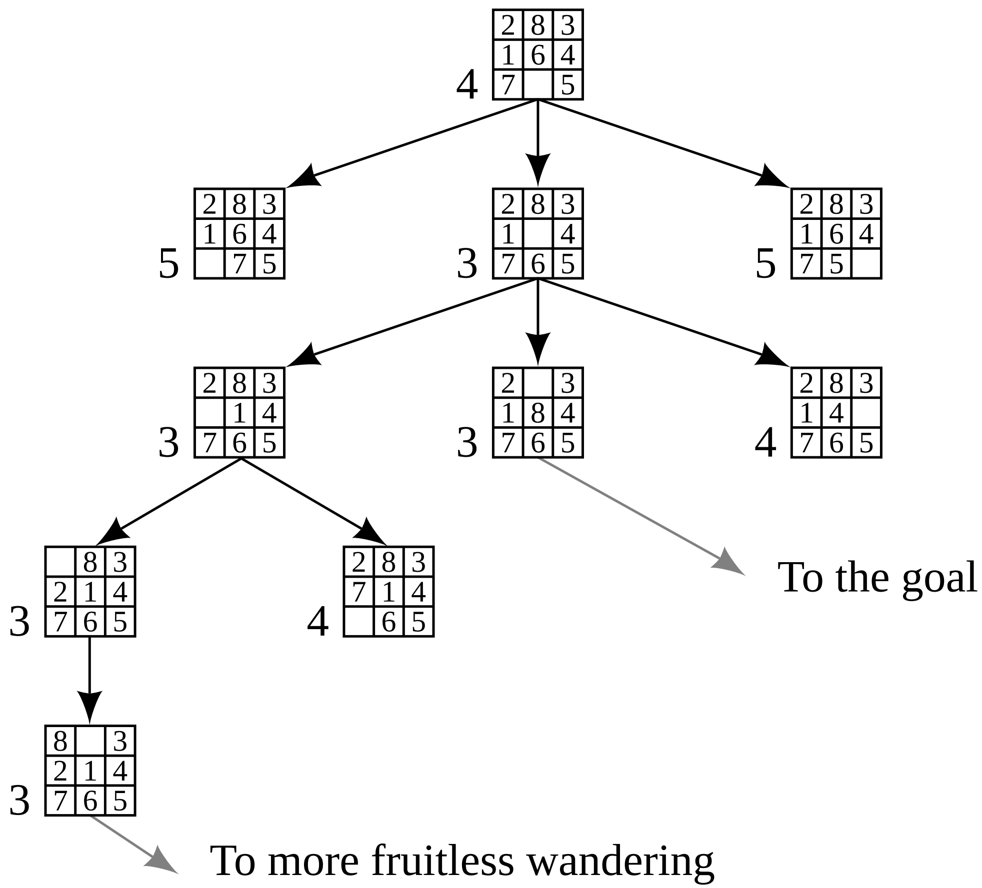
- And then we repeat.



- In the next level we find the goal.



- Greedy search finds solutions quickly.
- It doesn’t always find the best solution where there is more than one.
- Susceptible to false starts.
  - Chases good looking options that turn out to be bad.
- Only looks at *current* node. Ignores past!
- Also *myopic* (shortsighted).


4
2 8 3
1 6 4
7 5
5
3
5
3
3
4
3
4
3
To the goal
To more fruitless wandering

- For the 8-puzzle one good heuristic is:
  - count tiles out of place.
- Another is:
  - *Manhattan blocks' distance*
- The latter works for other problems as well:
  - Robot navigation.

A^* Search

- A^* is very efficient search strategy.
- Basic idea is to *combine*

uniform cost search
and
greedy search.

- We look at the *cost so far* and the *estimated cost to goal*.
- Gives heuristic f:

$$f(n) = g(n) + h(n)$$

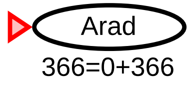
where

  - $g(n)$ is path cost of n;
  - $h(n)$ is expected cost of cheapest solution from n.

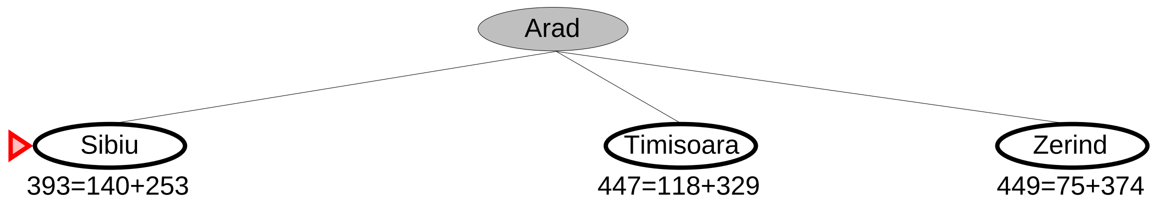
- Aims to mimimise *overall cost*.

```
function A-STAR-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
          fringe ← SORTBYFVALUE(fringe)
  end
```

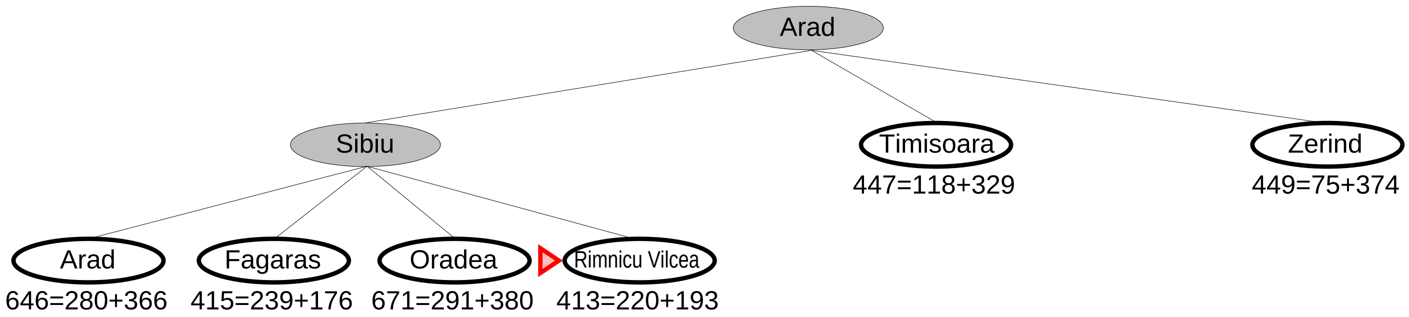
- Start with the initial node, this is the one we expand next.



- At the next level down, Sibiu has the lowest $f(\cdot)$ value.

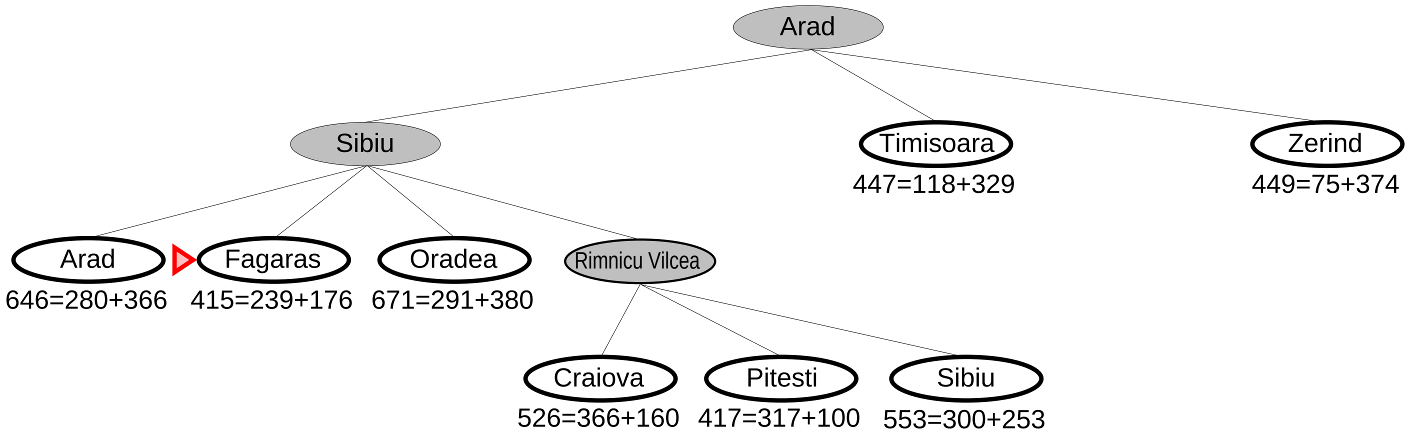


- At the next level, Rimnicu Vicea is the best-looking option.

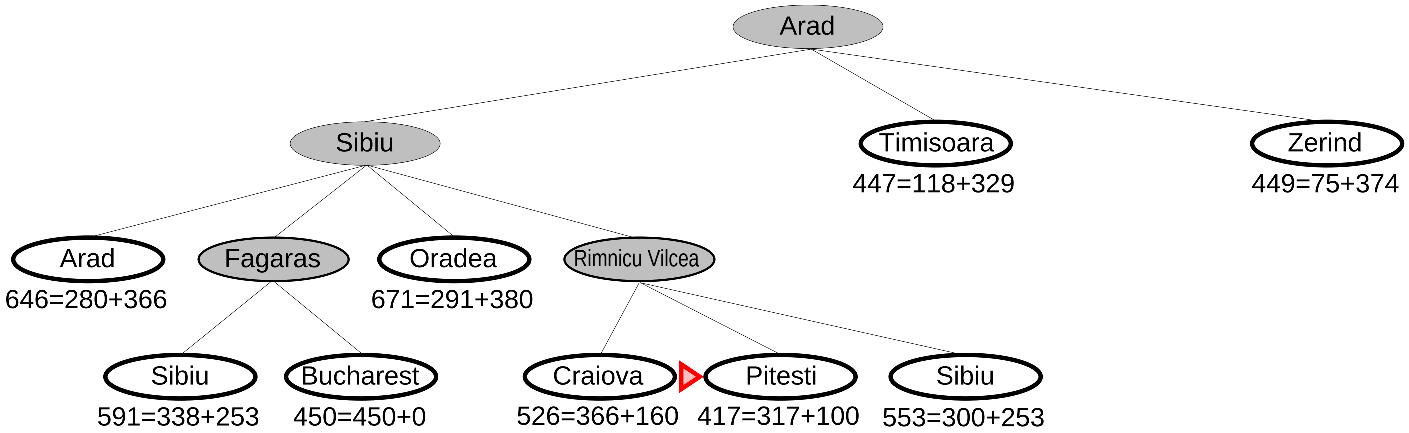


- Though it is further from the start than, for example, Timisoara, it is also closer to Bucharest.

- However, it is a false start, once we expand its children, they are worse options than Fagaras.



- And when we look at Fagaras' children, they include Bucharest.



The optimality of A^*

- A^* is optimal in precise sense—it is guaranteed to find a minimum cost path to the goal.
- There are a set of conditions under which A* will find such a path:
  1. Each node in the graph has a finite number of children.
  2. All arcs have a cost greater than some positive ϵ.
  3. For all nodes in the graph $h(n)$ always *underestimates* the true distance to the goal.
- The key here is the third bullet — the notion of *admissibility*.
- We will express this by saying a heuristic $h(\cdot)$ is admissible if

$$h(n) \leq h_T(n)$$

More informed search

- IF two versions of A^*, A_1^* and A_2^* use different functions h_1 and h_2,
- AND

$$h_1(n) < h_2(n)$$

  for all non-goal nodes,
- THEN we say that A_2^* is *more informed* than A_1^*.
- The better informed A^* is, the less nodes it has to expand to find the minimum cost path.

- As an example of ″more informed″ consider the 8-puzzle:
  - tiles out of place; and
  - Manhattan blocks distance.
- We need $h(n)$ to underestimate $h_T(n)$ to ensure admissibility.
- But, the closer the estimate, the easier it is to reject nodes which are not on the optimal path.
- This means less nodes need to be searched.

- There are techniques that go further than those we have studied:
  - Iterative deepening A^* (IDA^*)
  - Focussed Dynamic A^* (called D^*)
  - D^* Lite
  - Delayed D^*
  - Life-long planning A^* (called LPA^*)
  - PAO^*
- There are three directions we will take from here:
  - Adversarial search
  - Learning the state space.
  - Adding in more knowledge about the domain.

Summary

- This lecture introduced the basics of problem solving.
- In particular it discussed *state space* models and looked at some techniques for solving them.
  - Search for the goal.
  - Path through state space is the solution.
- We also looked at some techniques for search:
  - Breadth first.
  - Uniform cost
  - Depth first.
  - Iterative deepening
  - Best-first search
  - A^* search