Informed search algorithms

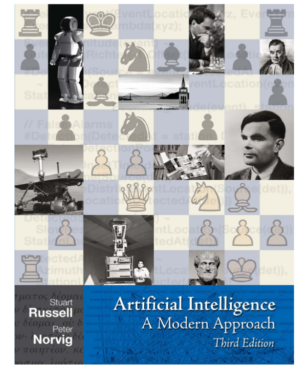
Michal Pěchouček, Milan Rollo

Department of Cybernetics
Czech Technical University in Prague

http://cw.felk.cvut.cz/doku.php/courses/ae3b33kui/start

Recommended literature

:: Artificial Intelligence: A Modern Approach (Third Edition) by Stuart Russell and Peter Norvig, 2007 Prentice Hall.



http://aima.cs.berkeley.edu/

Informed state space search strategies

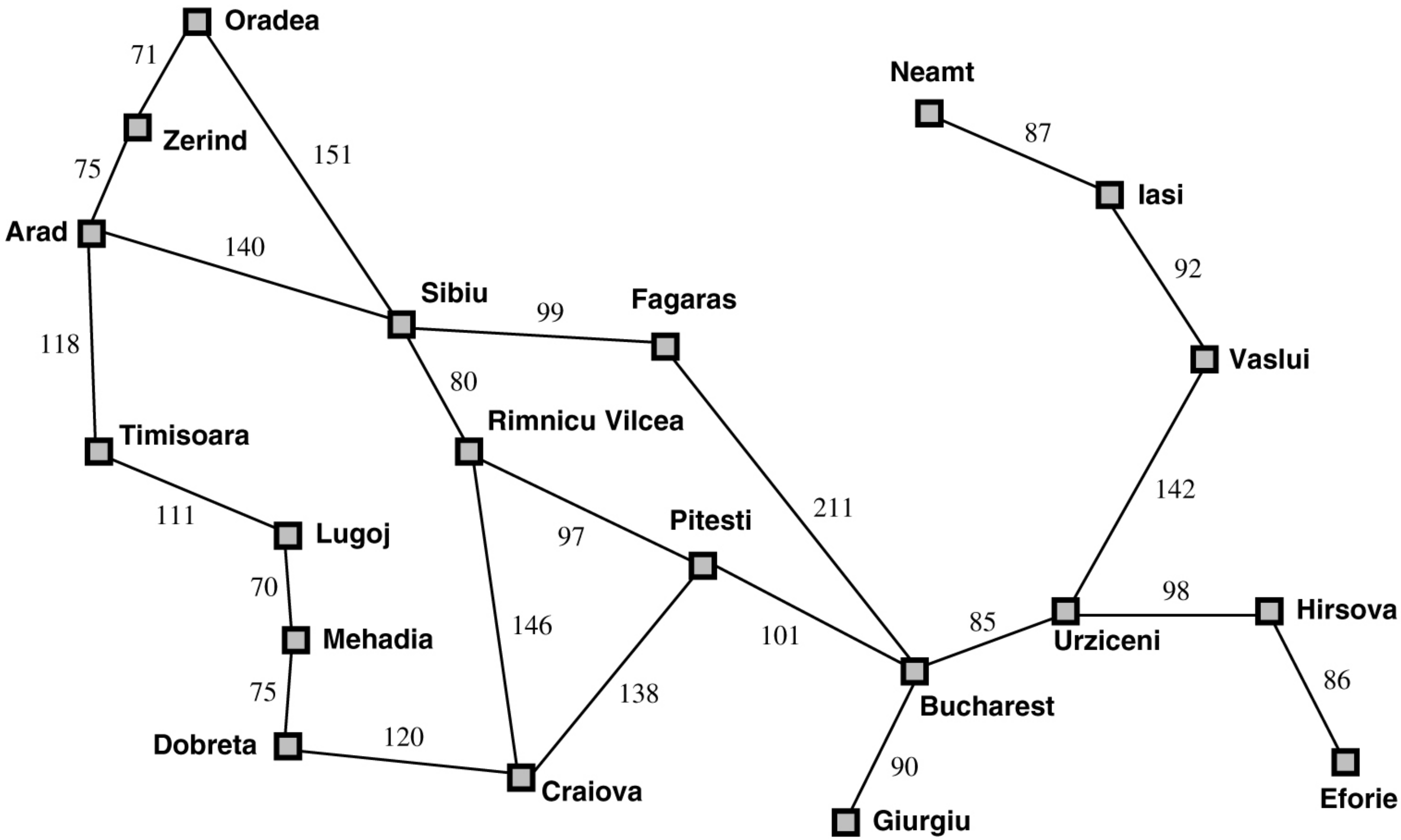
:: implement efficient ways to find an optimal solution and during that process they utilize qualitative information about various states of the state space.

During the search we use (at least one):

- specific information about a cost of given state in state space
- specific information about a cost of applying each possible action
- heuristic information, estimation suitability of use of given action with respect to the state space search efficiency

:: This information is used for the design of **heuristic** algorithm (called also as *Best-First-Search*), which selects a node for expansion. Such a node leads the search process to an optimal solution. When well-designed heuristic algorithm minimizes the search of those parts of state space that don't lead to optimal solution.

Informed state space search strategies



Best-first search

Design of the best-first search algorithm builds on a classic algorithm for uninformed state space search:

```
begin
    open := [Start], closed := []
    while (open <> []) do begin
        X := FIRST(open)
        closed := closed + [X], open := open - [X]
        if X = GOAL then return(SUCCESS)
        else begin
                E := expand(X)
                E := E - closed
               open := open + E
             end
   end
return(failure)
end.
```

Best-first search

only the selection of **first** element in a list is replaced by the selection of the **best** element

```
begin
    open := [Start], closed := []
    while (open <> []) do begin
        X := BEST(open)
        closed := closed + [X], open := open - [X]
        if X = GOAL then return(SUCCESS)
        else begin
                E := expand(X)
                E := E - closed
               open := open + E
             end
   end
return(failure)
end.
```

Evaluation function

:: When algorithm tries to select a best state to expand (e.g. s_n) from current state (e.g.. s_m) it works with following functions:

- $c(m, n)$ – cost of action we need to apply to get from state m to state n
- $g(m)$ – overall cost, sum of costs of all actions that were applied from the initial state to state m
- $h(n)$ – real or estimated total cost, sum of costs of all actions we need to apply from state n to the goal state.

:: it is thus necessary to propose an **evaluation function** f, that will integrate functions c, g and h in a sophisticated way and will ensure a reasonable behavior in a given domain.

Examples of evaluation function

- **Gradient search** (hill-climbing search)– where $\forall m, n : f(m, n) = c(m, n)$
  - easy to implement, fast, resistive to infinite loops – but often gets *stuck at a local optimum* !!!

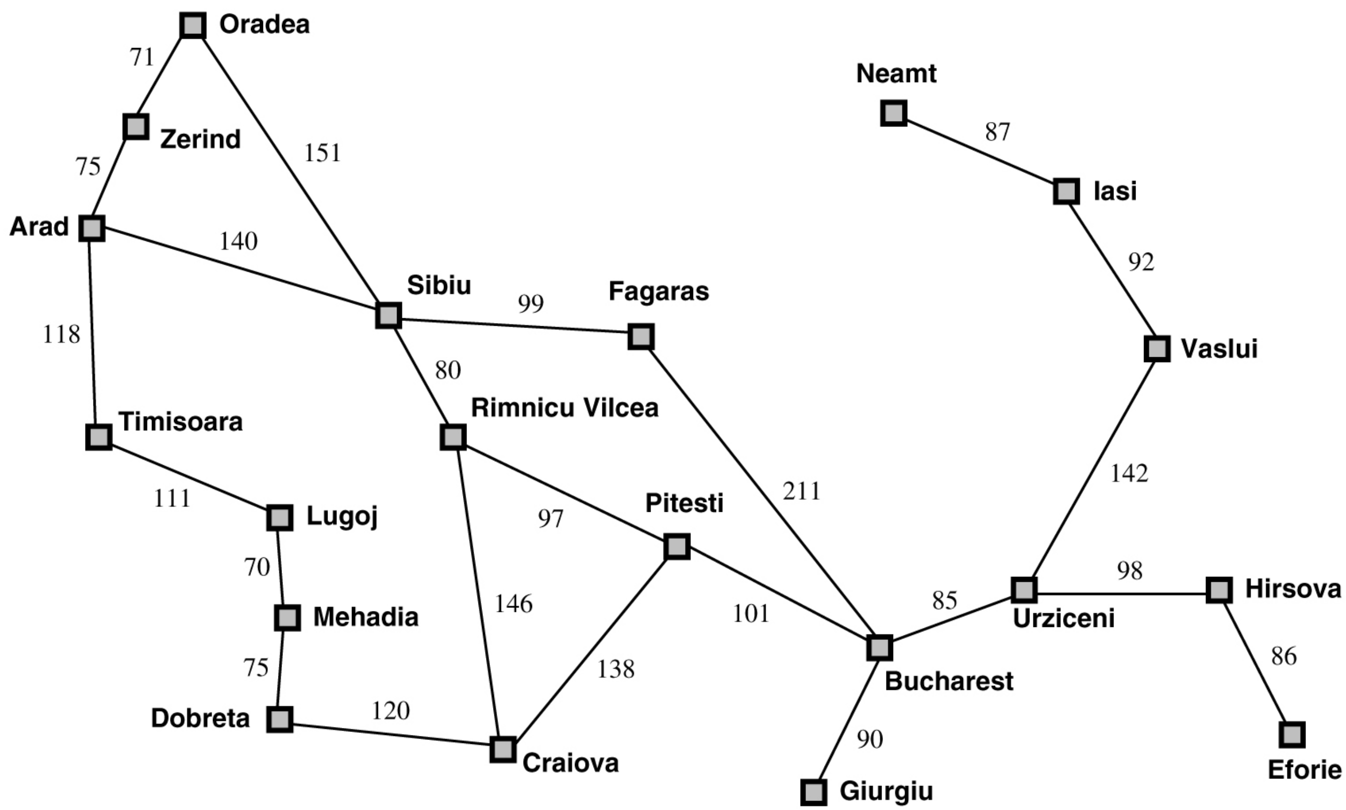
Examples of evaluation function

- **Gradient search** (hill-climbing search)– where $\forall m, n : f(m, n) = c(m, n)$
  - easy to implement, fast, resistive to infinite loops – but often gets *stuck at a local optimum* !!!
- **Breadth-first search** – $\forall m, n : c(m, n) = 1$ if there is an edge from m to n. Thus the $f(m, n) = g(m) + 1$
  - minimizes number of steps (depth) to the solution

Examples of evaluation function

- **Gradient search** (hill-climbing search)– where $\forall m, n : f(m,n) = c(m,n)$
  - easy to implement, fast, resistive to infinite loops – but often gets *stuck at a local optimum* !!!
- **Breadth-first search** – $\forall m, n : c(m,n) = 1$ if there is an edge from m to n. Thus the $f(m,n) = g(m) + 1$
  - minimizes number of steps (depth) to the solution
- **Greedy search algorithm** – $\forall m, n : f(m,n) = h(n)$ if there is an edge from m to n. Here the $h(n)$ is heuristic estimation of path cost from node n to closest goal node
  - non-optimal, incomplete

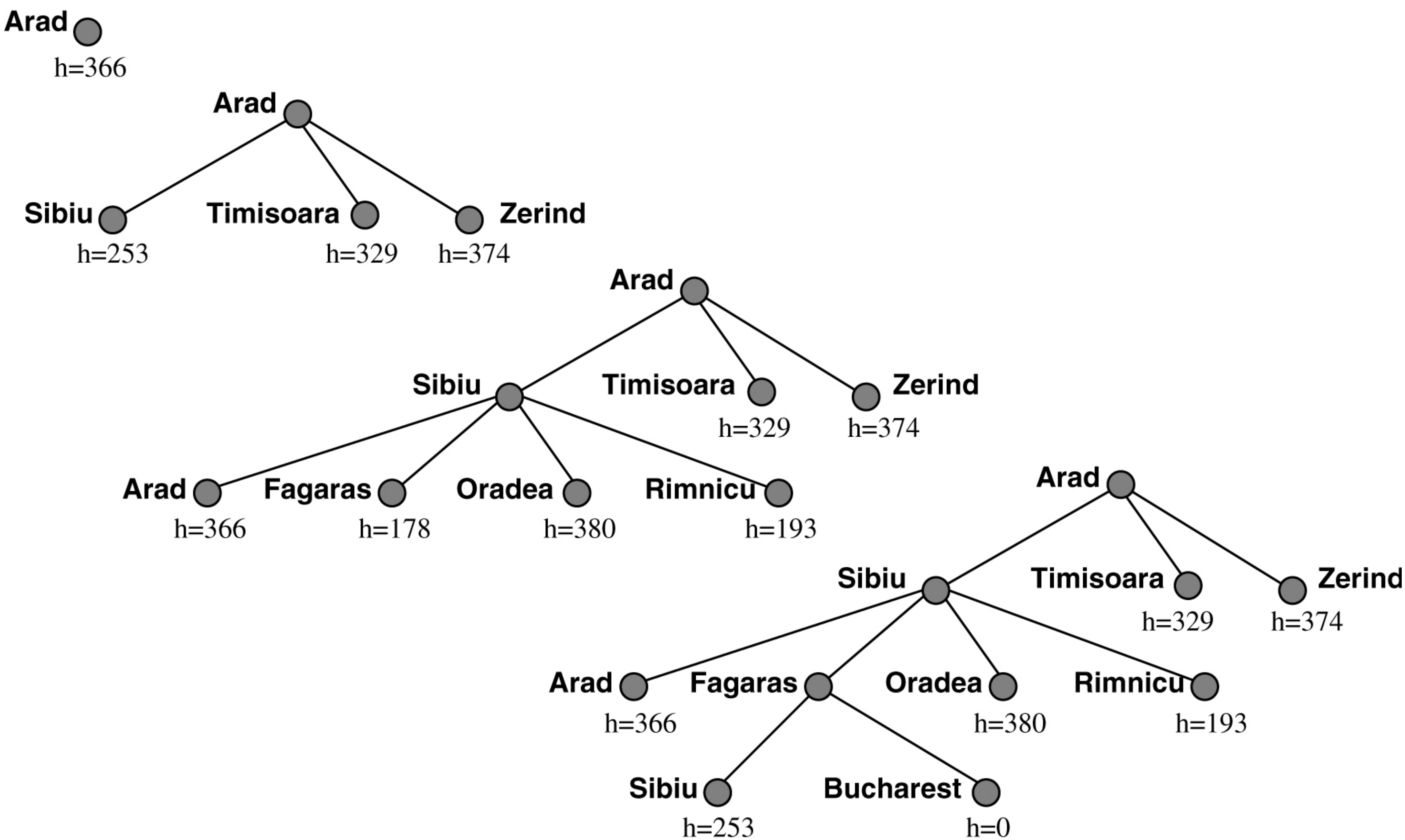
Greedy search



Straight–line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search



A^* algorithm

- A^* algorithm uses the best-first search approach, where each state has assigned an evaluation function:
  - $f(n) = g(n) + h(n)$
  - remark: (c,g a h) in a form $f(n, m) = c(m, n) + g(m) + h(n)$ could be written as $f(n) = g(n) + h(n)$ because $g(n) = g(m) + c(m, n)$, where argument m does no longer influence the value of the function.
- To set evaluation function f in a form $f(n) = g(n) + h(n)$ is a nontrivial problem because of the function $h(n)$ whose value is not known a priori and we have to estimate it.
- Due to the fact, that we optimize the behavior of algorithm, we want this estimation to be as precise as possible – we want the value of $h(n)$ to be as close to the value of $h^*(n)$ (real value). Function $h(n)$ is called **heuristic function**.
- Evaluation function $f(n)$ is thus an estimation of real values, that we will get by function $f^*(n) = g^*(n) + h^*(n)$ where $g(n) = g^*(n)$ (in most cases)

Admissibility of A^* algorithm

- Which features must the heuristic function $h(n)$ have? What will happen if $h(n) > h^*(n)$? And what if $h(n) < h^*(n)$?

Admissibility of A^* algorithm

- Which features must the heuristic function $h(n)$ have? What will happen if $h(n) > h^*(n)$? And what if $h(n) < h^*(n)$?
- For an algorithm to behave in a reasonable manner, *i.e to find an optimal solution first*, it must hold that:

$$\forall n : 0 \leq h(n) \leq h^*(n)$$

- When that holds, we say that heuristic function is **admissible**.
- A^* algorithm uses best-first search approach where each state is evaluated by function $f(n) = g(n) + h(n)$, where $h(n)$ is
- A^* algorithm will always find an optimal solution.

is BFS optimal?

Admissibility of A^* algorithm

- Which features must the heuristic function $h(n)$ have? What will happen if $h(n) > h^*(n)$? And what if $h(n) < h^*(n)$?
- For an algorithm to behave in a reasonable manner, *i.e to find an optimal solution first*, it must hold that:

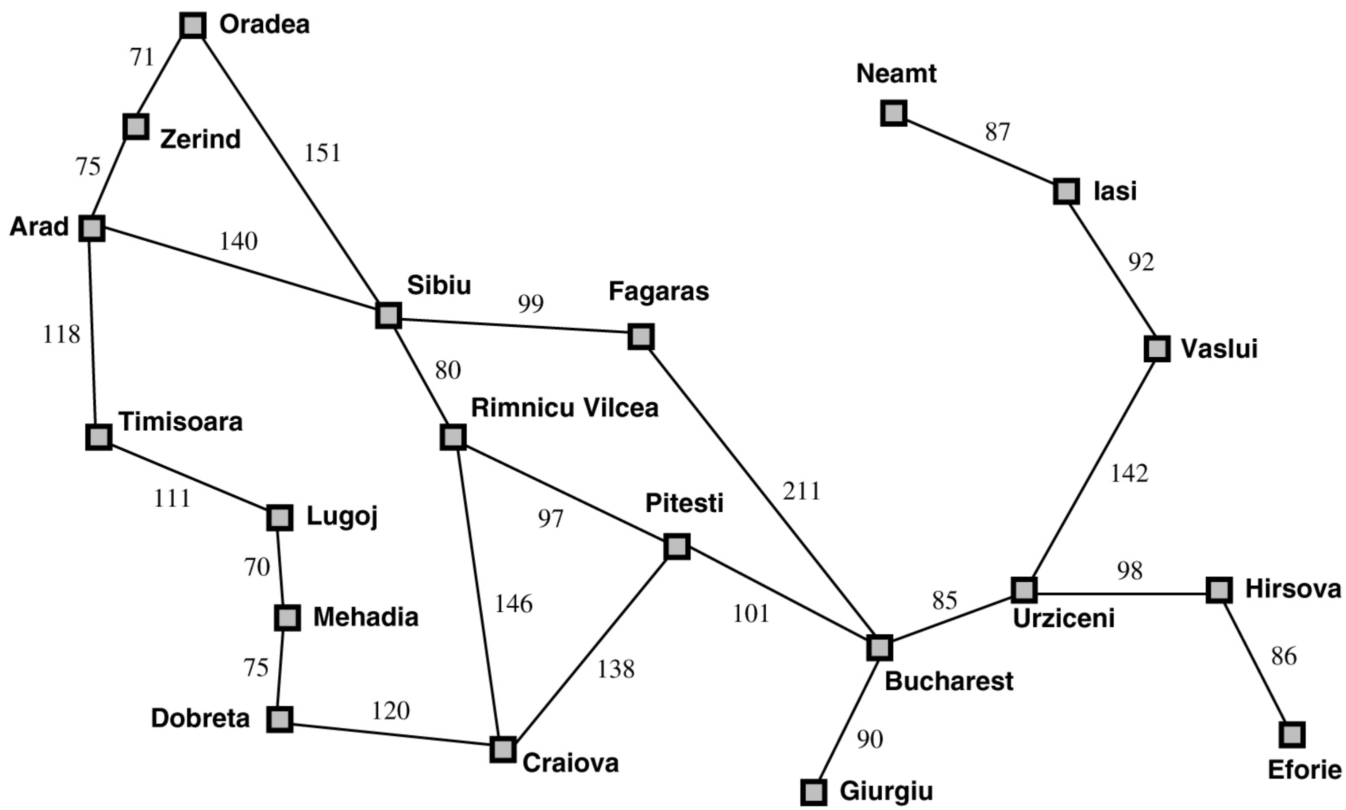
$$\forall n : 0 \leq h(n) \leq h^*(n)$$

- When that holds, we say that heuristic function is **admissible**.
- A^* algorithm uses best-first search approach where each state is evaluated by function $f(n) = g(n) + h(n)$, where $h(n)$ is
- A^* algorithm will always find an optimal solution.

is BFS optimal?

yes, because for BFS $f(n) = g(n) + h(n) = g(n)$. Thus $0 = h(n) < h^*(n)$ and heuristic is admissible.

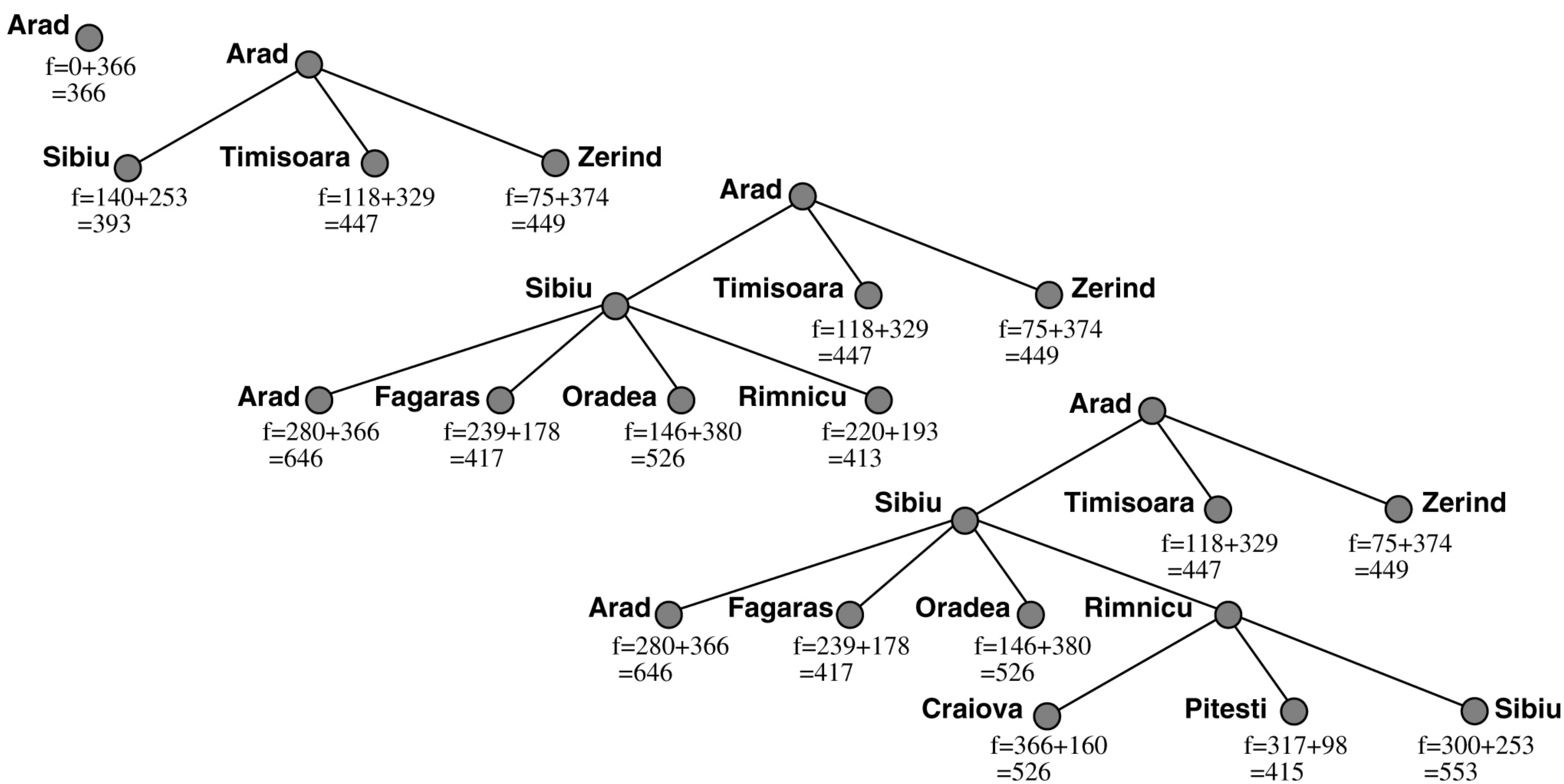
Example of heuristic for path search



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example of heuristic for path search



Additional properties of A*

```
begin
    open :=  [Start], closed :=  []
    while (open <>  []) do begin
        X :=  BEST(open)
        closed := closed + [X], open := open - [X]
        if X = GOAL then return(SUCCESS)
        else begin
                E := expand(X)
                E := E - closed
               open := open + E
             end
   end
return(failure)
end.
```

Additional properties of A*

clarification of operations in A^* at line 9 - 10:

- in case that node $e \in$ `E` is already in an `open` list
  - with value $f(e)$ better, than the node is not added into the `open` list at line `10`
  - with value $f(e)$ worse, than the better node is added into the `open` list at line `10` and the worst node is removed
- in case that node $e \in$ `E` is already in a `closed` list
  - with value $f(e)$ better, than that node is removed from the `E` list at line 9
  - with value $f(e)$ worse, than that node is not removed from `E` list at line 9, it is removed from the `closed` list.

Heuristic monotonicity

Heuristic function is **monotone**/consistent (localy admissible) if

i. $\forall n_1, n_2$, where n_1 expands into n_2: $h(n_1) - h(n_2) \leq cost(n_1, n_2)$, where $c(n_1, n_2)$ is real cost from n_1 to n_2

ii. $h(\texttt{goal}) = 0$.

each monotone heuristic function is admissible.

Heuristic monotonicity

Heuristic function is **monotone**/consistent (localy admissible) if

i. $\forall n_1, n_2$, where n_1 expands into n_2: $h(n_1) - h(n_2) \leq cost(n_1, n_2)$, where $c(n_1, n_2)$ is real cost from n_1 to n_2

ii. $h(\texttt{goal}) = 0$.

each monotone heuristic function is admissible.

Proof:

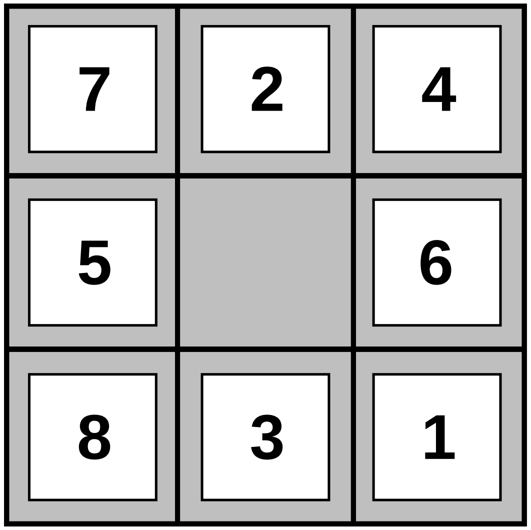
for $n_0 \rightarrow n_1 \quad \ldots \quad h(n_0) - h(n_1) \leq c(n_0, n_1)$ due to the monotonicity
for $n_1 \rightarrow n_2 \quad \ldots \quad h(n_1) - h(n_2) \leq c(n_1, n_2)$ due to the monotonicity
$\ldots$
for $n_{k-1} \rightarrow \texttt{goal} \quad \ldots \quad h(n_{k-1}) - h(\texttt{goal}) \leq c(n_{k-1}, \texttt{goal})$

if $h(\texttt{goal}) = 0$ then when we sum all lines $h(n_0) \leq c(n_0, \texttt{goal})$

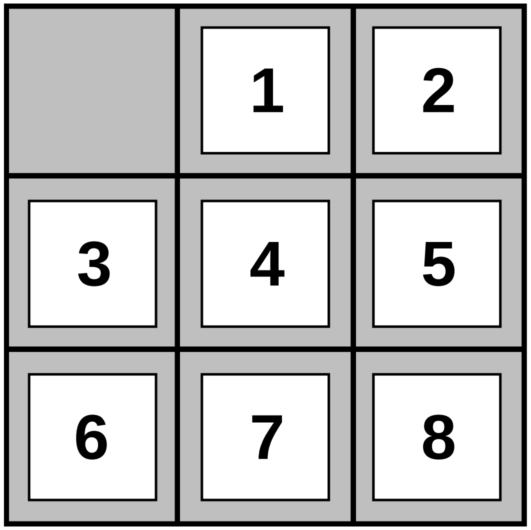
Dominance

- Having two admissible heuristics h_1 and h_2 so that $\forall n : h_1(n) \leq h_2(n)$, then we say that heuristic h_2 **dominates** h_1.
  - Both heuristics will find optimal solution, but h_2 needs to expand less nodes than h_1.
- We have to take care that the computational time of dominating heuristic will not last considerably longer than search of larger portion of state space.

Heuristics for 8-puzzle



Start State



Goal State

Heuristics for 8-puzzle

2 8 3 1 6 4 _ 7 5	5	6	0
2 8 3 1 _ 4 7 6 5	3	4	0
2 8 3 1 6 4 7 5 _	5	6	0
	Titles out of place	Sum of distances out of place	2x the number of direct tile reversals

1	2	3
8		4
7	6	5

Goal

Effective Branching Factor: average number of children b^* of each node. For depth d and total number of expanded nodes N then must hold $N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

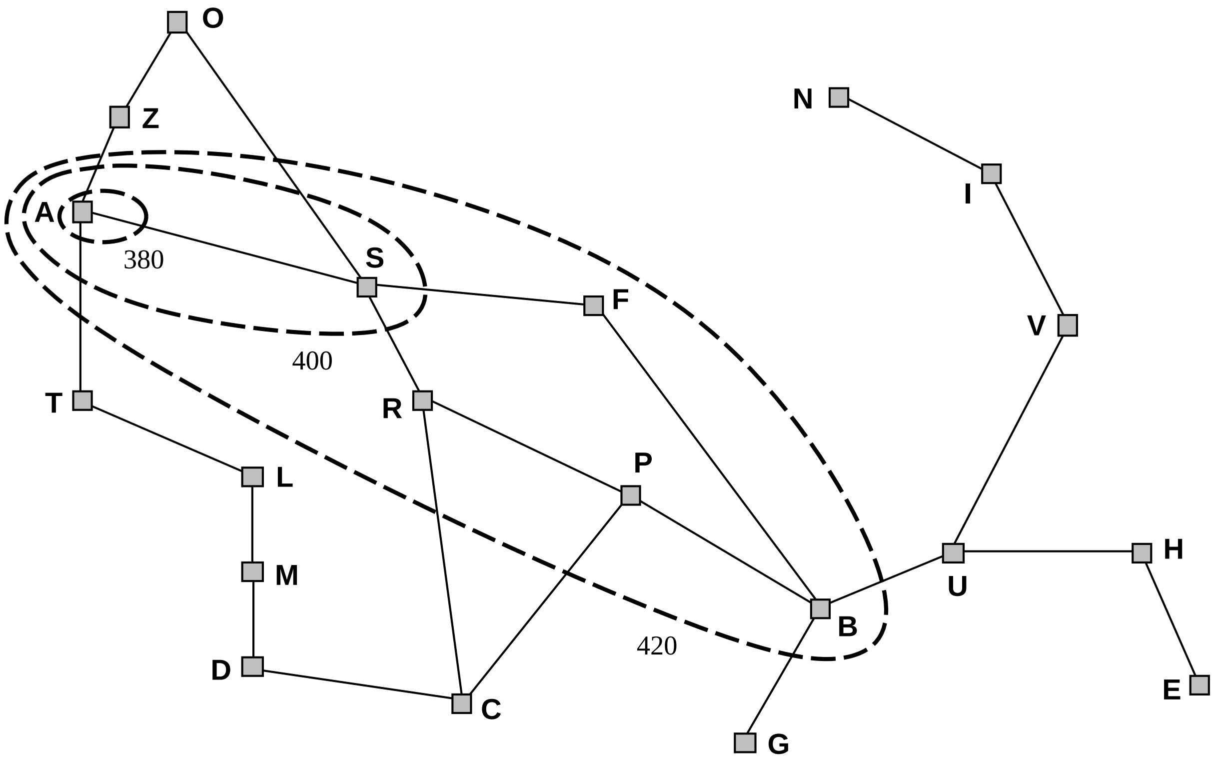
Optimal efficiency of A^*

:: We say that A^* is **optimally efficient**. It means that for any heuristic function there isn't any other optimal algorithm that will expand less nodes than A^*.

We know about A^* that algorithms is optimal, complete and optimally efficient. But it still doesn't mean that it is suitable for all search problems. Unfortunatally the memory comsumption still grows exponentially (it was prooved that this holds in cases when $|h(n) - h^*(n)| > O(\log h^*(n))$).

:: Time complexity is not the major problem of A^*. Due to the fact that A^* must keep all expanded states in a memory, it often happens that we run out of the memory sooner than we run out of time.

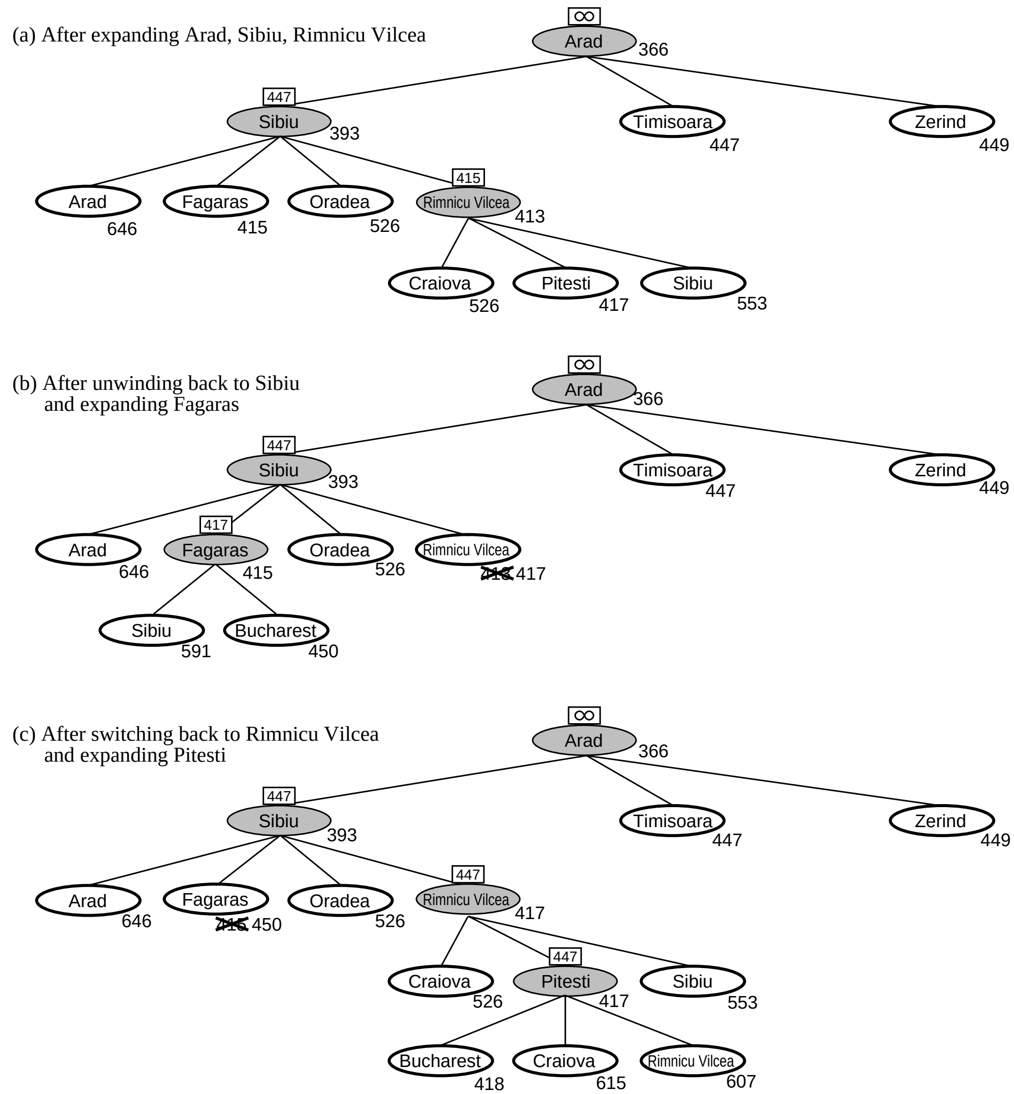
Optimal efficiency of A^*



Variants of algorithms improving memory usage

- **IDA**** – iterative deepening A^* algorithm: Works in a same way as an iterative-deepening depth-first search (IDDFS), with that difference that we don't increase the depth limit, but the least value of f which is higher than f from previous run.
- **RBFS** – Recursive best first search, recursive IDA*. It limits the value of f to the second best in given layer.

Variants of algorithms improving memory usage



Variants of algorithms improving memory usage

- **IDA*** – iterative deepening A^* algorithm: Works in a same way as an iterative-deepening depth-first search (IDDFS), with that difference that we don't increase the depth limit, but the least value of f which is higher than f from previous run.
- **RBFS** – Recursive best first search, recursive IDA*. It limits the value of f to the second best in given layer.

While IDA* is more memory efficient, RFBS will find a solution faster, because it holds bigger `open list`.

Variants of algorithms improving memory usage

- **IDA*** – iterative deepening A^* algorithm: Works in a same way as an iterative-deepening depth-first search (IDDFS), with that difference that we don't increase the depth limit, but the least value of f which is higher than f from previous run.
- **RBFS** – Recursive best first search, recursive IDA*. It limits the value of f to the second best in given layer.

While IDA* is more memory efficient, RFBS will find a solution faster, because it holds bigger `open list`.

- **MA*** – memory bounded A^* – uses all available memory. Simplified algorithm SMA* (simplified MA*) keeps predefined number of states in open-list. When the list is full the worst node is forgotten.

Variants of algorithms improving memory usage

- **IDA*** – iterative deepening A^* algorithm: Works in a same way as an iterative-deepening depth-first search (IDDFS), with that difference that we don't increase the depth limit, but the least value of f which is higher than f from previous run.
- **RBFS** – Recursive best first search, recursive IDA*. It limits the value of f to the second best in given layer.

While IDA* is more memory efficient, RFBS will find a solution faster, because it holds bigger `open list`.

- **MA*** – memory bounded A^* – uses all available memory. Simplified algorithm SMA* (simplified MA*) keeps predefined number of states in open-list. When the list is full the worst node is forgotten.

IDA* and RFBS are optimal (i.e., they cannot miss the best solution), MA* and SMA* can miss optimum and get stuck in a local extreme (when the `open list` size bound is small).