Block 1 – Knowledge Representation and Search

Chapter 5:
Heuristic Search

Block 1, Chapter 5- Index

Bibliography

S. Russell, P. Norvig. ***Artificial Intelligence . A modern approach.*** Prentice Hall, 3rd edition, 2010 (Chapter 3) http://aima.cs.berkeley.edu/

Alternatively:
S. Russell, P. Norvig. ***Artificial Intelligence . A modern approach.*** Prentice Hall, 2nd edition, 2004 (Chapters 3 and 4) http://aima.cs.berkeley.edu/2nd-ed/

1. Heuristic search

Informed (heuristic) search: one that uses problem-specific knowledge to guide search.

Heuristic search can find solutions more efficiently than an uninformed search. Heuristic search is recommended for complex problems of combinatorial explosion(e.g.: traveling salesman problem).

Why use heuristics? In order to efficiently solve a problem we must sometimes give up using a systematic search strategy which guarantees optimality and use instead a search strategy that returns a good solution though not the optimal one.

Heuristic search performs an intelligent search guidance and allows for pruning large parts of the search tree.

Why heuristics are appropriate?

1. Usually, we don't need to get the optimal solution in complex problem, but a *good* solution is just enough.
2. Heuristics may not return a good solution for the worst-case problem but this case seldom happens.
3. Understanding how heuristics work (or don't work) helps people understand and go deeply into the problem

1. Heuristic search

We will consider a general approach called best-first search:

- an instance of the TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function f(n)
- f(n) is a cost estimate so the node with the lowest cost is expanded first from the priority queue (in uninformed search, f(n) is a different function for each strategy so that nodes are inserted in the OPEN LIST according to the expansion order of the strategy)
- In fact, all search strategies can be implemented by using f(n); the choice of **f** determines the search strategy
- Two special cases of best-first search: *greedy best-first search* and *A**

2. Greedy search

Most best-first algorithms include as a component of **f** a heuristic function **h(n)**

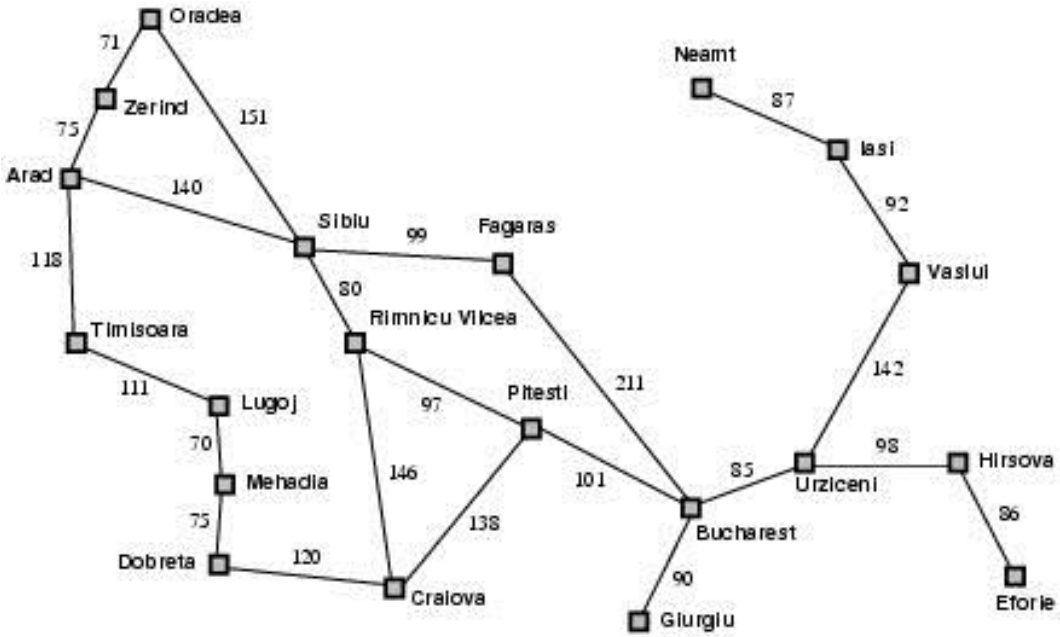
Heuristic function: *A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood*

h(n)= *estimated cost* of the cheapest path from the state at node **n** to the goal state

If **n** is the goal state then **h(n)=0**

Greedy best-first search expands the node that appears to be closest to goal; it expands the closest node to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: **f(n)=h(n)**

2. Greedy search: the Romania example



Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244

Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h_{SLD} = straight-line distance heuristic from a city to Bucharest

h_{SLD} can **NOT** be computed from the problem description itself

Examples:

h(Arad)=366

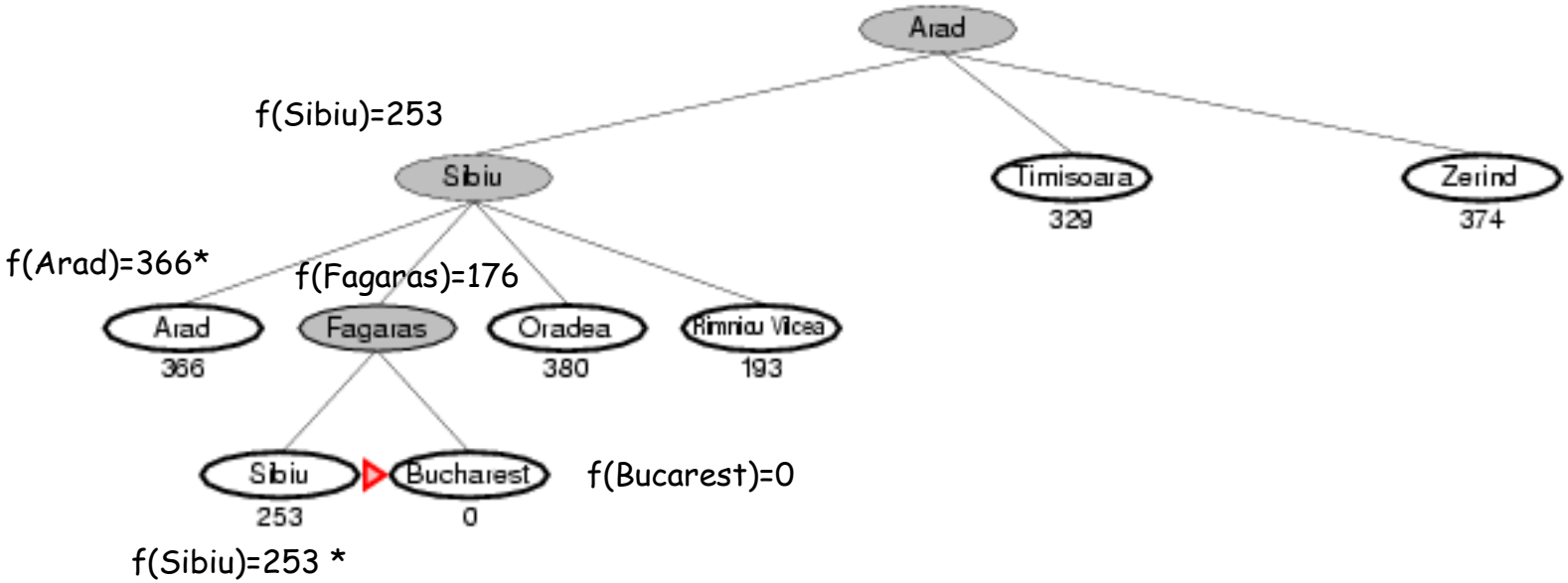
h(Fagaras)=176

h(Bucharest)=0

2. Greedy search: the Romania example

In this example f(n)=h(n)

- expand node that is closest to goal
- greedy best-first search



Reached goal: non-optimal solution
(alternative solution: Arad – Sibiu - Rimnicu Vilcea – Pitesti - Bucharest)

* By using the GRAPH-SEARCH version, we could check repeated states in the CLOSED list and the new appearance of the Arad node would not be inserted in the OPEN list.

2. Greedy search: evaluation

- Complete:
  - NO, it can get stuck in loops, e.g. (consider the problem to go from Iasi to Fagaras): Iasi → Neamt → Iasi → Neamt (dead end, infinite loop)
  - Resembles depth-first search (it prefers to follow a single path to the goal)
  - The GRAPH-SEARCH version is complete (check on repeated states in the CLOSED LIST)
- Optimal:
  - No, at each step it tries to get as close to the goal state as it can; this is why it is called greedy!
- Time complexity (execution time):
  - The worst-case time complexity is $O(b^m)$ where m is the maximum depth of the search space
  - Like worst case in depth-first search
  - Good heuristic can give dramatic improvement
  - The amount of the reduction depends on the particular problem and on the quality of the heuristic
- Space complexity (memory requirements):
  - $O(b^m)$ where m is the maximum depth of the search space

3. A* search

A* search is the most widely-known form of best-first search

It evaluates nodes by combining $g(n)$, the cost to reach the node **n**, and $h(n)$, the cost to get from the node to the goal state. $\mathbf{f(n)=g(n)+h(n)}$

The idea is to avoid paths that are already expensive. **f(n)** is the estimated total cost of the cheapest solution through **n**.

Algorithms that use an evaluation function of the form $\mathbf{f(n)=g(n)+h(n)}$ are called A algorithms

3. A* search

A* search uses an admissible heuristic function

A heuristic is admissible if it never overestimates the cost to reach the goal, i.e., it is optimistic

Formally:

- a heuristic $h(n)$ is admissible if for every node n, $\mathbf{h(n) \leq h^*(n)}$, where $h^*(n)$ is the true cost to reach the goal state from n
- Because A* search uses an admissible heuristic, it returns the optimal solution
- $h(n) >= 0$ so $h(G)=0$ for any goal G

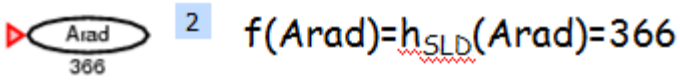
e.g. $h_{SLD}(n)$ never overestimates the actual road distance

3. A* search: the Romania example

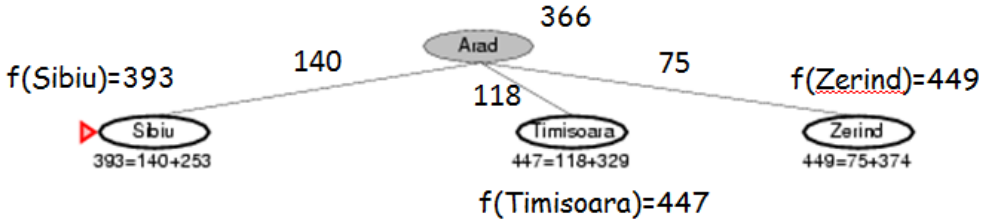
In this example $f(n)=g(n)+h(n)$

- $h(n)=h_{SLD}(n)$
- expand node with the lowest estimated total cost to the goal state
- A* search

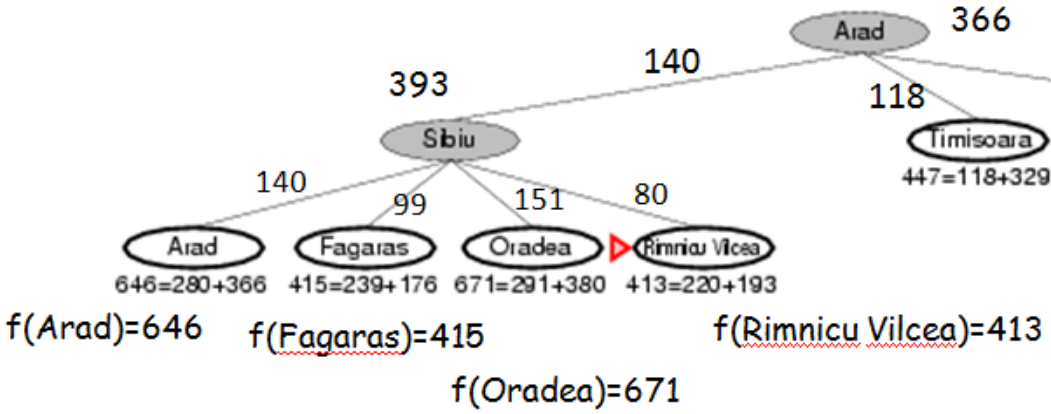
Iteration 1:



Iteration 2:



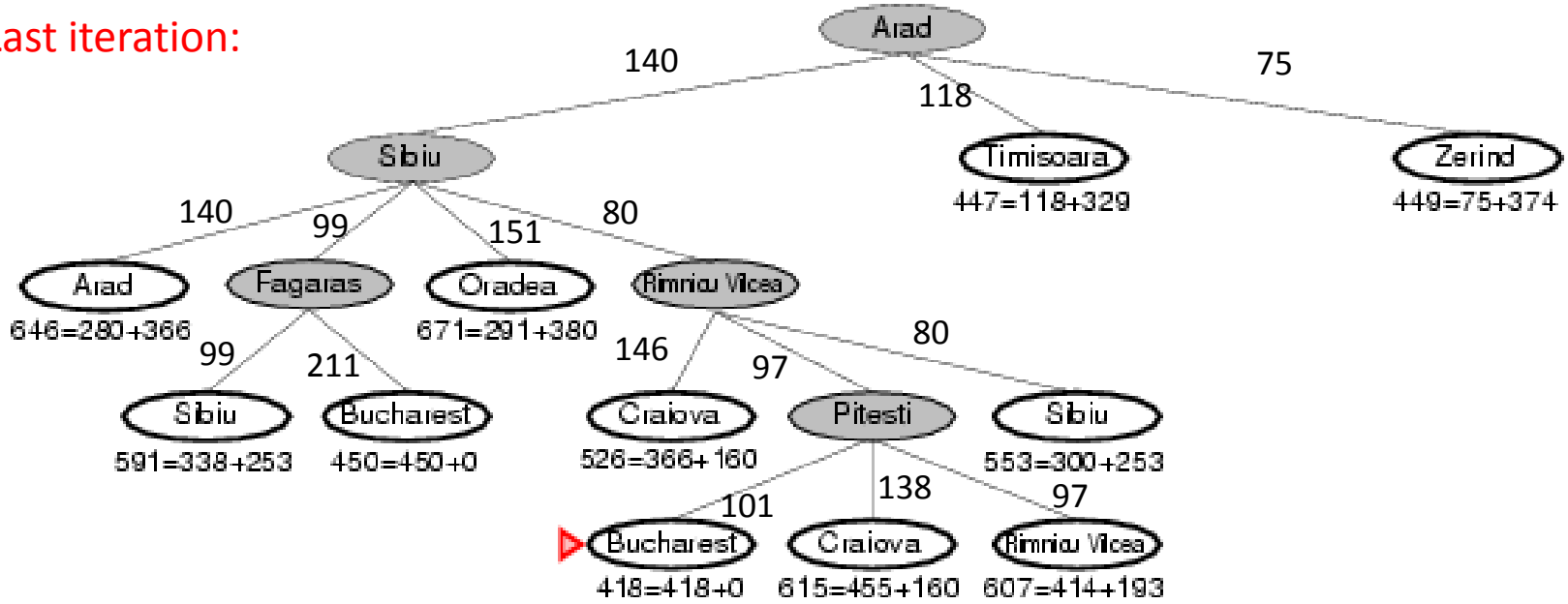
Iteration 3:



In GRAPH-SEARCH version: Arad is a repeated node; the Arad node in the CLOSED list has a better f-value.

3. A* search: the Romania example

Last iteration:

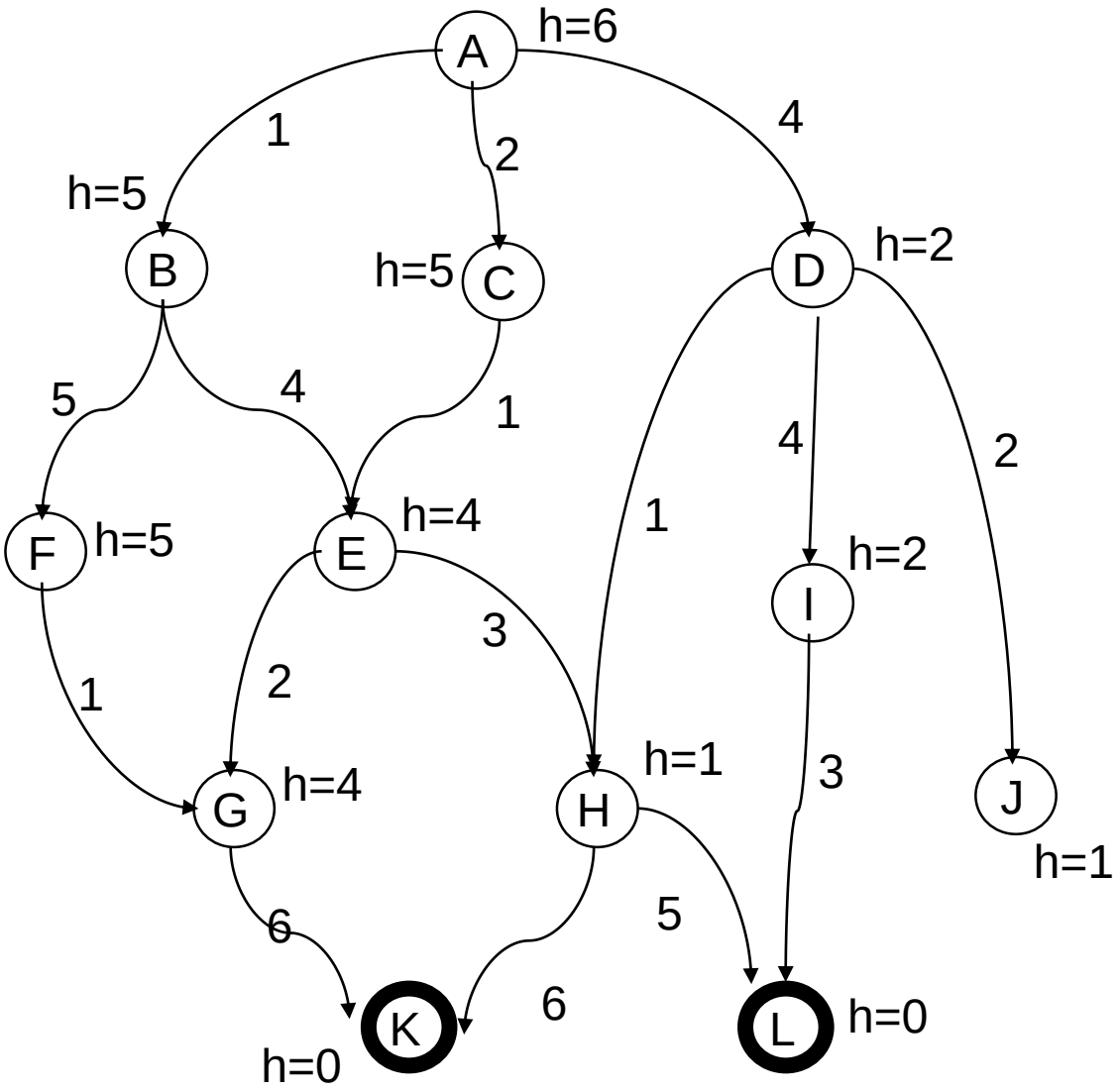


OPEN list= {Bucharest(418), Timisoara(447), Zerind(449), Craiova(526), Oradea (671)}

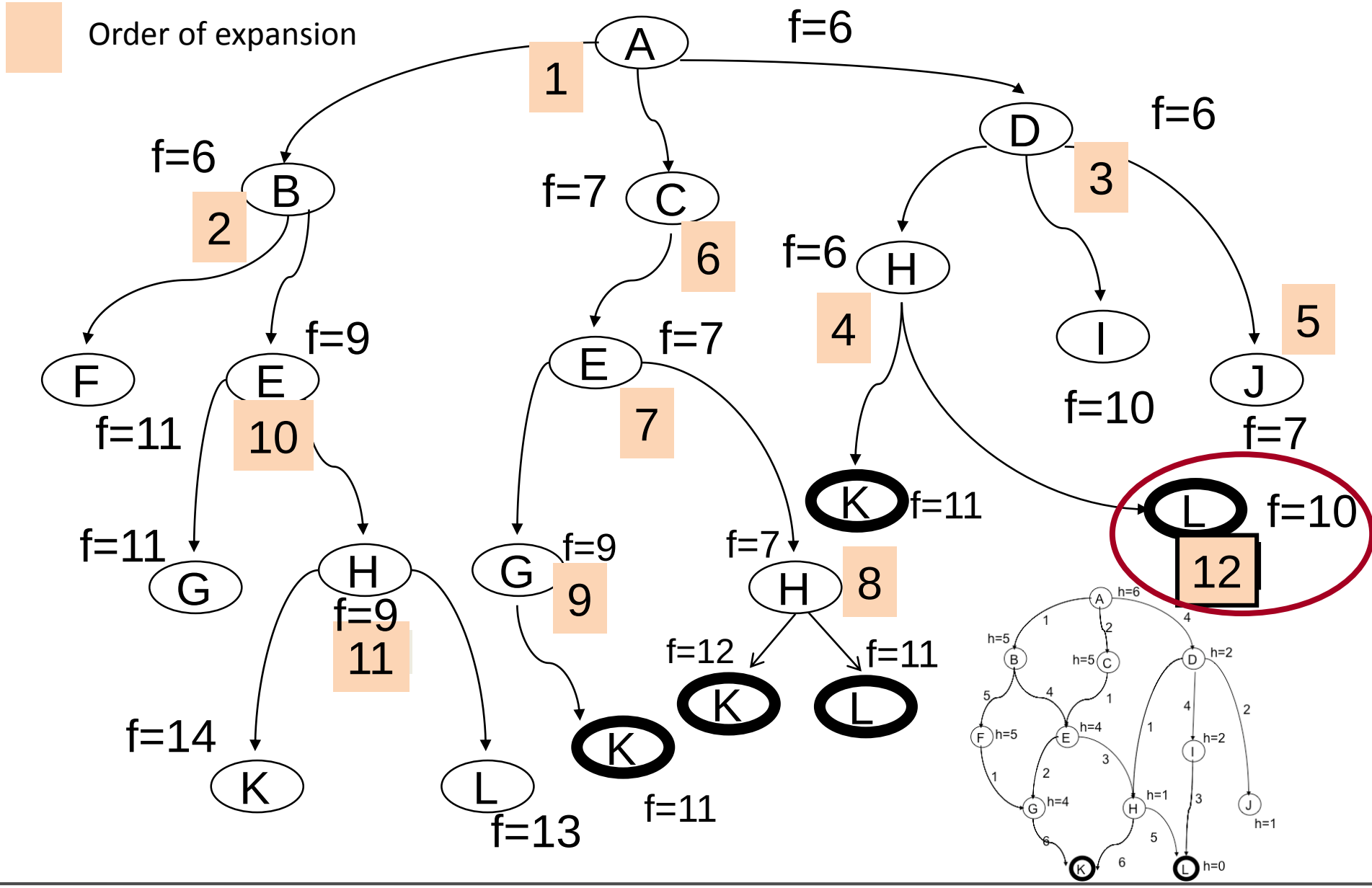
CLOSED list = {Arad, Sibiu, Rimnicu Vilcea, Fagaras, Pitesti}

Bucharest is already in the OPEN LIST (cost=450). The newly generated node has a lower estimated total cost (f-value=418) than the one in the OPEN LIST. We replace the Bucharest node in OPEN with the new found instance.

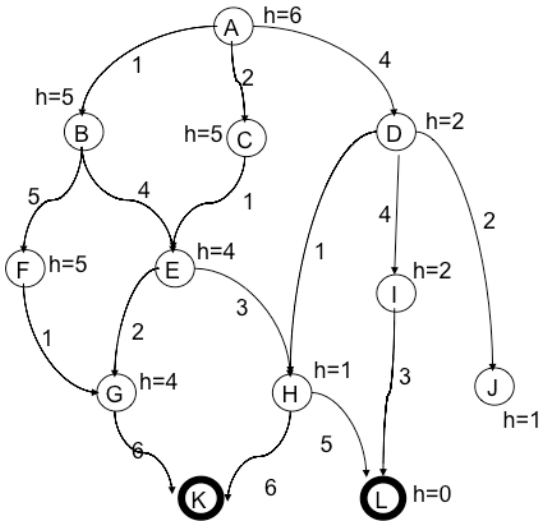
3. A* search: another example



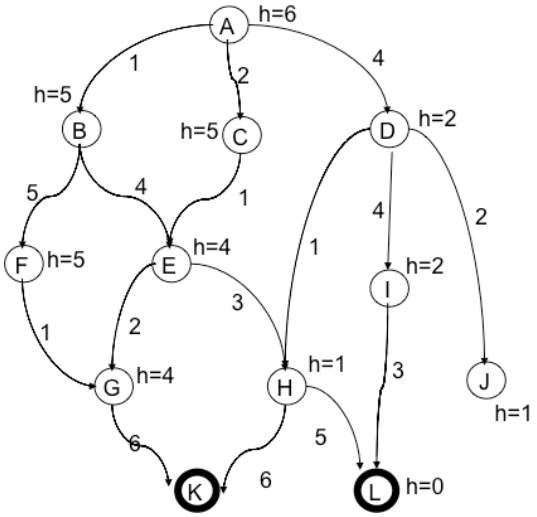
3. A* search: TREE-SEARCH version without control of repeated states



3. A* search: TREE-SEARCH version with control of repeated states



3. A* search: GRAPH-SEARCH version

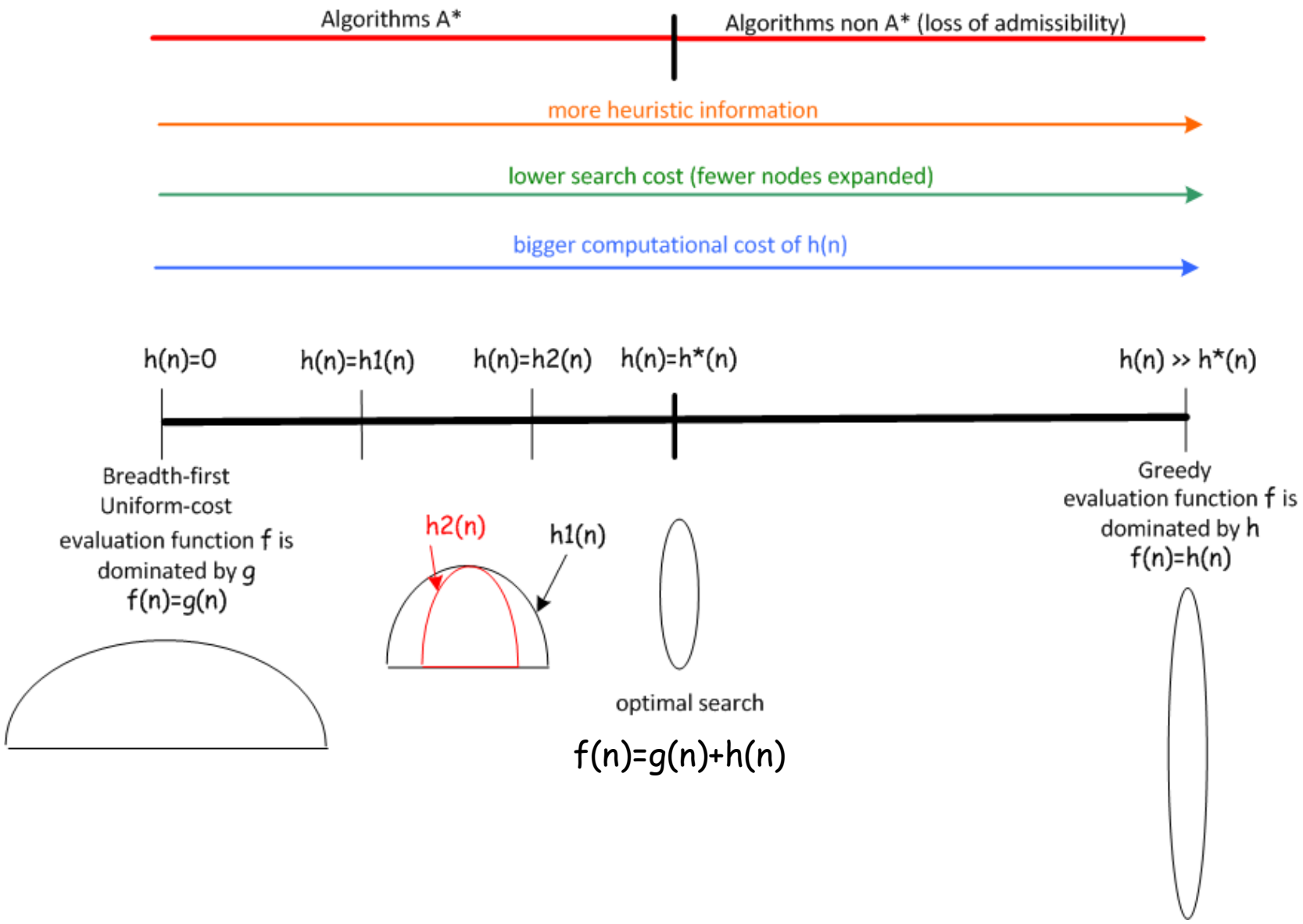


3. A* search: comparison and analysis

- Comparison to other search strategies:
  - Breadth-first search (optimal if all operators have the same cost). Equivalent to $f(n)=level(n)+0$, where $h(n)=0 \leq h^*(n)$
  - Uniform-cost (optimal). Equivalent to $f(n)=g(n)+0$ where $h(n)=0 \leq h^*(n)$
  - Depth-first (non-optimal). Not comparable to A*

- Heuristic knowledge:
  - $h(n)=0$, lack of knowledge
  - $h(n)=h^*(n)$, maximal knowledge
  - if $h2(n) \geq h1(n)$ for all n (both admissible) then $h2$ dominates $h1$ ($h2$ is more informed than $h1$); $h2$ will never expand more nodes than $h1$

3. A* search: comparison and analysis



3. A* search: comparison and analysis

h(n) = 0: no computational cost of h(n). (Slow search. Admissible.)
h(n)=h*(n): large computational cost of h(n). (Fast search. Admissible).
h(n)> h*(n): very high computational cost of h(n). (Very fast search. No admissible)

In general, h^* is not known but it is possible to state whether h is a lower bound of h^* or not.

For difficult problems, it is recommended to highly reduce the search space. In these cases, it is worth using $h(n) > h^*(n)$ to eventually find a solution at a reasonable cost even though this is not the optimal solution.

For each problem, find a balance between the search cost and the cost of the solution path found

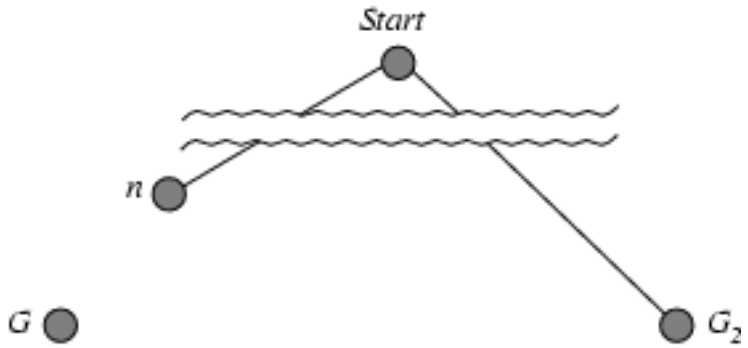
3. A* search: optimality

The optimality of a TREE-SEARCH A* algorithm is guaranteed if h(n) is an admissible heuristic

Admissibility:

- h(n) is admissible if it never oversestimates the cost to reach the goal ($\forall n,\ h(n) \leq h^*(n)$)
- Let G be a goal/objective node, and n a node on an optimal path to G. $f(n)$ never overestimates the true cost of a solution through n
  - $f(n) = g(n)+h(n) \leq g(n)+h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Prove that if h(n) is admissible, then A* is optimal



- Let *Start* be the initial state of a problem such that the optimal solution from *Start* leads to a solution node G: $f(G) = g(G)$
- Let $G2$ be another solution node ($f(G2) = g(G2)$) such that the cost of $G2$ is higher than the cost of G : $g(G2) > g(G)$ **(2)**
- Let n be a node on the optimal path to G
- Let's suppose that both n and $G2$ are in OPEN

If n is not chosen for expansion then $f(n) >= f(G2)$ **(3)**

If we combine **(1)** and **(3)** we get

$f(G2) <= f(n) <= g(G) \Rightarrow g(G2) \leq g(G)$

this contradicts **(2)** so n is chosen first for expansion

Therefore, the algorithm will return the optimal solution

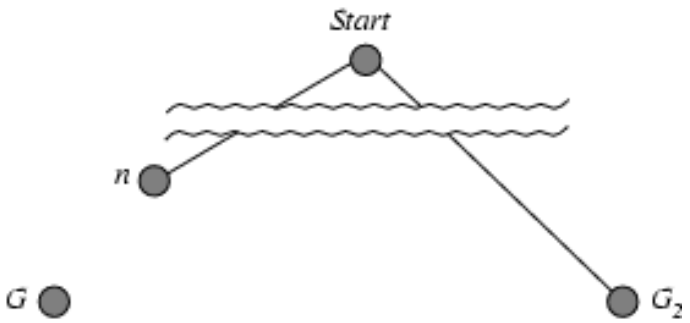
3. A* search: optimality

La optimalidad de un algoritmo A* TREE–SEARCH se garantiza si $h(n)$ es una heurística admisible

Admisibilidad:

- h(n) es admisible si $\forall n,\ h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G. $f(n)$ nunca sobreestima el coste del camino a través de n.
  - $f(n) = g(n)+h(n) \leq g(n)+h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Probar que si h(n) es admisible entonces A* es óptimo:



- Sea *Start* el estado inicial de un problema tal que la solución óptima desde *Start* conduce a un nodo solución G: $f(G) = g(G)$
- Sea $G2$ otro nodo solución ($f(G2) = g(G2)$) tal que el coste de $G2$ es mayor que el de G : $g(G2) > g(G)$ **(2)**
- Sea n un nodo de la lista OPEN en el camino óptimo a G. Y supongamos que n y $G2$ están en la lista OPEN

Si no se escoge n para expansión entonces $f(n) >= f(G2)$ **(3)**

Si combinamos **(3)** y **(1)** :

$f(G2) <= f(n) <= g(G) \Rightarrow g(G2) \leq g(G)$

Esto contradice **(2)** así que se escoge n

Por tanto, el algoritmo devolverá la solución óptima

3. A* search: optimality

The optimality of a GRAPH-SEARCH A* algorithm where repeated states in CLOSED are re-expanded is guaranteed if $h(n)$ is an admissible heuristic.

In a GRAPH-SEARCH A* algorithm with no re-expansion of repeated states in CLOSED, the optimal solution is guaranteed if we can ensure the first found path to any node is the best path to the node. This occurs if $h(n)$ is a consistent heuristic.

Consistency, also called monotonicity is a slightly stronger condition than admissibility:

$h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a:

$$h(n) <= h(n')+c(n,a,n')$$

Therefore, a GRAPH-SEARCH A* algorithm with no re-expansion of repeated states that uses a consistent heuristic $h(n)$ also returns the optimal solution:

- because it is guaranteed that when a node n is expanded, we have found the optimal solution from the initial state to n so it is not necessary to re-expand n in case of a repeated instance; i.e. the cost of the first instance of n will never be improved by any repeated node (they will be at least as expensive).
- a derivation of the consistency condition is that the sequence of expanded nodes by the GRAPH-SEARCH version of A* is in non-decreasing order of $f(n)$

3. A* search: evaluation

- Complete:
  - YES; if there exists at least one solution, A* finishes (unless there are infinitely many nodes with $f < f(G)$)
- Optimal:
  - YES, under consistency condition (for GRAPH-SEARCH version)
  - There always exists at least a node **n** on the OPEN LIST that belongs to the optimal solution path from the start node to a goal node
  - If C^* is the cost of the optimal solution:
    - A* expands all nodes with $f(n) < C^*$
    - A* might expand some nodes with $f(n)=C^*$
    - A* expands no nodes with $f(n) > C^*$
- Time complexity (execution time):
  - $O(b^{C^*/\text{min\_action\_cost}})$: exponential with path length
  - Number of nodes expanded is still exponential in the length of the solution
- Space complexity (memory requirements):
  - Exponential: it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms)
  - A* usually runs out of space long before it runs out of time
  - Hence, space is the major problem, not time

4. Design of heuristic functions

Heuristics are problem-dependent functions.

Given a particular problem, how might one come up with a heuristic function for the problem? In case of using an algorithm A*, how can one invent an admissible heuristic?

Common technique: ***Relaxed problem***

A problem with less restrictions on the operators is called a relaxed problem.

Admissible heuristics can be derived from the cost of an **exact solution to a relaxed version of the problem**. This is usually a good heuristic for the original problem.

4.1 Heuristics for the 8-puzzle problem

A tile in square A can move to square B if:

Restriction 1: B is adjacent to A
Restriction 2: B is the empty space (blank)

h1: misplaced tiles

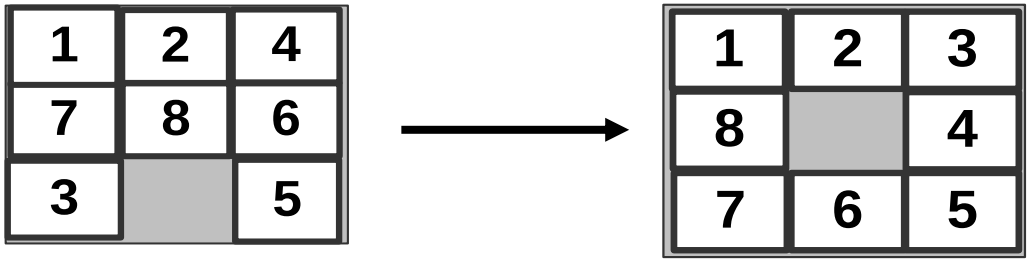
- It removes both restrictions
- Relaxed 8-puzzle for h1 : a tile can move anywhere
- As a result, h1(n) gives a very optimistic estimate (shortest solution)

h2: Manhattan distance

- It removes Restriction 2
- Relaxed 8-puzzle for h2: a tile can move to any adjacent square
- As a result, h2(n) gives an optimistic estimate (shortest solution)

The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

4.1 Heuristics for the 8-puzzle problem



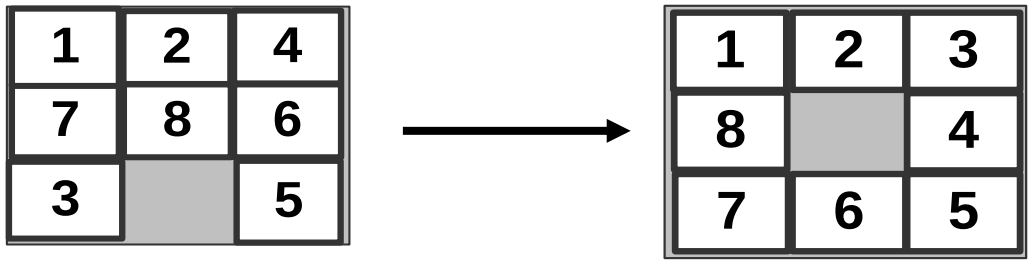
Initial state

Goal state

- The 8-puzzle problem
  - Average solution cost is about 22 steps (branching factor ≈ 3)
  - Exhaustive search to depth 22: $3^{22} \approx 3.1 \times 10^{10}$ states.
  - A good heuristic function can reduce the search process.

- **Heuristic 1:**
  - h1(n): number of misplaced tiles
  - h1(n)=5 for the example

4.1 Heuristics for the 8-puzzle problem



Initial state

Goal state

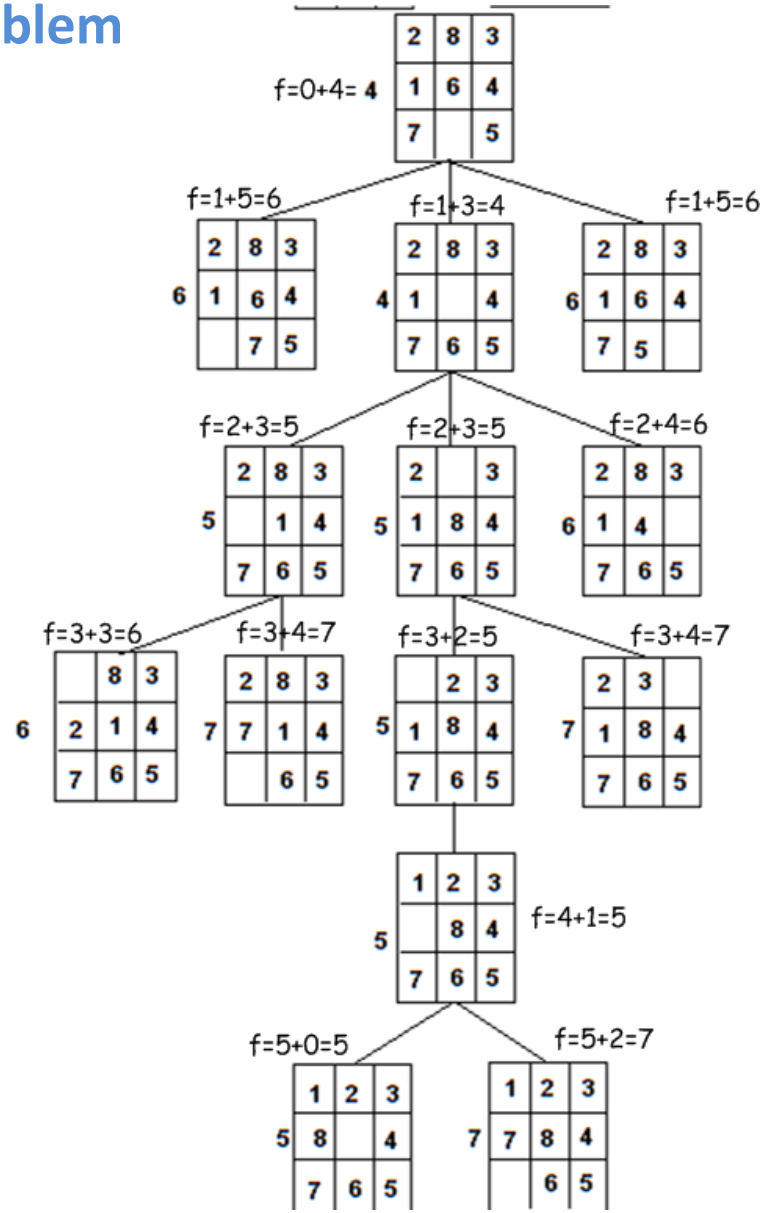
- **Heuristic 2:**
  - h2(n): Manhattan distance (the sum of the distances of the tiles from their goal positions)
  - h2(n)=1+2+4+1+1=9 for the example

Manhattan distance dominates misplaced tiles **(h2(n)>= h1(n), ∀n)**

h2 is better for search

4.1 Heuristics for the 8-puzzle problem

A* search (f(n)=g(n)+h(n)) using
h(n)= misplaced tiles

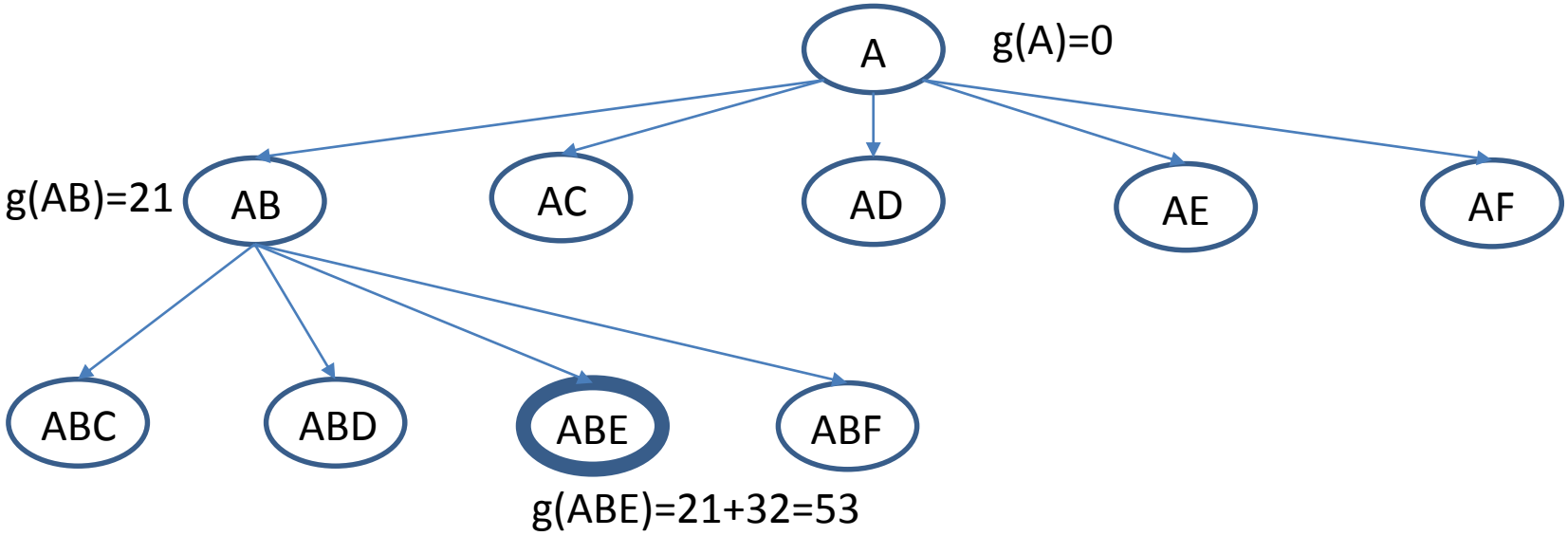


4.2 Heuristics for the traveling salesman problem

6 cities: A,B,C,D,E,F
States: sequences of cities starting in city A which represent partial routes
Start: A
Final: sequences starting and ending in A which are made up of all cities
Rules or operators: add at the end of each state one city which is not in the sequence already
Operators cost: distance between the last city in the sequence and the recently added city (see table)

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2 Heuristics for the traveling salesman problem



Cities that have not been reached yet: C, D, F, and back to A

h*(ABE) = min(ECDFA, ECFDA, EDCFA, EDFCA, EFCDA, EFDCA)
min(154, 92, 297, 251, 57, 73)

4.2. Heuristics for the traveling salesman problem

h(ABE) →
C, D, F, back to A

h2(n)= number of missing arcs multiplied by the arc of minimal cost

[h2(ABE)=4*5=20]

h3(n)= sum of the shortest *p* arcs if there are *p* missing arcs (undirected graph)

[h3(ABE)=5+5+7+7=24]

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heuristics for the traveling salesman problem

h(ABE) →
C, D, F, back to a A

h4(n)= sum of the shortest outgoing arcs of the cities which have not been left yet

[h4(ABE)=17 {leave E} +
5 {leave C} +
5 {leave D} +
9 {leave F} =36]

h7(n)= number of missing arcs multiplied by their average cost (non-admissible)

[h7(ABE)=4*40.33=161.33]

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

5. Evaluation of heuristic functions: computational cost

Difficult to apply mathematical analysis

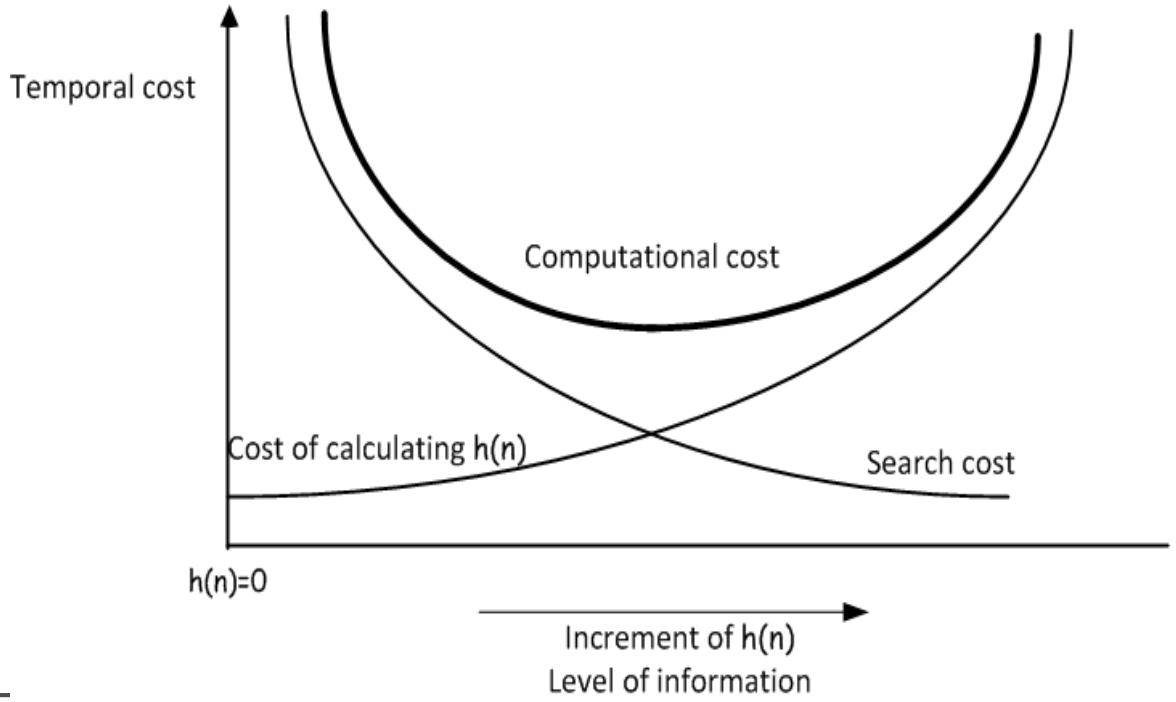
Often used statistical/experimental methods.

Objective: minimize the total cost of the problem, trading off computational expense and path cost

Computational cost (temporal cost):

Search cost: number of nodes generated or applied operators **+**

Cost of calculating h(n): cost for selecting the applicable operator.



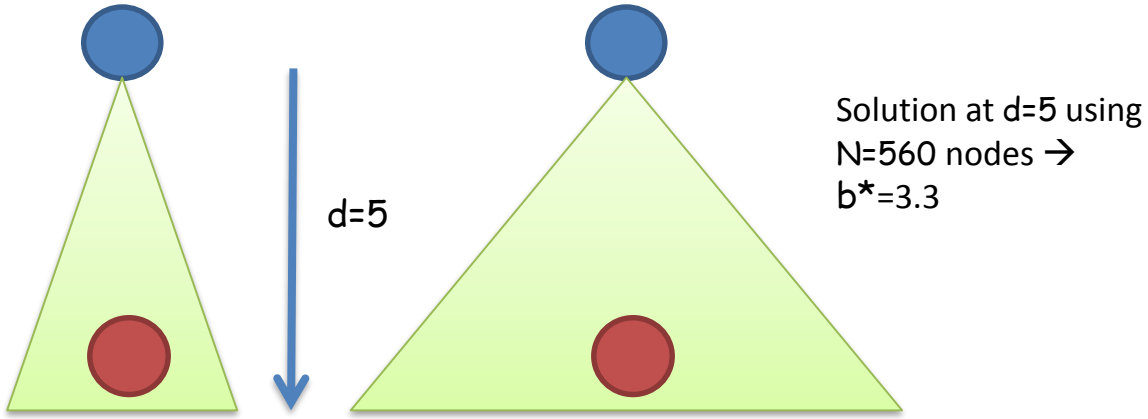
5. Evaluation of heuristic functions

Effective branching factor (b*)

If the total number of nodes generated by A* for a particular problem is N, and the solution depth is d, then b* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

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

For example, if A* finds a solution at depth 5 using 52 nodes, $b^*=1.92$



5. Evaluation of heuristic functions

- b* defines how sharply a search process is focussed toward the goal.
- b* is reasonably independent of path length (d)
- b* can vary across problem instances, but usually is fairly constant for sufficiently hard problems

- A well-designed heuristic would have a value of b* close to 1, allowing fairly large problems to be solved.
- A value of b* near unity corresponds to a search that is highly focussed toward the goal, with very little branching in other directions.
- Experimental measurements of b* on a small set of problems can provide a good guidance to the heuristic's overall usefulness