ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

04

Chapiter

Informed Search

2I1AE1: Artificial Intelligence

Régis Clouard, ENSICAEN - GREYC

"On fait la science avec des faits, comme on fait une maison avec des pierres : mais une accumulation de faits n'est pas plus une science qu'un tas de pierres n'est une maison."
Henri Poincaré

Today

- **Informed Search Algorithms**
  - Algorithms that can find a solution when the brute force algorithms failed due to the size of the search tree.

1) Best-First Search Algorithms
   - Greedy Best-First Algorithm
   - A* Algorithm
   - IDA* Algorithm
2) Heuristics

How to Speed Up the Search ?

- Uninformed search algorithms
  - Use only the information about the problem definition and past explorations e.g. cost of the path generated so far.
  - Hint to speed up the search process:
    - Bi-directional search
    - Closed-list (however, makes all algorithms exponential in worst-case space complexity, but makes them often better in average-case complexity)

- Informed search algorithms
  - Incorporate additional measure of a **potential of a state** to reach the goal.
    - Best-first search algorithms
  - **Caveat**: This has no influence on the solution itself, only on the search speed.

Best-First Search

- **Evaluation function,** denoted *f(n)*
  - Defines the **desirability** of a node *n* to be expanded next.
- Evaluation-function driven search
  - Among all candidates, expand first the node with the **best evaluation-function value *f(n)***.
- Implementation: same algorithm as for uninformed search:
  - **ADD-IN-LIST**: uses a **priority queue** that orders nodes in the decreasing order of their evaluation function value.

```
function GENERAL-SEARCH(problem) returns solution
  open-list ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  LOOP
     IF EMPTY(open-list) THEN return failure
     node ← REMOVE-FRONT-LIST(open-list)
     IF IS-GOAL(problem, STATE[node]) THEN return the related solution
     open-list ← ADD-IN-LIST(GET-SUCCESSORS(node, problem), open-list)
end
```

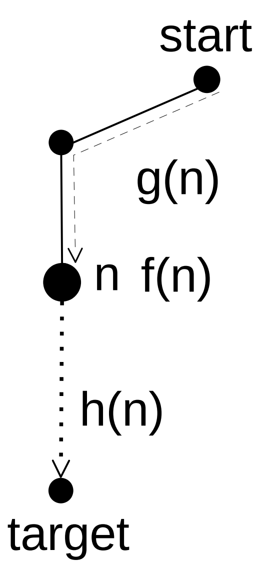
Evaluation Function

- Idea
  - Incorporate a **heuristic function** *h(n)* into the **evaluation function** *f(n)* to guide the search.

- Heuristic function
  - Measures a potential of a node to reach a goal.
    - Typically in terms of some distance to a goal.
  - Problem-dependent : designed for a given search problem.
  - Examples:
    - Traveler problem: the straight-line distance between the current city and the goal city.
    - Puzzle-8: number of well placed tiles.
- Recall: This heuristic has no influence on the result, only on the search speed.

Best-First Algorithms

- General formulation of the best-first algorithms
  - Algorithms differ in the design of evaluation function *f(n)*.
  - Notation
    - *f(n)*: estimated cost of the cheapest solution through *n*.
    - *g(n)*: path cost from the start to the node *n*.
    - *h(n)*: estimated cost of the cheapest path from *n* to the target.



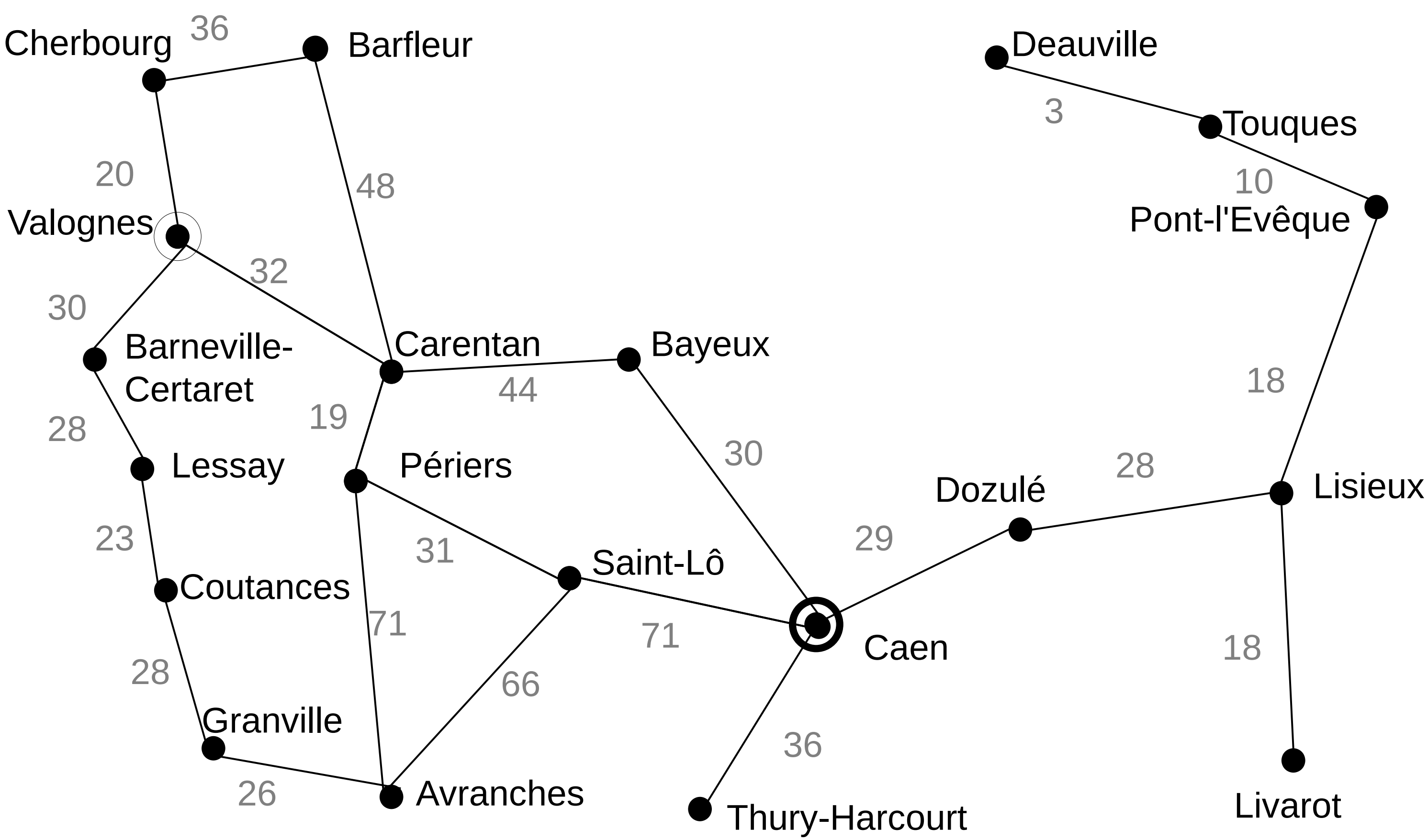
- Recall:
  - **Breadth-First Search**
    - $f_{BFS}(n)$ = visited time
  - **Depth-First Search**
    - $f_{DFS}(n)$ = - visited time
  - **Uniform Cost Search algorithm (aka Dijkstra)**
    - $f_{UCS}(n) = g(n)$

1. Greedy Best-First search Algorithm

- **Strategy**: Greedy best-first search algorithm expands the node that appears to be closest to goal.
  - Making the locally optimal choice at each stage.
  - Models the naive strategy: immediate gain.
- **Evaluation function**
  - $f(n) = h(n)$: estimate of cost from n to goal.

Traveler example

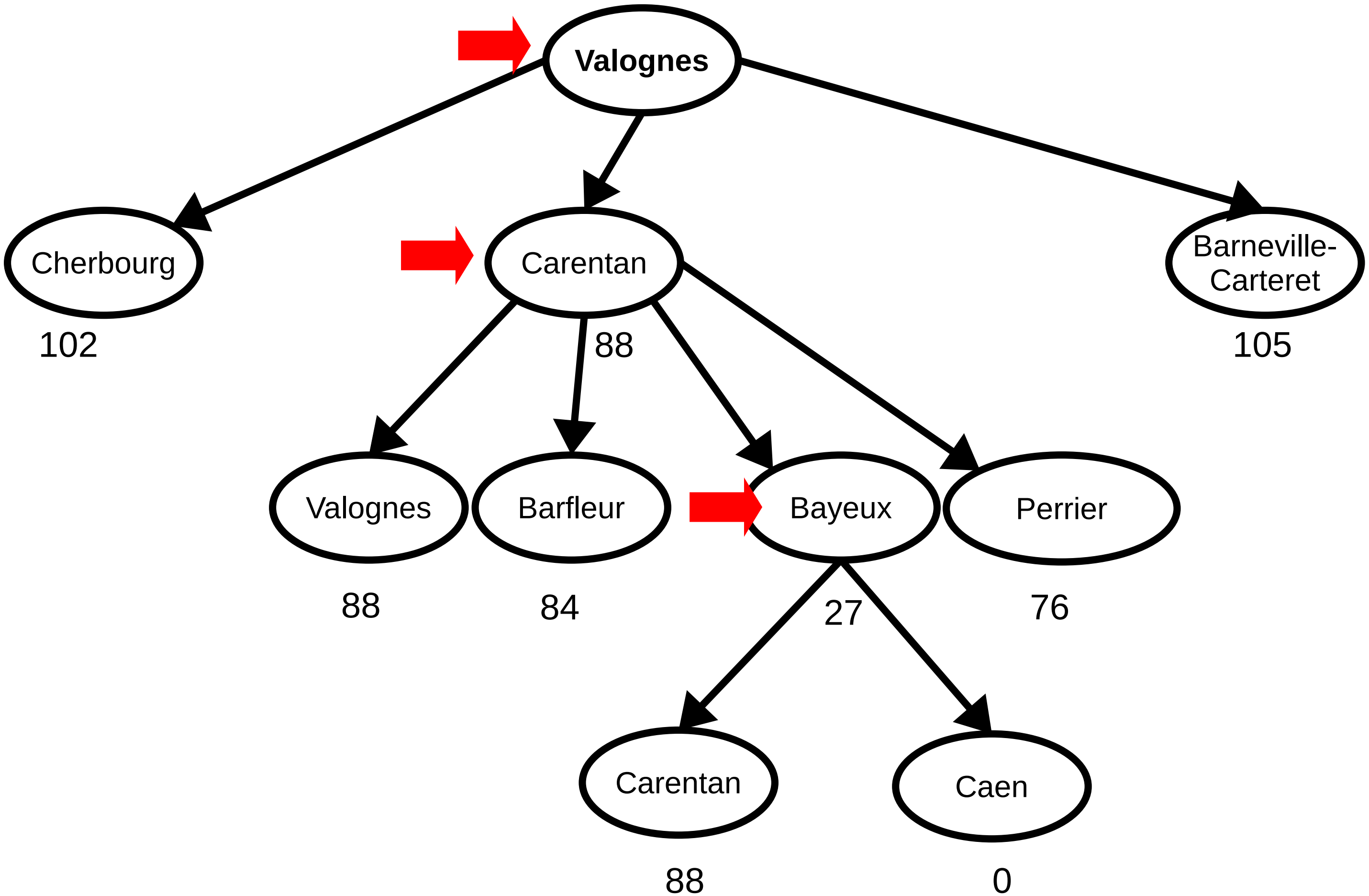
- Normandy map with straight-line distance to Caen



Town	Straight-line distance to Caen
Avranches	92
Barfleur	84
Barneville-Carteret	105
Bayeux	27
Caen	0
Carentan	88
Cherbourg	102
Coutances	79
Deauville	36
Dozulé	23
Granville	98
Lessay	83
Lisieux	43
Livarot	43
Périers	76
Pont-l'Évêque	40
Saint-Lô	53
Thury-Harcourt	23
Touques	37
Valognes	87

Greedy Best-First Search Example

e.g., *h(n)* = straight-line distance from *n* to Caen.



[gbfs1.sh; gbfs2.sh]

Properties of Greedy Search

- **Completeness**
  - No. Can get stuck in local minimum (eg. Knuth's conjecture with distance to expected number as heuristic).
- **Optimality**
  - No. Evaluation function disregards the real cost of the path. The minimum path is not necessarily the one with the minimum heuristic. The path cost is never used.
- **Time complexity**
  - *Worst-case* $O(b^m)$. Same as DFS but often better!
  - Best case: $O(bd)$ – If $h(n)$ is 100% accurate.
- **Space complexity**
  - *Worst-case* $O(b.m)$
  - Notice: with the closed list -> $O(b^m)$.

Recall

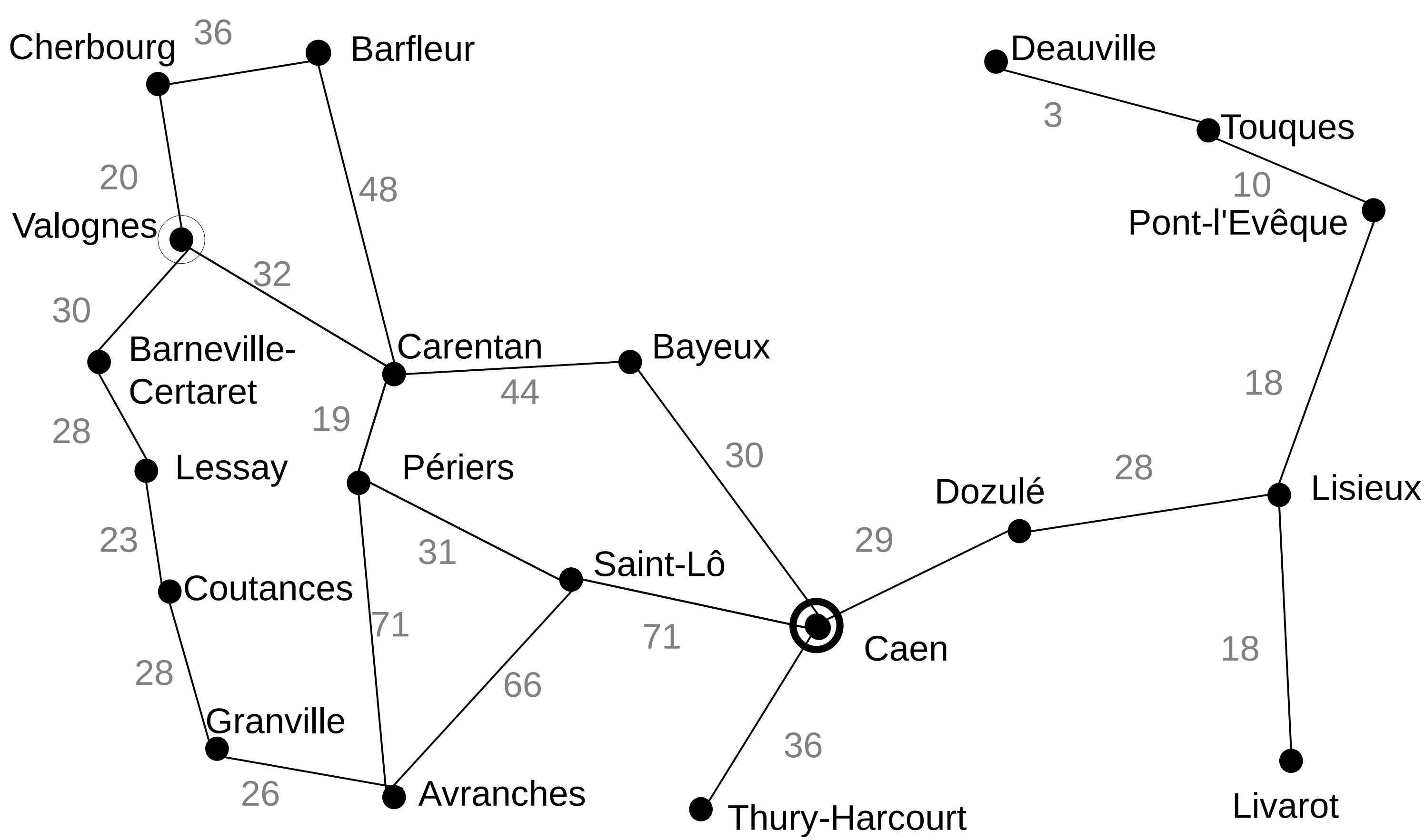
- *b: maximum branching factor*
- *d: depth of the optimal solution*

2. A* Search Algorithm

- The problem with the uniform-cost search is that it uses only past exploration information (path cost), no additional information is utilized.
  - $f(n) = g(n)$
- The problem with the greedy search is that it does not take into account the cost so far. It can keep expanding paths that can be already very expensive.
  - $f(n) = h(n)$
- A*: search takes into account both information
  - $f(n) = g(n) + h(n)$

Traveler example

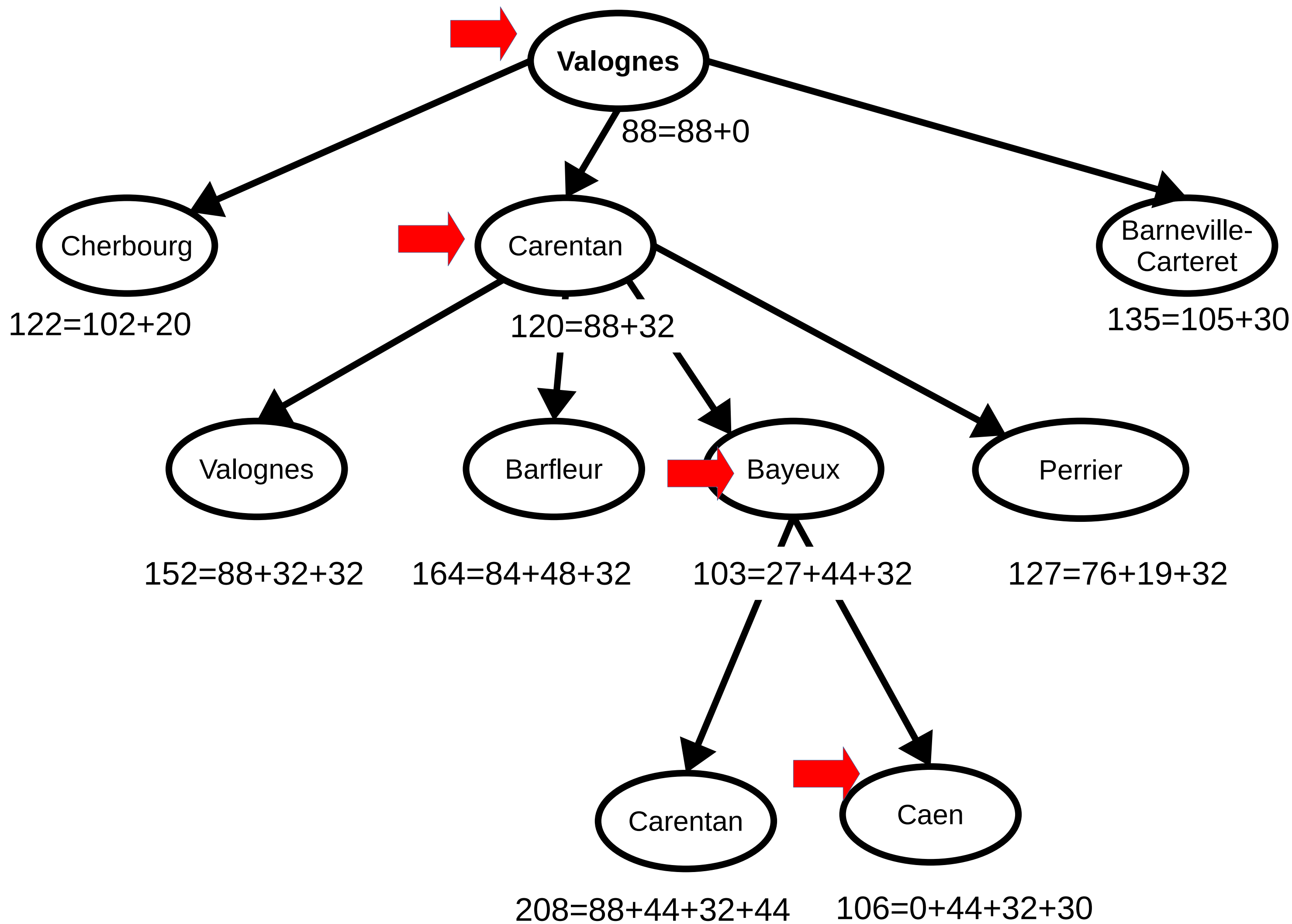
- Normandy map



Town	Straight-line distance to Caen
Avranches	92
Barfleur	84
Barneville-Carteret	105
Bayeux	27
Caen	0
Carentan	88
Cherbourg	102
Coutances	79
Deauville	36
Dozulé	23
Granville	98
Lessay	83
Lisieux	43
Livarot	43
Périers	76
Pont-l'Évêque	40
Saint-Lô	53
Thury-Harcourt	23
Touques	37
Valognes	87

A* Search Example

e.g., *h(n)* = straight-line distance from *n* to Caen.



[ass1.sh; ass2.sh]

Properties of A*

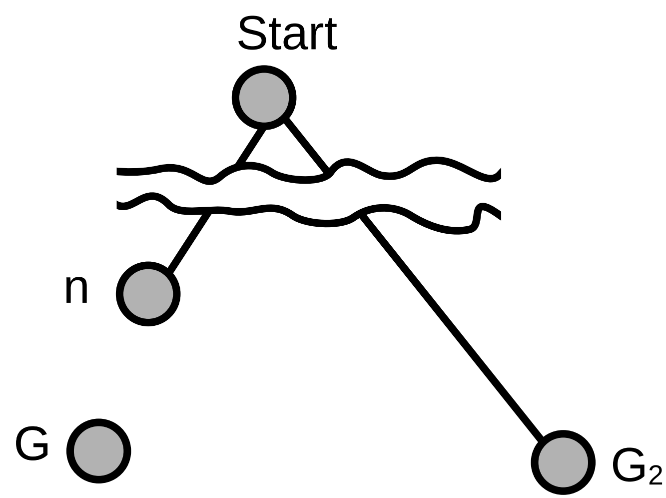
- **Completeness**
  - Yes, but only with non negative edge weights.
- **Optimality**
  - Yes in term of solution cost g(n), but only with **admissible heuristic**.
- **Time complexity**
- **Space complexity**

Admissible Heuristic

- Definition
  - A heuristic *h(n)* is admissible if:
    $\forall n,\ h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
  - An admissible heuristic never **overestimates** the cost to reach the goal, i.e., it is **optimistic**.
  - Worst-cases
    - if $h(n) = cst,\ \forall n$, this is equivalent to BFS.
    - If *h(n)* is not admissible this is equivalent to DFS.
- Example
  - h(n): straight-line distance is admissible because it underestimates the actual road distance (Euclide!).
- Theorem
  - If *h(n)* is admissible, A* is optimal.

Proof of optimality

- Suppose some suboptimal goal G_2 has been generated and is in the queue.
- Let *n* be an unexpanded node in the queue such that *n* is on a shortest path to an optimal goal *G*.



$f(G_2) = g(G_2) + h(G_2)$ by definition		
$f(G_2) = g(G_2)$		since G_2 is a goal $\rightarrow h(G_2) = 0$
$f(G) = g(G)$		since G is a goal $\rightarrow h(G) = 0$
$g(G_2) > g(G)$		since G_2 is suboptimal
$f(G_2) > f(G)$		
$h(n) \leq h^*(n)$		since h is admissible
$g(n) + h(n) \leq g(n) + h^*(n)$		since $f(G) = g(n) + h^*(n)$
$f(n) \leq f(G)$		since n is a step toward G

Hence $f(G_2) > f(G) \geq f(n)$, and A* will never select G_2 for expansion

Properties of A* Tree Search

- **Completeness**
  - Yes, but only with non negative edge weights.
- **Optimality**
  - Yes in term of solution cost g(n), but only with admissible heuristic.
- **Time complexity**
  - *Worst-case* $O(b^d)$. Exponential in worst case (= BFS).
  - But more interesting: average-case complexity can be close to $O(b.d)$ with good heuristic.
- **Space complexity**
  - *Worst-case* $O(b^{d+1})$. Exponential in worst case (= BFS).

3. IDA* Search Algorithm

- Problem: A* is memory greedy: $O(b^{d+1})$. It maintains all created states in memory (cf. BFS).
- Iterative deepening version of A*
  - Like Iterative Deepening algorithm -> space complexity O(bd).
    - But the cutoff used is the *f(g + h)* rather than the depth.
    - At each iteration, the cutoff value is the smallest *f(n)* of any node *n* that exceeded the cutoff on the previous iteration.
  - We can use the same closed list than IDS.

IDA* Search Algorithm (recursive version)


```
function IDA_STAR(root) returns solution
   bound ← h(root)
   path ← [root]
   LOOP
     t ← SEARCH(path, 0, bound)
     IF t = FOUND THEN return (path, bound)
     IF t = ∞ THEN return NOT_FOUND
     bound ← t
   END
 END

function SEARCH(path, g, bound)
   node ← path.last
   f ← g + h(node)
   IF f > bound THEN return f
   IF is_goal(node) THEN return FOUND
   min ← ∞
   FOR succ in successors(node) DO
     IF succ not in path THEN # avoid loops
       path.push(succ)
       t ← SEARCH(path, g + cost(node, succ), bound)
       IF t = FOUND THEN return FOUND
       IF t < min THEN min ← t
       path.pop()
     END
   END
   return min
END
```

[idas1.sh; idas2.sh]

Properties of IDA*

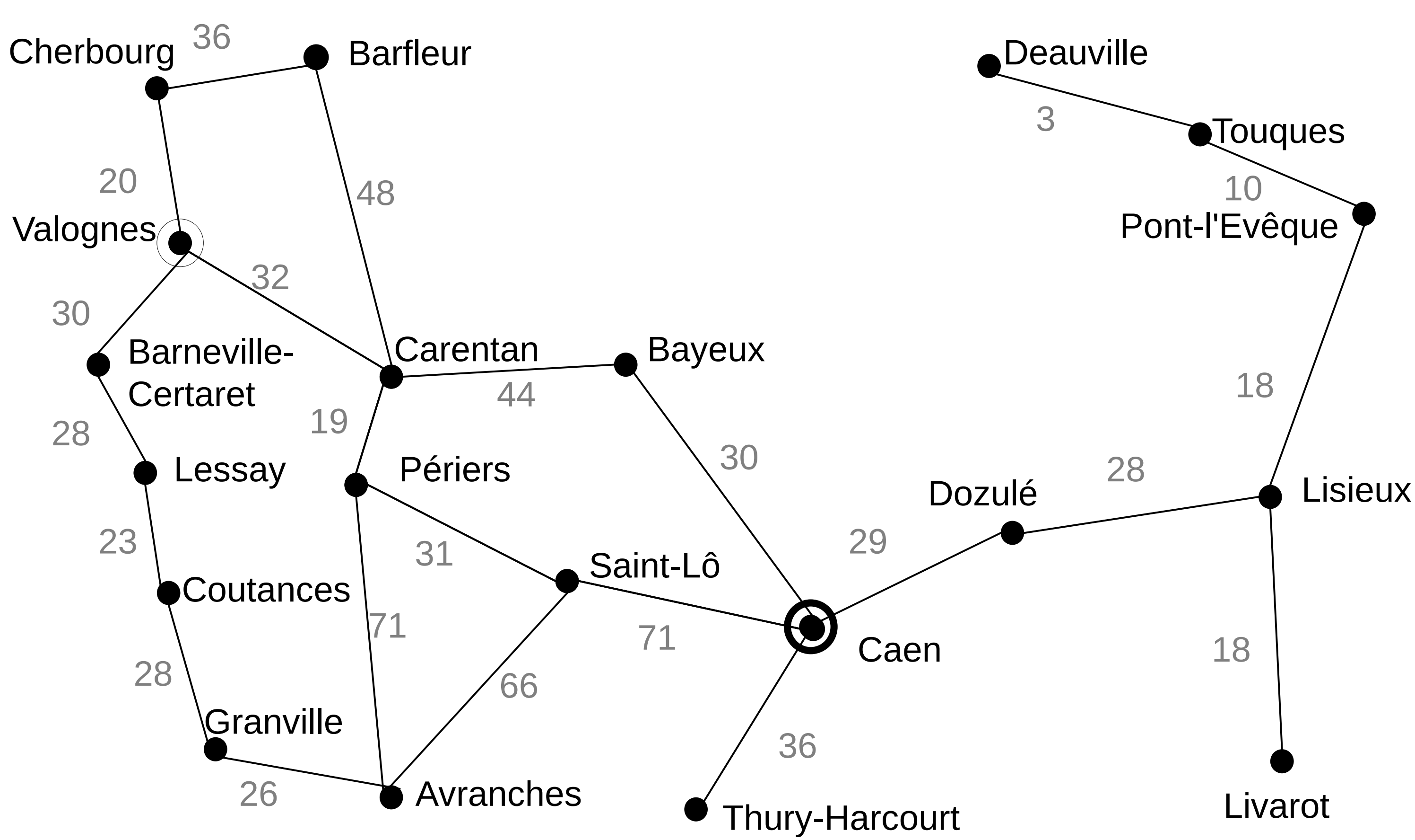
- **Completeness**
  - Yes, but only with non negative edge weights.
- **Optimality**
  - Yes in term of solution cost $g(n)$, with admissible heuristic.
- **Time complexity**
  - *Worst-case $O(b^d)$*
  - *In practice, it has an overcost of 10% compared to A*.*
- **Space complexity**
  - *Worst-case $O(bd)$*

4. How to Create Admissible Heuristics?

- Heuristic
  - Should be fast to compute (polynomial time complexity).
  - Should be close to the actual cost value.
- Recall:
  - Admissible heuristics $h(n) \leq h^*(n)$.
- Admissible heuristics can be derived from the exact solution cost of a **relaxed** version of the problem.
  - Relaxed problem is a problem with fewer restrictions on the actions than the real problem.

Relaxed Problems. Example 1

- Normandy map
  - Relaxed problem
    - No road between cities → straight-line distance.



Town	Straight-line distance to Caen
Avranches	92
Barfleur	84
Barneville-Carteret	105
Bayeux	27
Caen	0
Carentan	88
Cherbourg	102
Coutances	79
Deauville	36
Dozulé	23
Granville	98
Lessay	83
Lisieux	43
Livarot	43
Périers	76
Pont-l'Évêque	40
Saint-Lô	53
Thury-Harcourt	23
Touques	37
Valognes	88

Relaxed Problems. Example 2

- The 8-puzzle problem

Initial position

5	4	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

- Relaxed problem
  - The tile can move *anywhere.*
- Admissible heuristic
  - $h_1(n)$ = Number of misplaced tiles (**Hamming distance**).

Relaxed Problems. Example 3

- The 8-puzzle problem

Initial position

5	4	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

- Relaxed problem
  - The tile can move to *any of adjacent square.*
- Admissible heuristic
  - $h_2(n)$ = Sum of distances of all tiles from their goal positions (**Manhattan distance**).

Admissible Heuristics

- For example:

Initial position

5	4	
6	1	8
7	3	2

Goal position

1	2	3
4	5	6
7	8	

- Value of $h_1(n)$?

  = 7

- Value of $h_2(n)$?

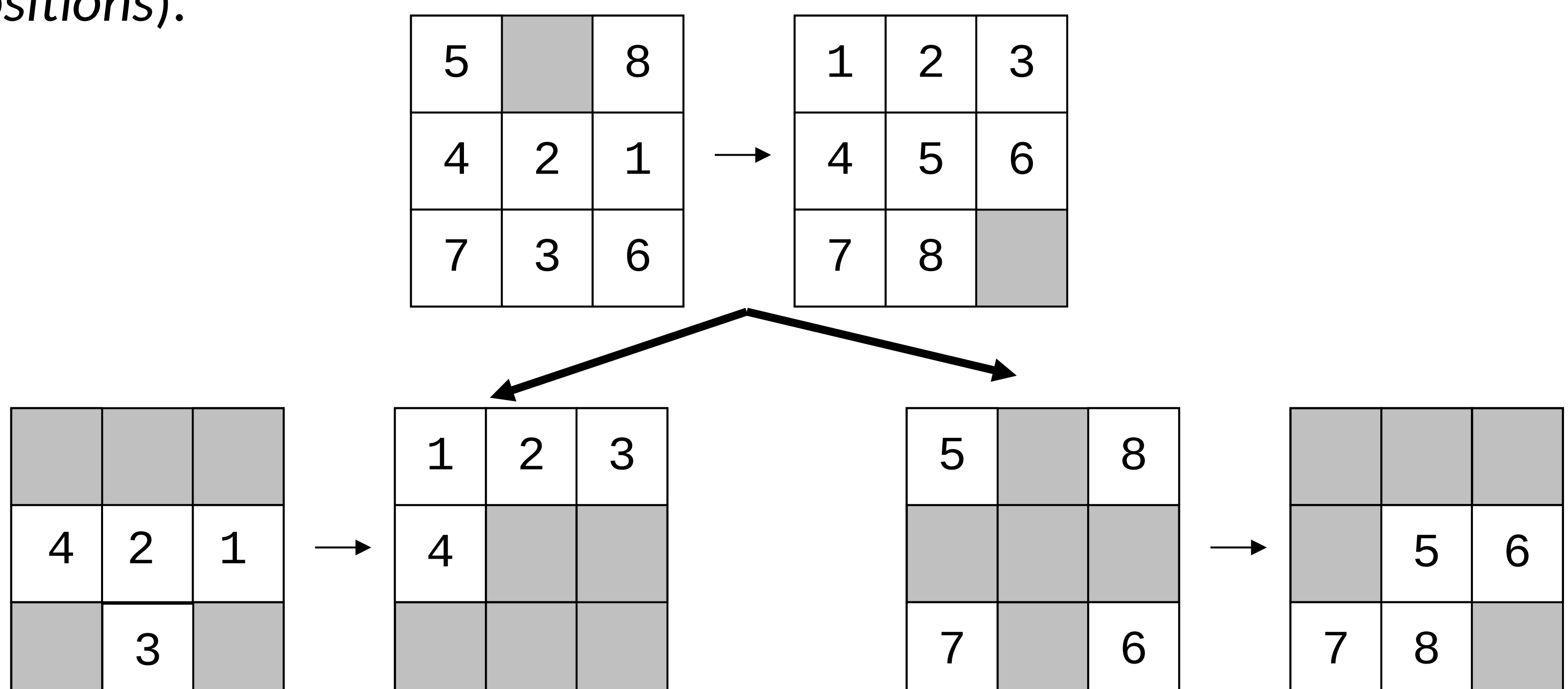
  = 2+3+3+2+2+2+0+2 = 16

Dominance

- h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all n.
  - both are admissible: $h_i(n) < h^*(n)$.
  - h_2 is better for search (closer to the real cost).
- Puzzle-8 typical search costs (average number of nodes expanded).
  - depth = 12
    - Iterative Deepening Search = 3,188,646 nodes
    - A*(h_1) = 539 nodes
    - A*(h_2) = 119 nodes
  - depth = 24
    - Iterative Deepening Search > $O(3^{24})$
    - A*(h_1) = 39,135 nodes
    - A*(h_2) = 1,641 nodes

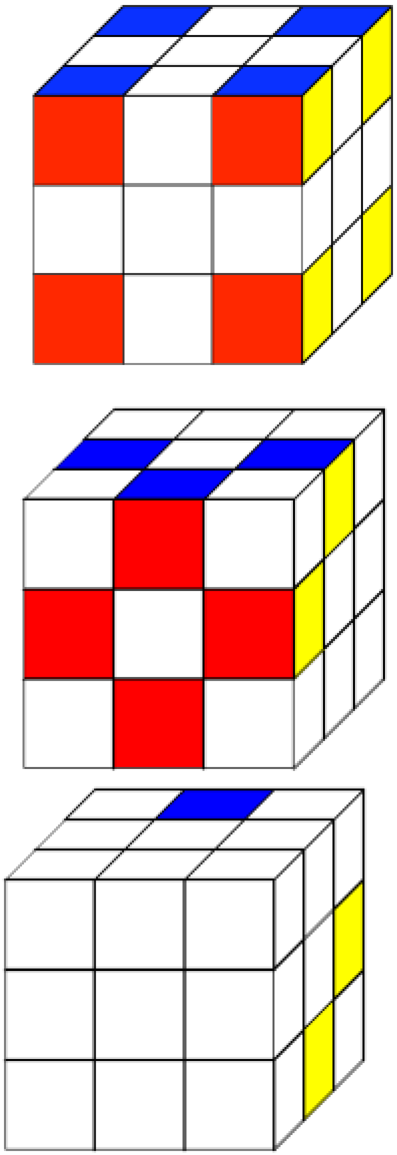
Can We Do Better?

- We could consider 2 less complex relaxed problems.
  - Consider 2 half puzzles (pieces [1, 4] and pieces [5, 8]).
  - Store the exact solution costs for every subproblems.
    - These distances could have been precomputed in a database
    - *Called:* ***Dynamic programming***
  - Heuristic $h_3(n)$ = d1234 + d5678 [disjoint pattern heuristic].
  - This heuristic h_3 dominates h_2 (ie, *sum of distances of all tiles from their goal positions*).



Relaxed Problems. Example 4

- Rubik's Cube (4.3 x 10^{19} states)
  - $h(n) = \max(h_c(n), h_{e1}(n), h_{e2}(n))$
    - hc: restricted to the corners.
    - h_{e1}: restricted to six edges.
    - h_{e2}: restricted to the rest of the edges.
- Dynamic Programming
  - Precomputed h_c, h_{e1}, h_{e2} using BFS (build solutions from start to each possible configuration and store distances)
    - 3 tables (5 minutes each, total size: 4 Mb).
- 1 day to solve a configuration with IDA* and the previous heuristic.
  - However optimal solution.
- Theorem: ≤ 20 moves for any configuration.
  - ie. d=20



Traveler problem

- To develop a shortest path search program (like a car GPS), you need to use hierarchical search algorithms (Hierarchical Contractions).
- Intuitively, there are axes that are more important than others.
  - A first phase creates short-cuts.
  - The second uses hierarchical A^* and bidirectional search.