AI-KI-B

Heuristic Search

Ute Schmid & Diedrich Wolter
Practice: Johannes Rabold

Cognitive Systems and Smart Environments
Applied Computer Science, University of Bamberg

last change: 3. Juni 2019, 10:05

Outline

Introduction of heuristic search algorithms, based on foundations of problem spaces and search

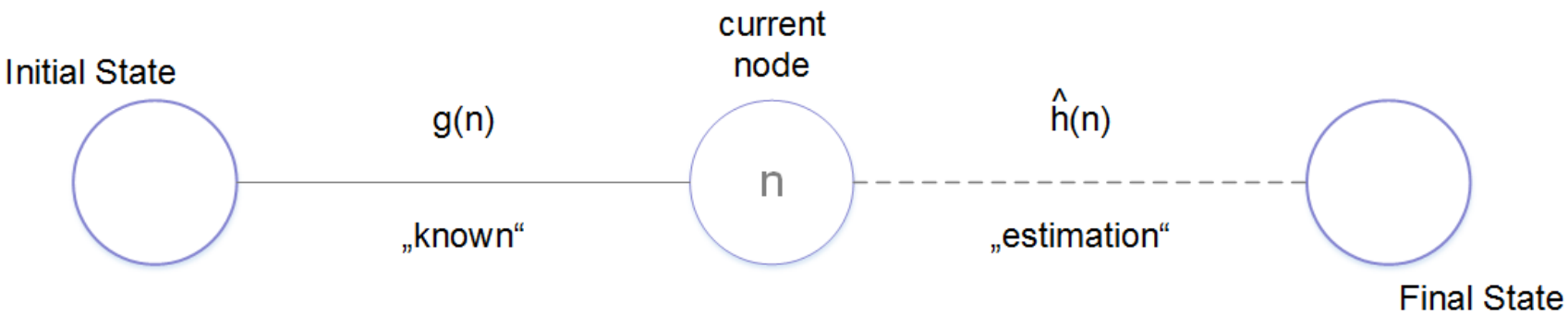
- Uniformed Systematic Search
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- Complexity of Blocks-World
- Cost-based Optimal Search
  - Uniform Cost Search
- **Cost Estimation (Heuristic Function)**
- **Heuristic Search Algorithms**
  - **Hill Climbing (Depth-First, Greedy)**
  - **Branch and Bound Algorithms (BFS-based)**
    - **Best First Search**
    - **A***
- Designing Heuristic Functions
- Problem Types

Cost and Cost Estimation

- "Real" cost is known for each operator.
  - Accumulated cost $g(n)$ for a leaf node n on a partially expanded path can be calculated.
  - For problems where each operator has the same cost or where no information about costs is available, all operator applications have equal cost values. For cost values of 1, accumulated costs $g(n)$ are equal to path-length d.
- Sometimes available:
  Heuristics for *estimating* the **remaining costs** to reach the final state.
  - $\hat{h}(n)$: estimated costs to reach a goal state from node n
  - "bad" heuristics can misguide search!

Cost and Cost Estimation cont.

Evaluation Function: $\hat{f}(n) = g(n) + \hat{h}(n)$



Cost and Cost Estimation cont.

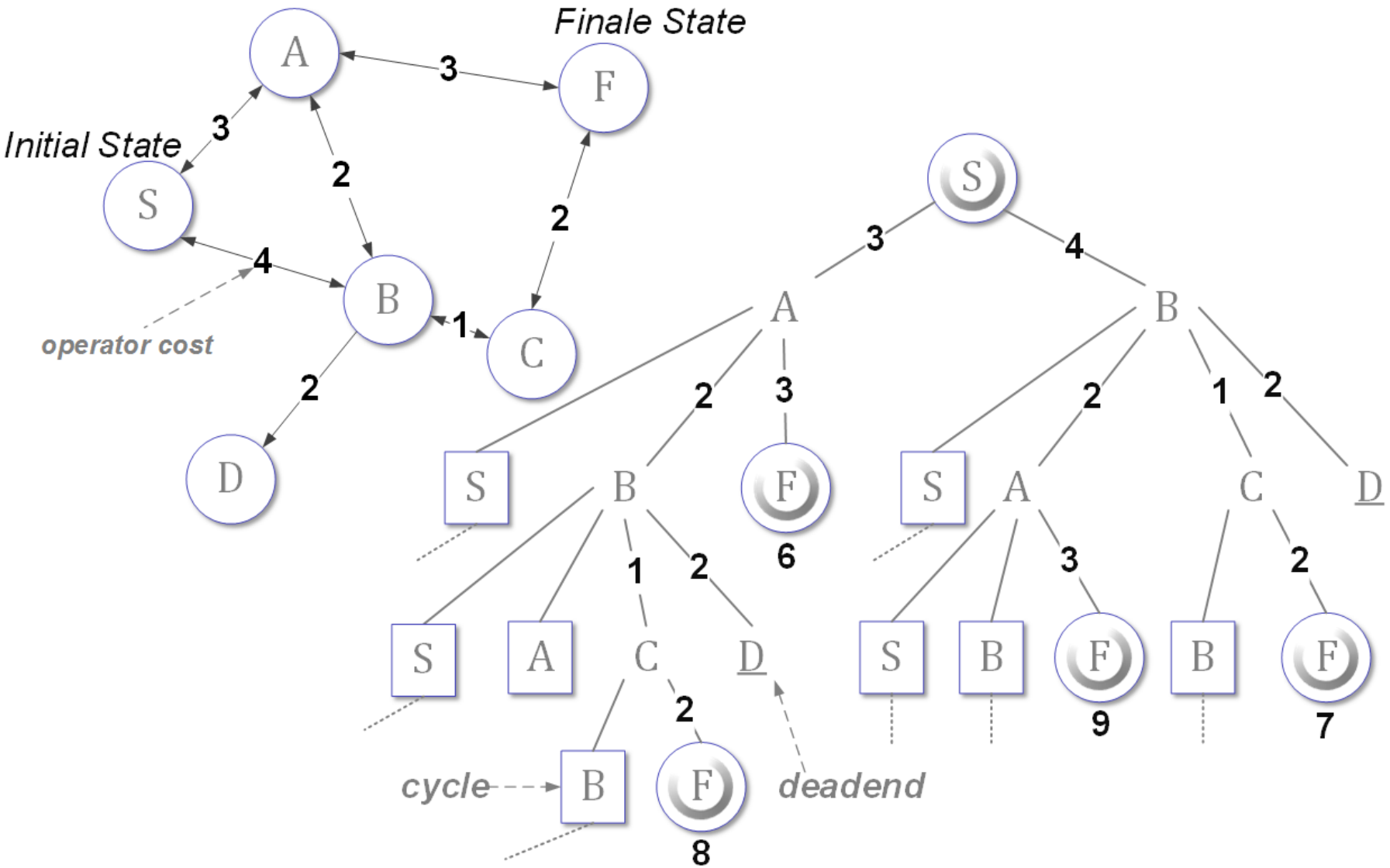
- "True costs" of an optimal path from an initial state s to a final state: $f(s)$.
- For a node n on this path, f can be decomposed in the already performed steps with cost $g(n)$ and the yet to perform steps with true cost $h(n)$.
- $\hat{h}(n)$ can be an estimation which is greater or smaller than the true costs.
- If we have no heuristics, $\hat{h}(n)$ can be set to the "trivial lower bound" $\hat{h}(n) = 0$ for each node n.
- If $\hat{h}(n)$ is a non-trivial lower bound, the optimal solution can be found in efficient time (see A*).

- **Hill Climbing:** greedy-Algorithm, based on depth-first search, uses only $\hat{h}(n)$ (not $g(n)$)
- **Best First Search** based on breadth-first search, uses only $\hat{h}(n)$
- **A*** based on breadth-first search (efficient branch-and bound algorithm), used evaluation function $f^*(n) = g(n) + h^*(n)$ where $h^*(n)$ is a *lower bound estimation* of the true costs for reaching a final state from node n.

Design of a search algorithm:

- based on depth- or breadth-first strategy
- use only $g(n)$, use only $\hat{h}(n)$, use both ($\hat{f}(n)$)

Example Search Tree



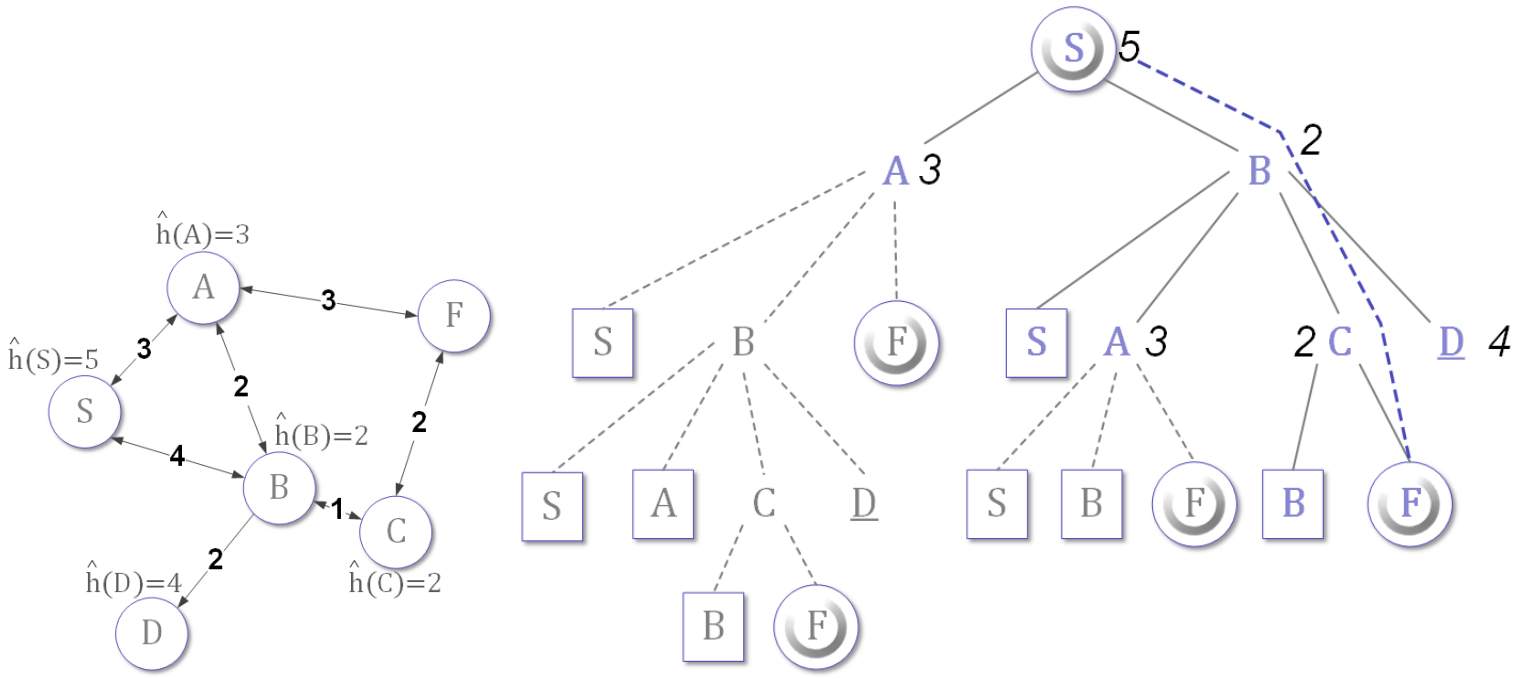
Hill Climbing Algorithm

Winston, 1992 To conduct a hill climbing search,

- Form a one-element stack consisting of a zero-length path that contains only the root node.
- Until the top-most path in the stack terminates at the goal node or the stack is empty,
  - Pop the first path from the stack; create new paths by extending the first path to all neighbors of the terminal node.
  - Reject all new paths with loops.
  - Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal.
  - Push the new paths, if any, on the stack.
- If the goal node is found, announce success; otherwise announce failure.

Hill Climbing Example

((S).5)
((S B).2 (S A).3)
((S B C).2 (S B A).3 (S B D).4 [(S B S).5] (S A).3)
([(S B C B).2] (S B C F).0 (S B A).3 (S B D).4 (S A).3)

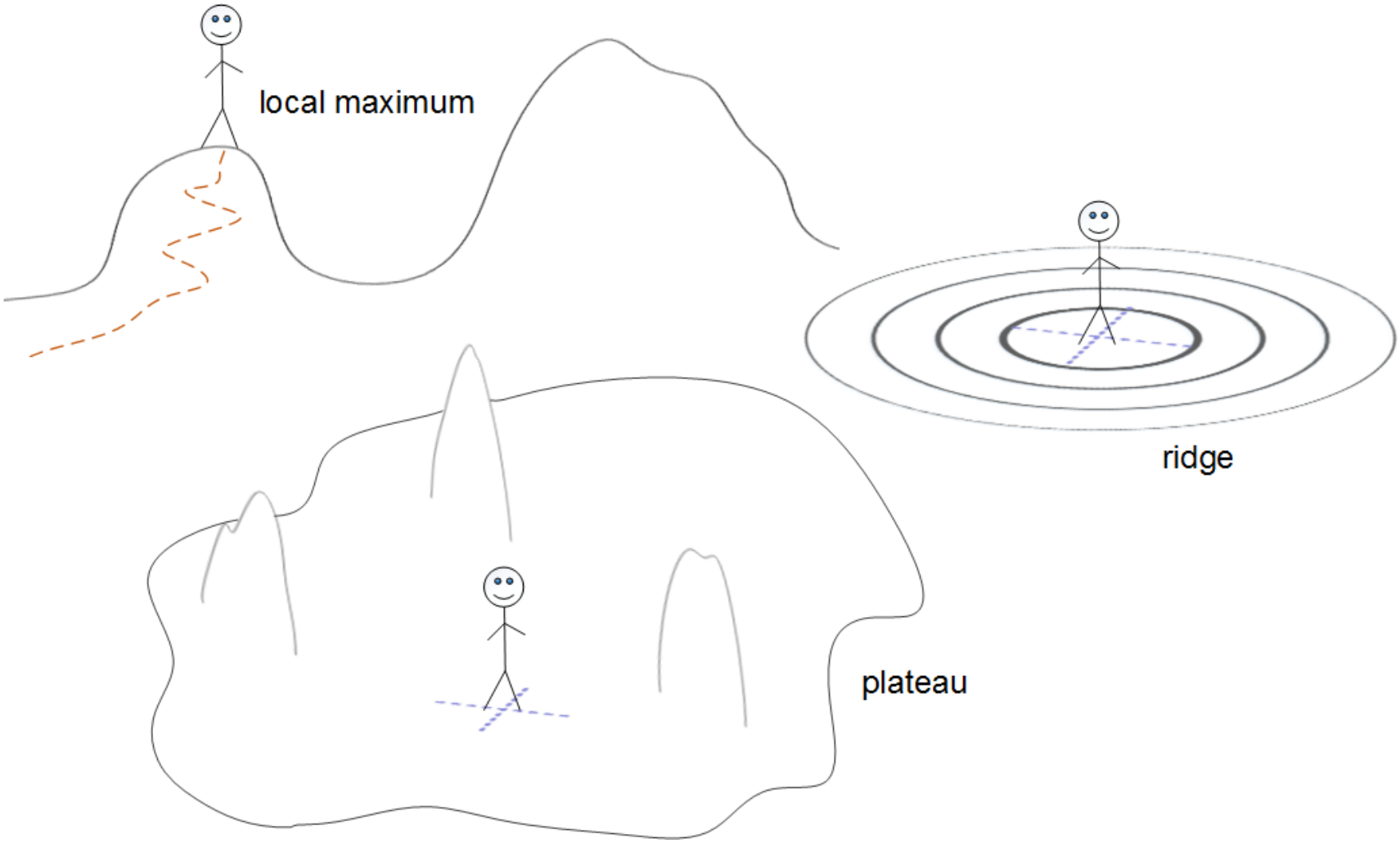


The heuristics was not optimal. If we look at the true costs (S A F) is the best solution!

Problems of Hill Climbing

- Hill climbing is a discrete variant of *gradient descend* methods (as used for example in back propagation).
- Hill climbing is a *local/greedy* algorithm:
  Only the current node is considered.
- For problems which are greedy solvable (local optimal solution = global optimal solution) it is guaranteed that an optimal solution can be found.
  Otherwise: danger of **local minima/maxima**
  (if $\hat{h}$ is a cost estimation: local **minima**!)
- Further problems:
  plateaus (evaluation is the same for each alternative),
  ridges (evaluation gets worse for all alternatives)

Problems of Hill Climbing cont.



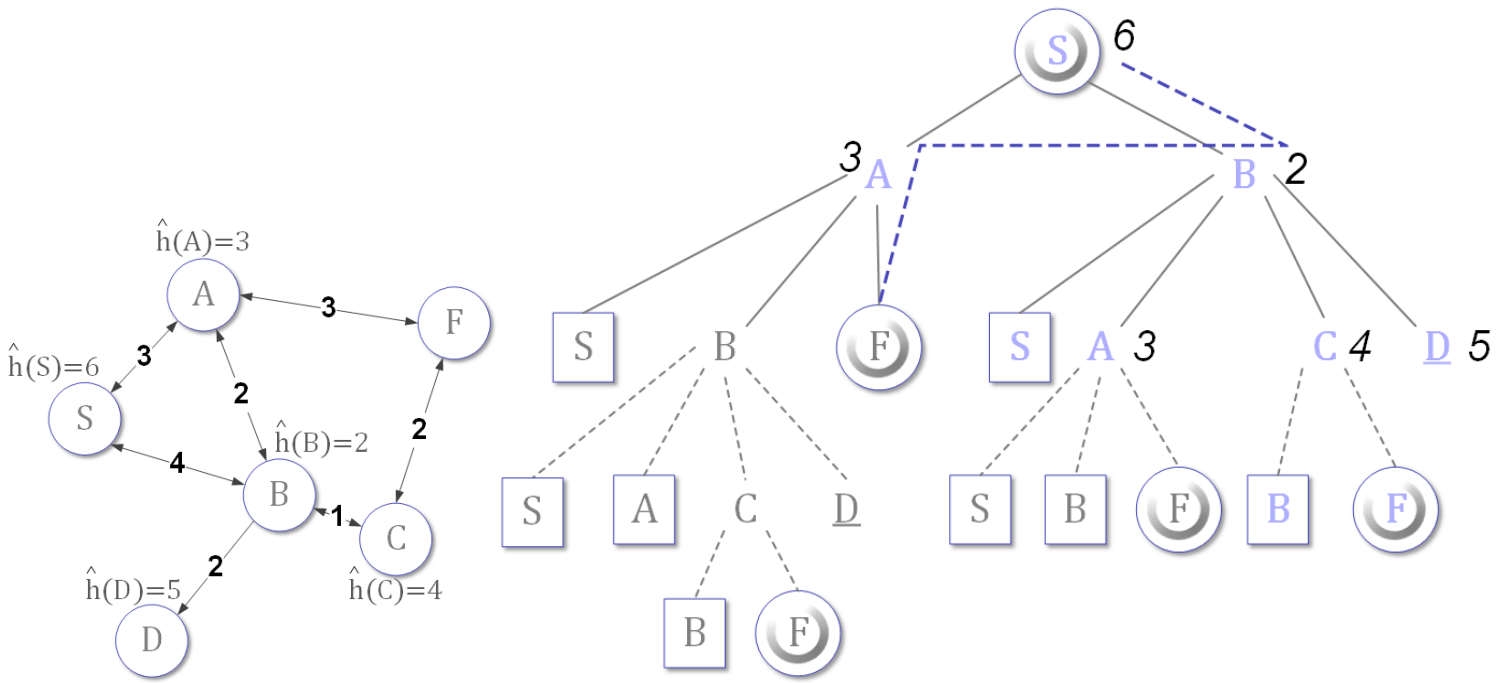
Best First Search Algorithm

Winston, 1992 To conduct a best first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the queue.
  - Sort entire queue by the estimated distances between their terminal nodes and the goal.
- If the goal node is found, announce success; otherwise announce failure.

Best First Example

((S).6)
((S B).2 (S A).3)
((S A).3 (S B A).3 (S B C).4 (S B D).5 [(S B S).6])
((S A F).0 (S A B).2 (S B A).3 (S B C).4 (S B D).5 [(S A S).6])



Best First Search Remarks

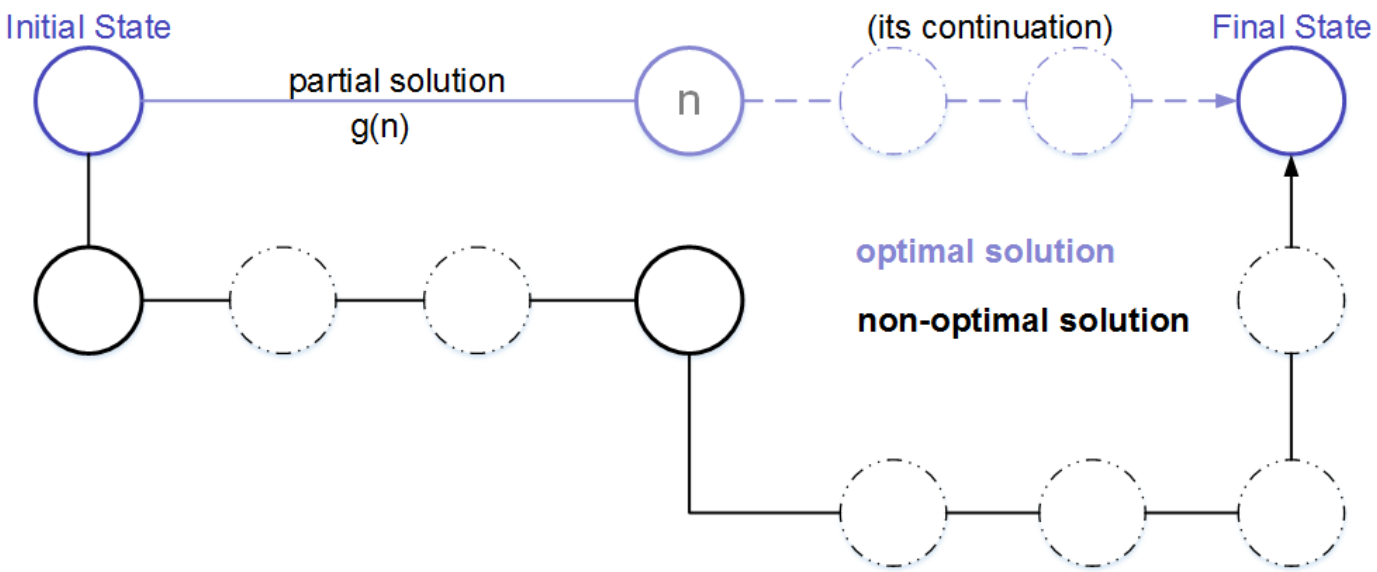
- Best First Search is not a local strategy:
  at each step the current *best* node is expanded, regardless on which partial path it is.
- It is probable but not sure that Best First Search finds an optimal solution.
  (depending on the quality of the heuristic function)

Optimal Search

- Inefficient, blind method: 'British Museum Algorithm'
  - Generate *all* solution paths and select the best.
  - Generate-and-test algorithm, effort $O(b^d)$
- Breadth-First search (for no/uniform costs) and Uniform Cost Search (for operators with different costs; Branch-and-Bound) find the optimal solution, but with a high effort
  - A* (Nilsson, 1971) is the most efficient branch-and-bound algorithm!

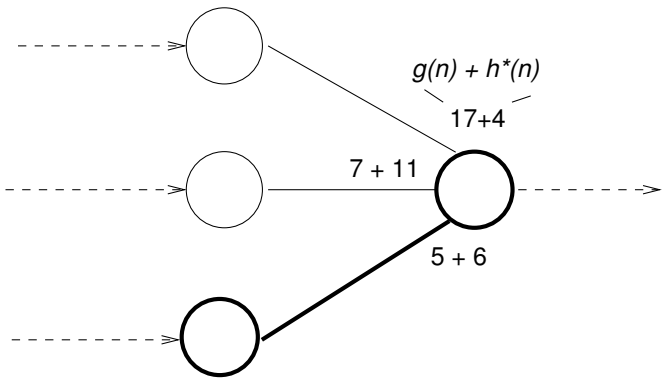
Reminder: Uniform Cost Search

- The complete queue is sorted by accumulated costs $g(n)$ with the path with the best (lowest) cost in front.
- Termination: If the *first* path in the queue is a solution.
- Why not terminate if the queue contains a path to the solution on an arbitrary position?
  Because there are partially expanded paths which have lower costs than the solution. These paths are candidates for leading to a solution with lower costs!



The Idea of A*

- Extend uniform cost search such, that not only the accumulated costs $g(n)$ but additionally an estimate for the remaining costs $\hat{h}(n)$ is used.
  $\hat{h}$ is defined such that it is a non-trivial lower bound estimate of the true costs for the remaining path (h^*).
  That is, use evaluation function $f^*(n) = g(n) + h^*(n)$.
- Additionally use the principle of 'dynamic programming' (Bellman & Dreyfus, 1962): If several partial paths end in the same node, only keep the best of these paths.



A* Algorithm

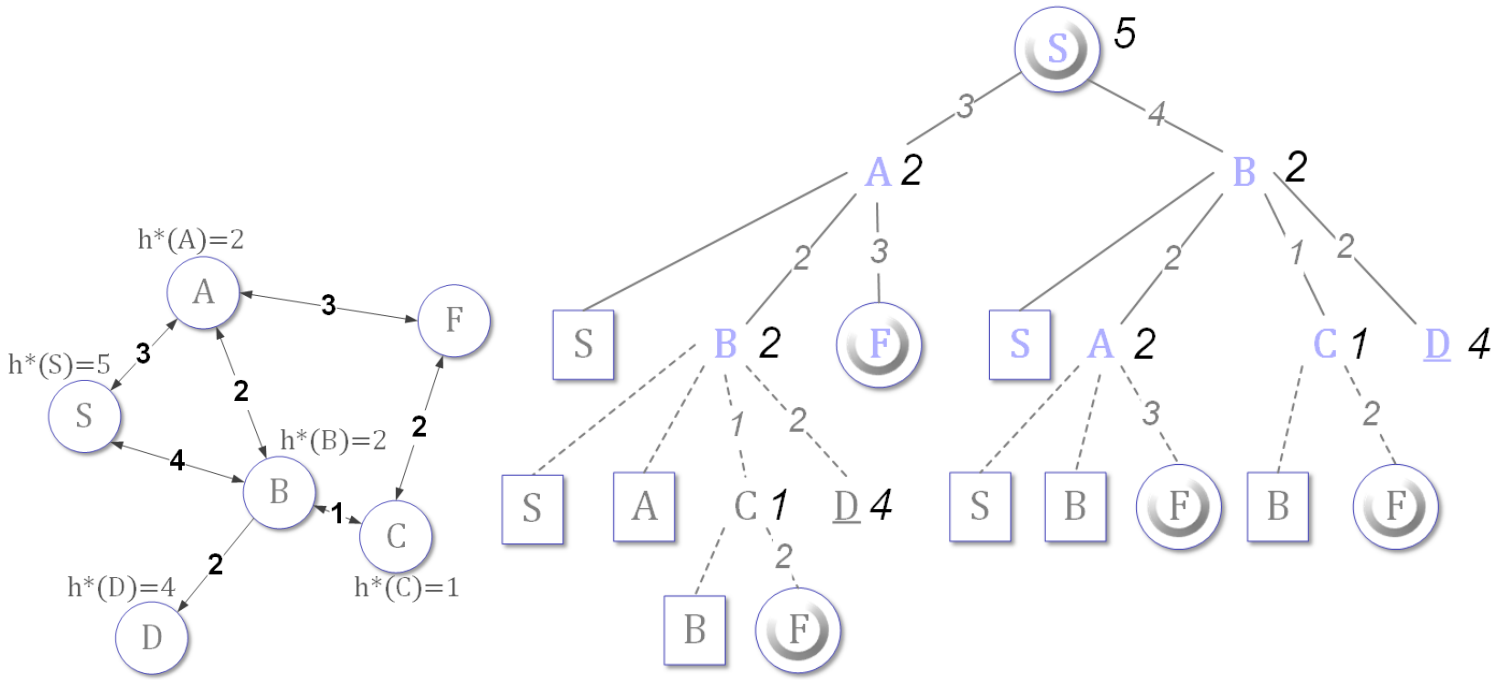
Winston, 1992 To conduct a A* search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by ext. the first path to all neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the remaining new paths, if any, to the queue.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front.
- If the goal node is found, announce success; otherwise announce failure.

A* Example

((S).0 + 5)
((S A).3 + 2 (S B).4 + 2)

([(S A S).11] (S A B).$\overbrace{3+2}^{g}$ +2 (S A F).3 + 3 + 0 (S B).6)
because of (S A B).7 and (S B).6: delete (S A B)
((S A F).6 (S B).6)



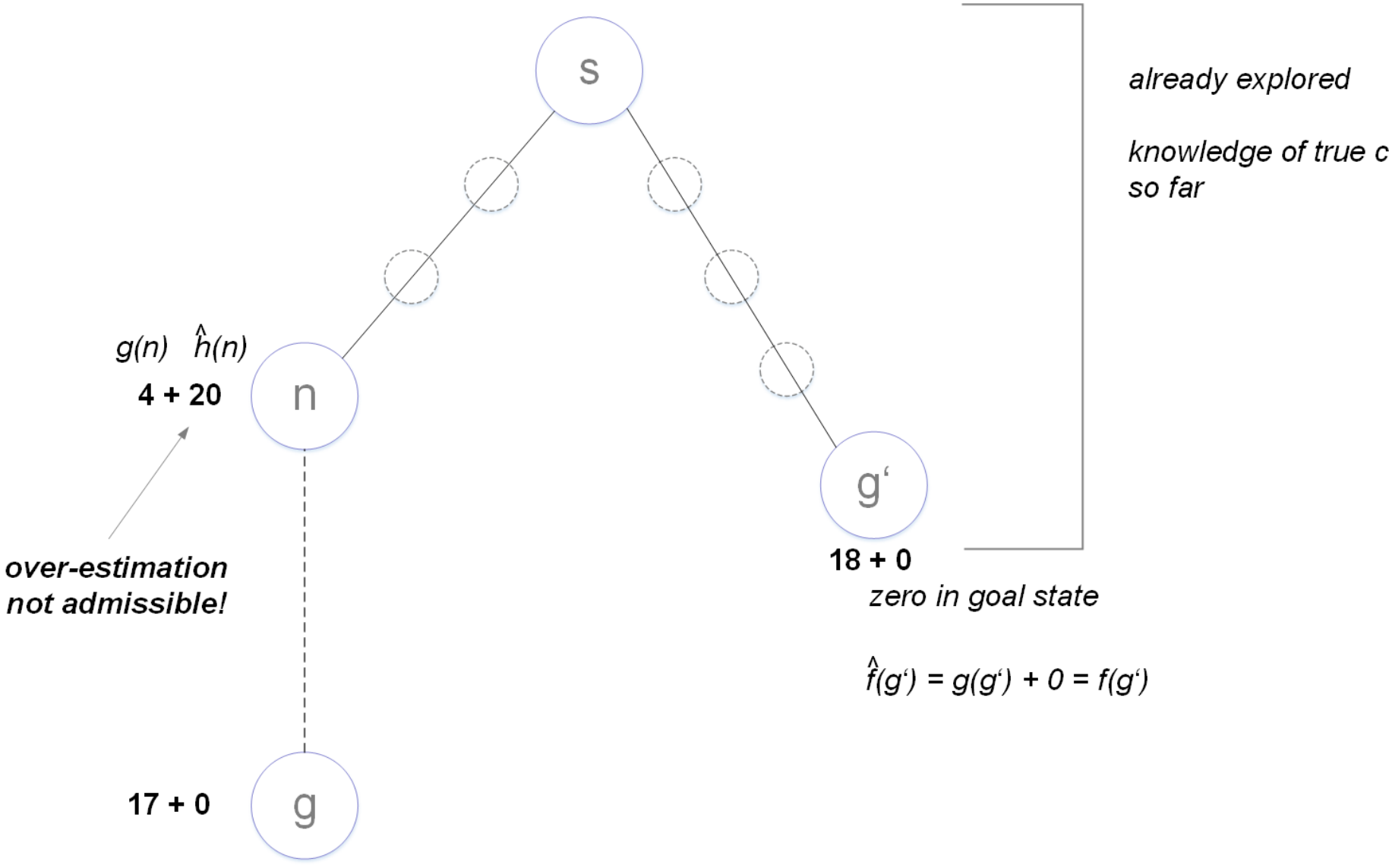
Admissibility of A*

- Theorem: If $\hat{h}(n) \leq h(n)$ for all nodes n and if all costs are greater than some small positive number δ, then A* always returns an optimal solution if a solution exists (is "admissible").
- Proof: in Nilsson (1971); we give the idea of the proof
  - Remember, that $f(n) = g(n) + h(n)$ denotes the "true costs", that is, the accumulated costs $g(n)$ for node n and the "true costs" $h(n)$ for the optimal path from n to a final state.
  - Every algorithm A working with an evaluation function $\hat{f}(n) = g(n) + \hat{h}(n)$ for which holds that $\hat{h}(n)$ is smaller or equal than the true remaining costs (including the trivial estimate $\hat{h}(n) = 0$ for all n) is guaranteed to return an optimal solution:

Admissibility of A* cont

- Each steps heightens the "security" of estimate $\hat{f}$ because the influence of accumulated costs grows over the influence of the estimation for the remaining costs.
- If a path terminates in a final states, only the accumulated costs from the initial to the final state are considered. This path is only returned as solution if it is first in the queue.
- If $\hat{h}$ would be an over-estimation, there still could be a partial path in the queue for which holds that $\hat{f}(n) > f(n)$.
- If $\hat{h}$ is always an under-estimation and if all costs are positive, it always holds that $\hat{f}(n) \leq f(n)$. Therefore, if a solution path is in front of the queue, all other (partial) paths must have costs which are equal or higher.

A* Illustration



Optimality of A*

- "Optimality" means here: there cannot exist a more efficient algorithm.
- Compare the example for uniform cost search and A*: both strategies find the optimal solution but A* needs to explore a much smaller part of the search tree!
- Why?
  Using a non-trivial lower bound estimate for the remaining costs don't direct search in a wrong direction!
- The somewhat lengthy proof is based on contradiction:
  Assume that A* expands a node n which is not expanded by another admissible algorithm A.

$$0 \leq h_1^*(n) \leq h_2^*(n) \leq \ldots \leq h_n^*(n) = h(n)$$

the tighter the lower bound, the more "well-informed" is the algorithm!

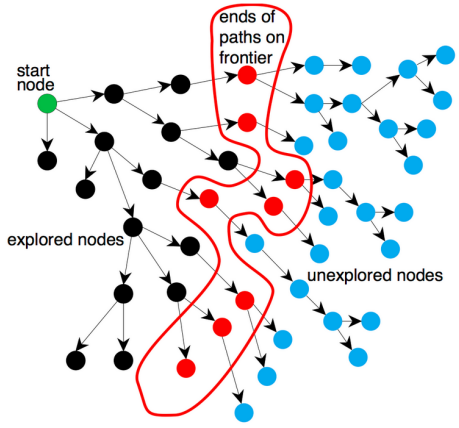
Optimality of A* cont.

- Alg. A does not expand n if it "knows" that any path to a goal through node n would have a cost larger or equal to the cost on an optimal path from initial node s to a goal, that is $f(n) \geq f(s)$.
- By rewriting $f(n) = g(n) + h(n)$ we obtain $h(n) = f(n) - g(n)$.
- Because of $f(n) \geq f(s)$ it holds that $h(n) \geq f(s) - g(n)$.
- If A has this information it must use a "very well informed" heuristics such that $\hat{h}(n) = f(s) - g(n)$!
- For A* we know that f^* is constructed such that holds $f^*(n) \leq f(s)$, because the heuristics is an under-estimation.
- Therefore it holds that $g(n) + h^*(n) \leq f(s)$
- and by rewriting that $h^*(n) \leq f(s) - g(n)$.
- Now we see that A used information permitting a tighter lower bound estimation of h than A*. It follows that the quality of the lower bound estimate determines the number of nodes which are expanded.

A generic graph searching algorithm

- Search algorithms follow the same abstract pattern
- Note the concept of a **frontier**: contains all of the paths that could form initial segments of paths from the start node to a goal node.
- Different search strategies are obtained by providing an appropriate implementation of the frontier!
- see: Poole & Mackworth, Artificial Intelligence: Foundations of Computational Agents, CUP, 2017; `https://artint.info/2e/html/ArtInt2e.Ch3.S4.html`

```
1: procedure Search(G, S, goal)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         s: start node
5:         goal: Boolean function of nodes
6:     Output
7:         path from s to a node for which goal is true
8:         or ⊥ if there are no solution paths
9:     Local
10:        Frontier: set of paths
11:    Frontier := {⟨s⟩}
12:    while Frontier ≠ {} do
13:        select and remove ⟨n_0, ..., n_k⟩ from Frontier
14:        if goal(n_k) then
15:            return ⟨n_0, ..., n_k⟩
16:        Frontier := Frontier ∪ {⟨n_0, ..., n_k, n⟩ : ⟨n_k, n⟩ ∈ A}
17:    return ⊥
```



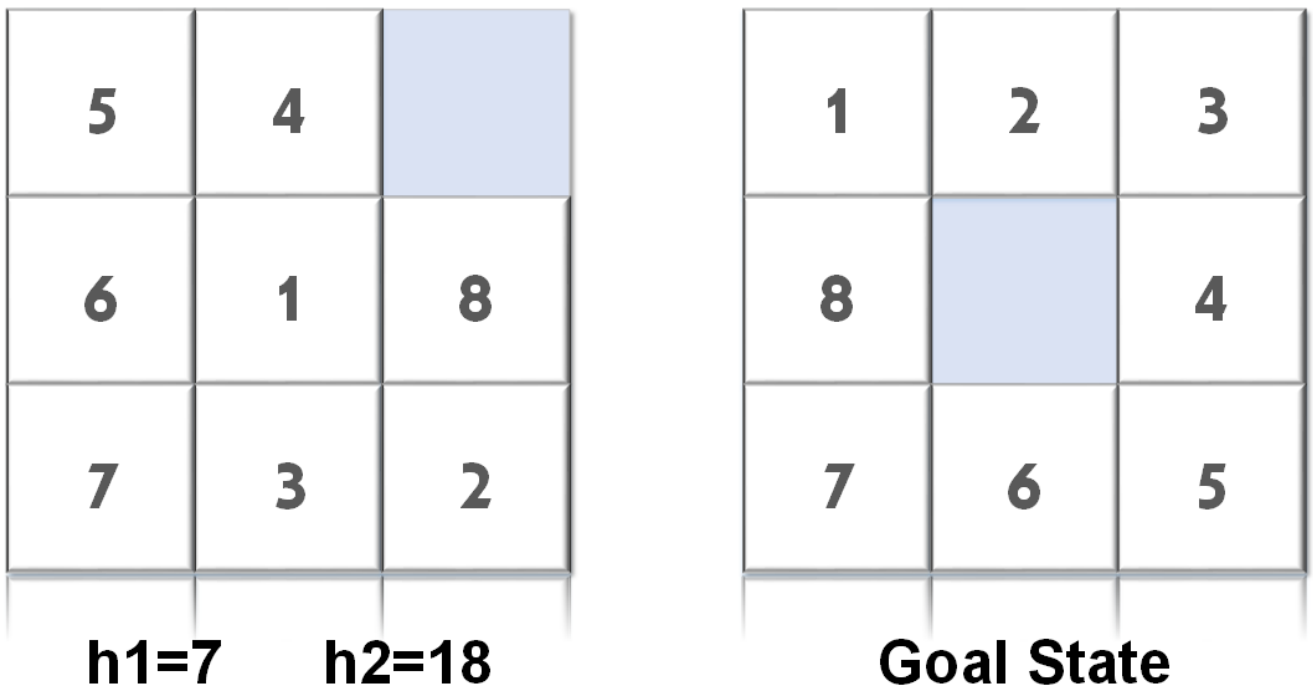
How to design a Heuristic Function?

- Often, it is not very easy to come up with a good heuristics.
- For navigation problems:
  use Euclidean distance between cities as lower bound estimate.
- For "puzzles":
  analyse the problem and think of a "suitable" rule. E.g.:
  Number of discs which are already placed correctly for Tower of Hanoi
- Chess programs (Deep Blue, Deep Fritz) rely on very carefully crafted evaluation functions. The "intelligence" of the system sits in this function and this function was developed by *human* intelligence
  (e.g. the grand master of chess Joel Benjamin, who contributed strongly to the evaluation function of Deep Blue).

Example: 8-Puzzle

admissible heuristics h^* for the 8-puzzle

- h_1^*: total number of misplaced tiles
- h_2^*: minimal number of moves of each tile to its correct location, i.e. total Manhattan distance



Excursus: Minkowski-Metric

$$d(\vec{v_1}, \vec{v_2}) = k\sqrt{\sum_{i=1}^{n} |v_{1i} - v_{2i}|^k}$$

- Distance d: in general between to feature vectors in n dimensional-space.
- For 8-Puzzle: 2 Dimensions (x-position and y-position).
- Minkowski parameter k: determines the metric
  - $k = 2$: the well known Euklidian distance $\sqrt{(\vec{v_1} - \vec{v_2})^2}$ (direct line between two points)
  - $k = 1$: City-block or Manhattan distance (summation of the differences of each feature)
  - $k \to \infty$: Supremum or dominance distance (only the feature with the largest difference is taken into account)
- Psychological investigations about metrics used by humans if they judge similarities, e.g., Lazarte, A. A., & Schönemann, P. H. (1991). Saliency metric for subadditive dissimilarity judgments of rectangles. Perception & psychophysics, 49(2), 142-158.

Summary: Search Algorithms

- Search algorithms are fundamental for many areas of computer science in general and for AI.
- Many AI technologies are based on logic or representation in other formal languages and search.
- **Depth-first** variants are in average more efficient than **breadth-first** variants, but there is no guarantee that an optimal solution can be found.
- Heuristic variant of depth-first search: **Hill-Climbing/greedy search**; heuristic variant of breadth-first search: **Best First search**
- Breadth-first variants with costs are called **branch-and-bound-algorithms**: branch from a node to all successors, bound (do not follow) unpromising paths
  - **A***: uses an admissible heuristic $0 \leq h^*(n) \leq h(n)$
    it gains its efficiency (exploring as small a part of the search tree as possible) by: dynamic programming and using a heuristic which is as well-informed as possible (tight lower bound)