CogSysI Lecture 8: Planning as Search: Heuristics

Intelligent Agents

Ute Schmid (lecture)
Emanuel Kitzelmann (practice)

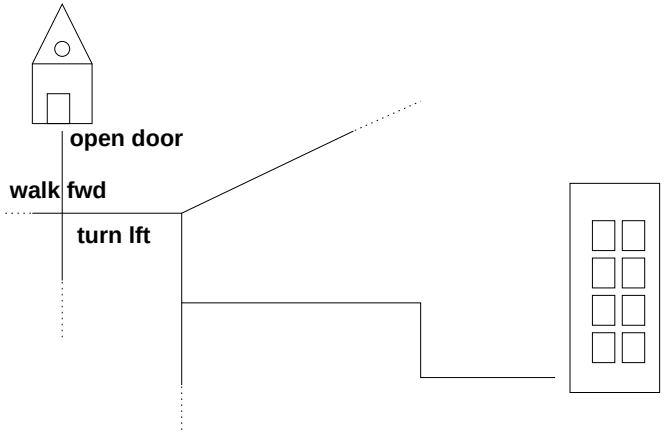
Applied Computer Science, Bamberg University

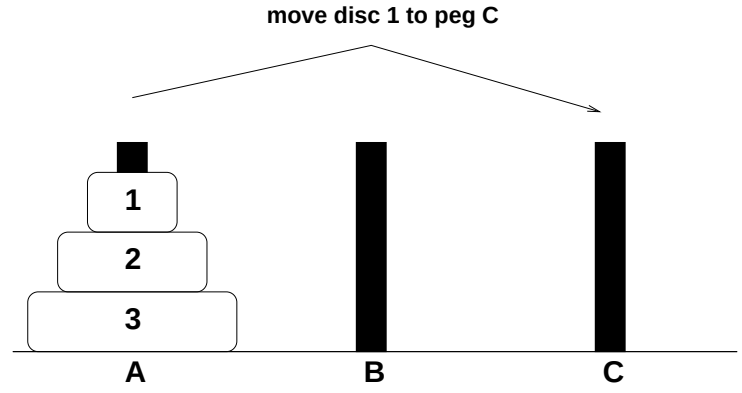
last change: June 6, 2008

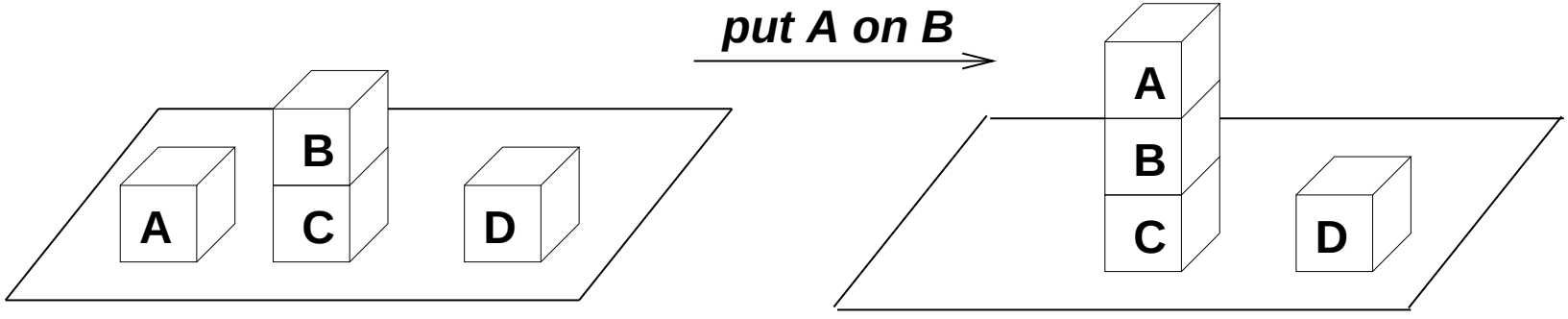
Problem Solving / Problem

- Problem: initial state, goal states, operators (with application conditions)
- Example: Way-finding
  - Initial state: *at-home*
  - Goal state: *at-work*
  - Operators: *open door* (if you are in front of door), *walk straight-forward*, *turn left*, ...
- Example: Tower of Hanoi
  - Initial state: *discs 1..3 on peg* A
  - Goal state: *discs 1..3 on peg* B
  - Operators: *put disc on another peg* (if disc is top of peg and if other peg is empty or top of other peg is larger then disc)
- Problem: more than one operator application necessary to reach goal

Example Problems







Types of Problems

- Ill-defined (McCarthy, 1956) / open (Minsky 1965):
  description of final state is incomplete/fuzzy
  e.g.: have a good life, create an aesthetic artefact
- Well-defined / closed:
  initial state and final states clearly defined
  - Interpolation problems: Unknown sequence of operators
- Further aspects: problem complexity, decomposability, ...

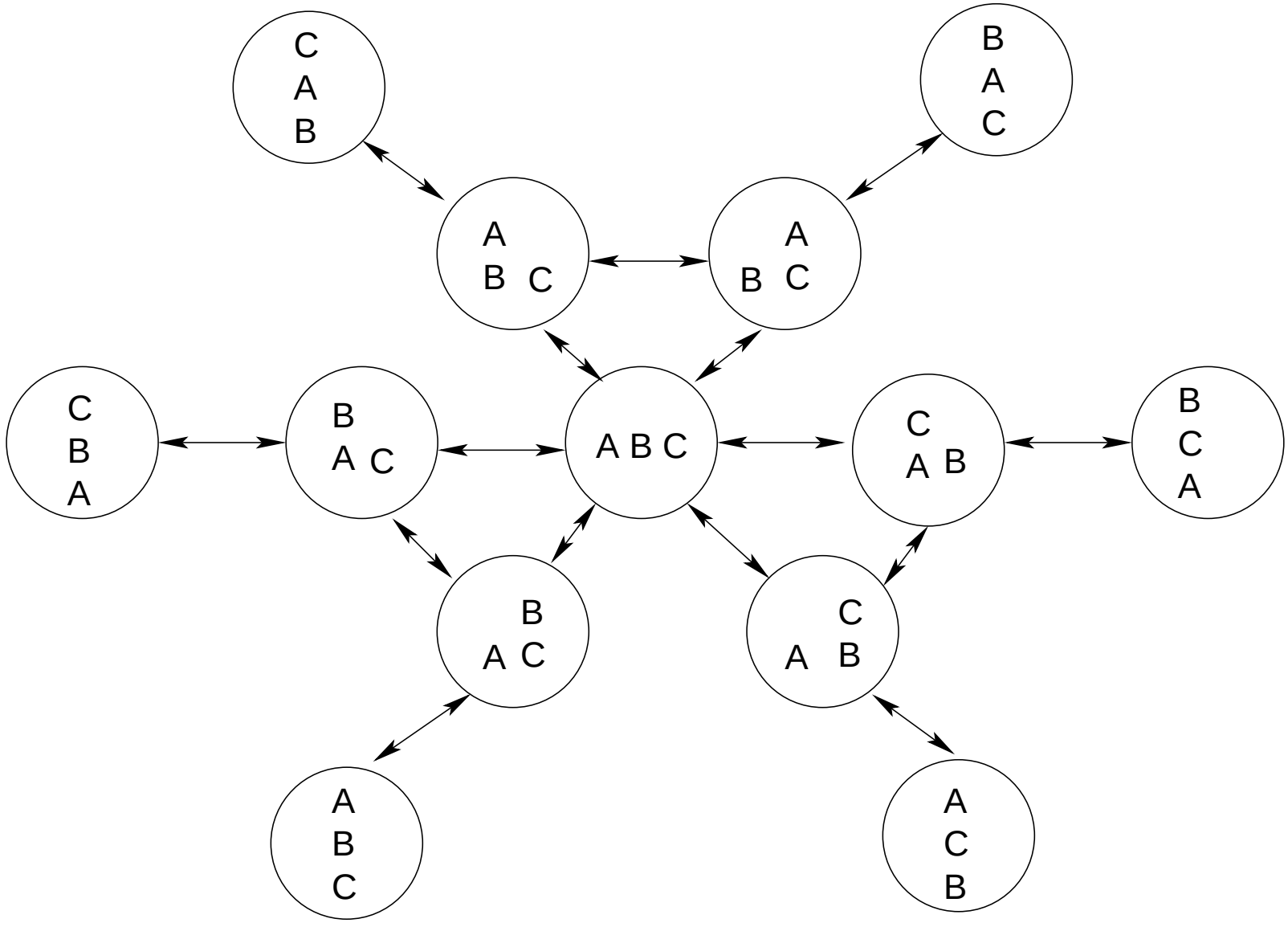
Basic Definitions

- **Problem**: $P = (S_i, S_e, O)$ with
  S_i: Set of initial states, S_e: Set of final/goal states, O : Set of operator/actions
- **Problem space**: Set of states S with $S_i \subset S$, $S_e \subseteq S$
- **Operator**: $o \in O$ with $O : S' \rightarrow S, S' \subseteq S$
  $\hookrightarrow$ Operators are typically *partial*, they have application conditions and cannot be applied in any state in S
- **Problem solving**: Transformation of an initial state into a final state by applying a (minimal) sequence of operators.
  Minimal: least number of steps, minimal costs
- Problem solving can be performed in a real environment (acting human or robot) or in a simulation (mental process, manipulation of descriptions)

State Space

- The state space is a graph (S, A) with:
  - S: the set of nodes (all states of a problem)
  - $A \subseteq S \times S$: the set of arcs, with $(s, s') \in A$ iff state s' can be reached from state s by applying an operator $o \in O$.
- **State-space semantics**: represents the structure of the problem. Each path in the graph from an initial state to a final state is an admissible solution.
- Also known as **problem space** (Newell & Simon, 1972).
- Note: for computational approaches to problem solving, we do not have the states, but only descriptions of states (which are partial)!
  We represent only such information which is relevant for the problem (e.g. we are not interested in the material or in the color of blocks).

State Space for Blocksworld



Problem Solving as Search

- If we can transform an initial state into a final state by applying *one* operator, we solve a **task** and not a **problem**. (We have the specialized program/the skill to reach our goal directly.)
- For interpolation problems we do not know, which sequence of operators is suitable ("admissible") to transform an initial state into a goal state.
- Human problem solvers typically use additional knowledge (besides the legal operators) to solve a problem: choose always that operator which makes the difference between the current state and the desired state as small as possible (heuristics used by GPS, see chapter on human problem solving).
- Algorithmic solution: search

Problem Solving as Search cont.

- There exist different **search strategies**:
  - Basic, uninformed ("blind") methods: random search, systematic strategies (depth-first, breadth-first)
  - Search algorithms for operators with different costs
  - Heuristic search: use assumptions to guide the selection of the next candidate operator
- In the following: We are not concerned how a single state transformation is calculated. We represent problems via problem graphs. **Please note:** In general, such a graph is *not* explicitely given. We do *not* know the complete set of states S (which can be very large). A part of the problem graph is constructed during search (the states which we explore).

Search Tree

- During search for a solution, starting from an initial state, a search tree is generated.
  - Root: initial state
  - Each path from the root to a leaf: (partial) solution
  - Intermediate nodes: intermediate states
  - Leafs: Final states or dead ends
- If the same state appears more than once on a path, we have a **cycle**. Without cycle-check search might not terminate! (infinite search tree)
- Again: A search tree represents *a part* of the problem graph

Problem Solving Strategy

As long as no final state is reached or there are still reachable, yet unexplored states:

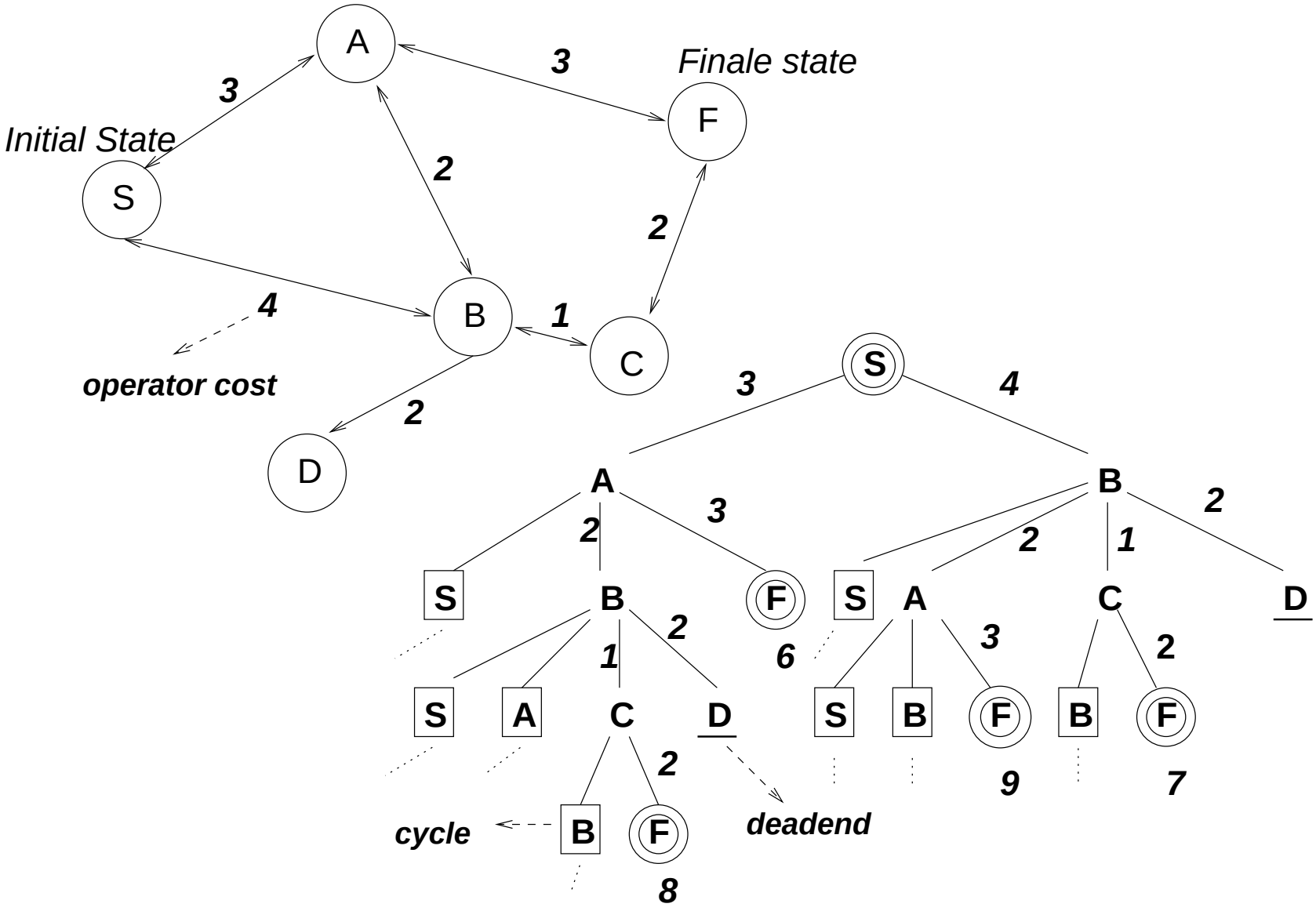
- Collect all operators which can be applied in the current state (Match state with application conditions)
- Select on applicable operator.
  In our example: alphabetic order of the labels of the resulting node.
  In general: give a preference order
- Apply the operator, generating a successor state

Remark: The Match-Select-Apply Cycle is the core of "production systems" (see chapter human problem solving)

Example

- In the following: abstract problem graph with nodes representing states and arcs representing operator applications. Numerical labels of the arcs: costs.
- Illustration: Navigation problem with states as cities and arcs as traffic routes; Blocksworld problem with states as constellations of blocks and arcs as put/puttable operators (might have different costs for different blocks); etc.

Example Search Tree



Depth-First Search

- Construct *one* path from initial to final state.
- Backtracking: Go back to predecessor state and try to generate another sucessor state (if none exists, backtrack again etc.), if:
  - the reached state is a deadend, or
  - the reached state was already reached before (cycle)
- Data structure to store the already explored states: **Stack**; depth-first is based on a “last in first out” (LIFO) strategy
- Cycle check: does the state already occur in the path.
- Result: in general not the shortest path (but the first path)

Effort of Depth-First Search

- In the best-case, depth-first search finds a solution in linear time $O(d)$, for d as average depth of the search tree: Solution can be found on a path of depth d and the first path is already a solution.
- In the worst-case, the complete search-tree must be generated: The problem has only one possible solution and this path is created as the last one during search; or the problem has no solution and the complete state-space must be explored.

Remark: Depth-First Search

- The most parsimonious way to store the (partial) solution path is to push always only the current state on the stack. Problem: additional infrastructure for backtracking (remember which operators were already applied to a fixed state)
- In the following: push always the partial solution path to the stack.

Depth-First Algorithm

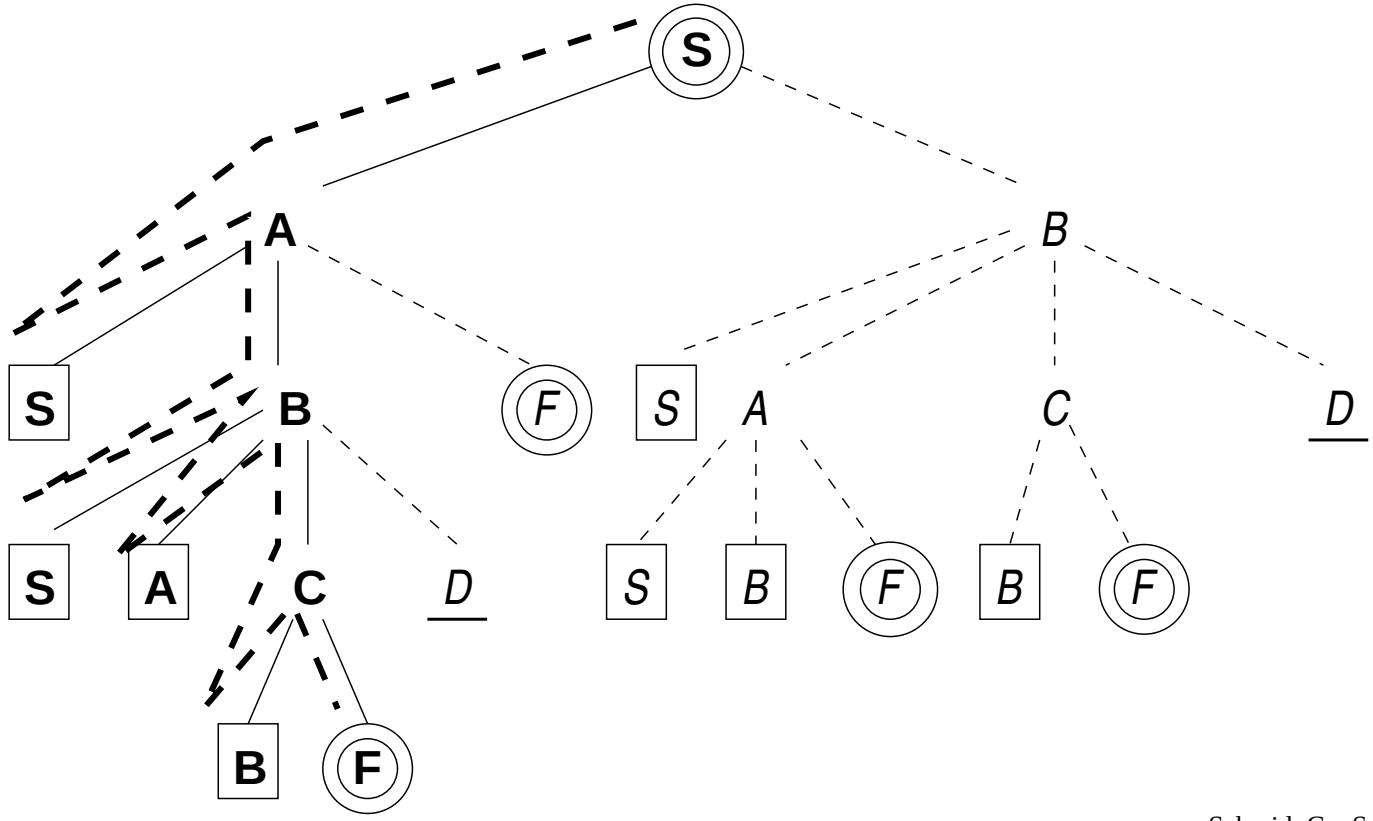
Winston, 1992

To conduct a depth-first 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.
  - Push the new paths, if any, on the stack.
- If the goal node is found, announce success; otherwise announce failure.

Depth-First Example

((S))
((S A) (S B))
((S A B) (S A F) [(S A S)] (S B))
((S A B C) (S A B D) [(S A B S)] (S A F) (S B))
([(S A B C B)] (S A B C F) (S A B D) (S A F) (S B))



Breadth-First Search

- The search tree is expanded levelwise.
- No backtracking necessary.
- Data structure to store the already explored states: **Queue**
  breadth-first is based on a “first in first out” (FIFO) strategy
- Cycle check: for finite state-spaces noty necessary for termination (but for efficiency)
- Result: shortest solution path
- Effort: If a solution can be first found on level d of the search tree and for an average branching factor b: $O(b^d)$

Breadth-First Algorithm

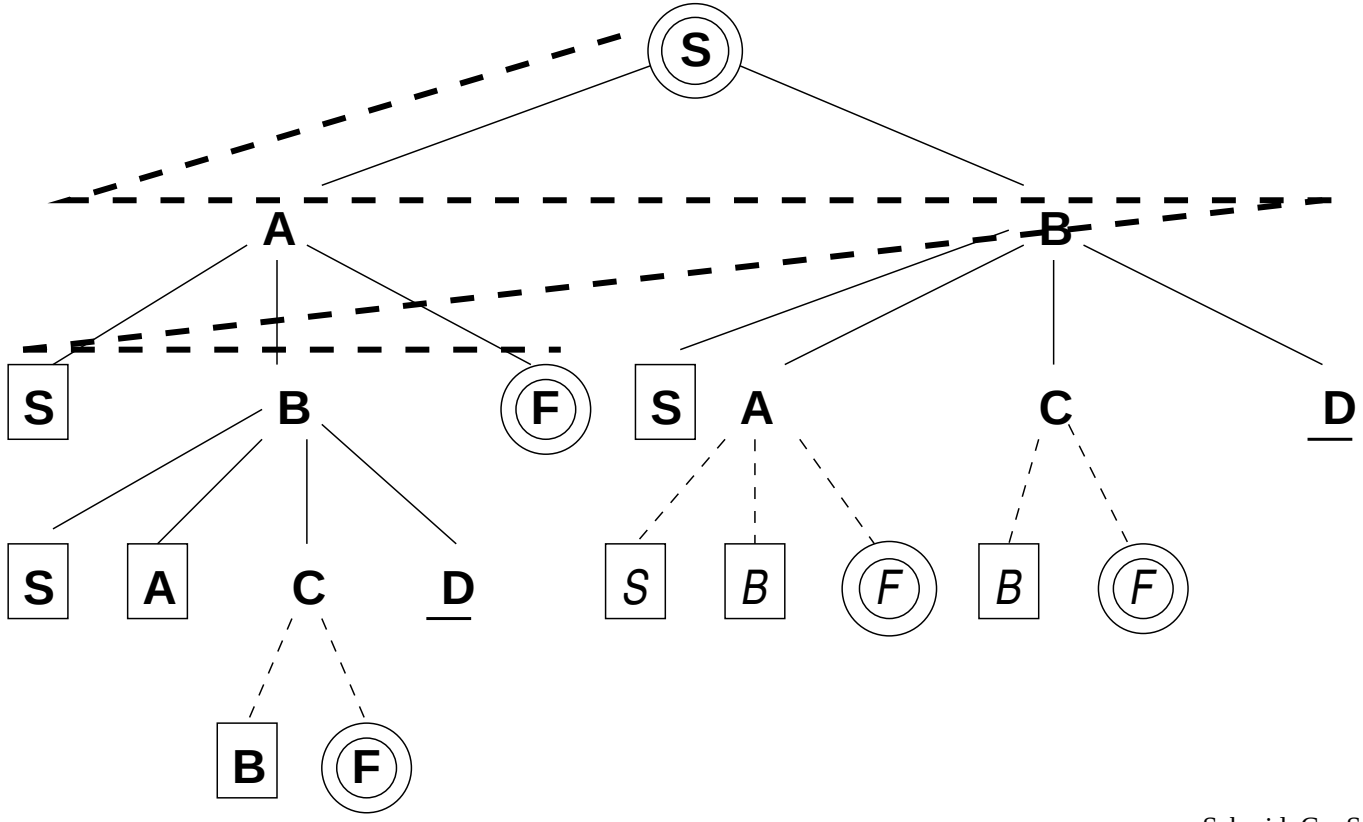
Winston, 1992

To conduct a breadth-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 *back* of the queue.
- If the goal node is found, announce success; otherwise announce failure.

Breath-First Example

((S))
((S A) (S B))
((S B) (S A B) (S A F) [(S A S)])
((S A B) (S A F) (S B A) (S B C) (S B D) [(S B S)])
((S A F) (S B A) (S B C) (S B D) (S A B C) (S A B D) [(S A B S)])



Evaluation of Search Strategies

- **Completeness:** The strategy is guaranteed to find a solution whenever there exists one
- **Termination:** The algorithm terminates with a solution or with an error message if no solution exists.
- **Soundness:** The strategy only returns admissible solutions.
- **Correctness:** Soundness + Termination
- **Optimality:** The strategy finds the "highest-quality" solution (minimal number of operator applications or minimal cost)
- **Effort:** How many time and/or how many memory is needed to generate an output?

Complexity of Blocksworld Problems

Remember: Problems can be characterized by their **complexity**, most problems considered in AI are NP-hard.

# blocks	1	2	3	4	5
# states	1	3	13	73	501
approx.	1.0×10^0	3.0×10^0	1.3×10^1	7.3×10^1	5.0×10^2
# blocks	6	7	8	9	10
# states	4051	37633	394353	4596553	58941091
approx.	4.1×10^3	3.8×10^4	3.9×10^5	4.6×10^6	5.9×10^7
#blocks	11	12	13	14	15
# states	824073141	12470162233	202976401213	3535017524403	65573803186921
approx.	8.2×10^8	1.3×10^{10}	2.0×10^{11}	3.5×10^{12}	6.6×10^{13}

Blocksworld problems are **PSpace-complete**: even for a polynomial time algorithm, an exponential amount of memory is needed!

Time and Memory Requirements

Depth	Nodes	Time	Memory
0	1	1 ms	100 Byte
2	111	0.1 sec	11 KiloByte
4	11.111	11 sec	1 MegaByte
6	10^6	18 min	111 MegaByte
8	10^8	31 h	11 GigaByte
10	10^{10}	128 days	1 TeraByte
12	10^{12}	35 years	111 TeraByte
14	10^{14}	3500 years	11.111 TeraByte

Breadth-first search with braching factor $b = 10$, 1000 nodes/sec, 100 bytes/node $\hookrightarrow$ Memory requirements are the bigger problem!

Evaluation of DFS and BFS

- **Soundness:** A node s is only expanded to such a node s' where (s, s') is an arc in the state space (application of a legal operator whose preconditions are fulfilled in s)
- **Termination:** For finite sets of states guaranteed.
- **Completeness:** If a finite length solution exists.
- **Optimality:** Depth-first no, breadth-first yes
- worst case $O(b^d)$ for both, average case better for depth-first $\hookrightarrow$ If you know that there exist many solutions, that the average solution length is rather short and if the branching factor is rather high, use depth-first search, if you are not interested in the optimal but just in some admissible solution.
- Prolog is based on a depth-first search-strategy.
- Typical planning algorithms are depth-first.

Uniform Cost Search

- Variation of breadth-first search for operators with different costs.
- Path-cost function $g(n)$: summation of all costs on a path from the root node to the current node n.
- Costs must be positive, such that $g(n) < g(successor(n))$.
  Remark: This restriction is stronger then necessary. To omit non-termination when searching for an optimal solution it is enough to forbid *negative cycles*.
- Always sort the paths in ascending order of costs.
- If all operators have equal costs, uniform cost search behaves exactly like breadth-first search.
- Uniform cost search is closely related to *branch-and-bound algorithms* (cf. operations research).

Uniform Cost Algorithm

To conduct a uniform cost 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 and *sort* the queue with respect to costs.
- If the goal node is found, announce success; otherwise announce failure.

Uniform Cost Example

(omitting cycles)

((S).0)

((S A).3 (S B).4)

((S A B).5 (S A F).6 (S B).4)

sort

((S B).4 (S A B).5 (S A F).6)

((S B A).6 (S B C).5 (S B D).6 (S A B).5 (S A F).6)

sort

((S A B).5 (S B C).5 (S A F).6 (S B A).6 (S B D).6)

((S A B C).6 (S A B D).7 (S B C).5 (S A F).6 (S B A).6 (S B D).6)

sort

((S B C).5 (S A B C).6 (S A F).6 (S B A).6 (S B D).6 (S A B D).7)

((S B C F).7 (S A B C).6 (S A F).6 (S B A).6 (S B D).6 (S A B D).7)

Uniform Cost Example cont.

((S B C F).7 (S A B C).6 (S A F).6 (S B A).6 (S B D).6 (S A B D).7)

sort

((S A B C).6 (S A F).6 (S B A).6 (S B D).6 (S A B D).7 (S B C F).7)

sort

((S A B C F).8 (S A F).6 (S B A).6 (S B D).6 (S A B D).7 (S B C F).7)

sort

((S A F).6 (S B A).6 (S B D).6 (S A B D).7 (S B C F).7 (S A B C F).8)

Note: Termination if first path in the queue (i.e. shortest path) is solution, only then it is guaranteed that the found solution is optimal!

Further Search Algorithms

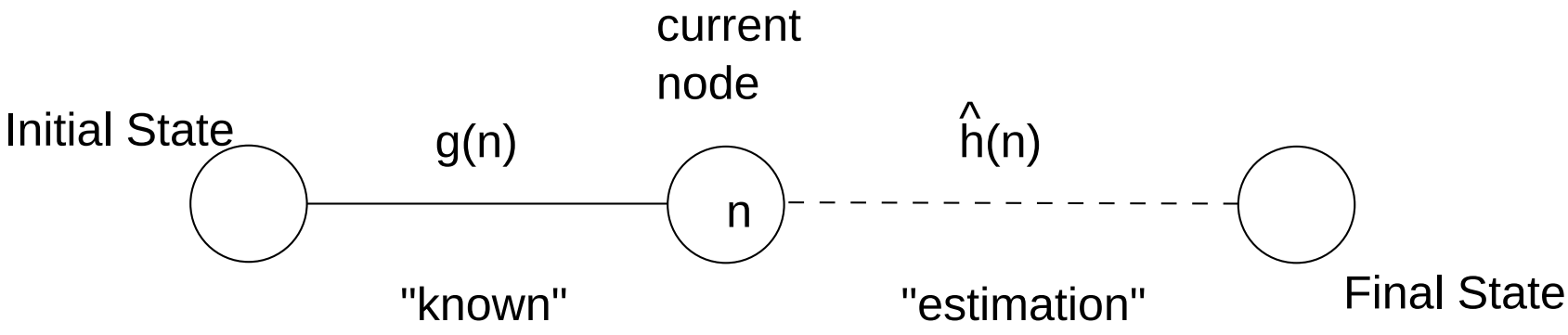
- **Depth-limited search:** Impose a cut-off (e.g. n for searching a path of length $n-1$), expand nodes with max. depth first until cut-off depth is reached (LIFO strategy, since variation of depth-first search).
- **Bidirectional search:** forward search from initial state & backward search from goal state, stop when the two searches meet. Average effort $O(b^{\frac{d}{2}})$ if testing whether the search fronts intersect has constant effort $O(1)$.
- In AI, the problem graph is typically not known. If the graph is known, to find *all* optimal paths in a graph with labeled arcs, **standard graph algorithms** can be used. E.g., the Dijkstra algorithm, solving the single source shortest paths problem (calculating the minimal spanning tree of a graph).

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*).

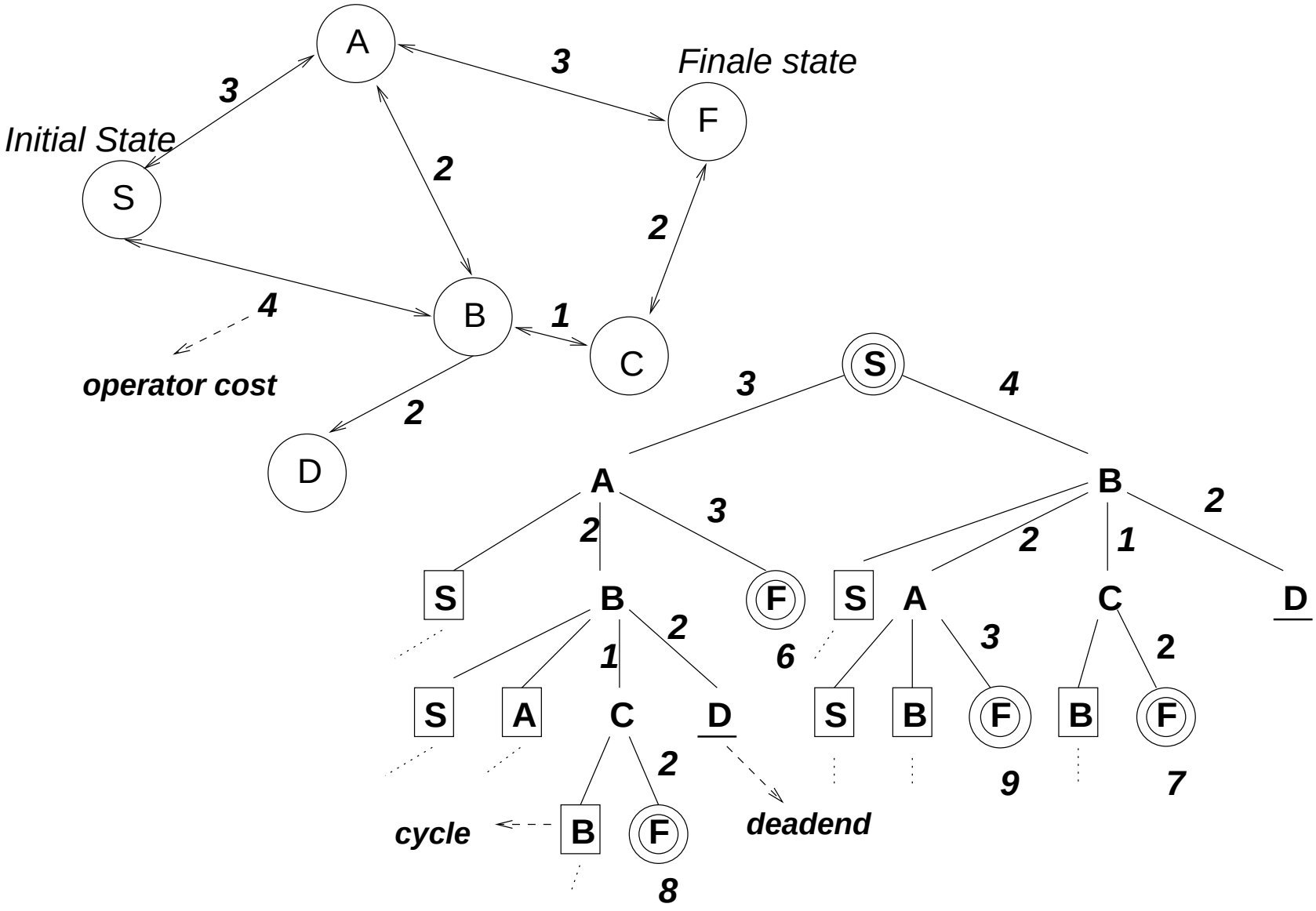
Heuristic Search Algorithms

- **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

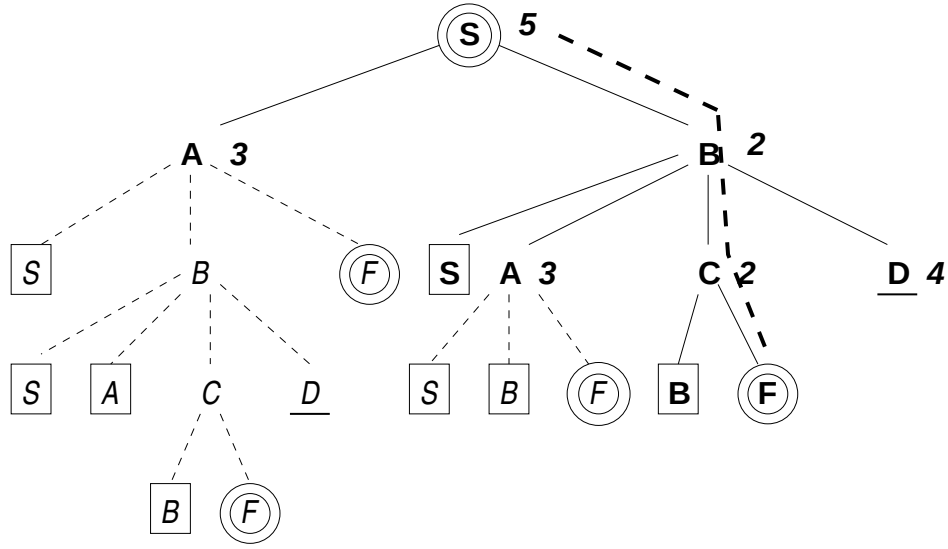
Winston, 1992 To conduct a hill climbing search,

- 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

h(S) = 5, h(A) = 3, h(B) = 2, h(C) = 2, h(D) = 4

((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)

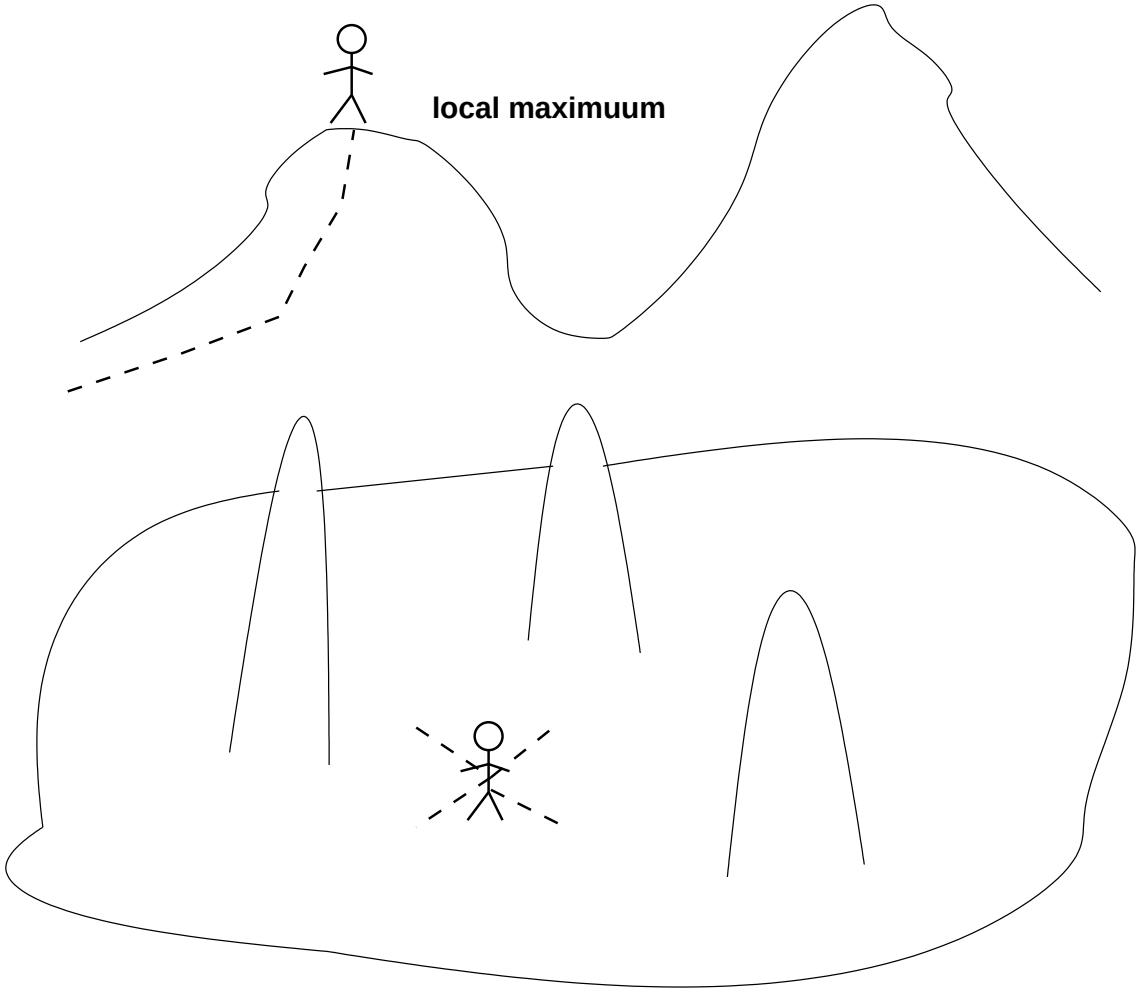


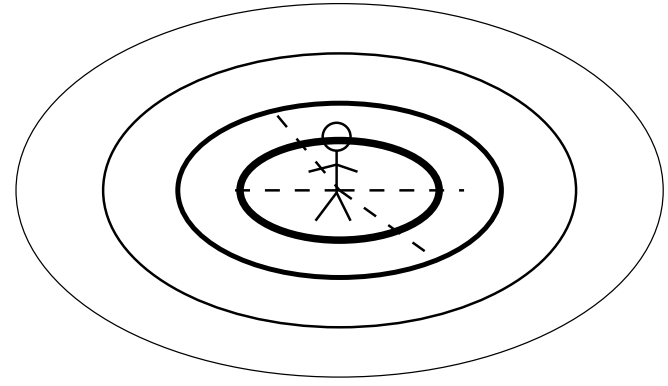
The heuristics was not optimal. If we look at the true costs (S A F) is the best solution!

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

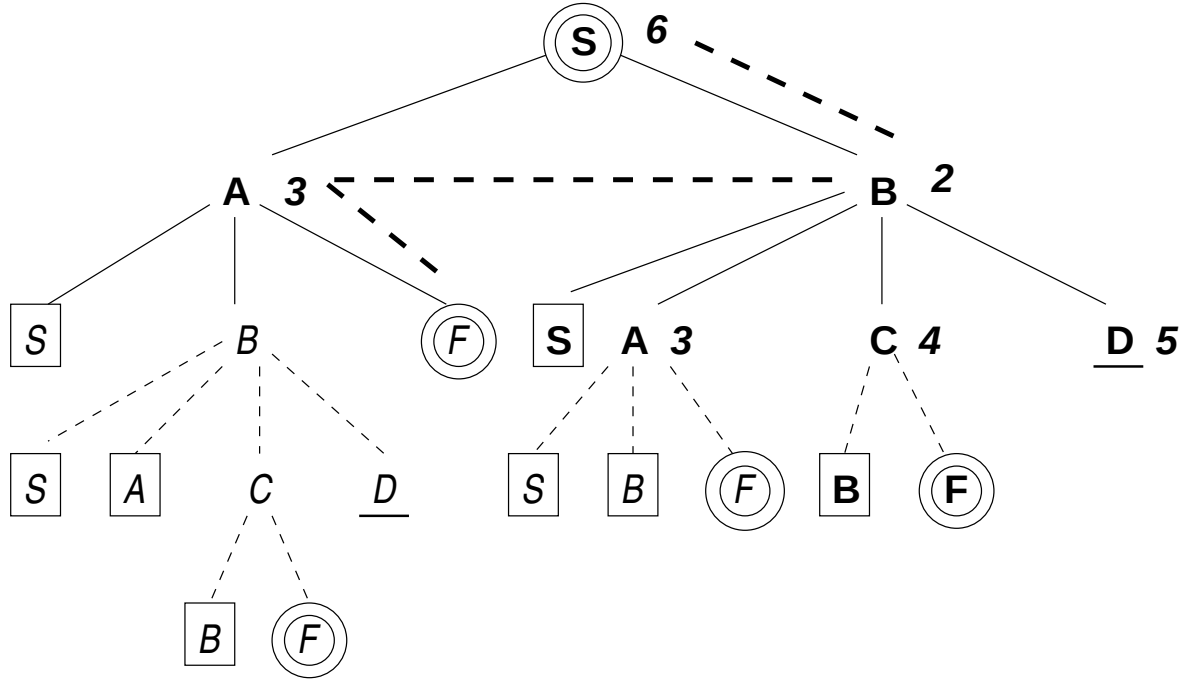
Winston, 1992 To conduct a best first search,

- 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

h(S) = 6, h(A) = 3, h(B) = 2, h(C) = 4, h(D) = 5

((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

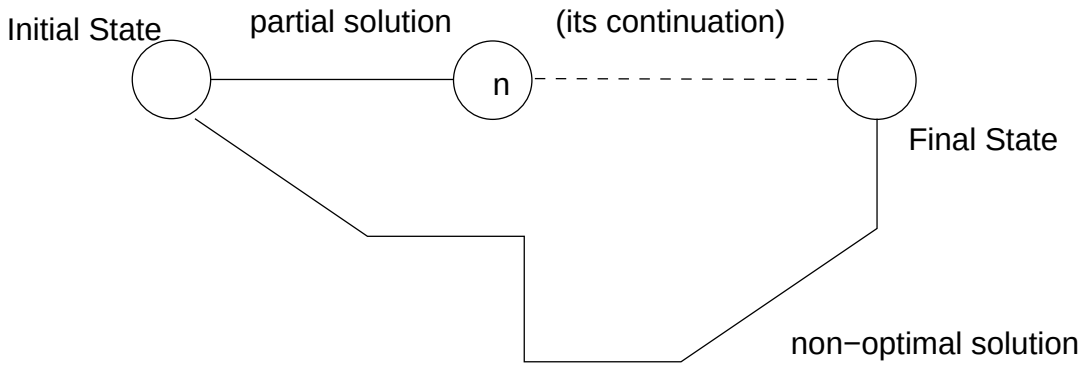
- Best First Search is not a local strategy: at each step the current *best* node is expanded, regardless on which partial path it is.
- It is probable but not sure that Best First Search finds an optimal solution. (depending on the quality of the heuristic function)

Optimal Search

- Inefficient, blind method: ‘British Museum Algorithm’
  - Generate *all* solution paths and select the best.
  - Generate-and-test algorithm, effort $O(d^b)$
- 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 (see lecture about uninformed search)
  - 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!

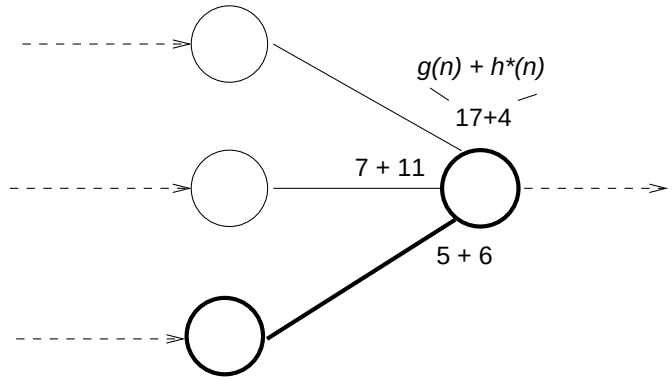


The Idea of A*

- 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.
  - 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.
  - Add the remaining new paths, if any, to the queue.
  - 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

$h^*(S) = 5,\ h^*(A) = 2,\ h^*(B) = 2,\ h^*(C) = 1,\ h^*(D) = 4$

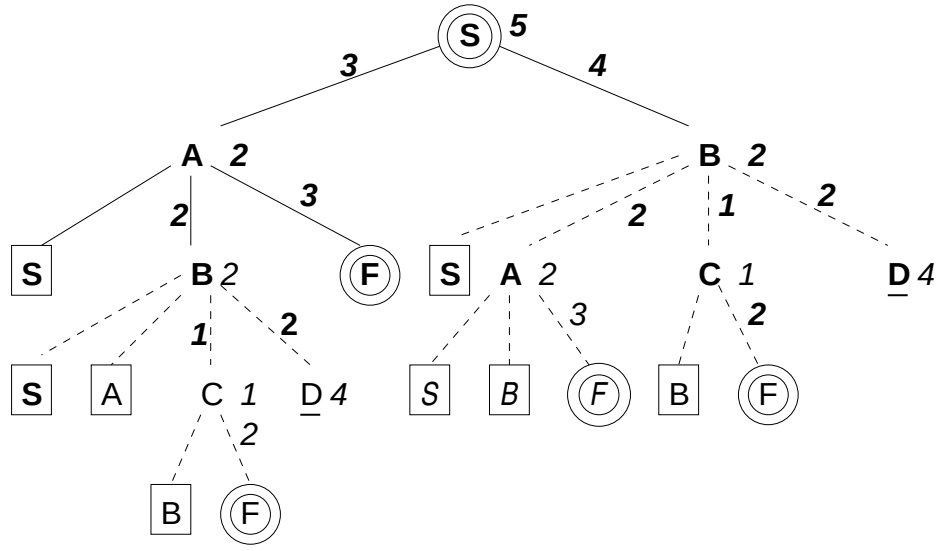
((S).$0 + 5$)

((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)



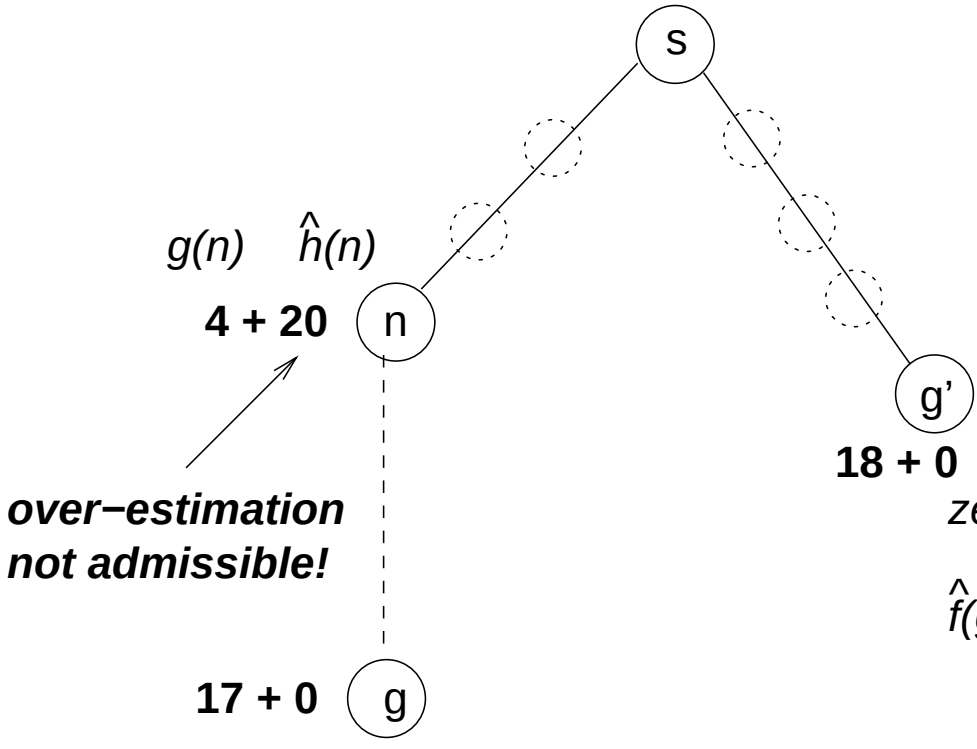
Admissibility of A*

- 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 hightens 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



already explored

knowledge of true costs so far

zero in goal state

$\hat{f}(g') = g(g') + 0 = f(g')$

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 lenghty proof is based on contradiction: Assume that A* expands a node n which is not expanded by another admissible algorithm A.

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

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.

How to design a Heuristic Function?

- Often, it is not very easy to come up with a good heuristics.
- For navigation problems: use Euclidian 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

5	4	
6	1	8
7	3	2

h1=7 h2=18

1	2	3
8		4
7	6	5

Goal State

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 vectures 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 inverstigations about metrics used by humans if they judge similarities (K.-F. Wender)

Summary: Search Algorithms

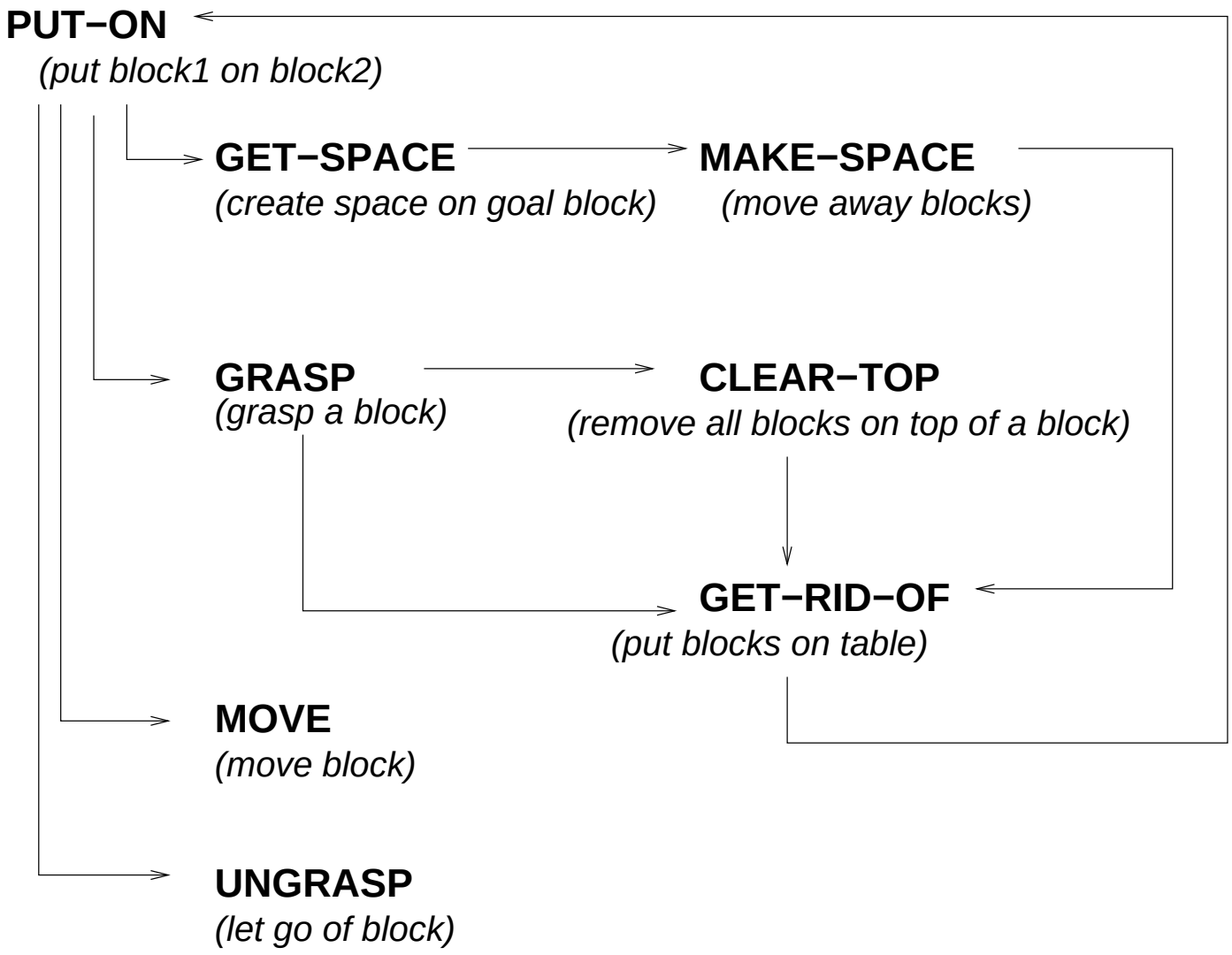
- Depth-first variants are in average more efficient than breadth-first variants, but there is no guarantee that an optimal solution can be found.
- Heuristic variant of depth-first search: Hill-Climbing/greedy search
  Heuristic varaint of breadth-first search: Best First search
- Breadth-first variants with costs are called branch-and-bound-algorithms: branch from a node to all successors, bound (do not follow) non-primising paths
  - Uniform-cost search: non-heuristic algorithm, only uses costs $g(n)$
  - A*: uses an admissible heuristic $h^*(n) \leq h(n)$
    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)

Problem Decomposition

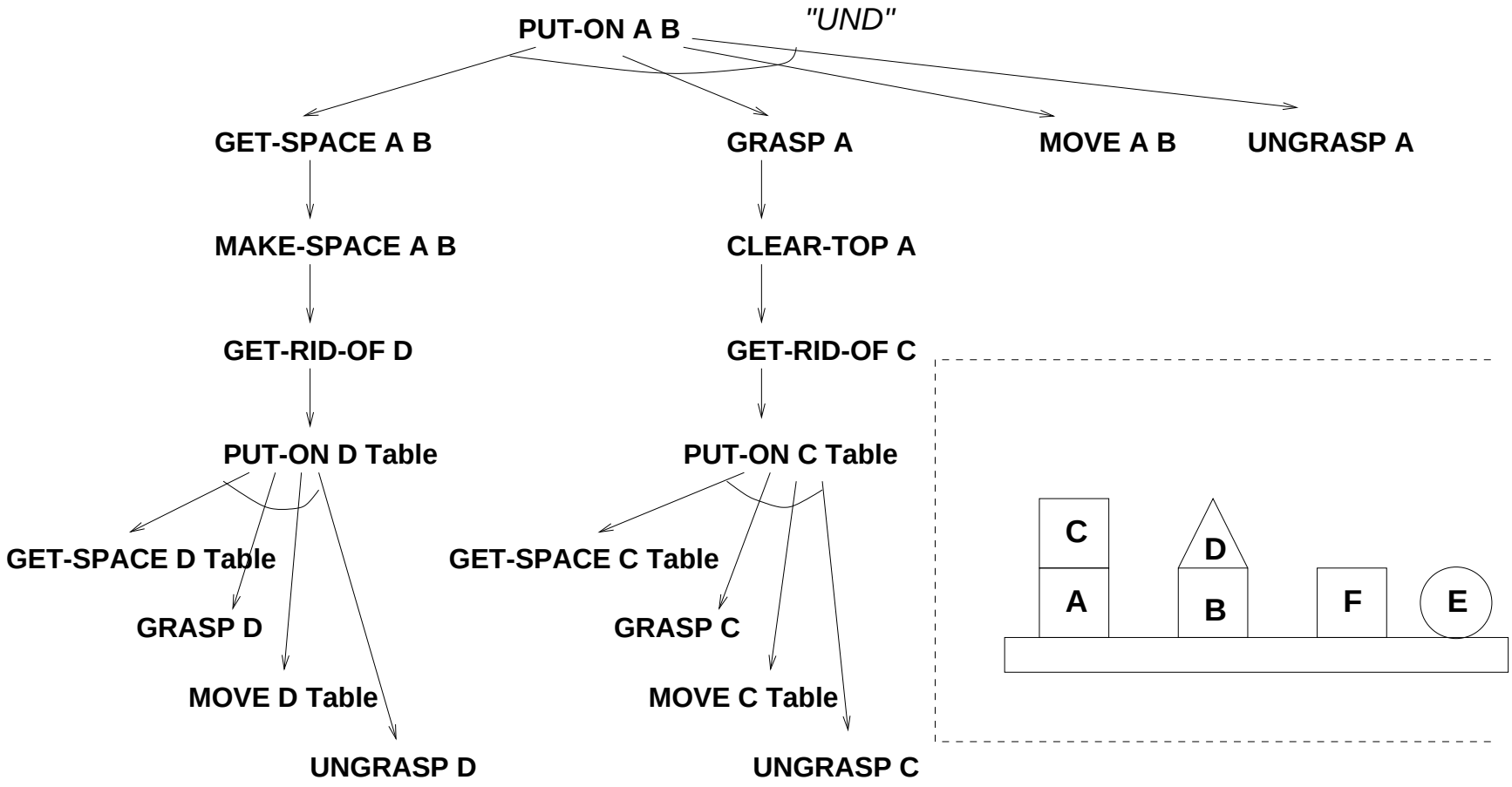
- Besides using heuristics, problem solving can be guided by knowledge about the problem structure.
- Problem decomposition: Dividing a problem in sub-problems
  $\hookrightarrow$ More complex production rules
- Advantage: dealing with smaller sub-problems and generating the solution by composition (“divide and conquer”)
- Representation: AND-OR Trees
  standard tree: each arc which exits a node represents an alternative (“or”); extension: specially mark edges which lead to sub-trees which must be all fulfilled for the current node to be fulfilled (“and”)

Example: MOVER

(Winston, 1992)



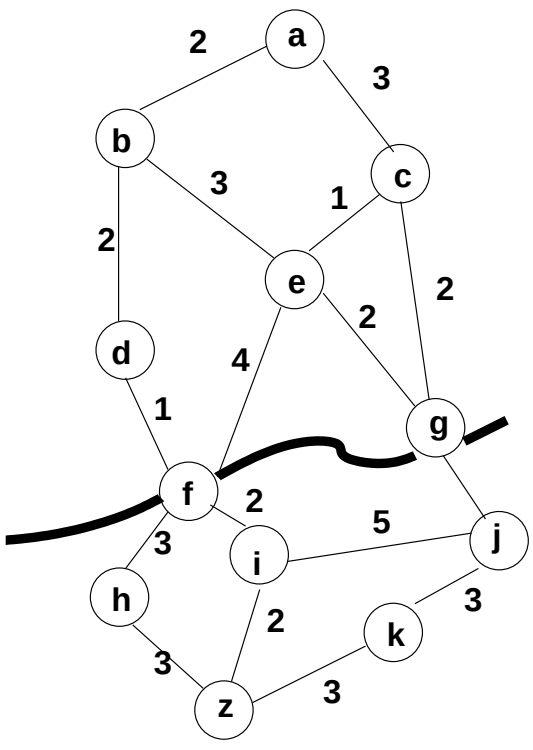
AND-OR Tree for MOVER

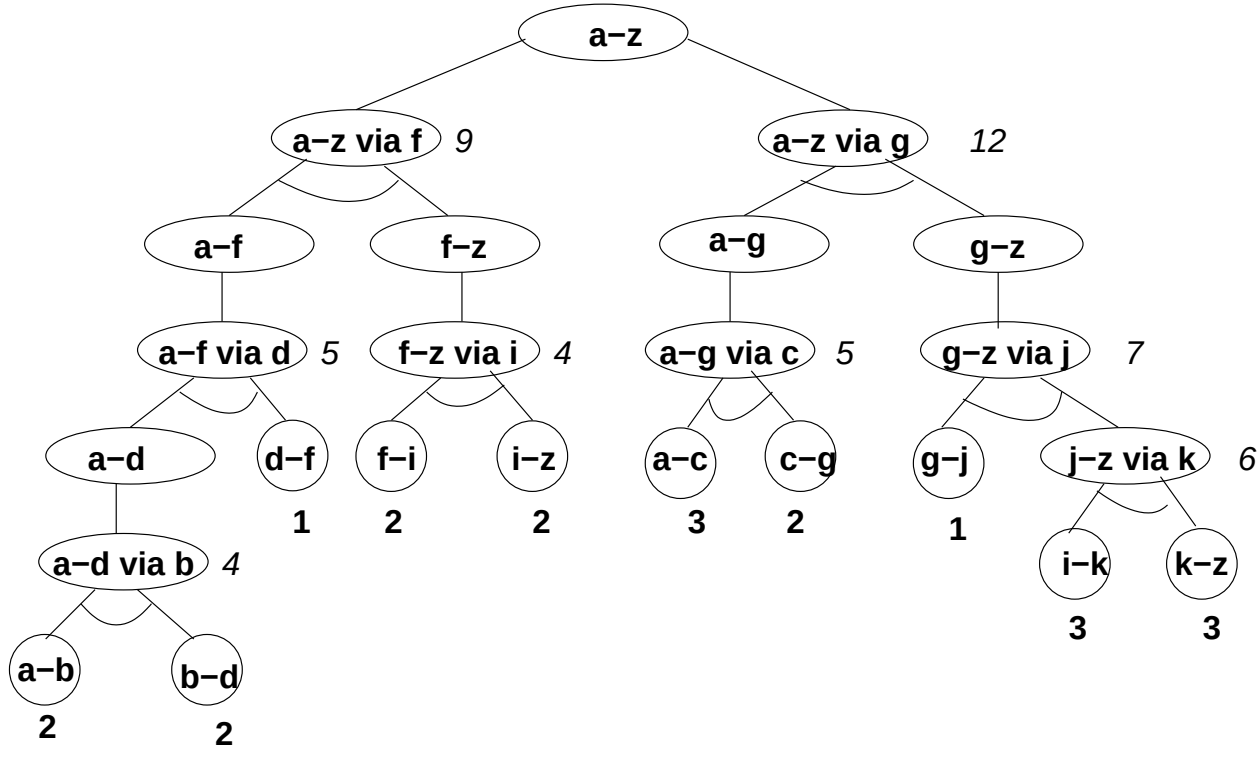


Example: Route Finding Problems

- “OR”-node: X - F (go from X to F), X - F is primitive, if F can be reached from X in one step (there exists an applicable operation)
  primitive sub-problems are leafs in the tree
- “AND”-node: X - Z via Y (go from X to Z via Y)
  “constraint”
- Problem solving: extracting an (optimal) AND-Tree
- Using costs: Each leaf is associated with its cost, the costs are propagated upwards in the tree, the AND-tree with the lowest costs is returned
- Algorithm: AO* (Nilsson)
- In planning: hierarchical planning

Example cont.





Two of many possible decompositions.