AI-KI-B

Problem Solving by Search in State-Spaces

Ute Schmid & Diedrich Wolter

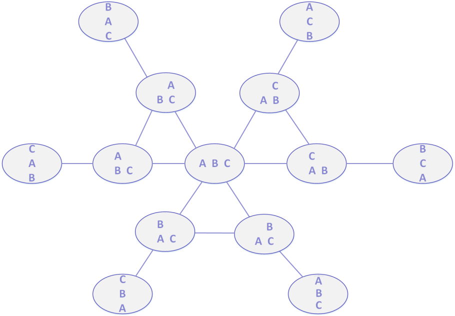
Practice: Johannes Rabold & student assistants

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

last change: 26. April 2020

In a nutshell: AI Search Algorithms

- Problem solving, planning, automated inference – and most other areas of AI – depend on intelligent search
- Basic concept: **state space** (or problem space)
  an abstract representation of the (real) states in the world and (real) actions which allow to reach one state from another



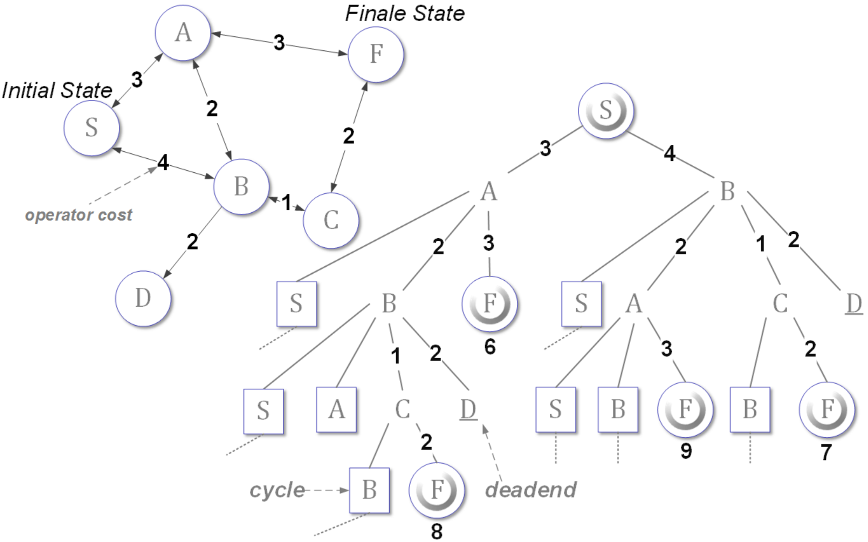
e.g.: initial state *tower 'C A B'* and goal state *tower 'A B C'* admissible sequence of actions: put C on table, put B on table, put A on table, put B on C, put A on B

In a nutshell: AI Search Algorithms

- Problem solving: search for a (shortest) path from an initial state to a state which fulfills a set of problem solving goals
- AI search algorithms mostly are variants of depth-first search where only a small part of states is explored – because state spaces are much too large to apply standard graph search algorithms (e.g. the Dijkstra algorithm)
- A good heuristics can reduce search effort dramatically. However, the heuristics must be carefully designed in a way that
  - the (shortest) solution has a chance to be found (**completeness**) and that
  - the result is indeed an admissible solution (**soundness**).

- **Introduction of running example**
- **Search Tree**
- **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*
- Generic Algortithm for Graph Search
- Designing Heuristic Functions

- In the following: We are not concerned with how a single state transformation is calculated (see state-space planning)
- We represent problems as graphs with abstract states (nodes) and abstract actions (arcs), i.e. state spaces
- If we label arcs with numbers, the numbers represent **costs** of the different actions (time, resources).
- Illustration: Navigation problem with nodes as cities and arcs as direct connections; blocksworld problems with nodes as constellations of blocks and arcs as put/puttable operators (might have different costs for different blocks)
- Please note: In general the state-space (graph) is not given explicitly!
- A part of the state space is constructed during search (the states which we explore) in form of a **search tree**.


Finale State
Initial State
operator cost
cycle
deadend
6
8
9
7

- 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

- 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)

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:

$\Rightarrow$ The **Match-Select-Apply Cycle** is the core of "production systems" (see human problem solving)

$\Rightarrow$ Select is realized with respect to the different search algorithms

Problem Solving

- A **problem** is defined by the following components:
  - Initial state
  - Actions or successor function $S(x)$, can be associated with costs
  - Goal test: explicit (current state equals goal state or current state includes all goals) or implicit (as boolean test, e.g. checkmate(x))
- A **problem solution** is a sequence of actions leading from the initial state to a goal state
- Problem solving and planning
  - Problem solving: focus on search algorithms
  - Planning: language for representing problem domains and problems + planning (i.e. search) algorithm (e.g. PDDL as representation language, FF-plan as algorithm)

State-Space and Admissibility

- A search algorithm generates a sequence of actions.
- This sequence of actions is called a solution of a problem or admissible,
  - if the sequence transforms the initial state in a goal state and
  - if all states on the solution path are states which are possible in the problem domain.
- Proofs of formal characteristics of a search algorithm (e.g. soundness and completeness) are based on the notion of state-space as the world (semantic) in which the action sequence is executed.
- The concept of a state space has been introduced by the AI pioneers Newell and Simon under the name of a problem space.

Depth-First Search

- Construct *one* path from initial to final state.
- Backtracking:
  Go back to predecessor state and try to generate another successor state (if none exists, backtrack again etc.), if:
  - the reached state is a dead-end, 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

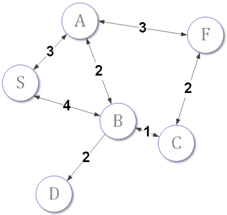
- The most parsimonious way to store the (partial) solution path is to push always only the current state on the stack.
  Problem: additional infrastructure for backtracking
  (remember which operators were already applied to a fixed state)
- In the following: push always the partial solution path to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

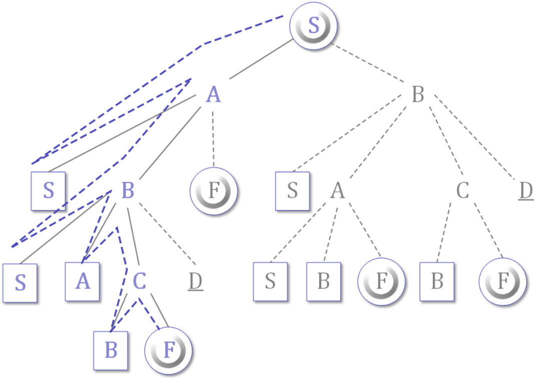
Winston, 1992

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.

((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))





- The search tree is expanded level-wise.
- 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 not 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: $O(b^d)$

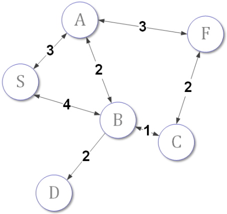
Breadth-First Algorithm

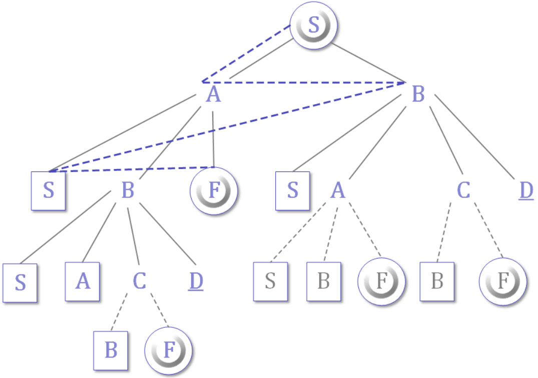
Winston, 1992

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.

((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)])





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	Kilo Byte
4	11.111	11 sec	1	Mega Byte
6	10^6	18 min	111	Mega Byte
8	10^8	31 h	11	Giga Byte
10	10^{10}	128 days	1	Tera Byte
12	10^{12}	35 years	111	Tera Byte
14	10^{14}	3500 years	11.111	Tera Byte

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

Evaluation of DFS and BFS

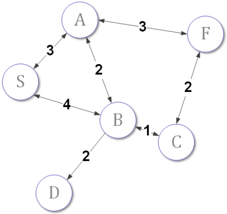
- **Soundness:** A node s is only expanded to such a node s' where (s, s') is an arc in the state space (application of a legal operator whose preconditions are fulfilled in s)
- **Termination:** For finite sets of states guaranteed.
- **Completeness:** If a finite length solution exists.
- **Optimality:** 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.
  Remark: Please note that $g(n)$ denotes the accumulated costs for one specific path from the root to node n: $\sum_{i=0}^{n} c(i, i+1)$ if nodes i and $i+1$ are on the selected path.
- 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).

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.



(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)
((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!
- A more efficient realisation of UCS would override the alphabetical order of terminal nodes when the terminal node is the goal state (of course while keeping the order wrt costs!)

- **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).