Artificial Intelligence 1

Prof. Dr. Frank Hopfgartner
Dr. Matthias Horbach

Institute for Web Science and Technologies (WeST)
University of Koblenz

Overview

Informed search 1/2

- Combinatorial explosion: many successor states imply long runtime
- Example chess: we have 20 successor states from the initial state alone (states later in game may have many more successor states)

Idea of informed search: "guess" the next best state

Even better: "guess well" by using domain knowledge to assess suitability of states (heuristics)

Informed search 2/2

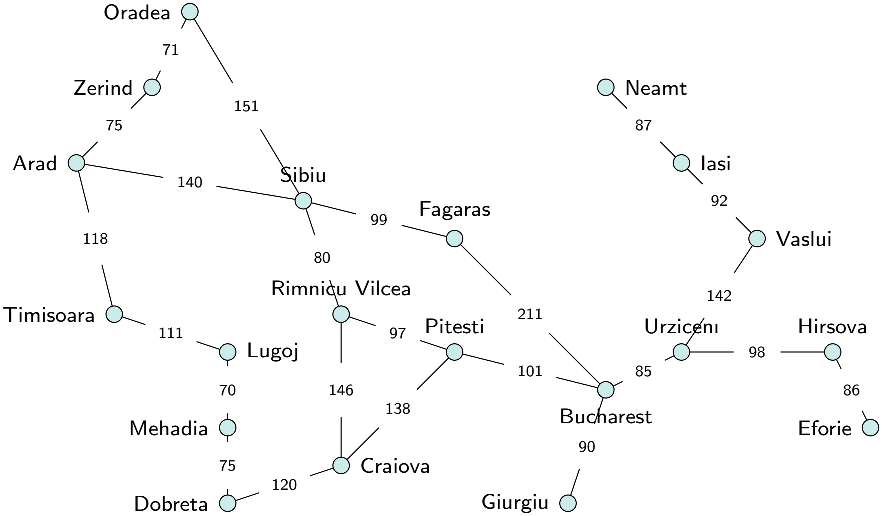
- Analysis (uninformed) search:
  - In general, search strategies are characterised by the way the next node to be explored is selected
  - Uninformed strategies sort nodes by problem-independent features (such as distances to starting node)
- Informed search (*best-first search*):
  - Apply *state evaluation function* $f(n)$ on every successors node n of the current node m
  - Select the successor node that maximises f (add this one to frontier)
  - Alternative: sort successor states with descending order and put them on a stack (enables back-tracking)

State evaluation functions

- The state evaluation function $f(n)$ implements a heuristic
- A heuristic is a problem-dependent *strategy*
  - Examples:
    - Chess: Prefer states where the opponent has less pieces
    - Sokoban: Prefer states where the sum of the distances of all rocks to their respective goal is minimal
  - Heuristics do not need to be correct and generally applicable
- A heuristic gives an estimation of the costs of the node to the goal
- In contrast to the actual costs, a heuristic should be easy to calculate

Another example: straight-line distance as heuristic for shortest paths

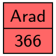
Example: shortest paths



Greedy best-first search

- Evaluation function is exactly the heuristic: $f(n) = h(n)$
- Underestimates the costs from a node n to the goal
- $h_{sld}(n) =$"straight-line distance from n to Bucharest"
- Greedy best-first search selects the node that is closest to the goal according to f

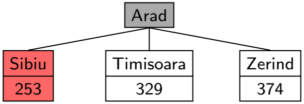
Example:



Greedy best-first search

- Evaluation function is exactly the heuristic: $f(n) = h(n)$
- Underestimates the costs from a node n to the goal
- $h_{sld}(n) =$ "straight-line distance from n to Bucharest"
- Greedy best-first search selects the node that is closest to the goal according to f

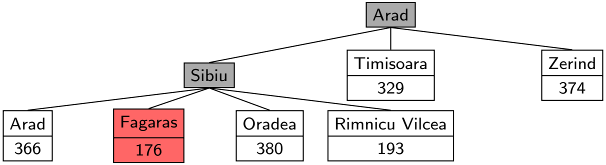
Example:



Greedy best-first search

- Evaluation function is exactly the heuristic: $f(n) = h(n)$
- Underestimates the costs from a node n to the goal
- $h_{sld}(n) =$ "straight-line distance from n to Bucharest"
- Greedy best-first search selects the node that is closest to the goal according to f

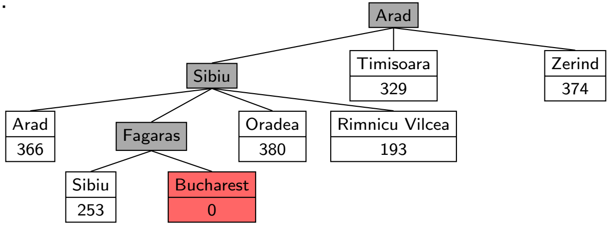
Example:



Greedy best-first search

- Evaluation function is exactly the heuristic: $f(n) = h(n)$
- Underestimates the costs from a node n to the goal
- $h_{sld}(n) =$ "straight-line distance from n to Bucharest"
- Greedy best-first search selects the node that is closest to the goal according to f

Example:

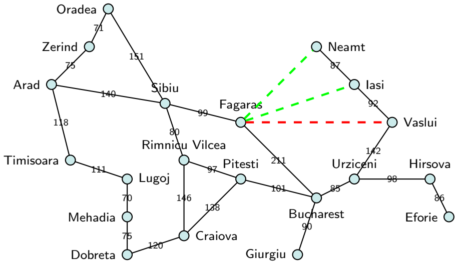


Greedy best-first search: properties 1/3

Completeness

- Greedy best-first search is *not* complete
- Loops may occur during search

Example: Iasi $\rightarrow$ Fagaras
Node selection: Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \ldots$



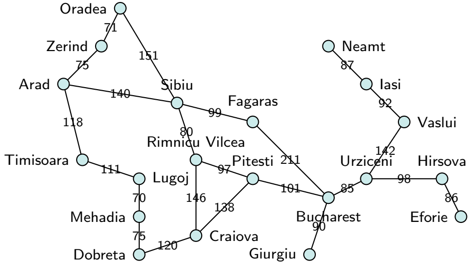
Note: search node $\neq$ "geographical" node

Greedy best-first search: properties 2/3

Optimality

- Greedy best-first search is *not* optimal
- Local optima do not need to be global optima

Example: Arad $\rightarrow$ Sibiu $\rightarrow$ Fagaras $\rightarrow$ Bucharest



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	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search: properties 3/3

Let n be the maximal number of successors of any one node,
e the distance to the goal node that is found first, and
d the maximal search depth
(=maximal distance of a childless node)

Theorem

Greedy best-first search is not complete, has runtime complexity $O(n^d)$, space complexity $O(1)$, and is not optimal.

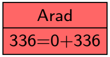
Remarks

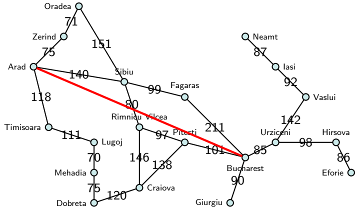
- Completeness can be guaranteed if we add duplicate detection
- Runtime is on average significantly better if a good heuristic is used
- In the best case we have runtime $O(e)$ (we find a direct path to the goal)
- With duplicate detection we have to maintain a data structure with all visited node: space complexity $O(n^d)$

A*

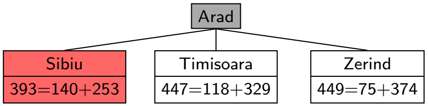
- Improved version of greedy best-first search
- Idea: Do not only use heuristic but also already incurred costs
- State evaluation function: $f(n) = g(n) + h(n)$
  - $g(n) =$ actual costs to get from the start node to the current node n
  - $h(n) =$ estimation of the costs to get from node n to the goal node
  - $f(n) =$ estimated costs to get from the start node to the goal via n

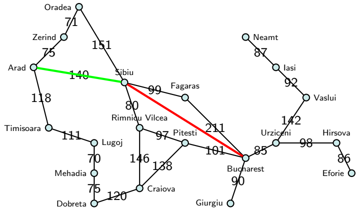
A* example


Arad
336=0+336

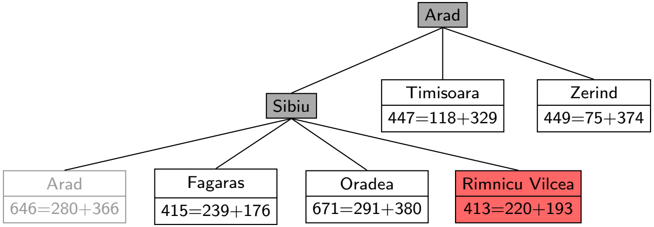

Oradea
Zerind
Arad
Sibiu
Fagaras
Rimnicu Vilcea
Pitesti
Timisoara
Lugoj
Mehadia
Dobreta
Craiova
Bucharest
Giurgiu
Urziceni
Hirsova
Eforie
Vaslui
Iasi
Neamt
71
75
151
140
118
99
80
211
97
101
146
138
111
70
75
120
90
85
98
86
142
92
87

A* example

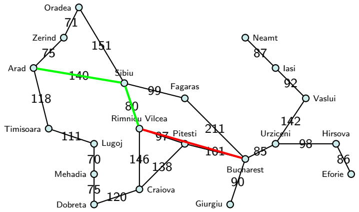

Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
146
138
101
Mehadia
75
120
Dobreta
Craiova
Bucharest
90
Giurgiu
Neamt
87
Iasi
92
Vaslui
142
Urziceni
85
98
Hirsova
86
Eforie

A* example

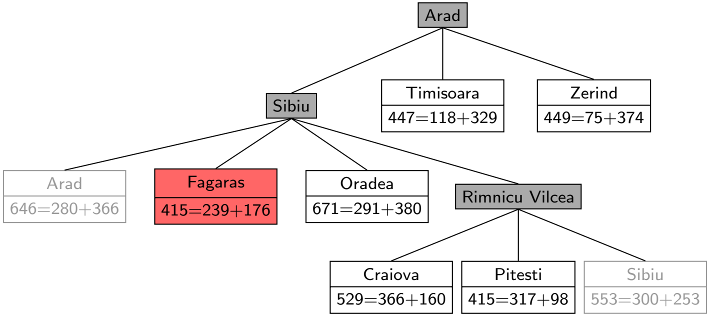


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

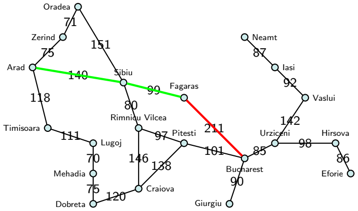


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
146
138
101
Mehadia
75
120
Craiova
Dobreta
Bucharest
90
Giurgiu
Neamt
87
Iasi
92
Vaslui
142
Urziceni
85
98
Hirsova
86
Eforie

A* example

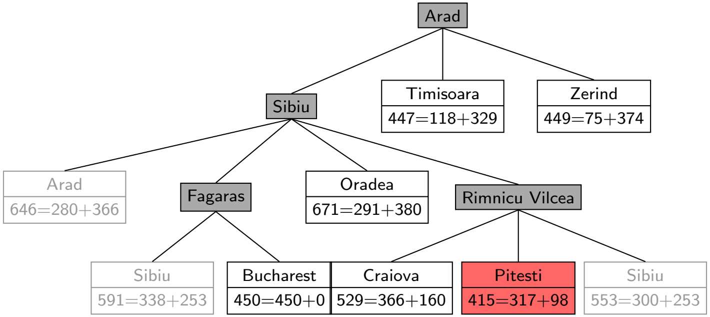


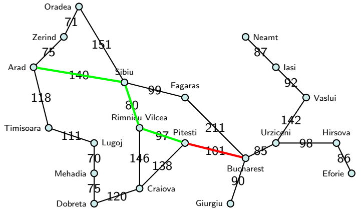
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
529=366+160
Pitesti
415=317+98
Sibiu
553=300+253



Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
146
138
101
Mehadia
75
120
Craiova
Dobreta
Bucharest
90
Giurgiu
Neamt
87
Iasi
92
Vaslui
142
Urziceni
85
98
Hirsova
86
Eforie

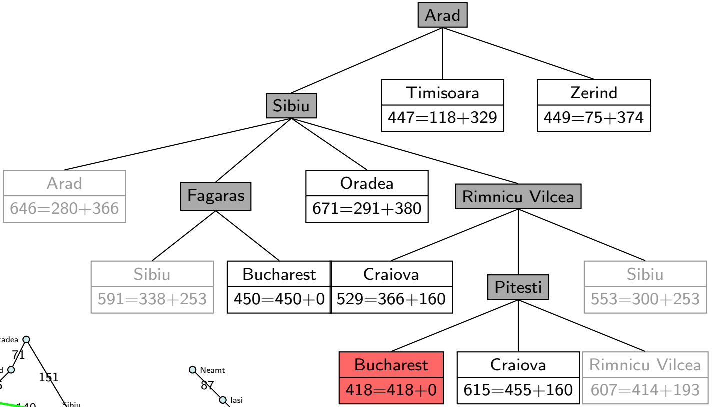
A* example

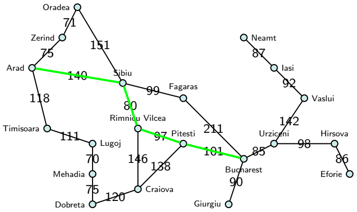




Note: although we have already found the goal (Bucharest), we select Pitesti as the next node (cf. with greedy best-first search).

A* example


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
529=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
146
138
101
Mehadia
75
120
Dobreta
Craiova
Bucharest
90
Giurgiu
85
Urziceni
98
Hirsova
86
Eforie
142
Vaslui
92
Iasi
87
Neamt

A* pseudo code

```
Algorithm astar
Input:
   Start node I, successor state function Succ(.),
   goal test function isGoal(.), heuristic h(.),
   edge cost function c(.,.)
Output: FAIL/SUCCESS

Frontier = {I}, Explored = {}, g(I) = 0, f(I) = g(I) + h(I)
while(true){
  if Frontier = {} terminate with FAIL
  Let n from Frontier with minimum heuristic cost f(n)
    and save n to Explored
  if isGoal(n) terminate with SUCCESS
  For each m in Succ(n):
    if m in Explored
      continue
    if m in Frontier
      g(m) = min { g(m) , g(n) + c(n, m) }
      f(m) = g(m) + h(m)
    if m not in Frontier
      g(m) = g(n) + c(n, m)
      f(m) = g(m) + h(m)
      insert m to Frontier
}
```

Properties 1/2

Let n be the maximal number of successors of any one node,
e the distance to the goal node that is found first,
d the maximal search depth (=maximal distance of a childless node), and
edge costs be bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ).

Theorem

A search is complete, has runtime complexity $O(n^d)$, space complexity $O(n^d)$, and is optimal if the heuristic is* **admissible***.*

Remarks

- Runtime and space complexity depend on the heuristic and the used data structures for `Frontier` and `Explored`
- Average case runtime and space complexity is usually significantly better when a good heuristic is used
- In order for A* to be optimal, *admissibility* of the heuristic is a mandatory property.

Properties 2/2

A* is also optimal in terms of efficiency: there is no other algorithm that can find a solution faster using the same heuristic.

Furthermore, A* is a very general algorithm:

- If $h(n) = 0$ for all nodes then A* is equivalent to the Dijkstra algorithm
- If $c(n, m) = 0$ for all edges then A* is equivalent to greedy best-first search
- If $c(n, m) = 1$ for all edges and $h(n) = 0$ for all nodes then A* is equivalent to breadth-first search
- If $c(n, m) = -1$ for all edges and $h(n) = 0$ for all nodes then A* is equivalent to depth-first search

Admissible heuristics

Let $c(n)$ be the minimal costs to get from node n to a goal node (e. g. the actual travelling distance from n to the goal).

Definition

A heuristic h is *admissible* iff $h(n) \leq c(n)$ for all n.

Example

$h_{sld}(n) =$ "straight-line distance from n to Bucharest" is admissible: the actual travelling distance from a place A to a place B is never smaller than the straight-line distance.

Monotonous heuristics

A sufficient criterion for admissibility is monotony.

Definition

A heuristic h is *monotonous* iff $h(n) \leq c(n, m) + h(m)$ for all nodes n and edges (n, m).

In other words, the costs of single transition are never overestimated.

Theorem

If h is monotonous then h is admissible.

When defining a heuristic it is therefore sufficient to consider only single transitions.

Dominating heuristics

Definition

If h_1 and h_2 are admissible heuristics, h_2 *dominates* h_1 iff $h_2(n) \geq h_1(n)$ for all nodes n.

If h_2 dominates h_1 then A* with h_2 is always faster than A* with h_1 (h_2 is always closer to the actual costs).

Theorem

If h_1 and h_2 are admissible then

$$h_3(n) = \max\{h_1(n), h_2(n)\}$$

is admissible and dominates both h_1 and h_2.

Heuristics for the 8 puzzle 1/2

7	2	4
5	■	6
8	3	1

initial state

1	2	3
4	5	6
7	8	■

goal state

- $h_{mis}(n) =$ "Number of misplaced tiles"
  - Admissible: every misplaced tile has to be moved at least once
  - Example (initial state above): $h_{mis}(start) = 6$
- $h_{man}(n) =$ "Sum of the manhattan distances of every tile to its goal position"
  - How many steps do we have to move every tile?
  - Admissible: every misplaced tile has to be moved at least this many times.
  - Example (initial state above): $h_{man}(start) = 14$

Heuristic for the 8 puzzle 2/2

7	2	4
5	▨	6
8	3	1

Startzustand

1	2	3
4	5	6
7	8	▨

Zielzustand

- $h_{mis}(n) =$ "Number of misplaced tiles"
- $h_{man}(n) =$ "Sum of the manhattan distances of every tile to its goal position"

Observation: h_{man} dominates h_{mis}

Selecting the right heuristic has much influence on the runtime (Standard laptop 2015):
8 puzzle A* with h_{mis}: $\approx$ 5 min.
8 puzzle A* with h_{man}: $\approx$ 15 sec.

Informed search: prospects

- A* is the algorithm that is used as the foundation of most used search algorithms today
- In its presented form it suffers from severe space complexity problems
- Extensions of A*:
    - IDA* (*Iterative Deepening A**): similar trick as extending depth-first search to iterative depth-first search
    - SMA* (*Simple Memory-bounded A**): whenever memory is exhausted delete nodes with worst f value

Chapter 3.2: Informed search

Summary

Chapter 3.2: Summary

- Idea of informed search: “guess well” the next best state
- Use heuristics to evaluate states
    - Admissible heuristics
    - Monotonous heuristics
    - Dominance
- Algorithms
    - Greedy best-first search
    - A*

Overview

Overview

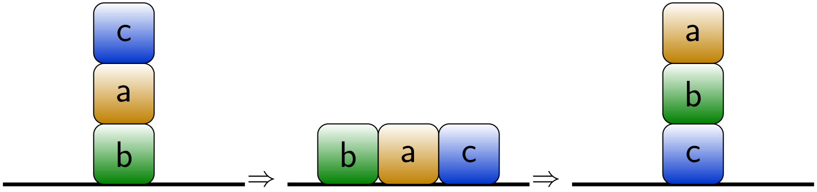
- Classical logics (propositional logic and first-order logic) enable knowledge representation for static situations: truth values cannot change, otherwise we get inconsistencies!
- Search problems are dynamic problems, actions change the state of the world

Desired:

- Using formal systems for the formalisation of search problems
- In particular: using such systems for *planning problems* (=search problems were we are interested in sequences of actions to reach a goal)

$\rightarrow$ situation calculus

Example: blocks world



Actions and states

Actions and states can be represented by first-order formulas:

Formalisation of actions through functional expressions:

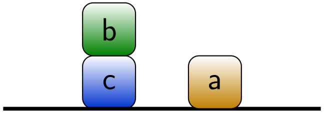
- Take a block X from the table and put it onto another block Y: stack(X,Y)
- Take block X from another block Y and put it on the table: unstack(X,Y)

Formalisation of states through predicates:

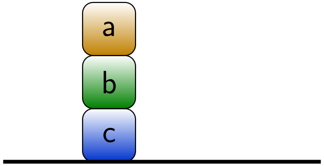
- on(X,Y): block X is on top of block Y
- ontable(X): block X is on the table
- clear(X): there is no other block on top of block X

Example

Initial state: ontable(c)
ontable(a)
on(b,c)
clear(a)
clear(b)



Goal state: on(a,b)
on(b,c)
ontable(c)
clear(a)



Situation calculus

Every predicate has an additional *situation parameter* s.

Initial situation (no action has been performed yet): s_0

Example: ontable(c,s_0), ontable(a,s_0),on(b,c,s_0), clear(a,s_0),clear(b,s_0)

Special binary function symbol *do*:

For an action a and a situation s is

$$do(a, s)$$

the situation when a has been performed in s.

Situations and states

- Situations are *world histories*, i. e., a sequence of changes on the world caused by certain actions
- Formally, a situation is a *term* and corresponds to a sequence of actions.

  $$do(stack(b, a), do(stack(a, c), do(unstack(b, c), s_0)))$$

  is the situation corresponding to the action sequence

  $$[unstack(b, c), stack(a, c), stack(b, a)]$$

- Actions cause changes in situations (and usually also changes in the state):

  $$\begin{aligned} &on(b, c, s) \\ &clear(c, do(unstack(b, c), s)) \end{aligned}$$

Actions, situations, and states

Different sequences of actions always lead to different situations, but can result in the same state.

Example

Blocks world with four blocks a, b, c, d, all being on the table.

$$
\begin{aligned}
s_1 &= do(stack(c, d), do(stack(a, b), s_0)) \\
s_2 &= do(stack(a, b), do(stack(c, d), s_0))
\end{aligned}
$$

$\rightarrow$ two different situations that result in the same state.

Therefore: situation $\neq$ state

Fluents

Fluents are predicates and functions where their truth value can change depending on the situtation.

Predicate+situation symbol $\equiv$ (relational) fluent
$=$ relation with situation-dependent truth value

Example

In situation s block b is on top of block c:

$$on(b, c, s)$$

In the situation that results from removing b from c in s, there is no block on c anymore:

$$clear(c, do(unstack(b, c), s))$$

Action preconditions

We have a special predicate symbol that encodes when an action is applicable in some situation:

$$poss(a, s)$$

For every action a we need to define *action preconditions* to define the applicability of actions.

Example

$$\begin{aligned} poss(stack(X, Y), S) \equiv\ & ontable(X, S) \wedge clear(X, S) \wedge \\ & clear(Y, S) \wedge X \neq Y \\ poss(unstack(X, Y), S) \equiv\ & on(X, Y, S) \wedge clear(X, S) \end{aligned}$$

Effect axioms

- Effect axioms describe effects actions have on situations
- We need to specify them for every (action, fluent) pair and usually distinguish between positive and negative effects.
- It is important to ensure that the action can be executed

$$\text{Action precondition} \Rightarrow \text{effect}$$
$$poss(a, s) \Rightarrow Fluent(\ldots, do(a, s))$$

Example

$$poss(stack(X, Y), S) \Rightarrow on(X, Y, do(stack(X, Y), S))$$
$$poss(stack(X, Y), S) \Rightarrow \neg clear(Y, do(stack(X, Y), S))$$
$$poss(stack(X, Y), S) \Rightarrow \neg ontable(X, do(stack(X, Y), S))$$

(stack,clear) and (stack,ontable) have no positive action effects

Frame axioms

- The counterpart to effect axioms are frame axioms
- They model which fluents are *not* changed by an action
  - When an object is dropped its color does not change

$$color(X, C, S) \Rightarrow color(X, C, do(drop(X), S))$$

  - An object X that is not broken does not get broken when an object Y is dropped, unless X is the same as Y and X is fragile.

$$\neg broken(X, S) \wedge (X \neq Y \vee \neg fragile(X)) \\ \Rightarrow \neg broken(X, do(drop(Y), s))$$

- Problem: these axioms *cannot* be derived directly from action preconditions and effect axioms.

Challenges in the situation calculus 1/2

- *Frame problem*: which fluents do not change when executing an action?
- *Qualification problem*: what are the exact preconditions of each action?

  Example: Precondition for $\mathit{unstack}(X, Y)$ is that X is on top of Y and X is clear. Possible (real-world) problems: X is too heavy, grappler of the robot is defect, X is glued onto Y, ...

Challenges in the situation calculus 2/2

- *Ramification problem*: what are the indirect effects of an action?

  Example: A robots moves a box from room $r1$ to room $r2$.

  direct effect: the position of the box is room $r2$

  indirect effect: a label on the box is now in room $r2$ as well

  however: the color of the box stays the same

  $\rightarrow$ problems with the frame problem as well

Limitations of the situation calculus

- Single agent: there are no exogenous actions of other agents in the situation calculus
- Temporal aspect: there is no duration of actions (actions take place immediately)
- No concurrency: no parallel actions, actions are always arranged in sequence
- Discrete actions: no continuous actions
- Only hypothetical actions: one cannot express that an actions has actually taken place or will be executed in the future
- Only primitive actions: actions cannot be composed from other actions

Computational problems in the situation calculus 1/2

The situation calculus is simple but the foundation of most action logics. It can be used, among other, to investigate the following computational questions:

- Projection: Given an initial situation and a sequence of actions, determine what formulas are true after executing that sequence
- Legality: A situation is legal, if it is either the initial situation or the outcome of applying an executable action on a legal situation. Determine whether a sequence of actions results in a legal situation.

Computational problems in the situation calculus 2/2

- Planning: Given an initial situation and a goal description, determine a sequence of actions such that the resulting situation satisfies the goal description
  - This is a particular search problem
    - Initial situation=start node
    - Goal description=Set of goal nodes
    - Execution of an action in a situation=successor of a node
    - Both uninformed and informed search algorithms can be applied
  - Note: situation calculus can be represented in second-order logic
    - Logical resolution algorithms can be applied for planning

STRIPS

- The situation calculus is a powerful modelling language for planning problems
- Practically quite inefficient
- Simplification: STRIPS = STanford Research Institute Problem Solver [1971]
  - Explicit modelling of state transitions through listing of changes
  - Sets of (grounded) atoms are used to represent states:

    STRIPS data base:

    $$\{ontable(a), on(b, c), ontable(c), clear(a), clear(b)\}$$

    STRIPS goal:

    $$\{on(a, b), on(b, c)\}$$

STRIPS operators 1/2

Operators

- are executed through direct manipulation of the data base
- consist of three parts
    1. a set C of ground atoms modelling *(pre-)conditions* of the operator;
    2. a set D of ground atoms, called D list (*delete list*);
    3. a set A of ground atoms, called A list (*add list*).

STRIPS operators 2/2

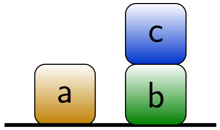
- An operator can be applied on a data base S if S satisfies all its conditions
- Application consists of two steps
  - First we delete all atoms of the D list from S.
  - Then all atoms of the A list are added to S. In particular, all atoms are retained in S that are not explicitly deleted.
- Solution to the frame problem in STRIPS: all elements of a STRIPS data base that are not explicitly mentioned in the postconditions of an operators are retained (*inertia*).

STRIPS example 1/3

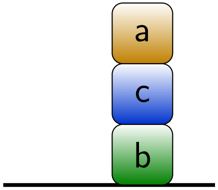
For *stack* and *unstack* from the blocks world we can define the following STRIPS operators:

stack(X,Y):	C: ontable(X), clear(X), clear(Y)
	D: ontable(X), clear(Y)
	A: on(X,Y)
unstack(X,Y)	C: on(X,Y), clear(X)
	D: on(X,Y)
	A: ontable(X), clear(Y)

STRIPS example 2/3



$\longrightarrow \textit{stack}(a, c) \longrightarrow$



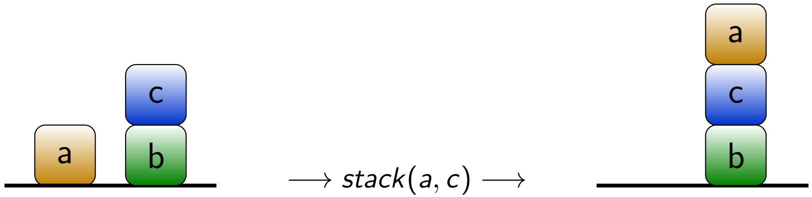
Definition stack(X,Y):

stack(X,Y):	C: ontable(X), clear(X), clear(Y)
	D: ontable(X), clear(Y)
	A: on(X,Y)

Instantiation stack(a,c):

stack(a,c):	C: ontable(a), clear(a), clear(c)
	D: ontable(a), clear(c)
	A: on(a,c)

STRIPS example 3/3



- Initial data base:
  $S_0 = \{ontable(a), clear(c), clear(a), on(c, b), ontable(b)\}$
- Operator is applicable: $\{ontable(a), clear(a), clear(c)\} \subseteq S_0$
- D list: $S_1 = S_0 \setminus D = \{clear(a), on(c, b), ontable(b)\}$
- A list: $S_2 = S_1 \cup A = \{clear(a), on(c, b), ontable(b), on(a, c)\}$

Backward-chaining algorithm for STRIPS: R-STRIPS

```
Algorithm r-strips
Input:
   Goal (=set of ground atoms) G,
   Initial data base (=set of ground atoms) Start
Output: Plan (=list of operators) P

P = []
S = Start
while G not in S{
  g = element in G that is not yet in S
  Op = a ground instance (C,D,A) of an operator such that g is in A
  P2 = r-strips(C,S)
  S = Op(P2(S))
  P = P + P2 + [Op]
}
return P
```

Note: this algorithm is a depth-first-search (does not always find an optimal plan)

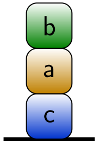
Example 1/2

New operator:

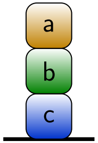
- $move(X, Y, Z)$: takes block X from block Y and puts it on block Z (corresponds to the sequence $[unstack(X, Y), stack(X, Z)]$)
- Definition:

move(X,Y,Z):	C: on(X,Y), clear(X), clear(Z)
	D: on(X,Y), clear(Z)
	A: on(X,Z), clear(Y)

Initial data base *Start* and goal *G*:



clear(b)
on(b,a)
on(a,c)
ontable(c)



on(a,b)
on(b,c)

Example 2/2

First partial goal:	on(a,b)
First operator:	move(a,c,b)

Selected operator sequence of r-strips:

[unstack(b,a), move(a,c,b), unstack(a,b), stack(b,c), stack(a,b)]

Prospects

- Conditional operators: „If *a* is on *c* then put *a* on the table“
- While loops: „As long as there is a block on top of another one, put the top block on the table“
- Non-deterministic selection: „Take some block and put it on top of *a*“
- Sensor information: „Read the entered number and stack as many blocks on top of each other“

Chapter 3.3: Situation calculus and STRIPS

Summary

Chapter 3.3: Summary

- Situation calculus
  - Situations $\neq$ states
  - Fluents: situation-dependent relations
  - Actions: preconditions and effects
- STRIPS
  - Simple planning language
  - Actions are triples (C, D, A)
  - Simple depth-first search algorithm r-strips