02285 AI and MAS, SP18
Classical automated planning

Curriculum for week 2: Sections 10.1 and 10.2 of Russell & Norvig, except Subsection 10.2.3 on Heuristics for planning.

Automated planning

Automated planning (or, simply, **planning**):

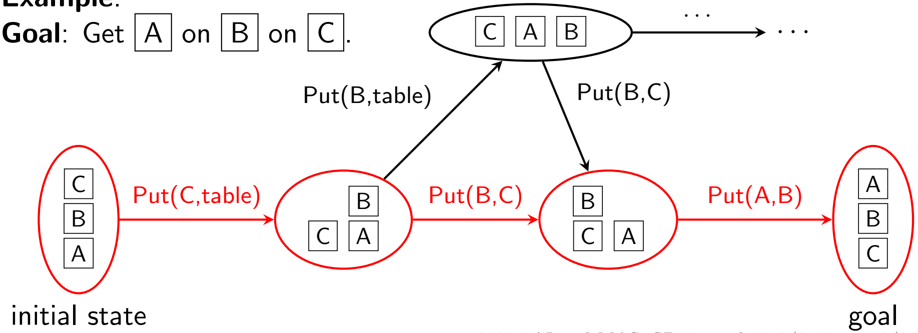
- A central subfield of AI.
- Aims at generating **plans (sequences of actions)** leading to desired outcomes.
- More precisely: Given a **goal formula**, an **initial state** and some **possible actions**, an **automated planner** outputs a plan that leads from the initial state to a state satisfying the goal formula.

Automated planning

Automated planning (or, simply, **planning**):

- A central subfield of AI.
- Aims at generating **plans** (**sequences of actions**) leading to desired outcomes.
- More precisely: Given a **goal formula**, an **initial state** and some **possible actions**, an **automated planner** outputs a plan that leads from the initial state to a state satisfying the goal formula.

Example.
Goal: Get $\boxed{A}$ on $\boxed{B}$ on $\boxed{C}$.



Planning as search

Automated planning can be seen as a **pure search problem** (cf. Chapter 3 of R&N).

Transform planning problem into search problem by building a **directed graph** (called the **state space** or **transition model**) where:

- **nodes**: possible states of the environment;
- **edges**: actions leading from one state to another.

$43 \cdot 10^{18}$ states

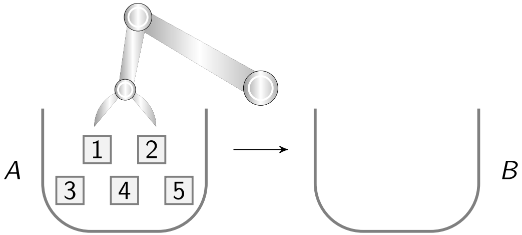
The **planning problem** then becomes the problem of finding a (shortest) path from the start state to the goal state.

Complexity: Linear in the size of the state space (using BFS).

Linear complexity sounds encouraging, but the problem is the size of the state spaces...

The Gripper problem

Gripper problem: Get n boxes from location A to location B using a gripper arm. A simplified version of one of the International Planning Competition (IPC) problems (1998–).



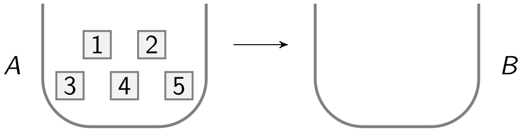
Initial state: The above (all boxes in A).

Goal state: All boxes in B.

Actions:
pick(x, y): pick up box x at location y with gripper arm;
drop(x, y): drop box x from arm at location y;
move(x, y): move gripper arm from location x to location y.

Simplified Gripper problem

Simplified Gripper problem: The arm is abstracted away.



Initial state: The above (all boxes in A).

Goal state: All boxes in B.

Actions. $MoveAB(x)$: move box number x from location A to location B; $MoveBA(x)$: move box x from B to A.

What is the *state space* of this planning problem? One node for each of the possible configurations of the boxes. E.g. represented by the set of boxes in A. Edges between nodes differing in the position of exactly one box. E.g. $\{1,2,3,4,5\} \overset{MoveAB(5)}{\rightarrow} \{1,2,3,4\}$.

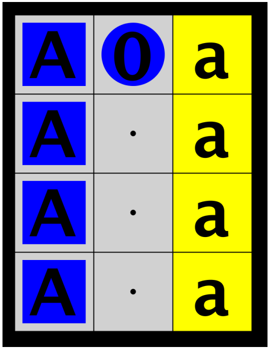
Simplified Gripper problem cont'd

The simplified Gripper problem as a **pure search problem**:

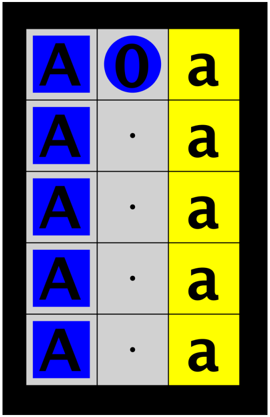
- Build **state space** (a graph).
- **Initial state**: all boxes in A.
- **Goal state**: all boxes in B.
- Find a path from the initial state to the goal state in the state space (use e.g. BFS or DFS).

What is the size of the state space, in Θ-notation, for an instance with n boxes? $\Theta(2^n)$, since each state can be represented by a subset of $\{1, 2, 3, \ldots, n\}$.

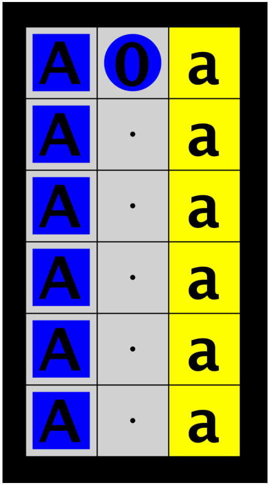
A Gripper-like problem in the setting of the programming project



gripper04

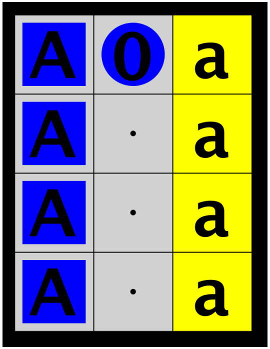


gripper05



gripper06

BFS on the Gripper-like problem



level	generated nodes
gripper04	3,755
gripper05	26,792
gripper06	184,326
gripper07	1,221,887
gripper08	7,956,581

Every time we add one box, the number of generated nodes is multiplied by ≈ 7. Why is that? It grows exponentially. In the simplified gripper problem, the size of the state space only grows with 2^n, so one additional box only leads to a state space of twice the size. However, in the programming project version, there are many more places to put each box.

BFS vs DFS on the Gripper-like problem

level	BFS generated nodes	DFS generated nodes
gripper04	3,755	3,090
gripper05	26,792	8,425
gripper06	184,326	88,374
gripper07	1,221,887	666,771
gripper08	7,956,581	2,660,195

For BFS, the number of nodes generated grows exponentially with the number of boxes. Is that true of DFS as well? Depends on order in which actions are considered. If random, it will move boxes back and forth randomly and also become exponential.

Heuristics for Gripper

Conclusion on the above: To make planning feasible even in the simplest domains, it is necessary to have **efficient heuristics**.

Designing an efficient heuristics for the (simplified) Gripper problem is not difficult.

How? Count number of boxes in A.

Which search strategy would work well using this heuristics? Greedy best-first search.

What will be the complexity of the resulting algorithm? $\Theta(n^2)$ where n is number of boxes.

Informed search on the Gripper-like problem

Two heuristics:

- $h_{gc}(s)$ is the **goal-count heuristics**, that is, $h_{gc}(s) =$ the number of misplaced boxes.
- $h_w(s)$ is a heuristics for the warmup assignment (the teacher's solution).

Number of generated nodes for the various search strategies and heuristics:

level	BFS	$A^\star/h_{gc}$	greedy/h_{gc}	$A^\star/h_w$	greedy/h_w
gripper04	3,755	2,148	92	64	63
gripper05	26,792	7,768	127	83	70
gripper06	184,326	37,924	328	106	78
gripper07	1,221,887	140,268	246	126	113
gripper08	7,956,581	477,701	289	152	118
gripper10	-	7,196,983	364	190	154
gripper20	-	-	1,169	397	324
gripper50	-	-	3,165	1,010	865

Drawback of planning as search

Planning as search:

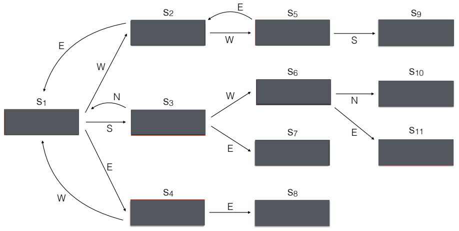
- States of the system are represented as **atoms** (nodes) with no internal structure.
- Without heuristics we are simply doing “blind search” (zero intelligence).
- Atomic graph representation only allows for problem-specific heuristics.

Designing problem-specific heuristics for each new problem is not very “intelligent” (but nevertheless the reality in most areas of computer science).

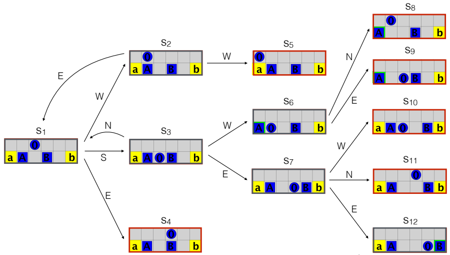
Solution (at least partial): Give **structure** to the states of the state space in a uniform and domain-independent way allowing one to automatically generate heuristics in a uniform way.

Finding the sweet spot

No structure on states: blind search, computationally *infeasible*.

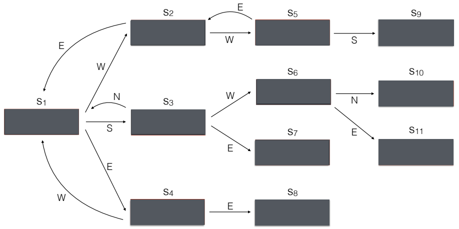


Use full state structure: heuristic search using *manual* heuristics.

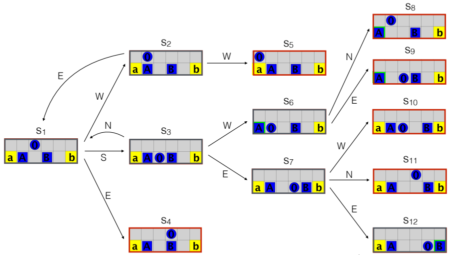


Finding the sweet spot

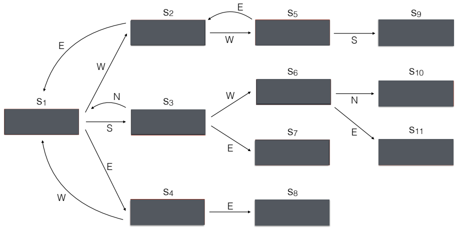
No structure on states: blind search, computationally *infeasible*.



Use full state structure: heuristic search using *manual* heuristics.



Finding the sweet spot

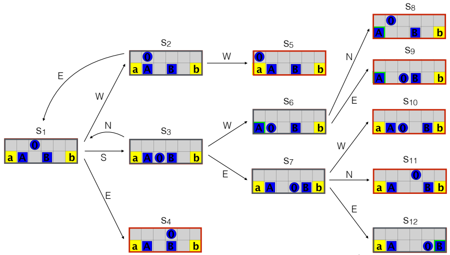


No structure on states: blind search, computationally *infeasible*.

Sweet spot: computationally *feasible* **and** *automatic* heuristics.

AUTOMATED PLANNING: Generic structure on states, generic heuristics

Use full state structure: heuristic search using *manual* heuristics.



State representation example

How to give structure to states in the Gripper example? We can e.g. use **predicate logic** (**first-order logic**) (cf. R&N Chapter 8).

Initial state:

$$In(1, A) \wedge In(2, A) \wedge \cdots \wedge In(n, A)$$

Goal state:

$$In(1, B) \wedge In(2, B) \wedge \cdots \wedge In(n, B).$$

In is a binary **predicate** (**predicate symbol**), $1, 2, \ldots, n$, A and B are **constants**.

Result of performing the action $MoveAB(2)$ in the initial state:

$$In(1, A) \wedge In(2, B) \wedge In(3, A) \wedge \cdots \wedge In(n, A).$$

Basic notions from (function-free) first-order logic

- *n*-**ary predicate**: A symbol denoting an *n*-ary relation. Takes *n* arguments. **Examples**: *In* is a 2-ary predicate. **Conventions**: We use capitalised words for predicate like *In*, *Move*, *Buy*, *Kill*.
- **Variables**: As usual. **Conventions**: We use small letters like x, y, z, a, b.
- **Constants**: A symbol denoting an object. **Conventions**: As for predicates.
- **Terms**: Constants and variables (we don't use function symbols).
- **Atom**: An *n*-ary predicate applied to *n* terms. **Examples**: $In(x, y)$, $In(x, 1)$, $In(1, 2)$.
- **Literal**: An atom (**positive literal**) or its negation (**negative literal**). **Examples**: $In(x, y)$, $\neg In(x, y)$.
- **Ground atom/literal**: All arguments are constants.

The logic has equality, that is, $=$ is among the predicates (infix notation).

Planning using logic

Advantage of structural representation of states: Allows us to define general and domain-independent heuristics in terms of these representations.

Example of domain-independent heuristics:

$$h(s) = \text{the number of goal literals that } \textit{do not} \text{ hold in } s.$$

In the Gripper problem the state

$$s = In(1, A) \wedge In(2, B) \wedge In(3, A) \wedge \cdots \wedge In(n, A)$$

has $h(s) = n - 1$.

Note: This gives exactly the goal count heuristics, counting the number of misplaced boxes. Also works with other goal configurations, e.g. all even numbered boxes in A, all odd-numbered in B.

Intermediate summary

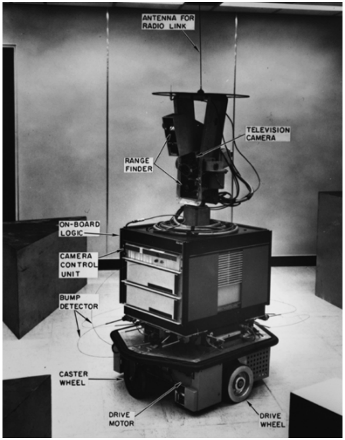
- Planning as pure search: combinatorial explosion even for the simplest problems—unless we have efficient domain-specific heuristics.
- Avoiding domain-specific heuristics for each new planning problem: put logical structure on states.

Example. Assume our goal is to get milk, bananas, and a cordless drill. We would get closer to our goal by starting out buying milk than e.g. by buying coffee or exotic underwear. This is immediately captured by planning using logical representations of states, but not without structure on the states.

Can logical state representations combined with generic heuristics avoid the problem of combinatorial explosion altogether? No, planning is PSPACE-complete.

A language for automated planning: PDDL

- **PDDL** stands for *Planning Domain Definition Language*.
- De facto standard for representing planning problems using (function-free first-order) logic.
- PDDL has a programming language-like syntax, but we'll use a more logically oriented syntax here (as does the textbook).
- PDDL is an extension of the **STRIPS** language (*STanford Research Institute Problem Solver*). STRIPS was originally employed in the late 1960's by *Shakey*, a box-pushing robot.



Shakey, the "first electronic person" according to Life Magazine, 1970.

From search to PDDL planning (+ PDDL states)

	search	**PDDL**
states	graph nodes	conjunction of atoms
actions	graph edges	action schemas
goal	graph node(s)	conjunction of literals
plan	path	sequence of actions

State (in PDDL): A conjunction of ground atoms.

Examples of states:

$$In(1, A) \wedge In(2, B) \wedge In(3, A). \tag{1}$$

$$At(Shop) \wedge Have(Milk) \wedge Have(Exotic\ Underwear). \tag{2}$$

We work under the **closed-world assumption** (**CWA**): everything not explicitly claimed to be true by the state description is assumed to be false.

Example. In the state described by (1) e.g. the following holds: $\neg In(1, B)$ and $\neg In(2, A)$.

Action schemas in PDDL

In PDDL, **actions** are represented by **action schemas** (**operators**). **Example**:

ACTION: $\mathit{MoveAB}(x)$
PRECONDITION: $\mathit{Box}(x) \wedge \mathit{In}(x, A)$
EFFECT: $\mathit{In}(x, B) \wedge \neg \mathit{In}(x, A)$

Definition. **Action schemas** (**operators**) consist of three elements:

1. **Action name**. A predicate applied to variables. **Examples**: $\mathit{MoveAB}(x)$, $\mathit{Buy}(x, y)$, $\mathit{Pick}(x, y)$.
2. **Precondition**. A conjunction of literals. **Examples**: $\mathit{Box}(x) \wedge \mathit{In}(x, A)$, $\mathit{At}(y) \wedge \mathit{Sells}(y, x)$, $\mathit{At}(y) \wedge \mathit{In}(x, y) \wedge \mathit{Free}$.
3. **Effect**. A conjunction of literals. **Examples**: $\mathit{In}(x, B) \wedge \neg \mathit{In}(x, A)$, $\mathit{Have}(x)$, $\mathit{Carry}(x) \wedge \neg \mathit{In}(x, y) \wedge \neg \mathit{Free}$.

Constraint: Preconditions and effects may only contain variables occurring in the action name.

Goals, planning domain and planning problems

Goal: A conjunction of literals. **Example**: $In(1, B) \wedge In(2, B) \wedge In(3, B)$.

Planning domain: A set $\mathcal{A}$ of action schemas.

A **planning problem** is a triple $(\mathcal{A}, s_0, g)$ where:

1. $\mathcal{A}$ is a domain (a set of action schemas).
2. s_0 is a state called the **initial state**.
3. g is a goal.

Example of a planning problem (simplified Gripper).

1. **Action schemas**:

   ACTION: $MoveAB(x)$
   PRECONDITION: $Box(x) \wedge In(x, A)$
   EFFECT: $In(x, B) \wedge \neg In(x, A)$

   ACTION: $MoveBA(x)$
   PRECONDITION: $Box(x) \wedge In(x, B)$
   EFFECT: $In(x, A) \wedge \neg In(x, B)$
2. **Initial state**: $Box(1) \wedge \cdots \wedge Box(n) \wedge In(1, A) \wedge \cdots \wedge In(n, A)$.
3. **Goal**: $In(1, B) \wedge \cdots \wedge In(n, B)$.

Ground actions

Ground action: An action obtained by instantiating the variables of an action schemas to a choice of constants.

Example. Action schema:

ACTION: $MoveAB(x)$
PRECONDITION: $Box(x) \wedge In(x, A)$
EFFECT: $In(x, B) \wedge \neg In(x, A)$

A couple of ground actions (instantiations) of the schema:

ACTION: $MoveAB(1)$
PRECONDITION: $Box(1) \wedge In(1, A)$
EFFECT: $In(1, B) \wedge \neg In(1, A)$

ACTION: $MoveAB(2)$
PRECONDITION: $Box(2) \wedge In(2, A)$
EFFECT: $In(2, B) \wedge \neg In(2, A)$

The ground actions are the executable actions in the domain.

Applicability of a ground action

Applicability: A ground action a is **applicable** in a state s if its precondition is logically entailed by s. In logical notation:

$$s \models \textsc{Precondition}(a).$$

Example. In the state $Box(1) \wedge Box(2) \wedge In(1, A) \wedge In(2, B)$ the action $MoveAB(1)$ is applicable, but not $MoveAB(2)$.

How many instantiations of $MoveAB(x)$ are applicable in the initial state $Box(1) \wedge \cdots \wedge Box(n) \wedge In(1, A) \wedge \cdots \wedge In(n, A)$? n.

Equivalent formulation of applicability of a in s: All positive literals in $\textsc{Precondition}(a)$ occur in s and none of the negative literals in $\textsc{Precondition}(a)$ occur in s.

The result of executing an action in a state

Add list of a ground action a: The set of positive literals in $\textsc{Effect}(a)$. Denoted $\textsc{Add}(a)$. Intuitively: The atoms that come to hold when a is executed.

Delete list of a ground action a: The set of negative literals in $\textsc{Effect}(a)$. Denoted $\textsc{Del}(a)$. Intuitively: The atoms that no longer hold when a has been executed.

Example. Consider the following ground action:

$\textsc{Action}$: $MoveAB(1)$
$\textsc{Precondition}$: $Box(1) \wedge In(1, A)$
$\textsc{Effect}$: $In(1, B) \wedge \neg In(1, A)$

We have $\textsc{Add}(MoveAB(1)) = \{In(1, B)\}$ and $\textsc{Del}(MoveAB(1)) = \{In(1, A)\}$.

Definition. The **result** s' of executing an applicable action a in a state s is given by (identifying states with sets of literals = **set semantics**):

$$s' = (s - \textsc{Del}(a)) \cup \textsc{Add}(a).$$

Solutions to planning problems

Solution to a planning problem $(\mathcal{A}, s_0, g)$: A plan (an action sequence) leading from the start state s_0 to a state satisfying the (existential closure of the) goal g.

More precisely, it is a sequence of ground actions $a_1, a_2, \ldots, a_n$ from $\mathcal{A}$ satisfying:

- a_1 is applicable in the initial state s_0.
- For all $i = 2, \ldots, n$: a_i is applicable in the result of executing the sequence of actions $a_1, \ldots, a_{i-1}$ in the initial state s_0.
- The state resulting from executing the sequence of actions $a_1, \ldots, a_n$ in the initial state s_0 satisfies the (existential closure of the) goal g.

Solutions can be computed e.g. with Graph-Search (BFS, DFS, $A^\star$, greedy best-first search), where the states are conjunctions of literals.

Heuristics in progression planning

- We still need an accurate **heuristics** for progression planning to be practical (consider e.g. the Gripper problem).
- The logical structure on the states allows us to define a number of completely domain-independent heuristics. We already considered one very simple such: count the number of unsatisfied goal literals (called the **goal count heuristics**).

Domain-independent heuristics for progression planning will be the main subject next week.

SELF-STUDY

Another planning problem example

Consider a world containing the following two action schemas:

ACTION: $Buy(x, y)$ (buy item x at location y)
PRECONDITION: $Buyable(x) \wedge Place(y) \wedge At(y) \wedge Sells(y, x)$
EFFECT: $Have(x)$

ACTION: $Go(x, y)$
PRECONDITION: $Place(x) \wedge Place(y) \wedge At(x)$
EFFECT: $At(y) \wedge \neg At(x)$

Assume we are given the following initial state and goal:

Init state: $Buyable(Milk) \wedge Buyable(Bananas) \wedge Buyable(Drill) \wedge Place(Home) \wedge Place(Netto) \wedge Place(ToolShop) \wedge At(Home) \wedge Sells(Netto, Milk) \wedge Sells(Netto, Bananas) \wedge Sells(ToolShop, Drill)$.

Goal: $At(Home) \wedge Have(Milk) \wedge Have(Bananas) \wedge Have(Drill)$.

A possible **solution** (plan) leading from start to goal is:

$Go(Home, Netto), Buy(Milk, Netto), Buy(Bananas, Netto),$
$Go(Netto, ToolShop), Buy(Drill, ToolShop), Go(Home)$.

Self-Study

Designing planning domains and problems

Observations:

- Planning domains and problems can be designed in many **distinct**—but equivalent—ways.
- Designing them efficiently is **non-trivial** (one of the biggest challenges when working with applications).
- **Bad domain design** can potentially **destroy** the advantage over planning as pure search: Take one unique literal per node of the state space of the problem.