# AI - NOTES

## Chapter 1: Introduction

**Artificial intelligence** (AI) refers to systems that display intelligent behavior by analyzing their environment and taking actions – with some degree of autonomy – to achieve specific goals. AI systems can both be software and hardware based systems.

The difference between a human or a system acting can be shown by the **Turing test** (1950): a human and an unknown guest interact with each other. After having this chat, the human has to say if the interlocutor is a machine or a human.

There are 4 main categories of behavior related to an artificial intelligence system:

- **acting humanly:** the system is supposed to perform actions with intelligence when performed by people
- **acting rationally:** the system should have an appropriate behavior in complex situation
- **thinking humanly:** the system should think like the machine were a human like mind
- **thinking rationally:** the system should have the mental faculties in terms of computational models

Each of these categories rely on different applications (ex.thinking humanly -> cognitive science\neuroscience, thinking rationally -> logics, acting humanly -> aeronautics, acting rationally -> logics, mathematics, …)

**Rational agents** are central entities in AI: they perceive from an environment and then decide how to act. So they can be referred to as functions f mapping a set of percept history P* and actions A: $f: P^* \rightarrow A$. For any given class of environments and tasks, we seek the agent with the best performance.

AI is applied to every (or almost every) domain, such as neuro-science, medicine, economy, finance, mobility (…) and finds its root far away in time, but starts having success in the second half of the 19th century up to now.

AI is a conglomerate of several sub-areas such as planning, search, KR, CSP & SAT, machine learning, robotics, computer vision, natural language processing, multi-agent systems, and uncertainty.

Open challenges: generalizability, causality, normativity.

Some issues with AI: ethics, privacy, impact of human intelligence, disadvantages with automation, …

## Chapter 2: Rational Agents

The ability to *act successfully* is central in artificial intelligence as well as the notion of *agents* who are supposed to act in the right way.

An **agent** is a system (as a human, robot, …) that perceives an environment through **sensors** and acts through **actuators** according to that environment.
The agent should act right in terms of **maximizing a performance**. (Ex. the performance of a vacuum cleaner can be the level of cleanliness, the $m^2$ cleaned area in an hour…).
Doing the right thing is often intractable, so the idea is to *try to do the best thing possible given the available resources*. In fact, *rationality is not equal to omniscience.*
A rational agent can be defined in terms of a function f:MxPxK -> A, where M is a performance measure, P stands for percepts, K is the knowledge and A is the output actions.
The action a is said to be **optimal** if it maximizes the performance.
The agent is **rational** if it always chooses an optimal a.
The agent can be seen as the sum of two components: a program and an architecture, so there may occur problems related to resources and computational limitations. Then, it is reasonable to approximate the best (or right) action.

There can be distinguished several classes of agents:

- **table-driven agents**: the function consists in a lookup table of actions to be taken for every possible state of the environment. So they can work only for a small number n of environment states, since the look-up table has size of $t^n$.
- **reflex-agents**: Decisions or actions are taken based on the current percept, which does not depend on the rest of the percept history. These agents react only to the predefined rules also called "condition-reaction" rules. It works best when the environment is fully observable.
- **reflex model-based agents:** they are made to deal with partial accessibility; they do this by keeping track of the part of the world it can see now. It does this by keeping an internal state that depends on what it has seen before so it holds information on the unobserved aspects of the current state.
- **goal-based agents**: they can make decisions based on previous experiences, knowledge, user input, and the desired goal. The goal-based agent distinguishes itself through its ability to find a solution according to the required output. So they don't only see the history, but also the future (the goal).
- **utility-based agents**: the actions are based not only on what the goal is, but the best way to reach that goal. In short, reaching a goal involves a cost. Utility is a measure of the possibility to reach a goal with a cost path.
- **learning-agents**: the knowledge of an agent can start from 0, so they need to learn through experience what to do.

**Different classes of environments**

1) **Fully observable**: sensors won't fail capture information since everything is accessible, this is an ideal situation
2) **Partially observable**: we need to consider some options for things we can't know, this is what happens in real applications
3) **Deterministic**: the next state is determined by the current state and the action I will perform
4) "**Uncertain**": we can distinguish 3 types of uncertainty
    - stochastic: probability is used to estimate uncertainty
    - non deterministic: actions have multiple outcomes
    - strategic: i have control on my own agent but not those of others
5) **Episodic**: the episode can be seen as perception+action, the actions depend only on an episode

6) **Sequential**: the action the agent will perform depends on all the actions it has taken
7) **Static**: the environment is not changing while the agent acts
8) **Dynamic**: the environment changes while the agent performs so it has to consider the new actions
9) **Semi Dynamic**: the environment doesn't change, but performance score of agent changes
10) **Discrete:** wrt time, environment states and agent's actions and percepts
11) **Continuous:** wrt time, environment states and agent's actions and percepts
12) **Single-Agent:** if only one agent is performing
13) **Multi-Agent:** if more than one agent is performing in a competitive or cooperative way

Depending on these types of environment AI areas are classified, as well as considering if they are domain specific or general and the architectures on which they run.

# Chapter 3: Classical Search, Blind Search

In this chapter the search problem will be discussed.
An example of a search problem is: going from a point A to a point B following some road segments by maximizing a performance based on some measurements (distance path, time…).
**Classic applications**: route planning, Puzzles, detecting bugs, …

We can distinguish between between two types of searches:
- **Systematic Search**: no limit on the number of search nodes kept in memory. Complete.
- **Local Search**: keep only one (or a few) node(s) at a time. Incomplete.

Typically these problems are harder than NP. So, for an easier problem we assume:
- discrete environment
- single-agent
- fully observability
- determinism
- static environment

To understand this argument clearly we need some notations.

A **problem** is associated with a **description ΠΤ** which specifies a **state space Θ.**

So π is the description of the problem and Θ is the state space for that description.
- **State spaces** are (annotated) graphs.
- **Solutions** are paths to goal states.
- The solution is **optimal** if the corresponding path is the cheapest.

We can also define the state space in math form with a 6-tuple Θ = (S, A, c, T, I, $S^G$), where:
- S is the set of states
- A is a finite set of actions
- c is a cost function
- T is a deterministic transition relation (s,a,s') linking a state s, an action a and a final state s' reached by s applying the action a
- I in S is the initial state
- $S^G$ is the set of goal states

Θ has **transition** (s,a,s') if (s,a,s') belongs to T.
Θ has **unit costs** if c(a)=1 for all a belonging to A.
Θ is **solvable** if there exists a solution for I.
Let's give some terminology:

- s' is **successor** of s if s → s'
- s is **predecessor** of s' if s → s'
- s' is **reachable** from s if there exists a sequence of transitions from s reaching s' with a set of actions a1,..., an and a set of states s0,...,s'.
- a1, …, an and s0, …, s' are called **paths**.
- s is **solvable** if some s' belonging to $S^G$ is reachable from s, else, s is a **dead end**.

A solution for a state s is a path from s to some s' belonging to $S^G$. If a solution exists then theta is solvable. The solution is optimal if its cost is minimal among all solutions for s.
Some examples: Vacuum cleaner, Missionaries and Cannibals, 15-Puzzle, Route Planning…

**Issues**
State spaces may be huge, so search problems are computationally hard.

There exists some types of description of the search problem:

- explicit descriptions are when Θ = π and they can't compactly describe large state spaces
- Blackbox descriptions represent the API (programming interface) which provides functionalities to construct the state space
- Declarative/Whitebox descriptions provide a general language describing the problem. They are the most efficient since they can provide the knowledge we have for that problem and they enable general problem solving

**How to search?**

1) Start from the initial state;
2) What action can I apply for that state?
3) Successful states are taken until goal is reached

**Other terminology**

**Search node n:** Contains a state reached by the search, plus information about how it was reached.
**Path cost g(n):** The cost of the path reaching n.
**Optimal cost g*** : The cost of an optimal solution path. For a state s, $g*(s)$ is the cost of a cheapest path reaching s.
**Node expansion:** Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the state s itself is also said to be expanded.
**Search strategy:** Method for deciding which node is expanded next.
**Open list:** Set of all nodes that currently are candidates for expansion. Also called frontier. **Closed list:** Set of all states that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

We can choose which structure will represent states and actions, the two most common methods are trees and graphs. This decision will also define the algorithm used to search.

We want search algorithms to be complete (find a solution) and optimal (the returned solution is optimal). But some issues are relevant to be considered: time (how long does it take to find a solution) and space complexity (how much memory the search requires).
Time and space complexity are influenced by two main factors:

- **branching factor b** (how many successors does each state have)
- **goal depth d** (at which level we find solution)

**Types of search:**

- **blind search** doesn't require any input beyond the api (rarely effective in practice)
  - breadth-first search
  - depth-first search
  - uniform-cost search
  - iterative deepening search
- **informed search** requires an additional input

**Breadth-First Search**

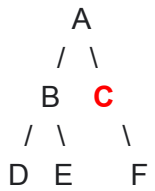
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Expands nodes in the order they were produced, so expand the shallowest unexpanded node. (FIFO)

**Properties**:

- **completeness**: Yes, In trees there are no loops or infinite nodes. Then there's always a unique path connecting each pair of nodes, so the search will explore all the nodes if necessary and find the solution
- **Optimality:** Yes for unit action costs, since it expands nodes in breadth so per level
- **Time complexity**: $O(b^d)$ if it's node generation, $O(b^{d+1})$ if it's node expansion
- **Space Complexity**: same of time

**Ex.**




Expansion: {A,B, *C*}, Visit: {A,B,**C**,*D,E,F*}

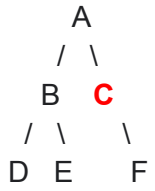
**Depth-First Search**

**function** Recursive Depth-First Search$(n,$ *problem*$)$ **returns** a solution, or failure
  **if** *problem.GoalTest*$(n.$*State*$)$ **then return** the empty action sequence
  **for each** *action* $a$ **in** *problem.Actions*$(n.$*State*$)$ **do**
    $n' \leftarrow$ *ChildNode*$($*problem*$,n,a)$
    *result* $\leftarrow$ Recursive Depth-First Search$(n'$, *problem*$)$
    **if** *result* $\neq$ *failure* **then return** $a \circ$ *result*
  **return** *failure*

Expand the most recent nodes, so expand the deepest unexpanded node. (LIFO)
**Properties:**

- **completeness**: No
- **Optimality**: No
- **Space complexity**: O(bm) if m is the maximal depth reached
- **Time Complexity**: O(b^m) worst case, if we choose the right branch O(bl)

**Ex.**




Expansion: {A,B,*C*}, Visit: {A,B,D,E,**C**,*F*}

**Uniform-Cost Search**

**function** Uniform-Cost Search(*problem*) **returns** a solution, or failure
  *node* ← a node $n$ with $n$.*State*=*problem.InitialState*
  *frontier* ← a priority queue ordered by ascending $g$, only element $n$
  *explored* ← empty set of states
  **loop do**
    **if** *Empty?(frontier)* **then return** failure
    $n$ ← *Pop(frontier)*
    **if** *problem.GoalTest*($n$.*State*) **then return** *Solution*($n$)
    *explored* ← *explored* ∪ $n$.State
    **for each** *action* $a$ **in** *problem.Actions*($n$.*State*) **do**
      $n'$ ← *ChildNode(problem,n,a)*
      **if** $n'$.State$\notin$ [*explored* ∪ States(*frontier*)] **then** *Insert*($n'$, $g(n')$, *frontier*)
      **else if** ex. $n''$ $\in$*frontier* s.t. $n''$.State= $n'$.State and $g(n') < g(n'')$ **then**
        replace $n''$ in *frontier* with $n'$

Expand nodes with lowest path cost g(n) (frontier ordered by path cost, lowest first). So, expand the least-cost unexpanded node.

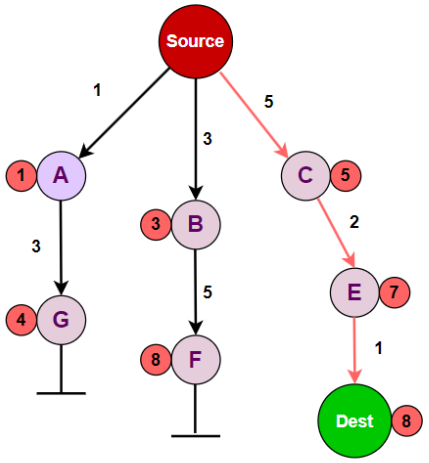
*Lemma*: Uniform-cost search is equivalent to Dijkstra's algorithm on the state space graph. (Obvious from the definition of the two algorithms)
*Theorem*: Uniform-cost search is optimal. (Because Dijkstra's algorithm is optimal).

**Properties**:

- **Completeness:** Yes
- **Optimality:** Yes
- **Time complexity:** O($b^{1+\lfloor g^*/eps \rfloor}$) where g* denotes the cost of an optimal solution and eps is the positive cost of the cheapest action
- **Space Complexity:** same of time

**Ex.**

Visited Nodes: **source, A, B, G, C, F, Dest**



**Iterative Deepening Search**

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

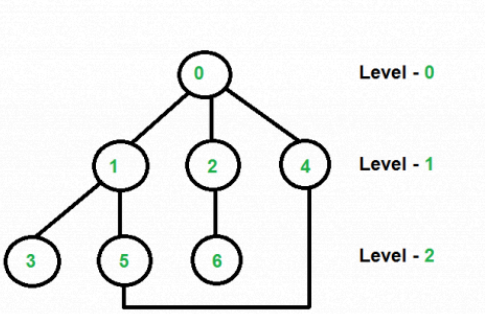
Supply depth-first search with a predetermined depth limit and find the best limit. → Goal is to combine the benefits of depth-first and breadth-first search.

**Properties:**

- **completeness**: Yes
- **Optimality:** Yes
- **Time complexity:** O($b^d$)
- **Space Complexity**: O(bd)

**Ex.**

| Depth | Iterative Deepening Depth First Search |
|---|---|
| 0 | 0 |
| 1 | 0 1 2 4 |
| 2 | 0 1 3 5 2 6 4 5 |
| 3 | 0 1 3 5 4 2 6 4 5 1 |




The explanation of the above pattern is left to the readers.

# Chapter 4: Classical Search, Informed Search

Applications: GPS, robotics, Video Games, network security…
In **Blind Search** we follow a rigid procedure to trace a path to go from an initial point to a final goal state. So we have zero knowledge of the problem it is solving because it doesn't exploit it. Instead, **Informed search** provides the "goodness" of expanding a node, given in the form of a heuristic function. So we use knowledge of the description of the problem using the heuristic function, so the information about the goal state.
We need now to give a definition of heuristic function.

**Definition of heuristic function:**
Heuristic is a Greek term meaning "I find".

Given a problem Π with states S, a heuristic is a function mapping S to real positive numbers including zero and infinity with the property that if s of S is a goal state then h(s) = 0. Formally $h: S -> R_0^+ U \{\infty\}$ such that for each goal state in S, h(s) = 0.

**Note that the value of h depends only on the state s, not on the search node (i.e., the path we took to reach s).**

Since it represents the distance between the current state s and the goal state, then it is a distance function. In principle we can choose the distance function we want (ex. Manhattan Distance, …). We want it to be:

- accurate: close to the actual distance
- fast: no overhead

These two things are in contradiction (if h=h*, h is accurate but too much overhead, if h=0, no overhead but no information too). So we need a trade-off between the two. To do this we use a **relaxed version of the problem,** i.e. an easier version.

**Perfect heuristic:**

The perfect heuristic h* is the function assigning every s ∈ S the cost of a cheapest path from s to a goal state, or ∞ if no such path exists (if no path exists we are in a dead end which corresponding h is ∞).

**"Almost perfect" heuristics:**

"Almost perfect" heuristics if |h*(n) − h(n)| ≤ c for a constant c

**Properties of h:**

Let Π be a problem with state space Θ and states S, and let h be a heuristic function for Π.

1) **Admissibility:** for all s ∈ S, we have h(s) ≤ h*(s). It represents a lower bound on goal distance, meaning that an admissible heuristic never overestimates the cost to the goal.
2) **Consistency:** for all transitions s → s' within the action a, in Θ, we have h(s) − h(s') ≤ c(a). It means that when applying an action a, the heuristic value cannot decrease by more than the cost of a.

**Proposition: Consistency → Admissibility**

Proof:

We need to show that h(s) ≤ h*(s) for all s. For states s where h*(s) = ∞, this is trivial. For all other states, we show the claim by induction over the length of the cheapest path to a goal state.

*Base case:* s is a goal state. Then h(s) = 0 by definition of heuristic functions,
so h(s) ≤ h*(s) = 0 as desired.

*Step case*: Assume the claim holds for all states s' with a cheapest goal path of length n. Say s has a cheapest goal path of length n + 1, the first transition of which is s → s' within a . By consistency, we have h(s) − h(s') ≤ c(a) and thus (a) h(s) ≤ h(s') + c(a).

By construction, s' has a cheapest goal path of length n and thus, by induction hypothesis, (b) h(s') ≤ h*(s'). By construction, (c) h*(s) = h*(s') + c(a). Inserting (b) into (a), we get
h(s) ≤ h*(s') + c(a). Inserting (c) into the latter, we get h(s) ≤ h*(s) as desired.

Admissible heuristics are ***typically*** consistent.

**Inadmissible heuristics:**

Inadmissible heuristics typically arise as approximations of admissible heuristics that are too costly to compute.

The approach on the search problem with an heuristic function is referred to as Best First Search which involves an evaluation function f(n) which provides the measure of desirability of expanding the node n. Two main algorithms used in Best-First-Search are: Greedy Best-First Search and A*.

**Greedy Best-First Search**

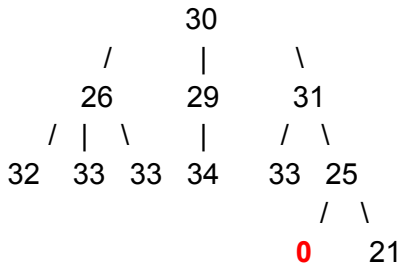
**function** Greedy Best-First Search(*problem*) **returns** a solution, or failure
  *node* ← a node $n$ with $n$.*state*=*problem.InitialState*
  *frontier* ← a priority queue ordered by ascending $h$, only element $n$
  *explored* ← empty set of states
  **loop do**
    **if** *Empty?(frontier)* **then return** failure
    $n$ ← *Pop(frontier)*
    **if** *problem.GoalTest*($n$.*State*) **then return** *Solution*($n$)
    *explored* ← *explored* ∪$n$.State
    **for each** *action* $a$ **in** *problem.Actions*($n$.*State*) **do**
      $n'$ ← *ChildNode(problem,*$n$,$a$*)*
      **if** $n'$.State∉*explored* ∪ States(*frontier*) **then** *Insert*($n'$, $h(n')$, *frontier*)

In this case the function f(n) is equal to the heuristic function h(n): f(n)=h(n). This approach actually means that we expand the node which is the closest to the goal state, according to the heuristic function. It is called greedy because it expands the node APPEARING to be the closest, in fact the heuristic gives us an estimation of this distance, not the real one.

**Properties:**

- **Completeness: Yes,** thanks to duplicate elimination and our assumption that the state space is finite.
- **Optimality: No**
- **Search space:**
  - if all costs are strictly positive it is linear in the length of the solution
  - if there are 0-cost actions, it may still be exponentially big

Ex.




Expansion = {30, 26, 29, 31, 25}
Visit = {30, 26, 32, 33, 33, 29, 34, 31, 33, 25, 0}

**A* algorithm**

```
function A*(problem) returns a solution, or failure
  node ← a node n with n.State=problem.InitialState
  frontier ← a priority queue ordered by ascending g + h, only element n
  explored ← empty set of states
  loop do
      if Empty?(frontier) then return failure
      n ← Pop(frontier)
      if problem.GoalTest(n.State) then return Solution(n)
      explored ← explored ∪ n.State
      for each action a in problem.Actions(n.State) do
        n' ← ChildNode(problem,n,a)
        if n'.State ∉ explored ∪ States(frontier) then
          Insert(n', g(n') + h(n'), frontier)
        else if ex. n'' ∈frontier s.t. n''.State= n'.State and g(n') < g(n'') then
            replace n'' in frontier with n'
```

It uses as a measure of desirability of a node f(n) both the heuristic function h(n) and a function measuring the cost accumulated so far to reach n g(n). So f(n) = h(n)+g(n).
It explores nodes by increasing g+h.

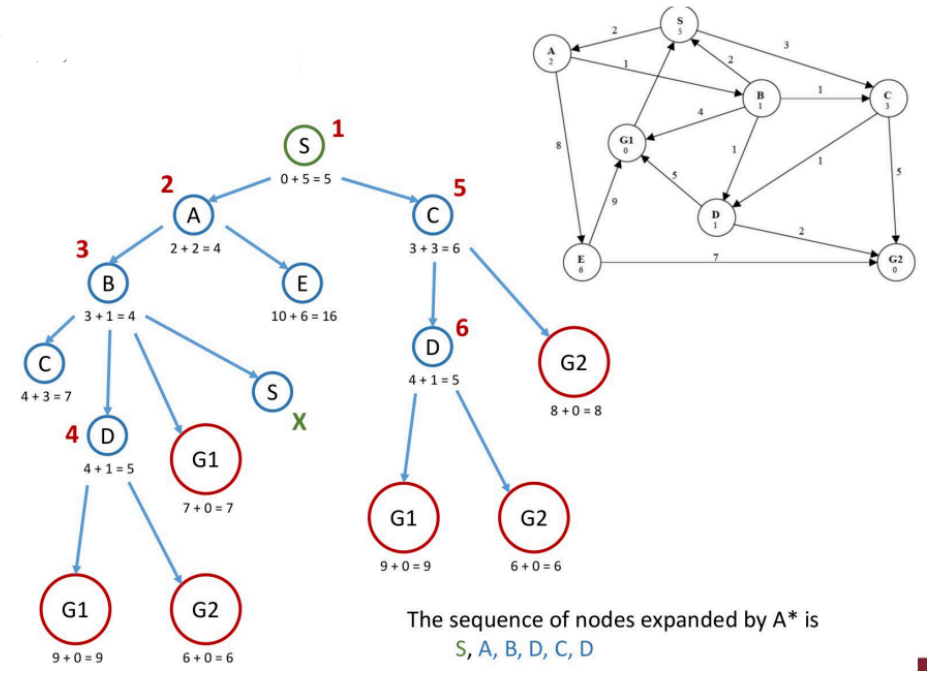
**Properties:**

- **Completeness: Yes,** if the heuristic is consistent
- **Optimality: Yes**, loops or dead ends are avoid thanks to g(n) and h(n)
- **Search Space:**
  - If all action costs are strictly positive, and we break ties (g(n) + h(n) = g(n') + h(n')) by smaller h, it is linear
  - otherwise it may still be exponentially big

**Proof: Optimality of A***

A*'s optimality is proved by contradiction. First, it is assumed that g is an optimal goal state with a path cost of f(g), that s is a suboptimal goal state with a path cost of g(s) > f(g), and that n is a node on an optimal path to g. We are assuming that A* selects s (the suboptimal goal) instead of n (the node on the optimal path) from the open list. Since h is admissible, (optimistic), f(g) >= f(n). (The actual path cost is greater than or equal to the path cost estimated by the heuristic at n.) If n is not chosen over s for expansion by A*, f(n) >= f(s). (The heuristic chooses the node with the lowest estimated F path cost.) Thus, f(g) >= f(s). Since s is a goal state, h(s) = 0. (The estimation from the current node to the final node must be 0.)
So f(s) = g(s). (f(s) = g(s) + h(s).) Thus, f(g) >= g(s). This contradicts the statement that S is suboptimal so it must be true that A* never chooses a suboptimal path. Since A* only can have as a solution a node that it has selected for expansion, it is optimal.

Ex.




# Chapter 5: Local Search

It is a kind of search much more experimental (lots of parameters are involved and their combinations) w.r.t Bind Search and informal search.
With this approach, completeness and optimality are not relevant and also path in some cases is not important. I only consider local surroundings.
When path is irrelevant and the state space is set of “complete” configurations and the task is to find the optimal configuration, then we can use iterative improvement algorithms which keep a single state, the current one, and try to improve it.
Thus, Local search is not systematic. This leads to:

- usage of very little memory-> constant space
- finding reasonable solution in large or infinite state spaces

**Hill-Climbing Algorithm (or Greedy Local Search or Gradient Descent)**

```
function Hill-Climbing(problem)
  n ← a node n with n.state=problem.InitialState
  loop do
      n' ← among child nodes n' of n with minimal h(n'),
           randomly pick one
      if h(n') ≥ h(n) then return the path to n
      n ← n'
```

Initially I'm in the initial node n then I consider all the children of n, let's call them n', with minimal h(n'). RANDOMLY, pick one. Then if h(n') >= h(n), return the path to n, else n=n' and restart.
This algorithm stops when no more immediate improvements can be made.
**Properties**:

- **Completeness:** No (search ends when no improvements can be done so I may end up in local minimum)
- **Optimality**: No (same reason)
- **Time complexity**: state space can have a huge bound (time depends on when the algorithm stop) and a single run is done
- **Memory complexity**: O(b) at any moment (basic usage of memory)

The fact that I may be stuck in a local minimum is a big issue and it can be solved by **restarting the algorithm** when reaching a local minimum i.e. when all neighbors are worse (h(n')>h(n)).
When neighbors have the same h, h(n')=h(n) we are in plateaus and actually I don't know where to go. I can solve this by doing **random walks**, which are really slow. These two strategies are not systematic, they have lots of parameters to be set (typically ML is used for this).

**Simulated Annealing**

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE − current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

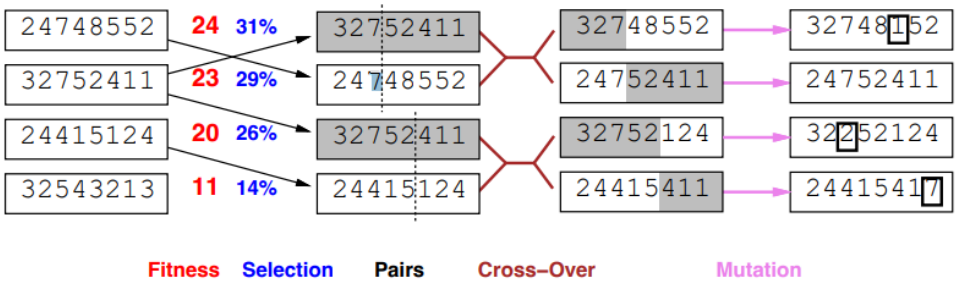
Annealing is a metallurgic technique used to harden a material with a decreasing temperature. This is the idea this algorithm uses.
Probability of accepting a "'bad" move decreases exponentially with the "badness" of the move and the temperature going down.
Again, this algorithm is not systematic: it depends on T and E.

**Local Beam Search**
The idea is to keep k states in memory instead of 1.
At each step, all successors of k states are generated:
→ If any is a goal, stop
→ If not, choose the top k of the successors and repeat.
Useful information is passed among parallel search threads.
The problem is that we can end up in the same local minima from the k nodes.
The idea to solve this issue is to use the stochastic beam search which introduces bias to the generated k nodes and select them with this term.

**Genetic Algorithms**
Recall of natural selection introduced in beam search

- Start with **population:** set of k randomly generated states
- Each state (individual) represented via a string on finite alphabet (e.g. 8-queens state represented via a string with 8 digits: positions of the queens in the columns).
- Next generation of states: Each state ranked by **fitness function** (objective function). → It should return higher values for better states (e.g. number of non-attacking pairs of queens).
- Pairs selected at random for **reproduction.** → Probability of being chosen directly proportional to fitness score.
- For each pair, **crossover** points are chosen randomly. Offspring production by crossing over the parent strings at the crossover point.
- **Random mutation** of some locations with small probability.

Easy intuition on 8-queen:




# Chapter 6: Adversarial Search

**Intro:**
The agent is facing the opponent(s) trying to reach their own goal. So they have to consider the presence of others.
AI is widely used in games since:

- they require a form of intelligence
- they capture a pure form of competition
- they are precise so that a well shaped formalization can be given

For these reasons, games are a gold sub-area of AI.

Moreover, lots of real applications can be defined or modeled as games.

The first game playing computer was introduced in 1941.

The restrictions we'll use to deal with games are:
- discreteness of states
- finite number of states
- fully observability -> no hidden information
- deterministic outcome of each move
- two players game
- no infinite runs of the game

The two players are in a game competing against each other. Each player plays in turn. So there won't be simultaneous moves.
The first player is called Max, the second player is called Min. So we can define the game as a MinMax Game. Max wants to maximize the utility, while Min wants to minimize it. So it's a zero-sum game. The utility is a value in the terminal states.

**Definitions**

The game state space is a 6-tuple $\Theta = (S, A, T, I, S^T, u)$, where:
- S is the set of states made by the disjoint union of $S^{MAX}$, $S^{MIN}$, $S^T$
- A is the set of actions made by the disjoint union of $A^{MAX}$, $A^{MIN}$
- T is the deterministic transition relation
- I is the initial state
- $S^T$ is the set of terminal states
- u is the utility function mapping terminal states to a real number: $u: S^T -> R$

if $a \in A^{MAX}$ $and$ $s -> s'$ $through$ $a$ $then$: $s \in S^{MAX}$ $and$ $s' \in S^{MIN}$
if $a \in A^{MIN}$ $and$ $s -> s'$ $through$ $a$ $then$: $s \in S^{MIN}$ $and$ $s' \in S^{MAX}$

**How to describe the game state space?**
- blackbox/API
- declarative description
- explicit

**Game terminology**
- States: positions
- Actions: moves
- End states: terminal states

**Why are games hard to solve?**
The number of possible solutions (and also the number of states which influence it) is potentially too high and computing all of them is intractable.
A solution for a game is called **strategy.**

It is a function in the form: $\sigma^x: S^x \ -> A^x$ , x {Min, Max}.

We say that a is applicable to s if $\sigma^x(s) = a$.
Since I don't know all the possible reactions of the opponent I have to be ready for all of them.
A strategy is **optimal** if it yields the best possible utility for X assuming perfect opponent play.

There are 3 kind of solutions:

1) ultra weak: we can predict if the X will win or lose by the starting positions
2) weak: strategy is optimal from the beginning of the game (so from that particular configuration)
3) strong: strategy is optimal for every valid state

Since the computation of all possible strategies is infeasible, we compute the next move given the current state.

**Minimax Search**: The name minimax gives the idea of alternation between Min and Max in playing. We suppose our player is Max, so we want to compute an optimal move for Max.

Input: State $s \in S^{Max}$, in which Max is to move.

```
function Minimax-Decision(s) returns an action
  v ← Max-Value(s)
  return an action a ∈ Actions(s) yielding value v

function Max-Value(s) returns a utility value
  if Terminal-Test(s) then return u(s)
  v ← −∞
  for each a ∈ Actions(s) do
     v ← max(v, Min-Value(ChildState(s, a)))
  return v

function Min-Value(s) returns a utility value
  if Terminal-Test(s) then return u(s)
  v ← +∞
  for each a ∈ Actions(s) do
     v ← min(v, Max-Value(ChildState(s, a)))
  return v
```

**Properties:**

- **completeness**: yes (if tree is finite)
- **optimality**: yes (against optimal component)
- **Time complexity**: O($b^m$), where b is the branching factor and m is the depth solution
- **Space complexity**: O(bm)

**Ex.**
(initialization)

MAX max -> -INF

MIN min min min

T 3 12 8 2 4 6 14 5 2

(depth first on left subtree)

MAX -INF

/

MIN +INF (3) +INF min

/ | \

T 3 12 8 2 4 6 14 5 2

(depth on second subtree)

MAX -INF (3)

/ |

MIN +INF (3) +INF (2) min

/ | \ / | \

T 3 12 8 2 4 6 14 5 2

(depth on last subtree)

MAX -INF (3)

/ | \

MIN +INF (3) +INF (2) +INF (14) (5) (2)

/ | \ / | \ / | \

T 3 12 8 2 4 6 14 5 2

So, we build a tree with:

- the root containing the max initialized with -inf
- then we have a min level initialized with +inf
- the terminal level containing the utilities

We start visiting the tree as if we were doing a depth first search and we update in a bottom-up approach the values of inner nodes n with its utility u(n) according to this rule:

- If it's Max's turn: Set u(n) to the maximum of the utilities of n's successor nodes.
- If it's Min's turn: Set u(n) to the minimum of the utilities of n's successor nodes.

**Pro:**

- easy and reasonable
- optimal action

**Contra:**

- infeasible (the tree is too large)

**How to fix these problems?**

The main issue is that the tree is too big. The solution might be imposing a depth limit called "horizon" and apply an evaluation function f to states even if they're not terminal.

**Evaluation function f**

f(s) = estimate of actual value of s.
If the state s is terminal then f(s) = u(s).

We want f to be accurate and fast, but these two characteristics are contradictory, so we need a trade off between them. Generally f is in the form f = w1f1+w2f2+...+wnfn, where wi are weights that can be learned automatically and fi are features described by the experts.

**Problems with horizon**

The introduction of the depth limit is itself an issue: how much should it be worth? how to choose its value? In general, it is very difficult to predict. So a solution may be an iterative deepening search with depth limit d=1,2,...
A better solution is the "Quiescence search" where we try to choose dynamically d, in order to give a "preference" to go in depth when we dive into unquiet positions.

**Alpha-Beta Search**

The idea is reducing search by cutting off some nodes (pruning).

**function** Alpha-Beta-Search$(s)$ **returns** an action
  $v \leftarrow$ *Max-Value*$(s, -\infty, +\infty)$
  **return** an *action* $a \in$ *Actions(s)* yielding value $v$

**function** Max-Value$(s, \alpha, \beta)$ **returns** a utility value
  **if** *Terminal-Test*$(s)$ **then return** $u(s)$
  $v \leftarrow -\infty$
  **for each** $a \in$ *Actions(s)* **do**
    $v \leftarrow \max(v,$*Min-Value*$($*ChildState*$(s, a), \alpha, \beta))$
    $\alpha \leftarrow \max(\alpha, v)$
    **if** $v \geq \beta$ **then return** $v$ /* Here: $v \geq \beta \Leftrightarrow \alpha \geq \beta$ */
  **return** $v$

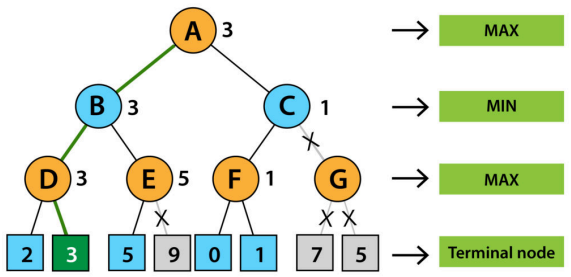
**function** Min-Value$(s, \alpha, \beta)$ **returns** a utility value
  **if** *Terminal-Test*$(s)$ **then return** $u(s)$
  $v \leftarrow +\infty$
  **for each** $a \in$ *Actions(s)* **do**
    $v \leftarrow \min(v,$*Max-Value*$($*ChildState*$(s, a), \alpha, \beta))$
    $\beta \leftarrow \min(\beta, v)$
    **if** $v \leq \alpha$ **then return** $v$ /* Here: $v \leq \alpha \Leftrightarrow \alpha \geq \beta$ */
  **return** $v$

Characteristics:

- **Alpha**: it is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is – ∞.
- **Beta**: it is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is + ∞.
- The condition for Alpha-beta Pruning is that α >= β.
- Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and beta can be updated only when it's MIN's chance.
- MAX will update only alpha values and MIN will update only beta values.
- The node values will be passed to upper nodes instead of values of alpha and beta are only passed to child nodes

**Properties** Best case of pruned nodes: $b^{m/2}$, Fast hardware: fast search, fast evaluating function, human expertise

**Ex.**



**Monte Carlo Tree Search - MCTS**

It is a more robust technique since it wants to overcome the issues of Alpha Beta.

- *problem of Alpha Beta:* accurate and fast evaluation function. *Solution*: MCTS evaluates positions by playing random games, so f(s) = average of utilities of these **simulations** or **playouts**. During simulations the moves to make are chosen according to the **playout policy** which uses self play and neural networks. To know where to start the simulation we prefer using a **selection policy** which focuses the computation on important paths of the tree. (Pure Monte-Carlo starts simulating at the current node, not efficient). The selection finds a balanced measure between exploration and exploitation with the Upper Confidence Bound, a formula that automatically balances exploration and exploitation to maximize total gains.
- *problem of Alpha Beta:* not so much exploration when there's a high branching factor. S*olution:* spend more time evaluating promising moves

**function** MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  *tree* ← NODE(*state*)
  **while** IS-TIME-REMAINING() **do**
    *leaf* ← SELECT(*tree*)
    *child* ← EXPAND(*leaf*)
    *result* ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  **return** the move in ACTIONS(*state*) whose node has highest number of playouts

The MCTS algorithm works in 4 steps:

1) **Selection:** traverse the tree starting at the root, applying the selection policy to choose successors until reaching a leaf node that has not been fully expanded.
2) **Expansion:** grow tree by generating a new child node according to playout policy. After this, we get a reward
3) **Simulation:** From the newly generated child node, perform a run of the game, selecting moves for both players according to the playout policy, to obtain the final reward.
4) **Backpropagation:** Use the result of the simulation to update all the search tree nodes going up to the root

When time to decide is over, choose the "best" move and play it.

**Properties:**

- chosen when branching factor becomes very high and the evaluation function is difficult
- is more robust than Alpha-Beta search
- can be applied to unknown games
- is a form of reinforcement learning

**Stochasticity:** stochastic games involve not only skills but also a component of luck. So the tree in those games adds nodes measuring the "chance".

# Chapter 7: Constraint Satisfaction Problem, I

A **constraint** is a condition that solutions have to satisfy.
A **soft constraint** is a condition that solutions may have (or not) to satisfy.

The constraint satisfaction **problem formulation** is:
Given a set of variables and a set of constraints, find a solution i.e. an assignment of variables to values satisfying the constraints.

**Applications:** Time scheduling, traveling tournament problem, radio frequency assignment,...

**Constraint network:** generic language to describe this kind of problem
A constraint network is defined as a triple consisting of (V, D, C), where:

- V is the finite set of variables
- D is the finite set of domains associated with the variables
- C is the (binary) set of constraints, so binary relations $C_{\{u,v\}} = C_{\{v,u\}} = C_{u,v}$, with $u, v \in V \; u != v$ and $C_{\{u,v\}} \subseteq D_u x D_v$

$C_{\{u,v\}}$ can be seen as the permissible assignment to u and v.
We can have more than one constraint over the same variables and the assignments of the variables will lie in the intersection of all the constraints involving these variables.

CSP solvers are generic algorithms solving such problems described by the constraint network.

There are also a lot of variants of constraint network:

- infinite domain
- unary constraint
- relations with over 2 variables

**Assignment**
Let γ = (V, D, C) be a constraint network.
A partial assignment is a function $a: V' \rightarrow \cup_{v \in V} D$ where V' ⊆ V and a(v) ∈ Dv for all v ∈ V'.
If V' = V , then a is a *total assignment*, or assignment in short.

A partial assignment assigns some variables to values from their respective domains. A total assignment is defined on all variables

**Consistency**

Let γ = (V, D, C) be a constraint network, and let a be a partial assignment.
We say that a is *inconsistent* if there exist variables u, v ∈ V on which a is defined, with $C_{uv}$ ∈ C and (a(u), a(v)) !∈ $C_{uv}$.
In that case, a violates the constraint $C_{uv}$. We say that a is consistent if it is not inconsistent.

**Solutions**
Let γ = (V, D, C) be a constraint network. If a is a total consistent assignment, then a is a solution for γ. If a solution to γ exists, then γ is solvable; otherwise, γ is inconsistent.

**Extensibility**
Let γ = (V, D, C) be a constraint network, and let a be a partial assignment. We say that a can be extended to a solution if there exists a solution a' that agrees with a on the variables where a is defined.

**Computational Complexity of CSP**

**Input size vs. solution space size:** Assume constraint network γ with n variables, all with domain size k.
The number of total assignments is $k^n$.
The size of the description of γ is nk for variables and domains. At most $n^2$constraints, each of size at most $k^2$ $-> \ O(n^2k^2)$
So, the number of assignments is exponentially bigger than the size of γ.

**Theorem (CSP is NP-complete)**:
It is NP-complete to decide whether or not a given constraint network γ is solvable.
**Proof.**
Membership in NP: Just guess a total assignment a and verify (in polynomial time) whether a is a solution.
**NP-Hardness**: The special case of graph coloring (our illustrative example) is known to be NP-hard.

Let's define as:
- **search:** depth-first enumeration of partial assignments
- **backtracking:** backtrack at inconsistent partial assignments
- **Inference:** Deducing tighter equivalent constraints to reduce search space (backtracking will occur earlier on).

**Naive Backtracking**

```
function NaïveBacktracking(a) returns a solution, or "inconsistent"
  if a is inconsistent then return "inconsistent"
  if a is a total assignment then return a
  select some variable v for which a is not defined
  for each d ∈ D_v in some order do
     a' := a ∪ {v = d}
     a'' := NaïveBacktracking(a')
     if a'' ≠ "inconsistent" then return a''
  return "inconsistent"
```

**Properties:**
- simple
- much more efficient than enumerating total assignments

- complete
- Backtracking does not recognize that a cannot be extended to a solution, unless a is already inconsistent

The order in which we consider variables and their values may have a huge impact on search space size! Naiıve backtracking does not specify in which order the variables are considered. Naiıve backtracking does not specify in which order the values of the chosen variable are considered.

- If no solution exists below current node: Doesn't matter, we will have to search the whole sub-tree anyway
- If a solution does exist below the current node: Does matter. If we always choose a "correct" value (from a solution) then no backtracking is needed.

Strategies to solve the problem of *variable ordering* can be:

1) **choosing most constrained variables first:** Always pick a variable v with minimal |{d ∈ Dv | a ∪ {v = d} is consistent}|.
   By choosing a most constrained variable v first, we reduce the branching factor (number of sub-trees generated for v) and thus reduce the size of our search tree.
   Extreme case: If |{d ∈ $D_v$ | a ∪ {v = d} is consistent}| = 1, then the value assignment to v is forced by our previous choices.
2) **choosing most constraining variable first:** Always pick v with maximal |{u ∈ V | a(u) is undefined, $C_{uv}$ ∈ C}|.
   By choosing a most constraining variable first, we detect inconsistencies earlier on and thus reduce the size of our search tree.

Strategies to solve the problem of *value ordering* can be:

1) **choosing least constraining value first**: For variable v, always pick d ∈ $D_v$ with minimal |{d' | d' ∈ $D_v$, a(u) is undefined, $C_{uv}$ ∈ C, (d' , d) !∈ $C_{uv}$}|.
   By choosing a least constraining value first, we increase the chances to not rule out the solutions below the current node.

# Chapter 8: Constraint Satisfaction Problem, II

Adding constraints without losing solutions = obtaining an equivalent network with a "tighter description" and hence with a smaller number of consistent partial assignments.

Decomposition methods exploit the structure of the constraint network. They identify separate parts (sub-networks) whose inter-dependencies are "simple" and can be handled efficiently.

**Inference:** Deducing additional constraints (unary or binary), that follow from the already known constraints, i.e., that are satisfied in all solutions.
Inference tightens γ without losing equivalence, during backtracking. This reduces the amount of search needed; that benefit must be traded off against the runtime overhead for making the inferences.

**Equivalent Constraint Networks:** Let γ = (V, D, C) and γ' = (V, D' , C') be constraint networks sharing the same set of variables. We say that γ and γ' are equivalent, written γ' ≡ γ, if every solution of γ is a solution of γ', and every solution of γ' is a solution of γ.

**Tightness**

Let γ = (V, D, C) and γ' = (V, D', C') be constraint networks sharing the same set of variables. We say that γ' is tighter than γ, if:

- for all v in V: $D_v' \subseteq D_v$
- for all u != v in V: either $C_{uv} \notin C \text{ or } C'_{uv} \subseteq C_{uv}$

γ' is strictly tighter than γ if at least one of these inclusions is strict.
In other words: "*γ' has the same constraints as γ, plus some others.*"

**Theorem**. Let γ and γ' be constraint networks s.t. γ' ≡ γ and γ' is tighter than γ. Then, γ' has the same solutions as γ but fewer consistent partial assignments than γ.
γ' is a better encoding of the underlying problem.

**Inference as a pre-process**:
Just once before the search starts. Little runtime overhead, little pruning power.

**Inference during search:**
At every recursive call of backtracking. Strong pruning power, may have large runtime overhead.

The more complex the inference, the smaller the number of search nodes, but the larger the runtime needed at each node.

**Backtracking With Inference**

**function** BacktrackingWithInference($\gamma, a$) **returns** a solution, or "inconsistent"
  **if** $a$ is inconsistent **then return** "inconsistent"
  **if** $a$ is a total assignment **then return** $a$
  $\gamma' :=$ a copy of $\gamma$  /* $\gamma' = (V, D', C')$ */
  $\gamma' :=$ Inference($\gamma'$)
  **if** exists $v$ with $D'_v = \emptyset$ **then return** "inconsistent"
  select some variable $v$ for which $a$ is not defined
  **for each** $d \in$ copy of $D'_v$ in some order **do**
    $a' := a \cup \{v = d\}$; $D'_v := \{d\}$  /* makes $a$ explicit as a constraint */
    $a'' :=$ BacktrackingWithInference($\gamma', a'$)
    **if** $a'' \neq$ "inconsistent" **then return** $a''$
  **return** "inconsistent"

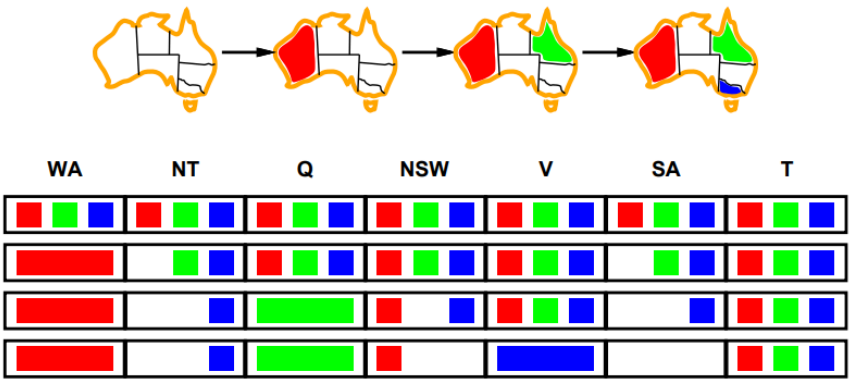
Inference(): Any procedure delivering a (tighter) equivalent network.
Inference typically prunes domains; indicate unsolvability by D'v = ∅.

When backtracking out of a search branch, retract the inferred constraints: these were dependent on a, the search commitments so far.

**Forward checking, version 1**

**function** ForwardChecking$(\gamma, a)$ **returns** modified $\gamma$
  **for each** $v$ where $a(v) = d'$ is defined **do**
    **for each** $u$ where $a(u)$ is undefined and $C_{uv} \in C$ **do**
      $D_u := \{d \mid d \in D_u, (d, d') \in C_{uv}\}$
  **return** $\gamma$

**Example of fwd:**




**Properties:**

- Forward checking is sound: Its tightening of constraints does not rule out any solutions. In other words: it guarantees to deliver an equivalent network. → Recall here that the partial assignment a is represented as unary constraints inside γ.
- Incremental computation: Instead of the first for-loop, use only the 2nd one every time a new assignment a(v) = d' is added.
- Cheap but useful inference method.
- Rarely a good idea to not use forward checking (or a stronger inference method subsuming it).
- removes values conflicting with an assignment already

**Problem**
Forward checking propagates information only "from assigned to unassigned" variables. No propagation between unassigned variables.

So we need a new definition: **Arc consistency.**
Let γ = (V, D, C) be a constraint network.
A variable u ∈ V is arc consistent relative to another variable v ∈ V if either $C_{uv}$ !∈ C, or for every value d ∈ $D_u$ there exists a value d' ∈ $D_v$ such that (d, d') ∈ $C_{uv}$.

The network γ is arc consistent if every variable u ∈ V is arc consistent relative to every other variable v ∈ V.
In other words: *Arc consistency = for every domain value and constraint, at least one value on the other side of the constraint "works".*
Arc consistency removes values that do not comply with any value still available at the other end of a constraint. This subsumes forward checking.

**Forward checking, version II**

**function** Revise$(\gamma, u, v)$ **returns** modified $\gamma$
  **for each** $d \in D_u$ **do**
    **if** there is no $d' \in D_v$ with $(d, d') \in C_{uv}$ **then** $D_u := D_u \setminus \{d\}$
  **return** $\gamma$

**Properties**

- Runtime, if k is maximal domain size: O($k^2$), based on the implementation where the test "(d, d') ∈ Cuv?" is constant time.

**AC-1**

**function** AC-1$(\gamma)$ **returns** modified $\gamma$
  **repeat**
    *changesMade* := *False*
    **for each** constraint $C_{uv}$ **do**
      Revise$(\gamma, u, v)$  /* if $D_u$ reduces, set *changesMade* := *True* */
      Revise$(\gamma, v, u)$  /* if $D_v$ reduces, set *changesMade* := *True* */
  **until** *changesMade* = *False*
  **return** $\gamma$

**Properties**

- Runtime, if n variables, m constraints, k maximal domain size: O($mk^2 * nk$): $mk^2$ for each inner loop, fixed point reached at the latest once all nk variable values have been removed.
- Redundant computations: u and v are revised even if their domains haven't changed since the last time.

**AC-3**

**function** AC-3$(\gamma)$ **returns** modified $\gamma$
  $M := \emptyset$
  **for each** constraint $C_{\{uv\}} \in C$ **do**
    $M := M \cup \{(u, v), (v, u)\}$
  **while** $M \neq \emptyset$ **do**
    remove any element $(u, v)$ from $M$
    Revise$(\gamma, u, v)$
    **if** $D_u$ has changed in the call to Revise **then**
      **for each** constraint $C_{\{w,u\}} \in C$ where $w \neq v$ **do**
        $M := M \cup \{(w, u)\}$
  **return** $\gamma$

**Properties**

- AC-3(γ) enforces arc consistency because at any time during the while-loop, if (u, v) !∈ M then u is arc consistent relative to v.
- Why only “where w != v”? v is the reason why Du just changed. Thus, if v was arc consistent relative to u before, that continues to hold: the values just removed from Du did not match any values from Dv anyway.

**Theorem (Runtime of AC-3).** Let γ = (V, D, C) be a constraint network with m constraints, and maximal domain size k. Then AC-3(γ) runs in time O($mk^3$).

**Proof.**

Each call to Revise(γ, u, v) takes time O($k^2$) so it suffices to prove that at most O(mk) of these calls are made. The number of calls to Revise(γ, u, v) is the number of iterations of the while-loop, which is at most the number of insertions into M. Consider any constraint Cuv. Two variable pairs corresponding to Cuv are inserted in the for-loop. In the while loop, if a pair corresponding to Cuv is inserted into M, then beforehand the domain of either u or v was reduced, which happens at most 2k times. Thus we have O(k) insertions per constraint, and O(mk) insertions overall, as desired.

**Constraint Graphs**

Let γ = (V, D, C) be a constraint network. The constraint graph of γ is the undirected graph whose vertices are the variables V and that has an arc {u, v} if and only if Cuv ∈ C.
The constraint graph captures the dependencies between variables. Separate connected components can be solved independently.

**Theorem (Disconnected Constraint Graphs).**
Let γ = (V, D, C) be a constraint network. Let ai be a solution to each connected component Vi of the network’s constraint graph. Then $a := \bigcup_i a_i$ i ai is a solution to γ.

**Proof.**
a satisfies all Cuv where u and v are inside the same connected component. The latter is the case for all Cuv.

*If two parts of γ are not connected, then they are independent.*

**Theorem (Acyclic Constraint Graphs)**.
Let γ = (V, D, C) be a constraint network with n variables and maximal domain size k, whose constraint graph is acyclic. Then we can find a solution for γ, or prove γ to be inconsistent, in time O($nk^2$).
*Constraint networks with acyclic constraint graphs can be solved in (low-order) polynomial time.*

### AcyclicCG

**Algorithm:** AcyclicCG($\gamma$)

1. Obtain a directed tree from $\gamma$'s constraint graph, picking an arbitrary variable $v$ as the root, and directing arcs outwards.[1]
2. Order the variables topologically, i.e., such that each vertex is ordered before its children; denote that order by $v_1, \ldots, v_n$.
3. **for** $i := n, n-1, \ldots, 2$ **do**:
   1. Revise($\gamma, v_{parent(i)}, v_i$).
   2. **if** $D_{v_{parent(i)}} = \emptyset$ **then return** "inconsistent"

   → Now, every variable is arc consistent relative to its children.
4. Run BacktrackingWithInference with forward checking, using the variable order $v_1, \ldots, v_n$.

   → This algorithm will find a solution without ever having to backtrack!

### Cutset

Let γ = (V, D, C) be a constraint network, and V' ⊆ V . V' is a cutset for γ if the sub-graph of γ's constraint graph induced by V \ V' is acyclic. V' is optimal if its size is minimal among all cutsets for γ.

A cutset is a subset of variables removing which renders the constraint graph acyclic. Cutset decomposition backtracks only on such a cuset, and solves a sub-problem with acyclic constraint graph at each search leaf.

### Cutset Conditioning

$V_0 :=$ a cutset; **return** CutsetConditioning($\gamma, V_0, \emptyset$)

**function** CutsetConditioning($\gamma, V_0, a$) **returns** a solution, or "inconsistent"
  $\gamma' :=$ a copy of $\gamma$; $\gamma' :=$ ForwardChecking($\gamma', a$)
  **if** exists $v$ with $D'_v = \emptyset$ **then return** "inconsistent"
  **if** exists $v \in V_0$ s.t. $a(v)$ is undefined **then** select such $v$
    **else** $a' :=$ AcyclicCG($\gamma'$); **if** $a' \neq$ "inconsistent" **then return** $a \cup a'$
      **else return** "inconsistent"
  **for each** $d \in$ copy of $D'_v$ in some order **do**
    $a' := a \cup \{v = d\}$; $D'_v := \{d\}$;
    $a'' :=$ CutsetConditioning($\gamma', V_0, a'$)
    **if** $a'' \neq$ "inconsistent" **then return** $a''$
  **return** "inconsistent"

### Properties

- Forward Checking required so that a ∪ a' is consistent in γ.
- Runtime is exponential only in |V0|, not in |V |
- Finding optimal cutsets is NP-hard, but practical approximations exist.

# Chapter 9: Propositional Reasoning, I

We want agents to think rationally and do it before acting.
”Thinking” = Reasoning about knowledge represented using logic.

**function** KB-AGENT(*percept*) **returns** an *action*
  **persistent**: *KB*, a knowledge base
    *t*, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  *action* ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  *t* ← *t* + 1
  **return** *action*

**Logic concepts**

**Syntax:** What are legal statements (formulas) ϕ in the logic? In propositional logic: Atomic propositions that can be either true or false, connected by connectives, “and, or, not”.
**Semantics**: Which formulas ϕ are true under which interpretation I, written I |= ϕ? In propositional logic: Assign value to every proposition, evaluate connectives.
Syntax and semantics are needed to represent knowledge.

**Entailment**: Which ψ follow from (are entailed by) ϕ, written ϕ |= ψ, meaning that, for all I s.t. I |= ϕ, we have I |= ψ?
**Deduction:** Which statements ψ can be derived from ϕ using a set R of inference rules (a calculus), written $\phi \mid=_R \psi$?
Entailment and deduction are used to reason about knowledge.

**Calculus soundness:** whenever $\phi \mid=_R \psi$, we also have ϕ |= ψ.
**Calculus completeness**: whenever ϕ |= ψ, we also have $\phi \mid=_R \psi$.

**Propositional logic** = canonical form of knowledge + reasoning.
**Applications:** Product configuration, Hardware verification, Software verification, CSP applications…

With propositional logic we want to formalize logics in order to understand the syntax and semantics and capture deductions.

Atoms Σ in propositional logic are Boolean variables.

**Definition (Syntax).** *Let $\Sigma$ be a set of atomic propositions. Then:*

1. $\bot$ *and* $\top$ *are* $\Sigma$*-formulas.* ("False", "True")
2. *Each* $P \in \Sigma$ *is a* $\Sigma$*-formula.* ("Atom")
3. *If* $\varphi$ *is a* $\Sigma$*-formula, then so is* $\neg\varphi$. ("Negation")

*If* $\varphi$ *and* $\psi$ *are* $\Sigma$*-formulas, then so are:*

4. $\varphi \wedge \psi$ ("Conjunction")
5. $\varphi \vee \psi$ ("Disjunction")
6. $\varphi \to \psi$ ("Implication")
7. $\varphi \leftrightarrow \psi$ ("Equivalence")

**Definition (Semantics).** *Let $\Sigma$ be a set of atomic propositions. An interpretation of $\Sigma$, also called a truth assignment, is a function $I : \Sigma \mapsto \{1, 0\}$. We set:*

$$
\begin{array}{lll}
I \models \top & & \\
I \not\models \bot & & \\
I \models P & \text{iff} & P^I = 1 \\
I \models \neg\varphi & \text{iff} & I \not\models \varphi \\
I \models \varphi \wedge \psi & \text{iff} & I \models \varphi \text{ and } I \models \psi \\
I \models \varphi \vee \psi & \text{iff} & I \models \varphi \text{ or } I \models \psi \\
I \models \varphi \to \psi & \text{iff} & \text{if } I \models \varphi, \text{ then } I \models \psi \\
I \models \varphi \leftrightarrow \psi & \text{iff} & I \models \varphi \text{ if and only if } I \models \psi
\end{array}
$$

*If $I \models \varphi$, we say that $I$ satisfies $\varphi$, or that $I$ is a model of $\varphi$. The set of all models of $\varphi$ is denoted by $M(\varphi)$.*

**Knowledge base:** A Knowledge Base (KB) is a set of formulas. An interpretation is a model of KB if I |= ϕ for all ϕ ∈ KB.

A formula ϕ is:

- **satisfiable** if there exists I that satisfies ϕ.
- **unsatisfiable** if ϕ is not satisfiable.
- **falsifiable** if there exists I that doesn't satisfy ϕ.
- **valid** if I |= ϕ holds for all I. We also call ϕ a tautology.

**Equivalency**
Formulas ϕ and ψ are equivalent, ϕ ≡ ψ, if M(ϕ) = M(ψ).

**Entailment.**
Let Σ be a set of atomic propositions. We say that a set of formulas KB entails a formula ϕ, written KB |= ϕ, if ϕ is true in all models of KB, i.e., M( $\wedge_{\psi \in KB}$ ) ⊆ M(ϕ). In this case, we also say that ϕ follows from KB.

**Contradiction Theorem**. KB |= ϕ if and only if KB ∪ {¬ϕ} is unsatisfiable.

**Proof.** "⇒": Say KB $\models \varphi$. Then for any $I$ where $I \models$ KB we have $I \models \varphi$ and thus $I \not\models \neg\varphi$. "⇐": Say KB $\cup \{\neg\varphi\}$ is unsatisfiable. Then for any $I$ where $I \models$ KB we have $I \not\models \neg\varphi$ and thus $I \models \varphi$.

In general we want to Determine whether ϕ is satisfiable, valid, etc. The method is building the truth table, enumerating all interpretations of Σ.

**The two quintessential normal forms:** (there are others as well)

- A formula is in conjunctive normal form (CNF) if it consists of a conjunction of disjunctions of literals:

$$\bigwedge_{i=1}^{n} \left( \bigvee_{j=1}^{m_i} l_{i,j} \right)$$

- A formula is in disjunctive normal form (DNF) if it consists of a disjunction of conjunctions of literals:

$$\bigvee_{i=1}^{n} \left( \bigwedge_{j=1}^{m_i} l_{i,j} \right)$$

## CNF Transformation (DNF Transformation: Analogously)

**Exploit the equivalences:**

1. $(\varphi \leftrightarrow \psi) \equiv [(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)]$ (Eliminate "$\leftrightarrow$")
2. $(\varphi \rightarrow \psi) \equiv (\neg\varphi \vee \psi)$ (Eliminate "$\rightarrow$")
3. $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ and $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$ (Move "$\neg$" inwards)
4. $[(\varphi_1 \wedge \varphi_2) \vee (\psi_1 \wedge \psi_2)] \equiv [(\varphi_1 \vee \psi_1) \wedge (\varphi_2 \vee \psi_1) \wedge (\varphi_1 \vee \psi_2) \wedge (\varphi_2 \vee \psi_2)]$ (Distribute "$\vee$" over "$\wedge$")

Given a propositional formula ϕ, we can in polynomial time construct a CNF formula ψ that is satisfiable if and only if ϕ is.

## Basic Concepts in Deduction

- **Inference rule**: Rule prescribing how we can infer new formulas.
  $\rightarrow$ For example, if the KB is $\{\ldots, (\varphi \rightarrow \psi), \ldots, \varphi, \ldots\}$ then $\psi$ can be deduced using the inference rule $\dfrac{\varphi, \varphi \rightarrow \psi}{\psi}$.
- **Calculus**: Set $\mathcal{R}$ of inference rules.
- **Derivation**: $\varphi$ can be derived from KB using $\mathcal{R}$, KB $\vdash_{\mathcal{R}} \varphi$, if starting from KB there is a sequence of applications of rules from $\mathcal{R}$, ending in $\varphi$.
- **Soundness**: $\mathcal{R}$ is sound if all derivable formulas do follow logically: if KB $\vdash_{\mathcal{R}} \varphi$, then KB $\models \varphi$.
- **Completeness**: $\mathcal{R}$ is complete if all formulas that follow logically are derivable: if KB $\models \varphi$, then KB $\vdash_{\mathcal{R}} \varphi$.

If R is sound and complete, then to check whether KB |= ϕ, we can check whether KB $|=_R$ ϕ.

**Resolution: Quick Facts**

**Input:** A CNF formula ψ.
**Method:** Calculus consists of a single rule, allowing to produce disjunctions using fewer variables.
We write ψ ` ϕ if ϕ can be derived from ψ using resolution.
**Output:** Can an impossible ϕ (the empty disjunction) be derived? “Yes”/”No”, where “yes” happens iff ψ is unsatisfiable.

**So how do we check whether KB |= ϕ?**
Proof by contradiction: Run resolution on ψ := CNF-transformation(KB ∪ {¬ϕ}). By the contradiction theorem, ψ is unsatisfiable iff KB |= ϕ.

Deduction can be reduced to proving unsatisfiability: “Assume, to the contrary, that KB holds but ϕ does not hold; then derive False”.

For the remainder of this chapter, we assume that the input is a set Δ of clauses:

**Terminology and Notation**

- A literal $l$ is an atom or the negation thereof (e.g., $P, \neg Q$); the negation of a literal is denoted $\bar{l}$ (e.g., $\overline{\neg Q} = Q$).
- A clause $C$ is a disjunction of literals. We identify $C$ with the set of its literals (e.g., $P \vee \neg Q$ becomes $\{P, \neg Q\}$).
- We identify a CNF formula $\psi$ with the set $\Delta$ of its clauses (e.g., $(P \vee \neg Q) \wedge R$ becomes $\{\{P, \neg Q\}, \{R\}\}$).
- The empty clause is denoted $\square$.

An interpretation I satisfies a clause C iff there exists l ∈ C such that I |= l.
I satisfies Δ iff, for all C ∈ Δ, we have I |= C.

**Definition (Resolution Rule).** *Resolution uses the following inference rule (with exclusive union $\dot{\cup}$ meaning that the two sets are disjoint):*

$$\frac{C_1 \dot{\cup} \{l\}, C_2 \dot{\cup} \{\bar{l}\}}{C_1 \cup C_2}$$

*If $\Delta$ contains parent clauses of the form $C_1 \dot{\cup} \{l\}$ and $C_2 \dot{\cup} \{\bar{l}\}$, the rule allows to add the resolvent clause $C_1 \cup C_2$. $l$ and $\bar{l}$ are called the resolution literals.*

**Lemma.** *The resolvent follows from the parent clauses.*

**Proof.** If $I \models C_1 \dot{\cup} \{l\}$ and $I \models C_2 \dot{\cup} \{\bar{l}\}$, then $I$ must make at least one literal in $C_1 \cup C_2$ true.

**Theorem (Soundness).** *If $\Delta \vdash D$, then $\Delta \models D$.* (Direct from Lemma.)

**Completeness**

**Is resolution complete?** Does $\Delta \models \varphi$ imply $\Delta \vdash \varphi$?

**BUT remember:** "Run resolution on $\psi :=$ CNF-transformation(KB $\cup \{\neg\varphi\}$): By the contradiction theorem, $\psi$ is unsatisfiable iff KB $\models \varphi$."

$\rightarrow$ This method *is* complete.

**Theorem (Refutation-Completeness).** $\Delta$ *is unsatisfiable iff* $\Delta \vdash \square$.

**Proof.** "If": Soundness. For "only if", we can prove that, if $\Delta \nvdash \square$, then $\Delta$ is satisfiable.

**Conclusions:**
Sometimes, it pays off to think before acting. In AI, "thinking" is implemented in terms of reasoning in order to deduce new knowledge from a knowledge base represented in a suitable logic. Logic prescribes a syntax for formulas, as well as a semantics prescribing which interpretations satisfy them. ϕ entails ψ if all interpretations that satisfy ϕ also satisfy ψ. Deduction is the process of deriving new entailed formulas. Propositional logic formulas are built from atomic propositions, with the connectives "and, or, not". Every propositional formula can be brought into conjunctive normal form (CNF), which can be identified with a set of clauses. Resolution is a deduction procedure based on trying to derive the empty clause. It is refutation-complete, and can be used to prove KB |= ϕ by showing that KB ∪ {¬ϕ} is unsatisfiable.

**Issues:**
Awkward to write for humans

# Chapter 10: Propositional Logic: SAT solver

The SAT Problem: Given a propositional formula ϕ, decide whether or not ϕ is satisfiable.

SAT can be viewed as a CSP problem in which all variable domains are Boolean, and the constraints have unbounded arity.
Given any constraint network γ, we can in low-order polynomial time construct a CNF formula ϕ(γ) that is satisfiable iff γ is solvable.
→ Anything we can do with CSP, we can (in principle) do with SAT.

### Local Search for SAT

**GSAT Algorithm**

**INPUT**: a set of clauses $\Delta$, MAX-FLIPS, and MAX-TRIES
**OUTPUT**: a satisfying truth assignment of $\Delta$, if found

```
for i := 1 to MAX-TRIES
   I := a randomly-generated truth assignment
   for j := 1 to MAX-FLIPS
      if I satisfies Δ then return I
      X := a proposition reversing whose truth assignment gives
             the largest increase in the number of satisfied clauses
      I := I with the truth assignment of X reversed
   end for
end for
return "no satisfying assignment found"
```

**SAT solver**: $\Delta$, returns interpretation I so that I |= $\Delta$.
**Complete SAT solver**: If such I does not exist, returns "unsatisfiable".

The DPLL procedure is a complete SAT solver.

### DPLL Procedure

```
function DPLL(Δ, I) returns a partial interpretation I, or "unsatisfiable"
/* Unit Propagation (UP) Rule: */
   Δ' := a copy of Δ; I' := I
   while Δ' contains a unit clause C = {l} do
      extend I' with the respective truth value for the proposition underlying l
      simplify Δ'      /* remove false literals and true clauses */
/* Termination Test: */
   if □ ∈ Δ' then return "unsatisfiable"
   if Δ' = {} then return I'
/* Splitting Rule: */
   select some proposition P for which I' is not defined
   I'' := I' extended with one truth value for P; Δ'' := a copy of Δ'; simplify Δ''
   if I''' := DPLL(Δ'', I'') ≠ "unsatisfiable" then return I'''
   I'' := I' extended with the other truth value for P; Δ'' := Δ'; simplify Δ''
   return DPLL(Δ'', I'')
```

### Properties of DPLL

**Unsatisfiable case**: What can we say if "unsatisfiable" is returned? → In this case, we know that $\Delta$ is unsatisfiable: Unit propagation is sound, in the sense that it does not reduce the set of solutions.

**Satisfiable case**: What can we say when a partial interpretation I is returned? → Any extension of I to a complete interpretation satisfies $\Delta$. (By construction, I suffices to satisfy all clauses.)

DPLL ≈ BacktrackingWithInference, with Inference() = unit propagation. Unit propagation is sound: It does not reduce the set of solutions.

**The Unit Propagation (UP) Rule . . .**

**while** $\Delta'$ contains a unit clause $\{l\}$ **do**
extend $I'$ with the respective truth value for the proposition underlying $l$
simplify $\Delta'$ /* remove false literals */

**. . . corresponds to a calculus:**

**Definition (Unit Resolution).** *Unit Resolution is the calculus consisting of the following inference rule:*

$$\frac{C \dot{\cup} \{\bar{l}\}, \{l\}}{C}$$

*That is, if $\Delta$ contains parent clauses of the form $C \dot{\cup} \{\bar{l}\}$ and $\{l\}$, the rule allows to add the resolvent clause $C$.*

→ Unit propagation = Resolution restricted to the case where one of the parent clauses is unit.

**UP/Unit Resolution: Soundness/Completeness**

**Soundness:**

Need to show: If $\Delta 0$ can be derived from $\Delta$ by UP, then $\Delta$ |= $\Delta 0$ .
Yes, because any derivation made by unit resolution can also be made by (full) resolution, which we already know has this property. (Intuitively: if $\Delta 0$ contains the unit clause {l}, then l must be made true so C ∪˙ {l} implies C.)

**Completeness:**

Need to show: If $\Delta$ |= $\Delta 0$ , then $\Delta 0$ can be derived from $\Delta$ by UP. No. UP makes only limited inferences, as long as there are unit clauses. It does not guarantee to infer everything that can be inferred.
Example: {{P, Q}, {P, ¬Q}, {¬P, Q}, {¬P, ¬Q}} is unsatisfiable but UP cannot derive the empty clause.

**DPLL vs. Resolution**

**Notation:** Define the number of decisions of a DPLL run as the total number of times a truth value was set by either unit propagation or the splitting rule.
**Theorem.** If DPLL returns “unsatisfiable” on $\Delta$, then $\Delta$ ` with a resolution derivation whose length is at most the number of decisions.
**Proof Sketch**. Consider first DPLL without the unit propagation rule. Consider any leaf node N, for proposition X, both of whose truth values directly result in a clause C that has become empty. Then for X = 0 the respective clause C must contain X; and for X = 1 the respective clause C must contain ¬X. Thus we can resolve these two clauses to a clause C(N) that does not contain X. C(N) can contain only the negations of the decision literals l1, . . . , lk above N. Remove N from the tree, then iterate the argument. Once the tree is empty, we have derived the empty clause. Unit propagation can be simulated via applications of the splitting rule, choosing a proposition that is constrained by a unit clause: One of the two truth values then immediately yields an empty clause

(not requested)
Theorem. If DPLL returns “unsatisfiable” on $\Delta$, then $\Delta \vdash$ with a resolution derivation whose length is at most the number of decisions

DPLL is an effective practical method for conducting resolution proofs.

In Fact: DPLL = tree resolution. This is a fundamental weakness! There are inputs $\Delta$ whose shortest tree-resolution proof is exponentially longer than their shortest (general) resolution proof.
→ In a tree resolution, each derived clause C is used only once (at its parent). The same C is derived anew every time it is used!
→ DPLL “makes the same mistakes over and over again”.
→ To the rescue: clause learning.

### Implication Graphs

**Notation/Terminology:** Literals set along a branch of DPLL

- Value of $P$ set by the splitting rule: choice literal, $P$ for $I(P) = 1$, respectively $\neg P$ for $I(P) = 0$.
- Value of $P$ set by the UP rule: implied literal $P$ respectively $\neg P$.
- Empty clause derived by UP: conflict literal $\square$.

**Definition (Implication Graph).** *Let $\Delta$ be a set of clauses, and consider any search branch $\beta$ of DPLL on $\Delta$. The implication graph $G^{\text{impl}}$ is a directed graph. Its vertices are the choice and implied literals along $\beta$, as well as a separate conflict vertex $\square_C$ for every clause $C$ that becomes empty.*
*When $\{l_1, \ldots, l_k, l'\} \in \Delta$ becomes unit with implied literal $l'$, $G^{\text{impl}}$ includes the arcs $\overline{l_1} \to l', \ldots, \overline{l_k} \to l'$. When $C = \{l_1, \ldots, l_k\} \in \Delta$ becomes empty, $G^{\text{impl}}$ includes the arcs $\overline{l_1} \to \square_C, \ldots, \overline{l_k} \to \square_C$.*

- How do we know that $\overline{l_1}, \ldots, \overline{l_k}$ are vertices in $G^{\text{impl}}$: Because $\{l_1, \ldots, l_k, l'\}$ became unit (respectively, empty).
- Vertices with indegree 0: Choice literals, and unit clauses of $\Delta$.

→ The implication graph is not uniquely determined by the choice literals.
Because: The implication graph also depends on “ordering decisions” made during UP: Which unit clause is picked first. Example: $\Delta$ = {{¬P, ¬Q}, {Q}, {P}}

### Conflict Graphs

→ A conflict graph captures “what went wrong” in a failed node.

**Definition (Conflict Graph).** *Let $\Delta$ be a set of clauses, and let $G^{\text{impl}}$ be the implication graph for some search branch of DPLL on $\Delta$. A conflict graph $G^{\text{confl}}$ is a sub-graph of $G^{\text{impl}}$ induced by a subset of vertices such that:*

1. *$G^{\text{confl}}$ contains exactly one conflict vertex $\square_C$.*
2. *If $l'$ is a vertex in $G^{\text{confl}}$, then all parents of $l'$, i.e. vertices $\overline{l_i}$ with a $G^{\text{impl}}$ arc $(\overline{l_i}, l')$, are vertices in $G^{\text{confl}}$ as well.*
3. *All vertices in $G^{\text{confl}}$ have a path to $\square_C$.*

→ Conflict graph = Starting at a conflict vertex, backchain through the implication graph until reaching choice literals.

## Clause Learning

**Observe:** Conflict graphs encode *logical entailments*

$$\Delta \models (\bigwedge_{l \in choiceLits(G^{\text{confl}})} l) \rightarrow \bot$$

→ Given $\Delta$, setting all choice literals in a conflict graph results in failure.

**Observe:** We can re-write this!

$$\Delta \models \bigvee_{l \in choiceLits(G^{\text{confl}})} \bar{l}$$

**Proposition (Clause Learning).** *Let $\Delta$ be a set of clauses, and let $G^{\text{confl}}$ be a conflict graph at some time point during a run of DPLL on $\Delta$. Let $choiceLits(G^{\text{confl}})$ be the choice literals in $G^{\text{confl}}$. Then $\Delta \models \{\bar{l} \mid l \in choiceLits(G^{\text{confl}})\}$.*

→ The negation of the choice literals in a conflict graph is a valid clause.

## The Effect of Learned Clauses

→ What happens after we learned a new clause $C$?

1. **We add $C$ into $\Delta$.** → Example: $C = \{\neg P, \neg Q\}$.
2. **We retract the last choice $l'$.**
   → Example: Retract the choice $l' = Q$.

**Observation:** $C = \{\bar{l} \mid l \in choiceLits(G^{\text{confl}})\}$. Before we learn the clause, $G^{\text{confl}}$ must contain the most recent choice $l'$: otherwise, the conflict would have occured earlier on. So $C = \{\overline{l_1}, \ldots, \overline{l_k}, \overline{l'}\}$ where $l_1, \ldots, l_k$ are earlier choices.
→ Example: $l_1 = P$, $C = \{\neg P, \neg Q\}$, $l' = Q$.

**Observation:** Given the earlier choices $l_1, \ldots, l_k$, after we learned the new clause $C = \{\overline{l_1}, \ldots, \overline{l_k}, \overline{l'}\}$, $\overline{l'}$ is now set by UP!

3. **We set the opposite choice $\overline{l'}$ as an implied literal.**
   → Example: Set $\neg Q$ as an implied literal.
4. **We run UP and analyze conflicts.** Learned clause: earlier choices only!
   → Example: $C = \{\neg P\}$, see next slide.

## Clause Learning vs. Resolution

1) DPLL = tree resolution: Each derived clause C (not in $\Delta$) is derived anew every time it is used.
2) There exist $\Delta$ whose shortest tree-resolution proof is exponentially longer than their shortest (general) resolution proof.

**This is no longer the case with clause learning!**

1) We add each learned clause C to $\Delta$, can use it as often as we like.
2) Clause learning renders DPLL equivalent to full resolution

**WHICH clause(s) to learn?**
While we only select choiceLits(Gconfl), much more can be done.
For any cut through Gconfl, with choiceLits(Gconfl) on the "left-hand" side of the cut and the conflict literals on the right-hand side, the literals on the left border of the cut yield a learnable clause. Must take care to not learn too many clauses…
Origins of clause learning: Clause learning originates from explanation-based (no-good) learning developed in the CSP community. The distinguishing feature here is that the "no-good" is a clause: → The exact same type of constraint as the rest of $\Delta$.

**Where Are the Hard Problems?**

- SAT is **NP**-hard. Worst case for DPLL is $2^n$, with $n$ propositions.
- Imagine I gave you as homework to make a formula family $\{\phi\}$ where DPLL runtime necessarily is in the order of $2^n$.
  → I promise you're not gonna find this easy . . . (although it is of course possible: e.g., the "Pigeon Hole Problem").
- People noticed by the early 90s that, in practice, the DPLL worst case does not tend to happen.

So, what's the problem?
Science is about understanding the world.
→ Are "hard cases" just pathological outliers? Can we say something about the typical case?

Difficulty 1: What is the "typical case" in applications? E.g., what is the "average" Hardware Verification instance? → Consider precisely defined random distributions instead.

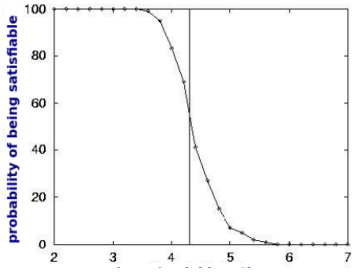
Difficulty 2: Search trees get very complex, and are difficult to analyze mathematically, even in trivial examples. Never mind examples of practical relevance…. → The most successful works are empirical. (Interesting theory is mainly concerned with hand-crafted formulas, like the Pigeon Hole Problem.)

**Phase Transitions in SAT**

**Fixed clause length model:** Fix clause length $k$; $n$ variables. Generate $m$ clauses, by choosing uniformly at random $k$ variables $P$ for each clause $C$, and, for each variable $P$, deciding uniformly at random whether to add $P$ or $\neg P$ into $C$.

**Order parameter:** Clause/variable ratio $\frac{m}{n}$.

**Phase transition:** (Fixing $k = 3$, $n = 50$)




**Does DPLL Care?** yes, it does! Extreme runtime peak at the phase transition!
**Why does DPLL care?**

Intuitive explanation:

Under-Constrained: Satisfiability likelihood close to 1. Many solutions, first DPLL search path usually successful. (“Deep but narrow”)
Over-Constrained: Satisfiability likelihood close to 0. Most DPLL search paths short, conflict reached after few applications of splitting rule. (“Broad but shallow”)
Critically Constrained: At the phase transition, many almost-successful DPLL search paths. (“Close, but no cigar”)

**The Phase Transition Conjecture**

“All NP-complete problems have at least one order parameter, and the hard to solve problems are around a critical value of this order parameter. This critical value (a phase transition) separates one region from another, such as over-constrained and under-constrained regions of the problem space.”

**Why Should We Care?**

Enlightenment: Phase transitions contribute to the fundamental understanding of the behavior of search, even if it’s only in random distributions. There are interesting theoretical connections to phase transition phenomena in physics.

**What can we use these results for?**

**Benchmark design:**

Choose instances from phase transition region.
→ Commonly used in competitions etc. (In SAT, random phase transition formulas are the most difficult for DPLL-style searches.)
Predicting solver performance: Yes, but very limited because: All this works only for the particular considered distributions of instances! Not meaningful for any other instances.

**Summary**

**SAT** solvers decide satisfiability of CNF formulas. This can be used for deduction, and is highly successful as a general problem solving technique (e.g., in Verification).

**DPLL** = backtracking with inference performed by unit propagation (UP), which iteratively instantiates unit clauses and simplifies the formula. DPLL proofs of unsatisfiability correspond to a restricted form of resolution. The restriction forces DPLL to “makes the same mistakes over again”.

**Implication graphs** capture how UP derives conflicts. Their analysis enables us to do clause learning. DPLL with clause learning is called CDCL. It corresponds to full resolution, not “making the same mistakes over again”.

**CDCL** is state of the art in applications, routinely solving formulas with millions of propositions.
In particular random formula distributions, typical problem hardness is characterized by phase transitions.

# Chapter 11. Predicate Logic Reasoning, Part I: Basics

Predicate Logic extends propositional logic with the ability to explicitly speak about objects and their properties.

- Even when we can describe the problem suitably, for the desired reasoning, the propositional formulation typically is way too large to write (by hand).
- Inner structure of vocabulary is ignored
- It is impossible to speak about infinite sets of objects!

**Practical Relevance/Applications:**

- Logic programming: Prolog et al.
- Databases: Deductive databases where elements of logic allow to conclude additional facts. Logic is tied deeply with database theory.
- Semantic technology. Mega-trend since > a decade. Use FOL fragments to annotate data sets, facilitating their use and analysis.

### The Alphabet of FOL

**Common symbols:**

- Variables: $x$, $x_1$, $x_2$, ..., $x'$, $x''$, ..., $y$, ..., $z$, ...
- Truth/Falseness: $\top$, $\bot$. (As in propositional logic)
- Operators: $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$. (As in propositional logic)
- Quantifiers: $\forall$, $\exists$.
  $\rightarrow$ Precedence: $\neg > \forall, \exists > \ldots$ (we'll be using brackets).

**Application-specific symbols:**

- Constant symbols ("object", e.g., $BlockA$, $BlockB, a, b, c, \ldots$)
- Predicate symbols, arity $\geq 1$ (e.g., $Block(.)$, $Above(.,.)$)
- Function symbols, arity $\geq 1$ (e.g., $WeightOf(.)$, $Succ(.)$)

**Definition (Signature).** *A signature $\Sigma$ in predicate logic is a finite set of constant symbols, predicate symbols, and function symbols.*

$\rightarrow$ In mathematics, $\Sigma$ can be infinite; not considered here.

### Syntax of FOL

$\rightarrow$ Terms represent objects:

**Definition (Term).** *Let $\Sigma$ be a signature. Then:*

1. *Every variable and every constant symbol is a $\Sigma$-term.* $[x,\ Garfield]$
2. *If $t_1, t_2, \ldots, t_n$ are $\Sigma$-terms and $f \in \Sigma$ is an $n$-ary function symbol, then $f(t_1, t_2, \ldots, t_n)$ also is a $\Sigma$-term.* $[FoodOf(x)]$

*Terms without variables are ground terms.* $[FoodOf(Garfield)]$

$\rightarrow$ For simplicity, we usually don't write the "$\Sigma$-".

$\rightarrow$ Atoms represent atomic statements about objects:

**Definition (Atom).** *Let $\Sigma$ be a signature. Then:*

1. *$\top$ and $\bot$ are $\Sigma$-atoms.*
2. *If $t_1, t_2, \ldots, t_n$ are terms and $P \in \Sigma$ is an $n$-ary predicate symbol, then $P(t_1, t_2, \ldots, t_n)$ is a $\Sigma$-atom.* $[Chases(Lassie, y)]$

*Atoms without variables are ground atoms.* $[Chases(Lassie, Garfield)]$

→ Formulas represent complex statements about objects:

**Definition (Formula).** *Let $\Sigma$ be a signature. Then:*

1. *Each $\Sigma$-atom is a $\Sigma$-formula.*
2. *If $\varphi$ is a $\Sigma$-formula, then so is $\neg\varphi$.*

*The formulas that can be constructed by rules 1. and 2. are literals.*

*If $\varphi$ and $\psi$ are $\Sigma$-formulas, then so are:*

4. $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, *and* $\varphi \leftrightarrow \psi$.

*If $\varphi$ is a $\Sigma$-formula and $x$ is a variable, then*

5. $\forall x\varphi$ *is a $\Sigma$-formula* ("Universal Quantification").
6. $\exists x\varphi$ *is a $\Sigma$-formula* ("Existential Quantification").

**Meaning of FOL formulas:**

Terms represent objects. [FoodOf (Garfield) = Lasagna]
Predicates represent relations on the universe. [Dog = {Lassie, Bello}]
Universally-quantified variables: "for all objects in the universe". Existentially-quantified variables: "at least one object in the universe".
→ Similar to propositional logic, we define interpretations, models, satisfiability, validity, …

**Semantics of FOL: Interpretations**

**Definition (Interpretation).** *Let $\Sigma$ be a signature. A $\Sigma$-interpretation is a pair $(U, I)$ where $U$, the universe, is an arbitrary non-empty set $[U = \{o_1, o_2, \dots\}]$, and $I$ is a function, notated as superscript, so that*

1. *$I$ maps constant symbols to elements of $U$: $c^I \in U$ $\quad [Lassie^I = o_1]$*
2. *$I$ maps $n$-ary predicate symbols to $n$-ary relations over $U$:*
   $P^I \subseteq U^n$ $\quad [Dog^I = \{o_1, o_3\}]$
3. *$I$ maps $n$-ary function symbols to $n$-ary functions over $U$:*
   $f^I \in [U^n \mapsto U]$ $\quad [FoodOf^I = \{(o_1 \mapsto o_4), (o_2 \mapsto o_5), \dots\}]$

→ We will often refer to $I$ as the interpretation, omitting $U$. Note that $U$ may be infinite.

**Definition (Ground Term Interpretation).** *The interpretation of a ground term under $I$ is $(f(t_1, \dots, t_n))^I = f^I(t_1^I, \dots, t_n^I)$. $[(FoodOf(Lassie))^I = FoodOf^I(Lassie^I) = FoodOf^I(o_1) = o_4]$*

**Definition (Ground Atom Satisfaction).** *Let $\Sigma$ be a signature and $I$ a $\Sigma$-interpretation. We say that $I$ satisfies a ground atom $P(t_1, \dots, t_n)$, written $I \models P(t_1, \dots, t_n)$, iff $(t_1^I, \dots, t_n^I) \in P^I$. We also call $I$ a model of $P(t_1, \dots, t_n)$.* $\quad [I \models Dog(Lassie)$ because $Lassie^I = o_1 \in Dog^I]$

## Semantics of FOL: Variable Assignments

**Definition (Variable Assignment).** *Let $\Sigma$ be a signature and $(U, I)$ a $\Sigma$-interpretation. Let $X$ be the set of all variables. A variable assignment $\alpha$ is a function $\alpha : X \mapsto U$.* $[\alpha(x) = o_1]$

**Definition (Term Interpretation).** *The interpretation of a term under $I$ and $\alpha$ is:*

1. $x^{I,\alpha} = \alpha(x)$ $[x^{I,\alpha} = o_1]$
2. $c^{I,\alpha} = c^I$ $[Lassie^{I,\alpha} = Lassie^I]$
3. $(f(t_1, \ldots, t_n))^{I,\alpha} = f^I(t_1^{I,\alpha}, \ldots, t_n^{I,\alpha})$
   $[(FoodOf(x))^{I,\alpha} = FoodOf^I(x^{I,\alpha}) = FoodOf^I(o_1) = o_4]$

**Definition (Atom Satisfaction).** *Let $\Sigma$ be a signature, $I$ a $\Sigma$-interpretation, and $\alpha$ a variable assignment. We say that $I$ and $\alpha$ satisfy an atom $P(t_1, \ldots, t_n)$, written $I, \alpha \models P(t_1, \ldots, t_n)$, iff $(t_1^{I,\alpha}, \ldots, t_n^{I,\alpha}) \in P^I$. We also call $I$ and $\alpha$ a model of $P(t_1, \ldots, t_n)$.*

$[I, \alpha \not\models Dog(FoodOf(x))\text{: } (FoodOf(x))^{I,\alpha} = o_4 \notin Dog^I]$

## Semantics of FOL: Formula Satisfaction

**Notation:** In $\alpha\frac{x}{o}$ we overwrite $x$ with $o$ in $\alpha$: for $\alpha = \{(x \mapsto o_1), (y \mapsto o_2), , \ldots\}$, $\alpha\frac{x}{o} = \{(x \mapsto o), (y \mapsto o_2), , \ldots\}$.

**Definition (Formula Satisfaction).** *Let $\Sigma$ be a signature, $I$ a $\Sigma$-interpretation, and $\alpha$ a variable assignment. We set:*

| | | |
|---|---|---|
| $I, \alpha \models \top$ *and* | $I, \alpha \not\models \bot$ | |
| $I, \alpha \models \neg\varphi$ | *iff* | $I, \alpha \not\models \varphi$ |
| $I, \alpha \models \varphi \wedge \psi$ | *iff* | $I, \alpha \models \varphi$ *and* $I, \alpha \models \psi$ |
| $I, \alpha \models \varphi \vee \psi$ | *iff* | $I, \alpha \models \varphi$ *or* $I, \alpha \models \psi$ |
| $I, \alpha \models \varphi \rightarrow \psi$ | *iff* | *if* $I, \alpha \models \varphi$, *then* $I, \alpha \models \psi$ |
| $I, \alpha \models \varphi \leftrightarrow \psi$ | *iff* | *if* $I, \alpha \models \varphi$ *if and only if* $I, \alpha \models \psi$ |
| $I, \alpha \models \forall x\varphi$ | *iff* | *for all* $o \in U$ *we have* $I, \alpha\frac{x}{o} \models \varphi$ |
| $I, \alpha \models \exists x\varphi$ | *iff* | *there exists* $o \in U$ *so that* $I, \alpha\frac{x}{o} \models \varphi$ |

*If $I, \alpha \models \varphi$, we say that $I$ and $\alpha$ satisfy $\varphi$ (are a model of $\varphi$).*

$[\varphi = \forall x[Dog(x) \rightarrow \exists y Chases(x, y)]$, $Dog^{I,\alpha} = \{Lassie^{I,\alpha}, Bello^{I,\alpha}\}$, $Chases^{I,\alpha} = \{(Lassie^{I,\alpha}, Garfield^{I,\alpha})\}$. Then $I, \alpha \not\models \varphi$ because Bello does not chase anything.]

### FOL Satisfiability

**Satisfiability**

A FOL formula $\varphi$ is:

- satisfiable if there exist $I, \alpha$ that satisfy $\varphi$.
- unsatisfiable if $\varphi$ is not satisfiable.
- falsifiable if there exist $I, \alpha$ that do not satisfy $\varphi$.
- valid if $I, \alpha \models \varphi$ holds for all $I$ and $\alpha$. We also call $\varphi$ a tautology.

**Entailment and Equivalence**

$\varphi$ entails $\psi$, $\varphi \models \psi$, if every model of $\varphi$ is a model of $\psi$.
$\varphi$ and $\psi$ are equivalent, $\varphi \equiv \psi$, if $\varphi \models \psi$ and $\psi \models \varphi$.

**Attention:** In presence of free variables!

$\rightarrow$ Do we have $Dog(x) \models Dog(y)$? No. Example: $Dog^I = \{o_1\}$, $\alpha = \{(x \mapsto o_1), (y \mapsto o_2)\}$. Then $I, \alpha \models Dog(x)$ but $I, \alpha \not\models Dog(y)$.

### Free and Bound Variables

$$\varphi := \forall x[R(\boxed{y}, \boxed{z}) \wedge \exists y(\neg P(y, x) \vee R(y, \boxed{z}))]$$

**Definition (Free Variables).** *By $vars(e)$, where $e$ is either a term or a formula, we denote the set of variables occuring in $e$. We set:*

$$\begin{aligned} free(P(t_1, \ldots, t_n)) &:= vars(t_1) \cup \cdots \cup vars(t_n) \\ free(\neg\varphi) &:= free(\varphi) \\ free(\varphi * \psi) &:= free(\varphi) \cup free(\psi) \text{ for } * \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\ free(\forall x\varphi) &:= free(\varphi) \setminus \{x\} \\ free(\exists x\varphi) &:= free(\varphi) \setminus \{x\} \end{aligned}$$

*$free(\varphi)$ are the free variables of $\varphi$. $\varphi$ is closed if $free(\varphi) = \emptyset$.*

$\rightarrow$ In the above $\varphi$, which variable appearances are free? The boxed ones.

$\rightarrow$ Knowledge Base (aka logical theory) = set of closed formulas. From now on, we asume that $\varphi$ is closed.

$\rightarrow$ We can ignore $\alpha$, and will write $I \models \varphi$ instead of $I, \alpha \models \varphi$.

### Normal Forms

Why normal forms?
Convenient: full syntax when describing the problem at hand.
Not convenient: full syntax when solving the problem.
“Solving the problem”? Decide satisfiability! → Uniform decision problem to tackle deduction as well as other applications.

Which normal forms?

- Prenex normal form: Move all quantifiers up front.
- Skolem normal form: Prenex, + remove all existential quantifiers while preserving satisfiability.
- Clausal normal form: Skolem, + CNF transformation while preserving satisfiability.

**Prenex Normal Form**

quantifier prefix + (quantifier-free) matrix

$$Qx_1Qx_2Qx_3 \dots Qx_n \ \varphi$$

**Step 1: Eliminate $\rightarrow$ and $\leftrightarrow$, move $\neg$ inwards**

1. $(\varphi \leftrightarrow \psi) \equiv [(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)]$ (Eliminate "$\leftrightarrow$")
2. $(\varphi \rightarrow \psi) \equiv (\neg\varphi \vee \psi)$ (Eliminate "$\rightarrow$")
3. $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ and $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$
   $\neg\forall x\varphi \equiv \exists x\neg\varphi$ and $\neg\exists x\varphi \equiv \forall x\neg\varphi$ (Move "$\neg$" inwards)

**Example:** $\neg\forall x[(\forall x P(x)) \rightarrow R(x)]$

$\rightarrow$ Eliminate $\rightarrow$ and $\leftrightarrow$: $\neg\forall x[\neg(\forall x P(x)) \vee R(x)]$.

$\rightarrow$ Move $\neg$ across first quantifier: $\exists x\neg[\neg(\forall x P(x)) \vee R(x)]$.

$\rightarrow$ Move $\neg$ inwards: $\exists x[(\forall x P(x)) \wedge \neg R(x)]$.

**Step 2: Move quantifiers outwards**

- $(\forall x\varphi) \wedge \psi \equiv \forall x(\varphi \wedge \psi)$, if $x$ not free in $\psi$.
- $(\forall x\varphi) \vee \psi \equiv \forall x(\varphi \vee \psi)$, if $x$ not free in $\psi$.
- $(\exists x\varphi) \wedge \psi \equiv \exists x(\varphi \wedge \psi)$, if $x$ not free in $\psi$.
- $(\exists x\varphi) \vee \psi \equiv \exists x(\varphi \vee \psi)$, if $x$ not free in $\psi$.

**Example "Animals":** $\forall x[\neg Dog(x) \vee \exists y Chases(x, y)]$

$\rightarrow$ Move "$\exists y$" outwards: $\forall x\exists y[\neg Dog(x) \vee Chases(x, y)]$.

**Example:** $\exists x[(\forall x P(x)) \wedge \neg R(x)]$

$\rightarrow$ We can't move "$\forall x$" outwards because $x$ is free in "$\neg R(x)$"!

### Prenex Normal Form: Variable Renaming

**Notation:** If $x$ is a variable, $t$ a term, and $\varphi$ a formula, then the instantiation of $x$ with $t$ in $\varphi$, written $\varphi\frac{x}{t}$, replaces all free appearances of $x$ in $\varphi$ by $t$. If $t = y$ is a variable, then $\varphi\frac{x}{y}$ renames $x$ to $y$ in $\varphi$.

**Lemma.** *If $y \notin vars(\varphi)$, then $\forall x\varphi \equiv \forall y\varphi\frac{x}{y}$ and $\exists x\varphi \equiv \exists y\varphi\frac{x}{y}$.*

**Step 2 Addition: Rename variables if needed**

For each Step 2 rule: If $x$ is free in $\psi$, then rename $x$ in $(\forall x\varphi)$ respectively $(\exists x\varphi)$ to some new variable $y$. Then, the rule can be applied.

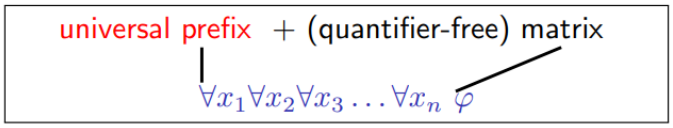
**Example:** $\exists x[(\forall x P(x)) \wedge \neg R(x)]$

$\rightarrow$ Rename $\frac{x}{y}$ in $(\forall x P(x))$: $\exists x[(\forall y P(y)) \wedge \neg R(x)]$.

$\rightarrow$ Move $\forall y$ outwards: $\exists x \forall y[P(y) \wedge \neg R(x)]$.

**Theorem.** *There exists an algorithm that, for any FOL formula $\varphi$, efficiently (i.e., in polynomial time) calculates an equivalent formula in prenex normal form.* (Proof: We just outlined that algorithm.)

### Skolem Normal Form




**Theorem (Skolem).** *Let $\varphi = \forall x_1 \ldots \forall x_k \exists y \psi$ be a closed FOL formula in prenex normal form, such that all quantified variables are pairwise distinct, and the $k$-ary function symbol $f$ does not appear in $\varphi$. Then $\varphi$ is satisfiable if and only if $\forall x_1 \cdots \forall x_k \psi\frac{y}{f(x_1,\ldots,x_k)}$ is satisfiable.* (Proof omitted.)

Note: Here, "0-ary function symbol" = constant symbol.

**Transformation to Skolem normal form**

Rename quantified variables until distinct. Then iteratively remove the outmost existential quantifier, using Skolem's theorem.

**Example.** $\exists x \forall y \exists z R(x, y, z)$ is transformed to:

$\rightarrow$ Remove "$\exists x$": $\forall y \exists z R(f, y, z)$. Remove "$\exists z$": $\forall y R(f, y, g(y))$.

$\rightarrow$ Note the arity/arguments of $f$ vs. $g$: "$x_1 \ldots x_k$" in the above!

**Notation:** A formula is in Skolem normal form (SNF) if it is in prenex normal form and has no existential quantifiers.

**Theorem.** *There exists an algorithm that, for any closed FOL formula $\varphi$, efficiently calculates an SNF formula that is satisfiable iff $\varphi$ is. We denote that formula $\varphi^*$.* (Proof: We just outlined that algorithm.)

**Example 1:** (a) $\varphi_1 = \exists y \forall x [\neg Dog(x) \vee Chases(x, y)]$: "There exists a $y$ chased by every dog $x$". (b) $\varphi_1^* = \forall x [\neg Dog(x) \vee Chases(x, f)]$: "The object named $f$ is chased by every dog $x$".

**Example 2:** (a) $\varphi_2 = \forall x \exists y [\neg Dog(x) \vee Chases(x, y)]$: "For every dog $x$, there exists $y$ chased by $x$". (b) $\varphi_2^* = \forall x [\neg Dog(x) \vee Chases(x, f(x))]$: "For every dog $x$, we can interprete $f(x)$ with a $y$ chased by $x$".

→ Satisfying existential quantifier for universally quantified variables $x_1, \ldots, x_k$ = choosing a value for a function of $x_1, \ldots, x_k$.

**Note:** $\varphi^*$ is *not* equivalent to $\varphi$. $\varphi^*$ implies $\varphi$, but not vice versa. Example for $I \models \varphi$ but $I \not\models \varphi^*$: $\varphi = \exists x\ Dog(x)$, $\varphi^* = Dog(f)$, $Dog^I = \{Lassie\}$, $f^I = \{Garfield\}$.

### Clausal Normal Form

universal prefix + disjunction of literals

$$\forall x_1 \forall x_2 \forall x_3 \ldots \forall x_n (l_1 \vee \cdots \vee l_n)$$

→ Written $\{l_1, \ldots, l_n\}$.

**Transformation to clausal normal form**

1. Transform to SNF: $\forall x_1 \forall x_2 \forall x_3 \cdots \forall x_n\ \varphi$.
2. Transform $\varphi$ to satisfiability-equivalent CNF $\psi$. (Same as in propositional logic.)
3. Write as set of clauses: One for each disjunction in $\psi$.
4. Standardize variables apart: Rename variables so that each occurs in at most one clause. (Needed for FOL resolution, **Chapter 12.**)

**Theorem.** *There exists an algorithm that, for any closed FOL formula $\varphi$, efficiently calculates a formula in clausal normal form that is satisfiable iff $\varphi$ is satisfiable.* (Proof: We just outlined that algorithm.)

### Summary

Predicate logic allows us to explicitly speak about objects and their properties. It is thus a more natural and compact representation language than propositional logic; it also enables us to speak about infinite sets of objects. Logic has thousands of years of history. A major current application in AI is Semantic Technology. First-order predicate logic (FOL) allows universal and existential quantification over objects. A FOL interpretation consists of a universe U and a function I mapping constant symbols/predicate symbols/function symbols to elements/relations/functions on U. In prenex normal form, all quantifiers are up front. In Skolem normal form, additionally there are no existential quantifiers. In clausal normal form, additionally the formula is in CNF. Any FOL formula can efficiently be brought into a satisfiability-equivalent clausal normal form.

# Chapter 12. Predicate Logic Reasoning, Part II: Reasoning

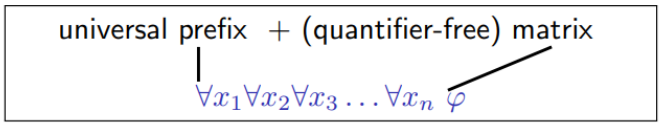
What do we need FOL for, then?

- "First-order logic as syntactic sugar for propositional logic."
- Remember all these propositions in the Wumpus world?
- Anyway, it's of course not that easy in general

How?
1 Bring into Skolem normal form (SNF).
2 Generate (the finite subsets of) the Herbrand expansion (up next).
3 Use propositional reasoning.

### Herbrand Expansion

**We assume:** Skolem normal form. (We don't require $\varphi$ to be in CNF.)




**Notation:** For any (finite) set $\theta^*$ of FOL formulas, denote by $CF(\theta^*)$ the set of constant symbols, and function symbols (arity $\geq 1$), occuring in $\theta^*$. If no constant symbol occurs in $\theta^*$, we add a new such symbol $c$ into $CF(\theta^*)$.

**Definition (Herbrand Universe).** *Let $\theta^*$ be a set of FOL formulas in SNF. Then the Herbrand universe $HU(\theta^*)$ over $\theta^*$ is the set of all ground terms that can be formed from $CF(\theta^*)$.*

**Example:** $\theta^* = \{\forall x[\neg Dog(x) \vee Chases(x, f(x))]\}$
$CF(\theta^*) =$ $\quad$ $HU(\theta^*) =$

**Definition (Herbrand Expansion).** *Let $\theta^*$ be a set of FOL formulas in SNF. The Herbrand expansion $HE(\theta^*)$ is defined as:*

$$HE(\theta^*) = \{\varphi\frac{x_1}{t_1}, \ldots, \frac{x_n}{t_n} \mid (\forall x_1 \ldots \forall x_n \varphi) \in \theta^*, t_i \in HU(\theta^*)\}$$

$\rightarrow$ Instantiate each matrix $\varphi$ with all terms from $HU(\theta^*)$. As $HE(\theta^*)$ contains ground atoms only, it can be interpreted as propositional logic.

**Example:** $\theta^* = \{\forall x[\neg Dog(x) \vee Chases(x, f(x))]\}$
$\rightarrow HE(\theta^*) = \{[\neg Dog(c) \vee Chases(c, f(c))],$
$[\neg Dog(f(c)) \vee Chases(f(c), f(f(c)))], \ldots\}.$

**Theorem (Herbrand).** *Let $\theta^*$ be a set of FOL formulas in SNF. Then, $\theta^*$ is satisfiable iff $HE(\theta^*)$ is satisfiable.* (Proof omitted.)

$\rightarrow$ **Observe:** Without function symbols, the Herbrand expansion is finite, and FOL reasoning is equivalent to propositional reasoning.

## Herbrand: The Infinite Case

→ **Recall:** Without function symbols, the Herbrand expansion is finite, and FOL reasoning is equivalent to propositional reasoning.

→ But what if there *are* function symbols?

**Theorem (Compactness of Propositional Logic).** *Any set $\theta$ of propositional logic formulas is unsatisfiable if and only if at least one finite subset of $\theta$ is unsatisfiable.* (Proof omitted.)

**Method:** Enumerate all finite subsets $\theta_1$ of the Herbrand expansion $HE(\theta^*)$, and test propositional satisfiability of $\theta_1$. $\theta$ is unsatisfiable if and only if one of the $\theta_1$ is. Only ... which $\theta_1$ will do the job?

→ If the Herbrand expansion is *infinite*, to show unsatisfiability (= to prove that some property does indeed follow from the KB), we must somehow choose a "relevant" finite subset thereof.

→ Direct FOL reasoning ameliorating this caveat: later in this chapter.

## What If θ is Satisfiable?

**Theorem (A).** *The set of unsatisfiable FOL formulas is recursively enumerable.*

**Proof.** Enumerate all FOL formulas $\varphi$. Incrementally for all of these in parallel, enumerate all finite subsets $\theta_1$ of the Herbrand expansion $HE(\varphi^*)$. Test propositional satisfiability of each $\theta_1$. By compactness of propositional logic, if $HE(\varphi^*)$ is unsatisfiable then one of the $\theta_1$ is.

**Theorem (B).** *It is undecidable whether a FOL formula is satisfiable.* (Proof omitted.)

**Corollary.** *The set of satisfiable FOL formulas is not recursively enumerable.* (Proof: Else, with Theorem (A), FOL satisfiability would be decidable, in contradiction to Theorem (B).)

→ If a FOL formula is unsatisfiable, then we can confirm this. Otherwise, we might end up in an infinite loop.

## Towards FOL Resolution

**Clausal normal form:**

universal prefix + disjunction of literals

$\forall x_1 \forall x_2 \forall x_3 \ldots \forall x_n (l_1 \vee \cdots \vee l_n)$

→ Written $\{l_1, \ldots, l_n\}$.

→ The quantifiers are omitted in the notation!

**Example:** $\{\{Nat(s(x)), \neg Nat(x)\}, \{Nat(1)\}\}$

**We want to somehow apply/adapt the resolution rule:**

$$\frac{C_1 \dot\cup \{l\}, C_2 \dot\cup \{\bar{l}\}}{C_1 \cup C_2}$$

### Substitutions

**Definition (Substitution).** *A substitution* $s = \{\frac{x_1}{t_1}, \ldots, \frac{x_n}{t_n}\}$ *is a function that substitutes variables* $x_i$ *for terms* $t_i$*, where* $x_i \neq t_i$ *for all* $i$*. Applying substitution* $s$ *to an expression* $\varphi$ *yields the expression* $\varphi s$*, which is* $\varphi$ *with all occurrences of* $x_i$ *simultaneously replaced by* $t_i$*.*

→ Variable instantiation and renaming, as used in the prenex and Skolem transformations as well as in the Herbrand expansion, is a special case of substitution.

**Example:** For $s = \{\frac{x}{y}, \frac{y}{h(a,b)}\}$, $P(x,y)s = P(y, h(a,b))$.

**Remember:** $x, y, z, v, w, \ldots$: variables; $a, b, c, d, e, \ldots$: constants.

### Composing Substitutions

**Definition (Composition).** *Given substitutions* $s_1$ *and* $s_2$*, by* $s_1 s_2$ *we denote the composed substitution, a single substitution whose outcome is identical to* $s_2 \circ s_1$*.*

**Example:** With $s_1 = \{\frac{z}{g(x,y)}, \frac{v}{w}\}$ and $s_2 = \{\frac{x}{a}, \frac{y}{b}, \frac{w}{v}, \frac{z}{d}\}$, we have $P(x,y,z,v)s_1 s_2 =$

**How to obtain $s_1 s_2$ given $s_1$ and $s_2$?**

1. Apply $s_2$ to the replacement terms $t_i$ in $s_1$.
2. For any variable $x_i$ replaced by $s_2$ but not by $s_1$, apply the respective variable/term pair $\frac{x_i}{t_i}$ of $s_2$.
3. Remove any pairs of variable $x$ and term $t$ where $x = t$.

**Example:** $\{\frac{z}{g(x,y)}, \frac{v}{w}\}\{\frac{x}{a}, \frac{y}{b}, \frac{w}{v}, \frac{z}{d}\} =$

### Properties of Substitutions

**For any formula $\varphi$ and substitutions $s_1$, $s_2$, $s_3$:**

→ $(\varphi s_1)s_2 = \varphi(s_1 s_2)$ by definition (composing functions).

→ $(s_1 s_2)s_3 = s_1(s_2 s_3)$ by definition (composing functions).

→ $s_1 s_2 = s_2 s_1$?

**(And by the way:)** (idempotence)

**Proposition.** *A substitution $s = \{\frac{x_1}{t_1}, \ldots, \frac{x_n}{t_n}\}$ is idempotent, i.e., $\varphi ss = \varphi s$ for all $\varphi$, iff $t_i$ does not contain $x_j$ for $1 \leq i, j \leq n$.*

**Proof.** "⇐": The second application of $s$ does not do anything because al $x_i$ have been removed. "⇒": if $t_i$ contains $x_j$ then the second application of $s$ replaces $x_j$ with $t_j \neq x_j$.

**Example:** For $s = \{\frac{x}{y}, \frac{y}{h(a,b)}\}$,
$P(x, y)s = P(y, h(a, b)) \neq P(x, y)ss = P(h(a, b), h(a, b))$.

### Unification

**Definition (Unifier).** *We say that a substitution $s$ is a unifier for a set of expressions $\{E_1, \ldots, E_k\}$ if $E_i s = E_j s$ for all $i, j$.*

**Notation:** We'll usually write $\{E_i\}$ for $\{E_1, \ldots, E_k\}$.

**Example:** $\{P(x, f(y, z), b), P(x, f(b, w), b)\}$

→ $s = \{\frac{y}{b}, \frac{z}{w}, \frac{x}{h(a,b)}\}$?

→ $s = \{\frac{y}{b}, \frac{z}{w}\}$?

**Definition (mgu).** *We say that $g$ is an mgu of $\{E_i\}$ if, for any unifier $s$ of $\{E_i\}$, there exists a substitution $s'$ such that $\{E_i\}s = \{E_i\}gs'$.*

→ If any unifier exists, then an idempotent mgu exists.

→ We'll next introduce an algorithm that finds it.

### Unification Algorithm: Disagreement Set

**Terminology:** The disagreement set of a set of expressions $\{E_i\}$ is the set of sub-expressions $\{t_i\}$ of $\{E_i\}$ at the first position in $\{E_i\}$ for which some of the $\{E_i\}$ disagree.

## Unification Algorithm

**Theorem.** *The following algorithm succeeds if and only if there exists a unifier for* $\{E_i\}$. *In the positive case, the algorithm returns an idempotent mgu of* $\{E_i\}$. (Proof omitted.)

$k \leftarrow 0$, $T_k = \{E_i\}$, $s_k = \{\}$;
**while** $T_k$ is not a singleton **do**
  Let $D_k$ be the disagreement set of $T_k$;
/* if $t_k$ contains $x_k$ then unification is impossible, cf. slide 24 */
  Let $x_k, t_k \in D_k$ be a variable and term s.t. $t_k$ does not contain $x_k$;
  **if** such $x_k, t_k$ do not exist **then exit** with output "failure";
  $s_{k+1} \leftarrow s_k\{\frac{x_k}{t_k}\}$; /* $t_k$ does not contain any of $x_1, \ldots, x_k$ */
  $T_{k+1} \leftarrow T_k\{\frac{x_k}{t_k}\}$; /* $x_k$ does not occur in $T_{k+1}$ */
  $k \leftarrow k + 1$;
**endwhile**
**exit** with output $s_k$;

## FOL Resolution: set up

**We assume:** Clausal normal form, variables standardized apart.

universal prefix + disjunction of literals

$$\forall x_1 \forall x_2 \forall x_3 \ldots \forall x_n (l_1 \vee \cdots \vee l_n)$$

$\rightarrow$ Written $\{l_1, \ldots, l_n\}$.

**Example:** $\{\{Nat(s(x)), \neg Nat(x)\}, \{Nat(y)\}\}$

**Reminder: Terminology and Notation**

- A literal $l$ is an atom or the negation thereof; the negation of a literal is denoted $\bar{l}$ (e.g., $\overline{\neg Q} = Q$).
- A clause $C$ is a set (=disjunction) of literals.
- Our input is a set $\Delta$ of clauses.
- The empty clause is denoted $\square$.
- A calculus is a set of inference rules.

**Derivations:** *We say that a clause $C$ can be derived from $\Delta$ using calculus $\mathcal{R}$, written $\Delta \vdash_{\mathcal{R}} C$, if (starting from $\Delta$) there is a sequence of applications of rules from $\mathcal{R}$, ending in $C$.*

$\rightarrow$ In contrast to propositional resolution, we will consider here three different resolution calculi $\mathcal{R}$.

**Soundness:** *A calculus $\mathcal{R}$ is sound if $\Delta \vdash_{\mathcal{R}} C$ implies $\Delta \models C$.*

**Completeness:** *A calculus $\mathcal{R}$ is refutation-complete if $\Delta \models \bot$ implies $\Delta \vdash_{\mathcal{R}} \square$, i.e., if $\Delta$ is unsatisfiable then we can derive the empty clause.*

- Together: $\Delta$ is unsatisfiable iff we can derive the empty clause.
- Propositional resolution is sound & refutation-complete or propositional $\Delta$.

**Reminder: Deduction $\approx$ Proof by Contradiction**

To decide whether KB $\models \varphi$, decide satisfiability of $\psi := \text{KB} \cup \{\neg\varphi\}$: $\psi$ is unsatisfiable iff KB $\models \varphi$.

### Propositional Resolution

**Definition (Propositional Resolution).** *Resolution uses the following inference rule (with exclusive union $\dot\cup$ meaning that the two sets are disjoint):*

$$\frac{C_1 \dot\cup \{l\}, C_2 \dot\cup \{\bar{l}\}}{C_1 \cup C_2}$$

*If $\Delta$ contains parent clauses of the form $C_1 \dot\cup \{l\}$ and $C_2 \dot\cup \{\bar{l}\}$, the rule allows to add the resolvent clause $C_1 \cup C_2$. $l$ and $\bar{l}$ are called the resolution literals.*

**Example**: $\{P, \neg R\}$ resolves with $\{R, Q\}$ to $\{P, Q\}$.

**Lemma.** *The resolvent follows from the parent clauses.*

**Proof.** If $I \models C_1 \dot\cup \{l\}$ and $I \models C_2 \dot\cup \{\bar{l}\}$, then $I$ must make at least one literal in $C_1 \cup C_2$ true.

### Binary FOL Resolution

**Definition (Binary FOL Resolution).** *Binary FOL resolution is the following inference rule:*

$$\frac{C_1 \dot\cup \{P_1\}, C_2 \dot\cup \{\neg P_2\}}{[C_1 \cup C_2]g}$$

*If $\Delta$ contains parent clauses of the form $C_1 \dot\cup \{P_1\}$ and $C_2 \dot\cup \{\neg P_2\}$, where $\{P_1, P_2\}$ can be unified and $g$ is an mgu thereof, the rule allows to add the resolvent clause $[C_1 \cup C_2]g$. $P_1$ and $\neg P_2$ are called the resolution literals.*

**Example:** From $\{Nat(s(x)), \neg Nat(x)\}$ and $\{Nat(1)\}$

**Lemma (Soundness).** *The resolvent follows from the parent clauses.*

### Why Do We Need To Standardize Variables Apart?

**Example:** $\Delta = \{\{Knows(John, x)\}, \{\neg Knows(x, Elizabeth), King(x)\}\}$
→ We should be able to conclude that?

**Unification 1:** $\{P_1, P_2\} = \{Knows(John, x), Knows(x, Elizabeth)\}$
→ Is there a unifier for $\{E_i\}$? No. We would have to substitute two different constants for $x$. (Cf. slide 28)

**Unification 2:** $\{P_1, P_2\} = \{Knows(John, x), Knows(y, Elizabeth)\}$
→ Is there a unifier for $\{E_i\}$? Yes: $\{\frac{x}{Elizabeth}, \frac{y}{John}\}$. (Cf. slide 28)

→ Standardizing the variables in clauses apart is sometimes necessary to allow unification.

(→ An alternative would be to not use unification, and instead substitute atoms separately to the same outcome; we don't consider this here.)

### Where Binary FOL Resolution Fails

**Notation:** Define $\mathcal{R}_{Binary} := \{\text{binary FOL resolution}\}$.

**Theorem.** *The calculus $\mathcal{R}_{Binary}$ is not refutation-complete.*

**Proof.** See example above.

However, $\mathcal{R}_{Binary}$ is sound:

**Theorem.** *The calculus $\mathcal{R}_{Binary}$ is sound.* (Proof: Lemma slide 33)

### Solution 1: Full FOL Resolution

→ Allow to unify *several* resolution literals:

**Definition (Full FOL Resolution).** *Full FOL resolution is the following inference rule:*

$$\frac{C_1 \dot\cup \{P_1^1, \ldots, P_1^n\}, C_2 \dot\cup \{\neg P_2^1, \ldots, \neg P_2^m\}}{[C_1 \cup C_2]g}$$

*where $\{P_1^1, \ldots, P_1^n, P_2^1, \ldots, P_2^m\}$ can be unified and $g$ is an mgu thereof.*

**Example:** $\Delta = \{\{P(x_1, x_2), P(x_2, x_1)\}, \{\neg P(y_1, y_2), \neg P(y_2, y_1)\}\}$
→ Can we derive $\square$ with full FOL resolution?

**Notation:** Define $\mathcal{R}_{Full} := \{\text{full FOL resolution}\}$.

**Theorem.** *The calculus $\mathcal{R}_{Full}$ is sound.*

**Proof.** It suffices to show that, for each application of the rule, the resolvent follows from the parents. That can be shown with the same argument as for binary FOL resolution (Lemma slide 33).

### Solution 2: Binary FOL Resolution + Factoring

→ Allow to unify literals *within* a clause:

**Definition (Factoring).** *Factoring is the following inference rule:*

$$\frac{C_1 \dot\cup \{l_1\} \dot\cup \{l_2\}}{[C_1 \cup \{l_1\}]g}$$

*where $\{l_1, l_2\}$ can be unified and $g$ is an mgu thereof. $[C_1 \cup \{l_1\}]g$ is called a factor of the parent clause $C_1 \dot\cup \{l_1\} \dot\cup \{l_2\}$.*

**Example:** $\Delta = \{\{P(x_1, x_2), P(x_2, x_1)\}, \{\neg P(y_1, y_2), \neg P(y_2, y_1)\}\}$

→ How can we apply factoring?

**Notation:** Define $\mathcal{R}_{FactBin} := \{\text{binary FOL resolution,factoring}\}$.

**Theorem.** *The calculus $\mathcal{R}_{FactBin}$ is sound.*

**Proof.** Due to the universal quantification, the factor follows from its parent. Done with Lemma slide 33.

### What About Completeness? The Lifting Lemma

**Lemma (Lifting Lemma).** *Let $C_1$ and $C_2$ be two clauses with no shared variables, and let $C_1^g$ and $C_2^g$ be ground instances of $C_1$ and $C_2$. Say that $C^g$ is a resolvent of $C_1^g$ and $C_2^g$. Then there exists a clause $C$ such that $C^g$ is a ground instance of $C$, and:*

1. *$C$ can be derived from $C_1$ and $C_2$ using $\mathcal{R}_{Full}$.*
2. *$C$ can be derived from $C_1$ and $C_2$ using $\mathcal{R}_{FactBin}$.*

**Theorem.** *The calculi $\mathcal{R}_{Full}$ and $\mathcal{R}_{FactBin}$ are refutation-complete.*

**Proof:**

Any set $\theta$ of FOL formulas is representable in clausal form $\Delta$.
↓
Assume $\Delta$ is unsatisfiable.
↓ ⟵ Herbrand, prop. compactness
Some finite set $\Delta'$ of ground instances is unsatisfiable.
↓ ⟵ Prop. resolution completeness
Propositional resolution can derive $\square$ from $\Delta'$.
↓ ⟵ Lifting Lemma
Each of $\mathcal{R}_{Full}$ and $\mathcal{R}_{FactBin}$ can derive $\square$ from $\Delta$.

**Summary**

The Herbrand universe is the set of all ground terms that can be built from the symbols used in a set θ of FOL formulas. The (propositional-logic) Herbrand expansion instantiates the formulas with these terms, and is satisfiable iff θ is.

For unsatisfiable θ, we can always find an unsatisfiable finite subset of the Herbrand expansion.

FOL resolution reasons directly about FOL formulas (in clausal normal form) It relies on unification to compare FOL terms.

Binary FOL resolution is like propositional resolution with unification. It is not refutation-complete.

To obtain a complete FOL resolution calculus, we can either allow to unify sets of resolution literals (full FOL resolution), or to unify literals within clauses (factoring).

The set of satisfiable FOL formulas is not recursively enumerable. Thus, neither the reduction to propositional logic, nor FOL resolution, guarantee to terminate in finite time.

# Chapter 13. Introduction to Planning

Ambition:
we would like to write one program that can solve all classical search problem

As we know, the description of a problem can be both:
- blackbox description: API providing functionality to construct the state space
- declarative description: problem description language
  The problem description language is a planning language.

**How does a planning language describe a problem?**
- logical description of all possible states
- logical description of the initial states I
- logical description of the goal condition G
- logical description of the set of actions A in terms of preconditions and effects

Solution, also called plan, is the sequence of actions from A, transforming I into a state that satisfies G.

Planning languages go way beyond classical search problems. There are variants for inaccessible, stochastic, dynamic, continuous, and multi-agent settings. But we will focus on classical search.

Applications: natural language generation, business process templates, automatic hacking

The planning system is the **planning domain definition language (PDDL).**

**General Problem solving is:**
+ powerful
+ quick
+ flexible
+ intelligent
- efficiency loss

How to make fully automatic algorithms effective?
The main issue is that state spaces are huge even for simple problems. Even solving “simple problems” automatically (without help from a human) requires a form of intelligence

**Algorithmic Problems in Planning**

We distinguish between:
- **satisficing planning:**
  where the input is a planning task and the output is a plan for that input, or unsolvable if that path doesn’t exist
- **optimal planning:**
  where the input is a planning task and the output is an *optimal* plan for that input, or unsolvable if that path doesn’t exist

Satisficing planning is much more effective in practice.
Programs solving these problems are called (optimal) planners, planning systems, or planning tools.

**Planning history**
**Newell and Simon (1963)** introduced general problem solving... not much happened (well not much we still speak of today)
**Stanford Research Institute** developed a robot named “Shakey”. They needed a “planning” component for making decisions. They took inspiration from general problem solving and theorem proving, and called the resulting algorithm “STRIPS” (Stanford Research Institute Problem Solver).

Then:

**Compilation into Logics/Theorem Proving:**
Popular when: Stone Age – 1990.
Approach: From planning task description, generate FOL formula $\phi$ that is satisfiable iff there exists a plan; use a theorem prover on $\phi$.

**Partial-Order Planning**:
Popular when: 1990 – 1995.
Approach: Starting at goal, extend partially ordered set of actions by inserting achievers for open sub-goals, or by adding ordering constraints to avoid conflicts.

**GraphPlan:**

Popular when: 1995 – 2000.
Approach: In a forward phase, build a layered “planning graph” whose “time steps” capture which pairs of actions can achieve which pairs of facts; in a backward phase, search this graph starting at goals and excluding options proved to not be feasible.
**Planning as SAT:**

Popular when: 1996 – today.
Approach: From planning task description, generate propositional CNF formula $\phi k$ that is satisfiable iff there exists a plan with k steps; use a SAT solver on $\phi k$, for different values of k.

**Planning as Heuristic Search:**
Popular when: 1999 – today.
Approach: Devise a method R to simplify (“relax”) any planning task $\Pi$; given $\Pi$, solve $R(\Pi)$ to generate a heuristic function h for informed search.
The International Planning Competition (IPC) is the competition.

**“STRIPS” Planning**

STRIPS = Stanford Research Institute Problem Solver. STRIPS is the simplest possible (reasonably expressive) logic-based planning language. STRIPS has only Boolean variables: propositional logic atoms. Its preconditions/effects/goals are as canonical as imaginable: Preconditions, goals: conjunctions of positive atoms. Effects: conjunctions of literals (positive or negated atoms). We use the common special-case notation for this simple formalism. I’ll outline some extensions beyond STRIPS later on, when we discuss PDDL.

**STRIPS Planning: Syntax**
A STRIPS planning task, short planning task, is a 4-tuple $\Pi = (P, A, I, G)$ where:

- P is a finite set of facts
- A is a finite set of actions and each action a is a triple a = ($pre_a$, $add_a$, $del_a$) of subsets of P referred to as the action's precondition, add list, and delete list respectively; We require that the set of add and del are disjoint.
- I (in P) is the initial state
- G (in P) is the goal.

We will often give each action a ∈ A a name (a string), and identify a with that name.

**STRIPS Planning: Semantics**

**Definition (STRIPS State Space).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The state space of $\Pi$ is $\Theta_\Pi = (S, A, T, I, S^G)$ where:*

- *The states (also world states) $S = 2^P$ are the subsets of $P$.*
- *$A$ is $\Pi$'s action set.*
- *The transitions are $T = \{s \xrightarrow{a} s' \mid pre_a \subseteq s, s' = (\llbracket s \rrbracket, a)\}$.*
  *If $pre_a \subseteq s$, then $a$ is applicable in $s$ and $(\llbracket s \rrbracket, a) := (s \cup add_a) \setminus del_a$. If $pre_a \not\subseteq s$, then $(\llbracket s \rrbracket, a)$ is undefined.*
- *$I$ is $\Pi$'s initial state.*
- *The goal states $S^G = \{s \in S \mid G \subseteq s\}$ are those that satisfy $\Pi$'s goal.*

*An (optimal) plan for $s \in S$ is an (optimal) solution for $s$ in $\Theta_\Pi$, i.e., a path from $s$ to some $s' \in S^G$. A solution for $I$ is called a plan for $\Pi$. $\Pi$ is solvable if a plan for $\Pi$ exists.*

*For $\vec{a} = \langle a_1, \ldots, a_n \rangle$, $(\llbracket s \rrbracket, \vec{a}) := (\llbracket \ldots \rrbracket(\llbracket \rrbracket \llbracket (\rrbracket s, a_1), a_2) \ldots, a_n)$ if each $a_i$ is applicable in the respective state; else, $(\llbracket s \rrbracket, \vec{a})$ is undefined.*

**Why Complexity Analysis?**

Two very good reasons:

1) It saves you from spending lots of time trying to invent algorithms that do not exist.
2) Killer app in planning: tractable fragments for heuristic functions.
   → Identify special cases that can be solved in polynomial time.
   → Relax the input into the special case to obtain a heuristic function!

→ I'll next remind you of the basic terms, then I'll illustrate both with an example. Afterwards we'll have a brief look at the complexity of the main decision problems in STRIPS planning.

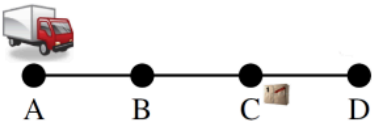
**NP and PSPACE**

**Def Turing machine:**
Works on a tape consisting of tape cells, across which its R/W head moves. The machine has internal states. There are transition rules specifying, given the current cell content and internal state, what the subsequent internal state will be, how what the R/W head does (write a symbol and/or move). Some internal states are accepting.
**Def NP:**
Decision problems for which there exists a non-deterministic Turing machine that runs in time polynomial in the size of its input. Accepts if at least one of the possible runs accepts.
**Def PSPACE**:
Decision problems for which there exists a deterministic Turing machine that runs in space polynomial in the size of its input.

**Relation**: Non-deterministic polynomial space can be simulated in deterministic polynomial space. Thus PSPACE = NPSPACE, and hence (trivially) NP ⊆ PSPACE. It is commonly believed that NP 6⊇ PSPACE (similar to P ⊆ NP).

**The "Only-Adds" Relaxation**

**Example:** "Logistics"




- Facts $P$: $\{truck(x) \mid x \in \{A, B, C, D\}\} \cup \{pack(x) \mid x \in \{A, B, C, D, T\}\}$.
- Initial state $I$: $\{truck(A),\ pack(C)\}$.
- Goal $G$: $\{truck(A),\ pack(D)\}$.
- Actions $A$: (Notated as "precondition ⇒ adds, ¬ deletes")
  - $drive(x, y)$, where $x, y$ have a road: "$truck(x) \Rightarrow truck(y), \neg truck(x)$".
  - $load(x)$: "$truck(x), pack(x) \Rightarrow pack(T), \neg pack(x)$".
  - $unload(x)$: "$truck(x), pack(T) \Rightarrow pack(x), \neg pack(T)$".

**Only-Adds Relaxation:** Drop the preconditions and deletes.

"$drive(x, y)$: $\Rightarrow truck(y)$"; "$load(x)$: $\Rightarrow pack(T)$"; "$unload(x)$: $\Rightarrow pack(x)$".

**Solving Only-Adds STRIPS Tasks**

Our problem: Given STRIPS task Π = (P, A, I, G). Find action sequence a leading from I to a state that contains G, when pretending that preconditions and deletes are empty.

$\vec{a} := \langle\rangle$
**while** $G \neq \emptyset$ **do**
  select $a \in A$
  $G := G \setminus add_a$
  $\vec{a} := \vec{a} \circ \langle a \rangle$; $A := A \setminus \{a\}$
**endwhile**
**return** $h := |\vec{a}|$

**Solution 1:**

- simplest possible approach

Is this h admissible? No. Admissibility is only guaranteed if we find a shortest possible $\vec{a}$; else, $\vec{a}$ might be longer than a plan for Π itself. Selecting an arbitrary action each time, $\vec{a}$ may be longer than needed.

$\vec{a} := \langle\rangle$
**while** $G \neq \emptyset$ **do**
  select $a \in A$ s.t. $|add_a|$ is maximal
  $G := G \setminus add_a$
  $\vec{a} := \vec{a} \circ \langle a \rangle$; $A := A \setminus \{a\}$
**endwhile**
**return** $h := |\vec{a}|$

**Solution 2:**

→ h admissible? No, large adda doesn't help if the intersection with G is small.

$\vec{a} := \langle\rangle$
**while** $G \neq \emptyset$ **do**
  select $a \in A$ s.t. $|add_a \cap G|$ is maximal
  $G := G \setminus add_a$
  $\vec{a} := \vec{a} \circ \langle a \rangle$; $A := A \setminus \{a\}$
**endwhile**
**return** $h := |\vec{a}|$

**Solution 3**

→ h admissible? Still no. Example: G = {A, B, C, D, E, F}; adda1 = {A, B}; adda2 = {C, D}; adda3 = {E, F}; adda4 = {A, C, E}.

## So what?

- Given STRIPS task $\Pi = (P, A, I, G)$.
- Find optimal $\vec{a}$ leading from $I$ to a state that contains $G$, when pretending that preconditions and deletes are empty.

→ $\vec{a}$ leads to $G \Leftrightarrow \bigcup_{a \in \vec{a}} add_a \supseteq G \Leftrightarrow$ the add lists in $\vec{a}$ cover $G$. QED.

**Decision Problems in (STRIPS) Planning**

**Definition (PlanEx).** By PlanEx, we denote the problem of deciding, given a STRIPS planning task Π, whether or not there exists a plan for Π. → Corresponds to satisficing planning.

**Definition (PlanLen).** By PlanLen, we denote the problem of deciding, given a STRIPS planning task Π and an integer B, whether or not there exists a plan for Π of length at most B. → Corresponds to optimal planning.

**Definition (PolyPlanLen).** By PolyPlanLen, we denote the problem of deciding, given a STRIPS planning task Π and an integer B bounded by a polynomial in the size of Π, whether or not there exists a plan for Π of length at most B. → Corresponds to optimal planning with “small” plans. Example of a planning domain with exponentially long plans? Towers of Hanoi.

**Lemma. PlanEx is PSPACE-hard.**

**Proof Sketch.** Given a Turing machine with space bounded by polynomial $p(|w|)$, we can in polynomial time (in the size of the machine) generate an equivalent STRIPS planning task. Say the possible symbols in tape cells are $x_1, \ldots, x_m$ and the internal states are $s_1, \ldots, s_n$, accepting state $s_{acc}$.

- The contents of the tape cells: $in(1, x_1), \ldots, in(p(|w|), x_1), \ldots, in(1, x_m), \ldots, in(p(|w|), x_m)$.
- The position of the R/W head: $at(1), \ldots, at(p(|w|))$.
- The internal state of the machine: $state(s_1), \ldots, state(s_n)$.
- Transitions rules $\mapsto$ STRIPS actions; accepting state $\mapsto$ STRIPS goal $\{state(s_{acc})\}$; initial state obvious.
- This reduction to STRIPS runs in polynomial-time because we need only polynomially many facts.

**Lemma. PlanEx is a member of PSPACE.**

**Proof.** Because **PSPACE** = **NPSPACE**, it suffices to show that PlanEx is a member of **NPSPACE**:

1. $s := I$; $l := 0$;
2. Guess an applicable action $a$, compute the outcome state $s'$, set $l := l + 1$;
3. If $s'$ contains the goal then succeed;
4. If $l \geq 2^{|P|}$ then fail else goto 2;

$\rightarrow$ Remembering the actual action *sequence* would take exponential space in case of exponentially long plans (cf. slide 39). But, to decide PlanEx, we only need to remember its length.

**Theorem (Complexity of PlanEx). PlanEx is PSPACE-complete.** *(Immediate from previous two lemmas)*

**Complexity of PlanLen**

PlanLen isn't any easier than PlanEx: Corollary. PlanLen is PSPACE-complete. Proof. Membership: Same as before but failing at l ≥ B. Hardness? Setting B := 2|P| , PlanLen answers PlanEx: If a plan exists, then there exists a plan that traverses each possible state at most once. PolyPlanLen is easier than PlanEx: Theorem. PolyPlanLen is NP-complete. Proof. Membership? Guess B actions and check whether they form a plan. This runs in polynomial time because B is polynomially bounded. Hardness: E.g., by reduction from SAT.

Bounding plan length does not help in the general case as we can set the bound to a trivial (exponential) upper bound on plan length. If we restrict plan length to be "short" (polynomial), planning becomes easier.

**Domain-Specific PlanEx vs. PlanLen**

In general, both have the same complexity.

Within particular applications, bounded length plan existence is often harder than plan existence.

This happens in many IPC benchmark domains: PlanLen is NP-complete while PlanEx is in P.

**Summary**
General problem solving attempts to develop solvers that perform well across a large class of problems. Planning, as considered here, is a form of general problem solving dedicated to the class of classical search problems. (Actually, we also address inaccessible, stochastic, dynamic, continuous, and multi-agent settings.) Heuristic search planning has dominated the International Planning Competition (IPC). We focus on it here. STRIPS is the simplest possible, while reasonably expressive, language for our purposes. It uses Boolean variables (facts), and defines actions in terms of precondition, add list, and delete list. PDDL is the de-facto standard language for describing planning problems. Plan existence (bounded or not) is PSPACE-complete to decide for STRIPS. If we bound plans polynomially, we get down to NP-completeness.

# Chapter 14. Planning Formalisms

**Transition Systems**
State space of planning task = a transition system.

**Definition (Transition System).** *A transition system is a 6-tuple* $\Theta = (S, L, c, T, I, S^G)$ *where:*

- $S$ *is a finite set of states.*
- $L$ *is a finite set of transition labels.*
- $c : L \mapsto \mathbb{R}_0^+$ *is the cost function.*
- $T \subseteq S \times L \times S$ *is the transition relation.*
- $I \in S$ *is the initial state.*
- $S^G \subseteq S$ *is the set of goal states.*

*The size of* $\Theta$ *is its number of states,* $size(\Theta) := |S|$.
*We say that* $\Theta$ *has the transition* $(s, l, s')$ *if* $(s, l, s') \in T$. *We also write this* $s \xrightarrow{l} s'$, *or* $s \to s'$ *when not interested in* $l$.
*We say that* $\Theta$ *is deterministic if, for all states* $s$ *and labels* $l$, *there is at most one state* $s'$ *with* $s \xrightarrow{l} s'$.
*We say that* $\Theta$ *has unit costs if, for all* $l \in L$, $c(l) = 1$.

**Terminology:** $\Theta = (S, A, c, T, I, S^G)$; $s, s', s_i \in S$

- $s'$ successor of $s$ if $s \to s'$; $s$ predecessor of $s'$ if $s \to s'$.
- $s'$ reachable from $s$ if there exists a sequence of transitions:

$$s = s_0 \xrightarrow{l_1} s_1, \ldots, s_{n-1} \xrightarrow{l_n} s_n = s'$$

  - $n = 0$ possible; then $s = s'$.
  - $l_1, \ldots, l_n$ is called path from $s$ to $s'$.
  - $s_0, \ldots, s_n$ is also called path from $s$ to $s'$.
  - The cost of that path is $\sum_{i=1}^{n} c(l_i)$.
- $s'$ reachable (without reference state) means reachable from $I$.
- Solution for $s$: path from $s$ to some $s' \in S^G$; optimal if cost is minimal among all solutions for $s$.
- $s$ is solvable if it has a solution; else, $s$ is a dead end.
- Solution for $I$ is called solution for $\Theta$; $\Theta$ is solvable if it has a solution.

## STRIPS Planning: Syntax

**Definition (STRIPS Planning Task).** *A STRIPS planning task is a 5-tuple* $\Pi = (P, A, c, I, G)$ *where:*

- $P$ *is a finite set of facts, also propositions.*
- $A$ *is a finite set of actions; each* $a \in A$ *is a triple* $a = (pre_a, add_a, del_a)$ *of subsets of* $P$ *referred to as the action's precondition, add list, and delete list respectively; we require that* $add_a \cap del_a = \emptyset$.
- $c : A \mapsto \mathbb{R}_0^+$ *is the cost function.*
- $I \subseteq P$ *is the initial state.*
- $G \subseteq P$ *is the goal.*

*We say that* $\Pi$ *has unit costs if, for all* $a \in A$, $c(a) = 1$. *We will often give each action* $a \in A$ *a name (a string), and identify* $a$ *with that name.*

→ Why do we allow 0-cost actions? Negligible cost (e.g. switch light on, take photo with smartphone), asking questions about only one kind of actions (e.g. Mars rover *take-picture* only).

## STRIPS Planning: Semantics

**Definition (STRIPS State Space).** *Let* $\Pi = (P, A, c, I, G)$ *be a STRIPS planning task. The state space of* $\Pi$ *is the labeled transition system* $\Theta_\Pi = (S, L, c, T, I, S^G)$ *where:*

- *The states (also world states)* $S = 2^P$ *are the subsets of* $P$.
- *The labels* $L = A$ *are* $\Pi$*'s actions; the cost function* $c$ *is that of* $\Pi$.
- *The transitions are* $T = \{s \xrightarrow{a} s' \mid a \in A[s], s' = s[\![a]\!]\}$, *where* $A[s] := \{a \in A \mid pre_a \subseteq s\}$ *are the actions applicable in* $s$; *for* $a \in A[s]$, $s[\![a]\!] := (s \setminus del_a) \cup add_a$; *for* $a \notin A[s]$, $s[\![a]\!]$ *is undefined,* $s[\![a]\!] := \bot$.
- *The initial state* $I$ *is identical to that of* $\Pi$.
- *The goal states* $S^G = \{s \in S \mid G \subseteq s\}$ *are those that satisfy* $\Pi$*'s goal.*

*An (optimal) plan for* $s \in S$ *is an (optimal) solution for* $s$ *in* $\Theta_\Pi$. *A solution for* $I$ *is called a plan for* $\Pi$. $\Pi$ *is solvable if a plan for* $\Pi$ *exists.*

*For* $\vec{a} = \langle a_1, \ldots, a_n \rangle$, $s[\![\vec{a}]\!] := \begin{cases} s & n = 0 \\ s[\![\langle a_1, \ldots, a_{n-1} \rangle]\!][\![a_n]\!] & n > 0 \end{cases}$

→ Is $\Theta_\Pi$ deterministic? Yes: the successor state $s'$ in $s \xrightarrow{a} s'$ is uniquely determined by $s$ and $a$, through $s' = s[\![a]\!]$.

## FDR Planning: Syntax

**Definition (FDR Planning Task).** *A finite-domain representation planning task, short FDR planning task, is a 5-tuple $\Pi = (V, A, c, I, G)$ where:*

- *$V$ is a finite set of state variables, each $v \in V$ with a finite domain $D_v$. We refer to (partial) functions on $V$, mapping each $v \in V$ into a member of $D_v$, as (partial) variable assignments.*
- *$A$ is a finite set of actions; each $a \in A$ is a pair $(pre_a, eff_a)$ of partial variable assignments referred to as the action's precondition and effects.*
- *$c : A \mapsto \mathbb{R}_0^+$ is the cost function.*
- *$I$ is a complete variable assignment called the initial state.*
- *$G$ is a partial variable assignment called the goal.*

*We say that $\Pi$ has unit costs if, for all $a \in A$, $c(a) = 1$.*

→ In FDR, a (partial) variable assignment represents a state in $I$, a condition in $pre_a$ and $G$, and an effect instruction in $eff_a$.

**Notation:** Pairs $(v, d)$ are facts, also written $v = d$. We identify partial variable assignments $p$ with fact sets. We write $V[p] := \{v \in V \mid p(v) \text{ is defined}\}$.

## FDR Planning: Semantics

**Definition (FDR State Space).** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task. The state space of $\Pi$ is the labeled transition system $\Theta_\Pi = (S, L, c, T, I, S^G)$ where:*

- *The states (also world states) $S$ are the complete variable assignments.*
- *The labels $L = A$ are $\Pi$'s actions; the cost function $c$ is that of $\Pi$.*
- *The transitions are $T = \{s \xrightarrow{a} s' \mid a \in A[s], s' = s[\![a]\!]\}$, where $A[s] := \{a \in A \mid pre_a \subseteq s\}$ are the actions applicable in $s$; for $a \notin A[s]$, $s[\![a]\!] := \bot$; for $a \in A[s]$, $s[\![a]\!](v) := \begin{cases} eff_a(v) & v \in V[eff_a] \\ s(v) & v \notin V[eff_a] \end{cases}$*
- *The initial state $I$ is identical to that of $\Pi$.*
- *The goal states $S^G = \{s \in S \mid G \subseteq s\}$ are those that satisfy $\Pi$'s goal.*

→ In $s[\![a]\!]$, instead of "adding/deleting" facts, we overwrite the previous variable values by $eff_a$.

→ Plan, optimal plan, $s[\![\vec{a}]\!]$ for action sequence $\vec{a}$: as before (slide 20).

## STRIPS vs. FDR in Practice

How do people use FDR?

Our surface language is PDDL, which corresponds to STRIPS. Most implemented planning tools are based on Fast Downward (FD) [Helmert (2009)], which reads PDDL input, then internally uses a "clever" STRIPS-2-FDR translation. That translation involves a PSPACE-complete sub-problem.

Why??? Practical Efficiency! Regression: FDR avoids myriads of unreachable state.

Causal Graphs: Capture variable dependencies; have a much clearer structure for clever FDR (e.g., acyclic vs. cyclic).

Complexity Analysis: Better with clearer causal graph.

Construction of Heuristic Functions: Better with multiple-valued variables and clearer causal graph.
Modeling: Anyway, FDR is more natural! (It's just one truck, after all.) Why does anybody use STRIPS? It's a legacy system.

**STRIPS vs. FDR Conversions**

1) **FDR-2-STRIPS:** For each variable v with domain {d1, . . . , dk}, make k STRIPS facts "v = d1", …, "v = dk".
2) **STRIPS-2-FDR:** Naive vs. clever variants

**Isomorphism**

**Definition (Isomorphism).** *Let $\Theta = (S, L, c, T, I, S^G)$ and $\Theta' = (S', L', c', T', I', S'^G)$ be transition systems. We say that $\Theta$ is isomorphic to $\Theta'$, written $\Theta \sim \Theta'$, if there exist bijective functions $\varphi : S \mapsto S'$ and $\psi : L \mapsto L'$ such that:*

(i) $\varphi(I) = I'$.
(ii) $s \in S^G$ *iff* $\varphi(s) \in S'^G$.
(iii) $(s, l, t) \in T$ *iff* $(\varphi(s), \psi(l), \varphi(t)) \in T'$.
(iv) *For all* $l \in L$, $c(l) = c'(\psi(l))$.

Isomorphic transition systems are identical modulo renaming states and actions.
Isomorphisms typically result from compilations between different formalisms; we will also sometimes use them as a technical device.

**FDR-2-STRIPS: Details**

**Definition (FDR-2-STRIPS).** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task. The STRIPS conversion of $\Pi$ is the STRIPS task $\Pi^{\text{STR}} = (P_V, A^{\text{STR}}, c, I, G)$ where:*

- $P_V = \{v = d \mid v \in V, d \in D_v\}$ *is the set of (STRIPS) facts.*
- $A^{\text{STR}} = \{a^{\text{STR}} \mid a \in A\}$ *where* $pre_{a^{\text{STR}}} = pre_a$, $add_{a^{\text{STR}}} = eff_a$, *and*
  $del_{a^{\text{STR}}} = \bigcup_{(v=d) \in eff_a} \begin{cases} \{v = pre_a(v)\} & \text{if } pre_a(v) \text{ is defined} \\ \{v = d' \mid d' \in D_v \setminus \{d\}\} & \text{otherwise} \end{cases}$
- *The cost function $c$ is defined by $c(a^{\text{STR}}) := c(a)$ for all $a^{\text{STR}} \in A^{\text{STR}}$.*
- *$I$ and $G$ are identical to those of $\Pi$.*

→ The adds establish the new variable values of $eff_a$; the deletes make sure to erase the previous values of those variables.

→ Take-home message: FDR variable/value pairs ≈ STRIPS facts!

**Proposition.** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task, and let $\Pi^{\text{STR}}$ be its STRIPS conversion. Then $\Theta_\Pi$ is isomorphic to the sub-system of $\Theta_{\Pi^{\text{STR}}}$ induced by those $s \subseteq P_V$ where, for each $v \in V$, $s$ contains exactly one fact of the form $v = d$. All other states in $\Theta_{\Pi^{\text{STR}}}$ are unreachable.*

### STRIPS-2-FDR: Naive Translation

**Definition (STRIPS-2-FDR).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The FDR conversion of $\Pi$ is the FDR task $\Pi^{\mathsf{FDR}} = (V_P, A^{\mathsf{FDR}}, c, I^{\mathsf{FDR}}, G^{\mathsf{FDR}})$ where:*

- $V_P = \{v_p \mid p \in P\}$ *is the set of variables, all Boolean.*
- $A^{\mathsf{FDR}} = \{a^{\mathsf{FDR}} \mid a \in A\}$ *where* $pre_{a^{\mathsf{FDR}}} = \{v_p = T \mid p \in pre_a\}$ *and* $eff_{a^{\mathsf{FDR}}} = \{v_p = T \mid p \in add_a\} \cup \{v_p = F \mid p \in del_a\}$.
- *The cost function $c$ is defined by $c(a^{\mathsf{FDR}}) := c(a)$ for all $a^{\mathsf{FDR}} \in A^{\mathsf{STR}}$.*
- $I = \{v_p = T \mid p \in I\}$; *and* $G = \{v_p = T \mid p \in G\}$.

→ All variables here have two possible values only, so this does not benefit at all from the added expressivity of FDR. Hence the designation "naïve".

**Proposition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $\Pi^{\mathsf{FDR}}$ be its STRIPS conversion. Then $\Theta_\Pi$ is isomorphic to $\Theta_{\Pi^{\mathsf{STR}}}$.*

### STRIPS-2-FDR: Clever Translation

**How to be clever?**

- Find sets $\{p_1, \ldots, p_k\}$ of STRIPS facts so that every reachable state $s$ makes exactly one $p_i$ true.

  → Deciding whether this holds, for a given $\{p_1, \ldots, p_k\}$, is **PSPACE**-complete (cf. slide 31). But one can design fast algorithms finding *some* such sets [Helmert (2009)].
- For each set $\{p_1, \ldots, p_k\}$ found, make *one* FDR variable $v$ with domain $\{d_1, \ldots, d_k\}$.
- This is implemented in the pre-processor of Fast Downward.

### Action Description Language (ADL)

Framework Definition: [Pednault (1989); Hoffmann and Nebel (2001)]. Problem: Like STRIPS but with first-order logic (FOL) formulas in prea and G, and conditional effects that execute only if their individual effect condition holds. Plan: Sequence of actions. (Yes, this is still "classical planning".) Example: If your action a opens the doors of an elevator, then each passenger gets out iff their individual condition ("Is this my destination floor?") holds. If you want to satisfy complex constraints ("Group A should never meet group B in the elevator") then prea gets nasty. (See the file miconic-ADL on Moodle.) Compilation: FOL formulas: Ground them (the universe is finite) and transform to DNF [Gazen and Knoblock (1997); Koehler and Hoffmann (2000)]. Conditional effects: Either enumerate all combinations of effects, or introduce artificial facts/actions enforcing an "effect evaluation phase" [Nebel (2000)]. State of the art: Get rid of FOL formulas but keep the conditional effects.

### Numeric and Temporal Planning

**Numeric Planning:** [Fox and Long (2003)]
$pre_a : fuelSupply \geq distance(x, y) * fuelConsumption$
$eff_a : fuelSupply := fuelSupply - distance(x, y) * fuelConsumption$

**Compilation:** Nothing known.

**Temporal Planning:** [Fox and Long (2003)]
$duration_a : distance(x, y)/speed$
$eff_a$ : *at Start* $\neg at(x)$, *at End* $at(y)$.

### Soft Goals and Trajectory Constraints

**Soft Goals:** [Gerevini *et al.* (2009)]
"I don't absolutely have to visit Darwin, but if I do, I get a certain amount $R$ of reward."

**Compilation:** Artificial actions that allow to forgo each weak goal, at cost $R$; minimize cost [Keyder and Geffner (2009)]. State of the art!

**Trajectory Constraints:** [Gerevini *et al.* (2009)]
"I must visit Perth before I visit Darwin."

**Compilation:** Artificial preconditions/effects, e.g. $visited(Perth)$ into precondition of driving to Darwin [Edelkamp (2006)]. State of the art!

### Conformant Planning

Framework Definition: [Smith and Weld (1998); Bonet and Givan (2006)]. Problem: There are many possible initial states (represented as a formula), and each action may have several possible effects. We have no observability during plan execution. Plan: Sequence of actions that achieves the goal regardless which initial state and action effects occur. Example: You're in a dark cave but don't know where exactly. The plan is to walk to the right until you reach a wall and can locate yourself. Then navigate to your goal by counting your steps. Compilation: Artificial "what-if" facts, like "If I was at A initially, then I am now at B" [Palacios and Geffner (2009)]. State of the art!

### Contingent Planning

Framework Definition: e.g., [Hoffmann and Brafman (2005)]. Problem: There are many possible initial states (represented as a formula), and each action may have several possible effects. We have partial observability during plan execution. Plan: Tree of actions that achieves the goal in each of its leaves. ("Plan ahead for all possible contingencies, i.e., situation aspects not known at planning time.") Example: Solving the Wumpus world: You walk some steps, then use sensing (for breeze and stench), and continue depending on the outcome. Compilation: Sample initial states, classical planning with artificial facts encoding knowledge yields a plan tree for those; in case a problem is detected during execution, re-plan with the new state of knowledge [Shani and Brafman (2011)]. Competitive with state of the art!

**Probabilistic Planning**
Framework Definition: e.g., [Younes et al. (2005)]. Problem: Each action specifies a probability distribution over its possible effects. We have full observability during plan execution. (Markov Decision Process (MDP) framework.) Plan: Policy that maps states to actions in a way that maximizes the expected reward. Example: Controlling a robot: If navigation comes with an imprecision (which it usually does), then the outcome of a "move" operation is uncertain. Compilation: Make classical problem that acts as if you could choose the outcomes; find a plan, and execute; if the plan fails, then re-plan from the current state [Yoon et al. (2007)]. State of the art for problems where "reactive behavior" is suitable (things may go wrong, but if they do, they can be easily repaired).

**Summary**

Transition systems are a kind of directed graph (typically huge) that encode how the state of the world can change.
Planning tasks are compact representations for transition systems, based on state variables; they are the input for planning systems.
In satisficing planning, we must find a solution to planning tasks (or show that no solution exists). In optimal planning, we must additionally guarantee that generated solutions are the cheapest possible.
Classical planning makes strong simplifying assumptions, but is very successful in practice and can be used by compilation to tackle more expressive planning problems.
In STRIPS, state variables are Boolean; in FDR, they may have arbitrary finite domains. The two formalisms can be compiled into each other. FDR is preferrable, but current planning technology is based on STRIPS for historical reasons.

# Chapter 15. PDDL

**What is PDDL?**
Once you decided for STRIPS/FDR/whatever, you still need to design an input syntax that your computer can read. That input syntax in the planning area is PDDL: The Planning Domain Definition Language. In particular, PDDL is used in the International Planning Competitons (IPC).
**Why PDDL?** It's just a fact of life: → PDDL is the de-facto standard input language in the planning area. → To complete this course you must know this language. (Initially, every group used their own input language = needing an interpreter every time you talk to your neighbor.)

**Schematic Encodings**

- Predicates instead of STRIPS propositions. Arity: number of vars.
- Action schemas instead of STRIPS actions. Arity: number of vars.
- Analogy: propositional logic vs. predicate logic (FOL).
- Set of objects in PDDL is finite!

Like FOL, PDDL describes the world in a schematic way relative to a set of objects. This makes the encoding much smaller and easier to write.
Most planners translate the schematic input into (propositional) STRIPS in a pre-process, by instantiating the variables in all possible ways. This is called grounding.

**Schematic Actions: Quantification**

**Quantification in Formulas**

Finite disjunctions $\varphi(o_1) \vee \cdots \vee \varphi(o_n)$ represented as $\exists x \in \{o_1, \ldots, o_n\} : \varphi(x)$.
Finite conjunctions $\varphi(o_1) \wedge \cdots \wedge \varphi(o_n)$ represented as $\forall x \in \{o_1, \ldots, o_n\} : \varphi(x)$.

**Quantification over Effects**

Finite list of conditional effects WHEN $\varphi(o_i)$ DO $\psi(o_i)$ represented as $\forall x \in \{o_1, \ldots, o_n\}$ : WHEN $\varphi(o_i)$ DO $\psi(o_i)$.

If an action schema has k parameters, and there are n objects each of these parameters can be instantiated with, then there are n k grounded actions. Same for predicates. Grounding is exponential in operator and predicate arity.
In practice, this is often ok, many domains have maximum arity 2 or 3.
However, this is NOT always so! (E.g., natural language generation)
Grounding typically leads to more efficient planning in the cases where it is feasible; in the other cases, lifted planning is needed.
There has been little research on lifted planning in the last 2 decades.

**PDDL Basics**

Variants used by almost all implemented planning systems. Supports a formalism comparable to what we have outlined above (including schematic operators and quantification). Syntax inspired by the Lisp programming language:
e.g., prefix notation for formulas (and (or (on A B) (on A C)) (or (on B A) (on B C)) (or (on C A) (on A B)))
The planner input is separated into a domain file (predicates, types, action schemas) and a problem file (objects, initial state, goal).

**PDDL Domain Files**

1. (define (domain )
2. A requirements definition (use ":adl :typing" by default).
3. Definitions of types (each object variable has a type).
4. Definitions of predicates.
5. Definitions of action schemas.

**Domain File Types and Predicates: Example Blocksworld**

```
(define (domain Blocksworld)
  (:requirements :adl :typing)
  (:types block - object
          blueblock smallblock - block)
  (:predicates (on ?x - smallblock ?y - block)
               (ontable ?x - block)
               (clear ?x - block))
```

### Action Schema: Example Blocksworld

```
(:action fromtable
    :parameters (?x - smallblock ?y - block)
    :precondition (and (not (= ?x ?y))
                       (clear ?x)
                       (ontable ?x)
                       (clear ?y))
    :effect
      (and (not (ontable ?x))
           (not (clear ?y))
           (on ?x ?y)))
```

### PDDL Grammar: Action Schema

```
(:action <name>
```

List of parameters:

```
(?x - type1 ?y - type2 ?z - type3)
```

The precondition is a formula:

```
  <predicate>
  (and <formula> ... <formula>)
  (or <formula> ... <formula>)
  (not <formula>)
  (forall (?x1 - type1 ... ?xn - typen) <formula>)
  (exists (?x1 - type1 ... ?xn - typen) <formula>)
```

The effect is a combination of literals, conjunction, conditional effects, and quantification over effects:

```
<predicate>
(not <predicate>)
(and <effect> ... <effect>)
(when <formula> <effect>)
(forall (?x1 - type1 ... ?xn - typen) <effect>)
```

### PDDL Problem Files

1. `(define (problem <name>)`
2. `(:domain <name>)`
   – to which domain does this problem belong?
3. Definitions of objects belonging to each type.
4. Definition of the initial state (list of ground predicates initially true).
5. Definition of the goal (a formula like action preconditions).

**Problem File: Example Blocksworld**

```
(define (problem example)
  (:domain Blocksworld)
  (:objects a b c - smallblock)
            d e - block
            f - blueblock)
  (:init (clear a) (clear b) (clear c)
         (clear d) (clear e) (clear f)
         (ontable a) (ontable b) (ontable c)
         (ontable d) (ontable e) (ontable f))

  (:goal (and (on a d) (on b e) (on c f)))
)
```

**(History)**

**Summary:**

PDDL is the de-facto standard for classical planning, as well as extensions to numeric/temporal planning, soft goals, trajectory constraints.
PDDL is used in the International Planning Competition (IPC).
PDDL uses a schematic encoding, with variables ranging over objects similarly as in predicate logic. Most implemented systems use grounding to transform this into a propositional encoding. PDDL has a Lisp-like syntax.

# Chapter 16. Causal Graphs

Causal graphs capture the structure of the planning task input, in terms of the direct dependencies between state variables. This can be exploited for many different purposes.

**Example Uses of Causal Graphs**

- Identifying a subclass of planning tasks where generating a partial delete of relaxation heuristic is tractable
- Avoiding redundant work in the search for a pattern collection when generating a pattern database heuristic
- Search space surface analysis
- Complexity analysis
- Designing and generating (yet more) heuristic functions
- System design
- Factorized planning

**Definition (Causal Graph).** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task. The causal graph of $\Pi$ is the directed graph $CG(\Pi)$ with vertices $V$ and an arc $(u, v)$ if $u \neq v$ and there exists an action $a \in A$ so that either*

1. *$pre_a(u)$ and $eff_a(v)$ are both defined; or*
2. *$eff_a(u)$ and $eff_a(v)$ are both defined.*

**Causal graphs capture variable dependencies:**

- Arc (1) $(u, v)$: we may have to change $u$ to be able to change $v$.
- Arc (2) $(u, v)$: changing $u$ may, as a side effect, change $v$ as well.
  $\rightarrow$ Note that we also get the arc $(v, u)$ in this situation, constituting a cycle between $u$ and $v$.

Class (1) (precondition-effect) causal graph cycles occur when there are “cyclic support dependencies”, where moving variable x requires a precondition on y which (transitively) requires a precondition on x.

Class (2) (effect-effect) causal graph cycles occur whenever an action has more than one effect. Their absence is equivalent to “unary” actions, each affecting only a single variable.

**Where Causal Graphs Fail**

Does CG(Π) depend on either of I or G? No, CG(Π) remains the same whichever I and G we choose. → This is a main weakness of causal graphs! They capture only the structure of the variables and actions, and can by design not account for the influence of different initial states and goals.

**Domain Transition Graphs**

**Definition (Domain Transition Graph).** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task, and let $v \in V$. The domain transition graph (DTG) of $v$ is the labeled directed graph $DTG(v, \Pi)$ with vertices $D_v$ and an arc $(d, d')$ labeled with action $a \in A$ whenever either (i) $pre_a(v) = d$ and $eff_a(v) = d'$, or (ii) $pre_a(v)$ is not defined and $eff_a(v) = d'$.*
*We refer to $(d, d')$ as a value transition of $v$. We write $d \xrightarrow{a}_{\varphi} d'$ where $\varphi = pre_a \setminus \{v = d\}$ is the (outside) condition. Where not relevant, we omit “$a$” and/or “$\varphi$”.*

$\rightarrow$ DTG captures “where a variable can go and how”.

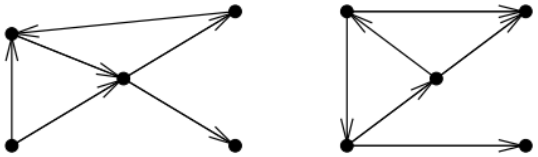
$\rightarrow$ Attention: “value transition $d \xrightarrow{a}_{\varphi} d'$” $\neq$ “state transition $s \rightarrow s'$”. (Value transition focuses on $v$, state transition encompasses all variables.)

**Definition (Invertible Value Transition).** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task, let $v \in V$, and let $d \rightarrow_{\varphi} d'$ be a value transition of $v$. We say that $d \rightarrow_{\varphi} d'$ is invertible if there exists a value transition $d' \rightarrow_{\varphi'} d$ where $\varphi' \subseteq \varphi$.*

$\rightarrow$ DTG captures whether “we can go back”.

DTGs capture the “travel routes” of individual variables. For domain size 2, these routes hardly capture any interesting structure.

**Task Decomposition: Unconnected Sub-Tasks**



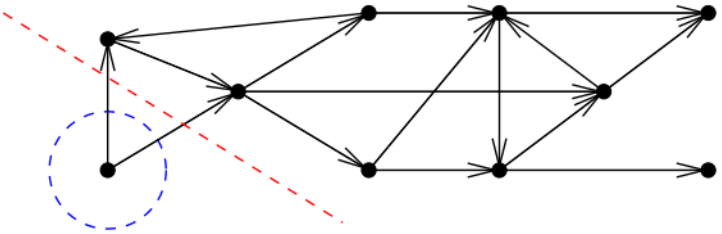
→ Unconnected parts of the task can be solved separately:

**Lemma.** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task, and let $V_1, V_2$ be a partition of $V$ such that $CG(\Pi)$ contains no arc between the two sets. Let $\Pi_i$, for $i \in \{1, 2\}$, be identical to $\Pi$ except that we use variables $V_i$, restrict $I$ and $G$ to $V_i$, and remove all actions $a$ where either $pre_a$ or $eff_a$ is defined on a variable outside $V_i$. Then, for any pair $\vec{a}_1$ and $\vec{a}_2$ of (optimal) plans for $\Pi_1$ and $\Pi_2$, $\vec{a}_1 \circ \vec{a}_2$ is an (optimal) plan for $\Pi$.*

**Proof Intuition:** Since $CG(\Pi)$ contains no arc between $V_1$ and $V_2$, every action either touches only variables from $V_1$, or touches only variables from $V_2$.

Hence any plan for $\Pi$ can be separated into independent sub-sequences touching $V_1$ respectively $V_2$, corresponding to plans for $\Pi_1$ respectively $\Pi_2$.

**Task Simplification: Invertible Root Variables**



→ Root variables with invertible & connected DTGs can be handled separately:

1. Remove $v$ from $\Pi$ to obtain $\Pi'$; find plan $\vec{a}$ for $\Pi'$.
2. Extend $\vec{a}$ with move sequence for $v$ that achieves all preconditions on $v$ as needed, then moves to $v's$ own goal (if any) at the end.

→ Intuition: $v$ is a “servant”. Thanks to its connected and invertible DTG, it can always go wherever it is needed.

→ Does this hold for optimal planning? No. The optimal plan for $\Pi'$ ignores the cost of moving $v$ so may incur unnecessarily high costs on $v$.

**Lemma.** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task, and let $v \in V$ be a root vertex in $CG(\Pi)$ such that $DTG(v, \Pi)$ is connected and all value transitions of $v$ are invertible. Let $\Pi'$ be identical to $\Pi$ except that we remove $v$, restrict $I$ and $G$ to $V \setminus \{v\}$, remove any assignment to $v$ from all action preconditions, and remove all actions $a$ where $eff_a(v)$ is defined. Then, from any plan $\vec{a}$ for $\Pi'$, a plan for $\Pi$ can be obtained in time polynomial in $|\Pi|$ and $|\vec{a}|$.*

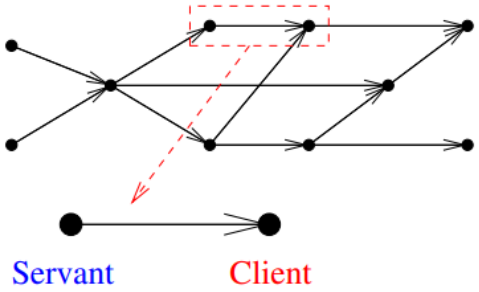
**Proof Intuition:** Since $v$ is a root in $CG(\Pi)$, the actions that affect $v$ affect no other variables, and have no preconditions on any variables other than $v$. In other words, "moving $v$ has no side effects, and does not need any outside help".

Since $DTG(v, \Pi)$ is connected and all value transitions of $v$ are invertible, $DTG(v, \Pi)$ is strongly connected i.e. from any value $d$ of $v$ we can reach any other value $d'$ of $v$. Hence "$v$ can always move to any value desired".

These two things together imply that, given a plan $\vec{a}$ for $\Pi'$, we can insert a suitable move sequence for $v$ into $\vec{a}$ to obtain a plan for $\Pi$.

### Complexity: Acyclic + Invertible

**Servants + clients, now in full:**




→ A plan can be constructed in polynomial time:

1. Order the variables topologically $v_1, \ldots, v_n$ from "servants" to "clients".
2. Iteratively apply step 1 on slide 22 to $v_1, \ldots, v_n$ in this order. Then $\Pi'$ is empty, and the empty plan $\vec{a} := \langle\rangle$ solves it.
3. Iteratively apply step 2 on slide 22 to $v_n, \ldots, v_1$ in this order.

→ Intuition: Iteratively deal with clients, then insert needed moves for servants.

**Theorem.** *Restrict the input to FDR tasks* $\Pi = (V, A, c, I, G)$ *such that* $CG(\Pi)$ *is acyclic and, for all* $v \in V$*, all value transitions of* $v$ *are invertible. Then PlanEx can be decided in polynomial time.*

→ Note: We do *not* require the DTGs to be connected here. If they were connected, $\Pi$ would be solvable and there would be no PlanEx to decide. Also, $\Pi$ can be solvable even for non-connected DTGs:

**Proof intuition [for reference]:** If every $v \in V$ can reach all target values – those requested in preconditions by its clients – in $DTG(v, \Pi)$, then, due to invertibility, these target values can be provided whenever they are requested.

If there exists $v \in V$ that can *not* reach all target values in $DTG(v, \Pi)$, then the plan cannot be constructed.

**Summary**

For general problem solving to be effective, it is essential to automatically detect and exploit problem structure. Causal graphs are the most prominent means to capture problem structure in planning; they are typically considered along with domain transition graphs. Causal graphs can be used for a variety of purposes, including task decomposition/simplification and complexity analysis. Simple decomposition/simplification methods are to split up unconnected components, remove invertible root variables, remove non-goal leaf variables. One tractable class is the special case where the causal graph is acyclic and all value transitions are invertible.

# Chapter 17. Progression and Regression

There are three independent choices to make:

1) search space
   **Progression.**
   → Search forward from initial state to goal.
   → Search states = world states.

   **Regression**.
   → Search backward from goal to initial state.
   → Search states = sub-goals we would need to achieve.
2) search algorithm
   **Blind search.**
   → Depth-first, breadth-first, iterative depth-first, . . .
   **Heuristic search (systematic)**. Aka informed search (systematic).
   → A ∗ , IDA∗ , . . .
   **Heuristic search (local).** Aka informed search (local).
   → Hill-climbing, simulated annealing, beam search

3) search control

**Heuristic function.** (For heuristic searches.)
→ Critical-path heuristics, delete-relaxation heuristics, abstraction heuristics, landmarks heuristics, .
**Pruning techniques**.
→ Helpful actions pruning, symmetry elimination, dominance pruning, partial-order reduction.

(1) **What is a "Search Space"?**

A (classical) search space is defined by the following three operations:

- InitialState(): Generate the start (search) state.
- GoalTest(s): Test whether a given search state is a goal state.
- ChildState(s, a): Generates the successor states (s') of search state s, by applying applicable action a.

Progression and regression instantiate this template in different ways.

**Progression**

**Search Space for Progression**

Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The progression search space of $\Pi$ is given by:

- InitialState() $= I$
- GoalTest$(s) = \begin{cases} \textbf{true} & \text{if } G \subseteq s \\ \textbf{false} & \text{otherwise} \end{cases}$
- ChildState$(s, a) = \{s' \mid \Theta_\Pi \text{ has the transition } s \xrightarrow{a} s'\}$

The same definition applies to FDR tasks $\Pi = (V, A, c, I, G)$.

→ **Start from the initial state, and apply actions until a goal state is reached.**
→ **Search space = state space ⇒ called state space search.**

**Regression**

**Search Space for Regression**

Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The regression search space of $\Pi$ is given by:

- InitialState() $= G$
- GoalTest$(g) = \begin{cases} \textbf{true} & \text{if } g \subseteq I \\ \textbf{false} & \text{otherwise} \end{cases}$
- ChildState$(g, a) = \{g' \mid g' = regr(g, a)\}$

The *same definition* applies to FDR tasks $\Pi = (V, A, c, I, G)$.

**→ Start at goal, and regress over actions to produce subgoals, until a subgoal is contained in the initial state.**

**Condition (*) required:** If $g' = regr(g, a)$, then for all $s'$ with $s' \models g'$, we have $s'[\![a]\!] = s$ where $s \models g$.

### Regressing Subgoals Over Actions: FDR

**Definition (FDR Regression).** *Let $(V, A, c, I, G)$ be an FDR planning task, $g$ be a partial variable assignment, and $a \in A$.*
*We say that $g$ is regressable over $a$ if*

1. $eff_a \cap g \neq \emptyset$;
2. *there is no $v \in V$ s.t. $v \in V[eff_a] \cap V[g]$ and $eff_a(v) \neq g(v)$; and*
3. *there is no $v \in V$ s.t. $v \notin V[eff_a]$, $v \in V[pre_a] \cap V[g]$, and $pre_a(v) \neq g(v)$.*

*In that case, the regression of $g$ over $a$ is $regr(g, a) = (g \setminus eff_a) \cup pre_a$; else $regr(g, a)$ is undefined, written $regr(g, a) = \bot$.*

→ Intuition: $a$ can make the conjunctive subgoal $g$ true if (i) it achieves part of $g$; (ii) it contradicts none of $g$; and (iii) the new subgoal we would have to solve is not self-contradictory.

**Proposition.** *This definition of $regr$ satisfies condition (*) on slide 15.*

### Regressing Subgoals Over Actions: STRIPS

**Definition (STRIPS Regression).** *Let $(P, A, c, I, G)$ be a STRIPS planning task, $g \subseteq P$, and $a \in A$. We say that $g$ is regressable over $a$ if*

1. $add_a \cap g \neq \emptyset$; *and*
2. $del_a \cap g = \emptyset$.

*In that case, the regression of $g$ over $a$ is $regr(g, a) = (g \setminus add_a) \cup pre_a$; else $regr(g, a)$ is undefined, written $regr(g, a) = \bot$.*

**Proposition.** *This definition of $regr$ satisfies condition (*) on slide 15.*

**Note the difference to FDR:**

- In (ii), instead of "contradicting variable values", we only look at the action's immediate deletes.
  → This condition is weaker; e.g., we fail to see that different truck positions yield contradictions as well (see next slide).
- Condition (iii) here is missing completely because in STRIPS there is no possibility for a subgoal to be "self-contradictory".
  → This is also weaker; e.g., we fail to see that subgoals requiring several different truck positions are self-contradictory (see next slide).

→ **Progression explores only reachable states, but may explore unsolvable ones.**
→ **Regression explores only solvable states, but may explore unreachable ones.**

**Progression and Relevance**

Observe: Progression doesn't know what's "relevant", i.e., what contributes to reaching the goal
→ Use heuristic function to guide the search towards the goal!

**Regression and Reachability**
Observe: Regression doesn't know what's "reachable", i.e., what contributes to reaching the initial state
→ Use heuristic function to guide the search towards the initial state!

**So Which One Should We Use?**

In favor of progression:
Regression has in the past often had serious trouble getting lost in gazillions of solvable but unreachable states.
Reachable dead end states tend to be less frequent in practice.
Progression allows easy formulation of searches for more complex planning formalisms (numbers, durations, uncertainty, you name it).
Which one works better depends on the input task and search algorithm, and there is no comprehensive understanding of this.

**Summary**

Search is required in planning because the problem is computationally hard. We consider classical search, that finds a path through a search space implicitly defined in terms of the operations InitialState(), GoalTest(s), and ChildState(s, a).
Progression is forward search from the initial state to the goal, in the state space. To be effective, it needs to be informed about relevance.
Regression is backward search from the goal to the initial state, in a space of subgoals that correspond to sets of world states. To be effective, it needs to be informed about reachability. FDR regression is a lot more effective than STRIPS regression, because its search space contains fewer unreachable states.

# Chapter 18. Heuristic Search

→ Heuristic function h estimates the cost of an optimal path from a state s to the goal; search prefers to expand states s with small h(s).

**Definition (Heuristic Function).** *Let $\Pi$ be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$. A heuristic function, short heuristic, for $\Pi$ is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. Its value $h(s)$ for a state $s$ is referred to as the state's heuristic value, or $h$ value.*

**Definition (Remaining Cost, $h^*$).** *Let $\Pi$ be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$. For a state $s \in S$, the state's remaining cost is the cost of an optimal plan for $s$, or $\infty$ if there exists no plan for $s$. The perfect heuristic for $\Pi$, written $h^*$, assigns every $s \in S$ its remaining cost as the heuristic value.*

→ Heuristic functions $h$ estimate remaining cost $h^*$.

→ These definitions apply to both STRIPS and FDR.

**What does it mean "estimate remaining cost"?**

- In principle, the "estimate" is an arbitrary function. In practice, we want it to be accurate (aka: informative), i.e., close to the actual remaining cost.
- We also want it to be fast, i.e., a small overhead for computing $h$.
- These two wishes are in contradiction! Extreme cases?
  → $h = 0$: No overhead at all, completely un-informative.
  → $h = h^*$: Perfectly accurate, overhead=solving the problem in the first place.

→ We need to trade off the accuracy of $h$ against the overhead of computing it.

→ What exactly is "accuracy"? How does it affect search performance?
Interesting and challenging subject!

## Properties of Individual Heuristic Functions

**Definition (Safe/Goal-Aware/Admissible/Consistent).** *Let $\Pi$ be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$, and let $h$ be a heuristic for $\Pi$. The heuristic is called:*

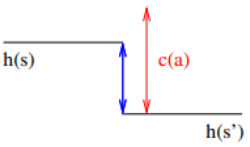
- *safe if, for all $s \in S$, $h(s) = \infty$ implies $h^*(s) = \infty$;*
- *goal-aware if $h(s) = 0$ for all goal states $s \in S^G$;*
- *admissible if $h(s) \leq h^*(s)$ for all $s \in S$;*
- *consistent if $h(s) \leq h(s') + c(a)$ for all transitions $s \xrightarrow{a} s'$.*

→ Relationships:

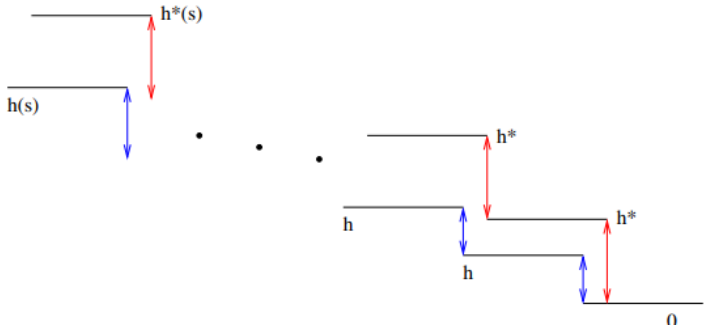
**Proposition.** *Let $\Pi$ be a planning task, and let $h$ be a heuristic for $\Pi$. If $h$ is admissible, then $h$ is goal-aware. If $h$ is admissible, then $h$ is safe. If $h$ is consistent and goal-aware, then $h$ is admissible. No other implications of this form hold.*

**Proof.** First two claims: Easy. Third claim: Next slide.

**Consistency = "heuristic value decreases by at most $c(a)$":**




**Consistent and goal-aware implies admissible:** Let $s$ be a state. $h^*(s)$ is the cost of an optimal solution path for $s$. Induction over that path, backwards from the goal: (on an optimal path, $h^*$ decreases by exactly $c(a)$ in each step)



**→ In practice, most heuristics are safe and goal-aware, and admissible heuristics are typically consistent**

**Domination Between Heuristic Functions**

**Definition (Domination).** *Let $\Pi$ be a planning task, and let $h$ and $h'$ be admissible heuristics for $\Pi$. We say that $h'$ dominates $h$ if $h \leq h'$, i.e., for all states $s$ in $\Pi$ we have $h(s) \leq h'(s)$.*

→ $h'$ dominates $h$ = "$h'$ provides a lower bound at least as good as $h$".

**Additivity of Heuristic Functions**

**Definition (Additivity).** *Let $\Pi$ be a planning task, and let $h_1, \ldots, h_n$ be admissible heuristics for $\Pi$. We say that $h_1, \ldots, h_n$ are additive if $h_1 + \cdots + h_n$ is admissible, i.e., for all states $s$ in $\Pi$ we have $h_1(s) + \cdots + h_n(s) \leq h^*(s)$.*

→ An ensemble of heuristics is additive if its sum is admissible.

**What Works Where in Planning?**

Blind (no h) vs. heuristic:
For satisficing planning, heuristic search vastly outperforms blind algorithms pretty much everywhwere.
For optimal planning, heuristic search also is better (but the difference is not as huge).

Systematic (maintain all options) vs. local (maintain only a few):
For satisficing planning, there are successful instances of each.
For optimal planning, systematic algorithms are required.

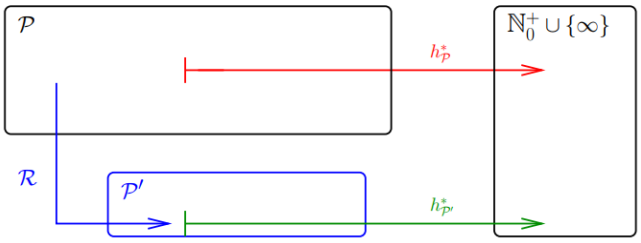
**Greedy best-first search** explores states by increasing heuristic value h. A ∗ explores states by increasing plan-cost estimate g + h. Complete but not optimal.
– Fast but not optimal ==⇒ satisficing planning

**A*** Complete and optimal
Optimal for admissible h ⇒ optimal planning, with such h

**Weighted A*** explores states by increasing weighted-plan-cost estimate g + W ∗ h. The weight is an algorithm parameter. For W > 1, weighted A ∗ is bounded suboptimal. → If h is admissible, then the solutions returned are at most a factor W more costly than the optimal ones.
Allows to interpolate between greedy best-first search and A ∗ , trading off plan quality against computational effort.

**Heuristic Functions from Relaxed Problems**

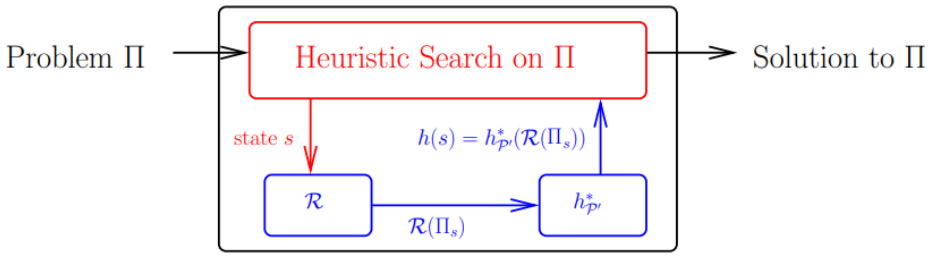
- **How to Relax**




- You have a class $\mathcal{P}$ of problems, whose perfect heuristic $h^*_{\mathcal{P}}$ you wish to estimate.
- You define a class $\mathcal{P}'$ of *simpler problems*, whose perfect heuristic $h^*_{\mathcal{P}'}$ can be used to *estimate* $h^*_{\mathcal{P}}$.
- You define a transformation – the relaxation mapping $\mathcal{R}$ – that maps instances $\Pi \in \mathcal{P}$ into instances $\Pi' \in \mathcal{P}'$.
- Given $\Pi \in \mathcal{P}$, you let $\Pi' := \mathcal{R}(\Pi)$, and estimate $h^*_{\mathcal{P}}(\Pi)$ by $h^*_{\mathcal{P}'}(\Pi')$.

## How to Relax During Search: Overview

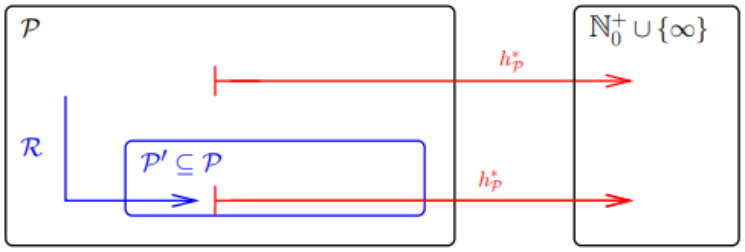
**Attention!** Search uses the real (un-relaxed) $\Pi$. The relaxation is applied **only within the call to** $h(s)$**!!!**




- Here, $\Pi_s$ is $\Pi$ with initial state replaced by $s$, i.e., $\Pi = (P, A, I, G)$ changed to $(P, A, s, G)$: The task of finding a plan for search state $s$.
- A common student mistake is to instead apply the relaxation once to the whole problem, then doing the whole search "within the relaxation".

## Only-Adds and Ignoring Deletes are "Native" Relaxations

**Native Relaxations:** Confusing special case where $\mathcal{P}' \subseteq \mathcal{P}$.




- Problem class $\mathcal{P}$: STRIPS planning tasks.
- Perfect heuristic $h^*_{\mathcal{P}}$ for $\mathcal{P}$: Length $h^*$ of a shortest plan.
- Transformation $\mathcal{R}$: Drop the (preconditions and) delete lists.
- Simpler problem class $\mathcal{P}'$ is a special case of $\mathcal{P}$, $\mathcal{P}' \subseteq P$: STRIPS planning tasks with empty (preconditions and) delete lists.
- Perfect heuristic for $\mathcal{P}'$: Shortest plan for only-adds respectively delete-free STRIPS task.

**Summary**

Heuristic functions h map states to estimates of remaining cost. A heuristic can be safe, goal-aware, admissible, and/or consistent. A heuristic may dominate another heuristic, and an ensemble of heuristics may be additive.

Greedy best-first search can be used for satisficing planning, A ∗ can be used for optimal planning provided h is admissible. Weighted A ∗ interpolates between the two.

Relaxation is a method to compute heuristic functions. Given a problem P we want to solve, we define a relaxed problem P'. We derive the heuristic by mapping into P' and taking the solution to this simpler problem as the heuristic estimate. During search, the relaxation is used only inside the computation of h(s) on each state s; the relaxation does not affect anything else.

# Chapter 19. Critical Path Heuristics

This chapter is about finding methods for relaxed planning tasks, where we want to compute an heuristic function *automatically.*
The basic idea is: “Approximate the cost of a goal set by the most costly subgoal g.” g may be a set of possible subgoals, so in general |g|>=1. If |g|=1, it is called singleton.

The concept of approximation is related to regression. So we’ll use regression to approximate the heuristic function.

**A Regression-Based Characterization of h***

**Definition** $(r^*)$. *Let* $\Pi = (P, A, c, I, G)$ *be a STRIPS planning task. The perfect regression heuristic* $r^*$ *for* $\Pi$ *is the function* $r^*(s) := r^*(s, G)$ *where* $r^*(s, g)$ *is the function that satisfies*

$$r^*(s,g) = \begin{cases} 0 & g \subseteq s \\ min_{a \in A, regr(g,a) \neq \bot} c(a) + r^*(s, regr(g,a)) & \text{otherwise} \end{cases}$$

(Reminder Chapter 17: $regr(g,a) \neq \bot$ if $add_a \cap g \neq \emptyset$ and $del_a \cap g = \emptyset$; then, $regr(g,a) = (g \setminus add_a) \cup pre_a$.)

So, The cost of achieving a subgoal g is 0 if it is true in s (so we are in a goal state); else, it is the minimum of using any action a to achieve g.

Proposition. Let Π = (P, A, c, I, G) be a STRIPS planning task. Then r* = h*.

But since h* is the perfect heuristic then we have to discard it, since not reachable.
So we use an approximation of h*: $h^m$.

if m = 1:

**Definition** $(h^1)$. *Let* $\Pi = (P, A, c, I, G)$ *be a STRIPS planning task. The critical path heuristic* $h^1$ *for* $\Pi$ *is the function* $h^1(s) := h^1(s, G)$ *where* $h^1(s, g)$ *is the function that satisfies*

$$h^1(s,g) = \begin{cases} 0 & g \subseteq s \\ min_{a \in A, regr(g,a) \neq \bot} c(a) + h^1(s, regr(g,a)) & |g| = 1 \\ \max_{g' \in g} h^1(s, \{g'\}) & |g| > 1 \end{cases}$$

Else:

**Definition ($h^m$).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $m \in \mathbb{N}$. The critical path heuristic $h^m$ for $\Pi$ is the function $h^m(s) := h^m(s, G)$ where $h^m(s, g)$ is the function that satisfies*

$$h^m(s,g) = \begin{cases} 0 & g \subseteq s \\ min_{a \in A, regr(g,a) \neq \bot} c(a) + h^m(s, regr(g,a)) & |g| \leq m \\ \max_{g' \subseteq g, |g'|=m} h^m(s, g') & |g| > m \end{cases}$$

For subgoal sets |g| ≤ m, use regression as in r* . For subgoal sets |g| > m, use the cost of the most costly m-subset g 0 .

“Critical path” = Cheapest path through the most costly subgoals gi.

→ For fixed m, $h^m$(s, g) can be computed in time polynomial in the size of Π

**Critical Path Heuristics: Properties**

**Proposition ($h^m$ is Admissible).** $h^m$ is consistent and goal-aware, and thus also admissible and safe.

**Proof Sketch.** Goal-awareness is obvious. We need to prove that $h^m(s) \leq h^m(s') + c(a)$ for all transitions $s \xrightarrow{a} s'$. Since $s \supseteq regr(s', a)$, a critical path $\vec{p}$ for $h^m(s')$ can be pre-fixed by $a$ to obtain an upper bound on $h^m(s)$: *all* subgoals at the start of $\vec{p}$ are contained in $s'$, and are achieved by $a$ in $s$.

Intuition: $h^m$ is admissible because it is always more difficult to achieve larger subgoals (so m-subsets can only be cheaper).

**Proposition** ($h^m$ gets more accurate as m grows). $h^{m+1}$ dominates $h^m$.
**Proof Intuition**: “It is always more difficult to achieve larger subgoals.”

**Proposition ($h^m$ is perfect in the limit)**.
There exists m s.t. $h^m$ = h* . Proof. Setting m := |P|, the case |g| > m will never be used, so $h^m$ = r*.

**Dynamic Programming**

Basic idea:
Consider all subgoals g with size ≤ m. Initialize $h^m$ (s, g) to 0 if g ⊆ s, and to ∞ otherwise. Then, keep updating the value of each g based on actions applied to the values computed so far, until the values converge.
We start with an iterative definition of h m that makes this approach explicit.
We define a dynamic programming algorithm that corresponds to this iterative definition.

**Iterative Definition of $h^m$**

**Definition (Iterative $h^m$).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $m \in \mathbb{N}$. The iterative $h^m$ heuristic $h_i^m$ is defined by*

$$h_0^m(s,g) := \begin{cases} 0 & g \subseteq s \\ \infty & \text{otherwise} \end{cases}$$

*and* $h_{i+1}^m(s,g) :=$

$$\begin{cases} \min[h_i^m(s,g), \min_{a \in A, regr(g,a) \neq \bot} c(a) + h_i^m(s, regr(g,a))] & |g| \leq m \\ \max_{g' \subseteq g, |g'|=m} h_{i+1}^m(s,g') & |g| > m \end{cases}$$

**Proposition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. Then the series $\{h_i^m\}_{i=0,..}$ converges to $h^m$.*

**Proof Sketch:** (i) Convergence: If $h_{i+1}^m(s,g) \neq h_i^m(s,g)$, then $h_{i+1}^m(s,g) < h_i^m(s,g)$; that can happen only finitely often because each decrease is due to a new path for $g$. (ii) If $h_{i+1}^m = h_i^m$ then $h_i^m$ satisfies the $h^m$ equation (direct from definition). (iii) No function greater than $h_i^m$ at any point can satisfy the $h^m$ equation (easy by induction over $i$).

**Dynamic Programming Algorithm**

**new** table $T_0^m(g)$, for all $g \subseteq P$ with $|g| \leq m$
For all $g \subseteq P$ with $|g| \leq m$: $T_0^m(g) := \begin{cases} 0 & g \subseteq s \\ \infty & \text{otherwise} \end{cases}$
**fn** $Cost_i(g) := \begin{cases} T_i^m(g) & |g| \leq m \\ \max_{g' \subseteq g, |g'|=m} T_i^m(g') & |g| > m \end{cases}$
**fn** $Next_i(g) := \min[Cost_i(g), \min_{a \in A, regr(g,a) \neq \bot} c(a) + Cost_i(regr(g,a))]$
$i := 0$
**do forever**:
  **new** table $T_{i+1}^m(g)$, for all $g \subseteq P$ with $|g| \leq m$
  For all $g \subseteq P$ with $|g| \leq m$: $T_{i+1}^m(g) := Next_i(g)$
  **if** $T_{i+1}^m = T_i^m$ **then** stop **endif**
  $i := i + 1$
**enddo**

**Proposition** $h_i^m(s,g) = Cost_i(g)$ *for all $i$ and $g$*

**→ This is very inefficient! (Optimized for readability.) We can use "Generalized Dijkstra" instead, maintaining the frontier of cheapest m-tuples reached so far.**

### Graphplan Representation: The Case m = 1

**1-Planning Graphs**

$F_0 := s$; $i := 0$
**while** $G \not\subseteq F_i$ **do**
  $A_i := \{a \in A \mid pre_a \subseteq F_i\}$
  $F_{i+1} := F_i \cup \bigcup_{a \in A_i} add_a$
  **if** $F_{i+1} = F_i$ **then** stop **endif**
  $i := i + 1$
**endwhile**

### 1-Planning Graphs vs. $h^1$

**Definition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The 1-planning graph heuristic $h^1_{\mathsf{PG}}$ for $\Pi$ is the function $h^1_{\mathsf{PG}}(s) := \min\{i \mid s \subseteq F_i\}$, where $F_i$ are the fact sets computed by a 1-planning graph (and the minimum over an empty set is $\infty$.)*

**Proposition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task with unit costs. Then $h^1_{\mathsf{PG}} = h^1$.*

**Proof Sketch:** Induction over the value $i$ of $h^1(s)$. Trivial for base case $i = 0$. For the step case, assume that $h^1_{\mathsf{PG}}(s) = h^1(s)$ for all $s$ where $h^1(s) \leq i$, and show the same property for all $s$ with $h^1(s) \leq i + 1$. $h^1_{\mathsf{PG}}(s) < i + 1$ directly contradicts the assumption. To show $h^1_{\mathsf{PG}}(s) \leq i + 1$, it suffices to observe that $h^1(pre_a) \leq i$ implies $h^1_{\mathsf{PG}}(pre_a) \leq i$ by assumption.

→ Intuition: A 1-planning graph is like our dynamic programming algorithm for $m = 1$, except that it represents not all facts but only those that have been reached (value $\neq \infty$), and instead of a fact-value table it only remembers that set.

### Graphplan Representation: The General Cas

**$m$-Planning Graphs**

$F_0 := s$; $M_0 := \emptyset$; $i := 0$
**fn** $Reached_i(g) := \begin{cases} True & g \subseteq F_i, \nexists g' \in M_i : g' \subseteq g \\ False & \text{otherwise} \end{cases}$
**while** not $Reached_i(G)$ **do**
  $A_i := \{a \in A \mid Reached_i(pre_a)\}$
  $F_{i+1} := F_i \cup \bigcup_{a \in A_i} add_a$
  $M_{i+1} := \{g \subseteq P \mid |g| \leq m, \forall a \in A_i : \text{not } Reached_i(regr(g, a))\}$
  **if** $F_{i+1} = F_i$ and $M_{i+1} = M_i$ **then** stop **endif**
  $i := i + 1$
**endwhile**

→ Intuition: All m-subsets g of Fi are reachable within i steps, except for those g listed in Mi (the “mutexes”)
→ Instead of listing the reached m-subsets, represent those that are not reached (and hope that there are fewer of those).

**Critical Path Heuristics in FDR**

All definitions, results, and proofs apply, exactly as stated, also to FDR planning tasks

The single non-verbatim-applicable statement, adapted to FDR:

**Proposition ($h^m$ is Perfect in the Limit).** *There exists $m$ s.t. $h^m = h^*$.*

**Proof.** Given the definition of $regr(g, a)$ for FDR (→ Chapter 17), it is easy to see by induction that every subgoal $g$ contains at most one fact for each variable $v \in V$. Thus, if we set $m := |V|$, then the case $|g| > m$ will never be used, so $h^m = r^*$.

→ In FDR, it suffices to set $m$ to the number of *variables*, as opposed to the number of *variable values* i.e. STRIPS facts (compare slide 12)!

**Summary**
The critical path heuristics h estimate the cost of reaching a subgoal g by the most costly m-subset of g. This is admissible because it is always more difficult to achieve larger subgoals. h can be computed using dynamic programming, i.e., initializing true m-subsets g to 0 and false ones to ∞, then applying value updates until convergence. This computation is polynomial in the size of the planning task, given fixed m. In practice, m = 1, 2 are used; m > 2 is typically infeasible. Planning graphs correspond to dynamic programming with unit costs, using a particular representation of reached/unreached m-subsets g.

# Chapter 20. Delete Relaxation Heuristics

Here, we want to delete relaxation.
Delete relaxation is a method to relax planning tasks, and thus automatically compute heuristic functions h.
The idea to exploit is: if we have obtained something true, then it will be true.

### The Delete Relaxation

**Definition (Delete Relaxation).**

1. *For a STRIPS action $a$, by $a^+$ we denote the corresponding delete relaxed action, or short relaxed action, defined by $pre_{a^+} := pre_a$, $add_{a^+} := add_a$, and $del_{a^+} :=$*
2. *For a set $A$ of STRIPS actions, by $A^+$ we denote the corresponding set of relaxed actions, $A^+ := \{a^+ \mid a \in A\}$; similarly, for a sequence $\vec{a} = \langle a_1, \ldots, a_n \rangle$ of STRIPS actions, by $\vec{a}^+$ we denote the corresponding sequence of relaxed actions, $\vec{a}^+ := \langle a_1^+, \ldots, a_n^+ \rangle$.*
3. *For a STRIPS planning task $\Pi = (P, A, c, I, G)$, by $\Pi^+ := (P, A^+, c, I, G)$ we denote the corresponding (delete) relaxed planning task.*

$\rightarrow$ "+" super-script = delete relaxed. We'll also use this to denote states encountered within the relaxation.

**Definition (Relaxed Plan).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $s$ be a state. An (optimal) relaxed plan for $s$ is an (optimal) plan for $\Pi_s^+$ where $\Pi_s = (P, A, c, s, G)$. A relaxed plan for $I$ is also called a relaxed plan for $\Pi$.*

*No delete effect in the relaxed world! So what is true, remains true.*

### State Dominance

**Definition (Dominance).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $s, s'$ be states. We say that $s'$ dominates $s$ if $s' \supseteq s$.*

$\rightarrow$ Dominance = "more facts true".

**Proposition (Dominance).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $s, s'$ be states where $s'$ dominates $s$. We have:*

1. *If $s$ is a goal state, then $s'$ is a goal state as well.*
2. *If $\vec{a}$ is applicable in $s$, then $\vec{a}$ is applicable in $s'$ as well, and $s'[\![\vec{a}]\!]$ dominates $s[\![\vec{a}]\!]$.*

**Proof.** (i) is trivial. (ii) by induction over the length $n$ of $\vec{a}$. Base case $n = 0$ is trivial. Inductive case $n \rightarrow n + 1$ follows directly from induction hypothesis and the definition of $s[\![a]\!]$.

$\rightarrow$ It is always better to have more facts true.

This is good because we have more chances to reach the goal and preconditions satisfied.

### The Delete Relaxation and State Dominance

**Proposition**. *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. Let $s$ be a state, and let $a \in A$ be applicable in $s$. Then:*

(i) *$s[\![a^+]\!]$ dominates $s$.*

(ii) *For any state $s'$ that dominates $s$, $s'[\![a^+]\!]$ dominates $s[\![a]\!]$.*

Ergo 1: Any real plan also works in the relaxed world.

**Proposition (Delete Relaxation is Over-Approximating).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, let $s$ be a state, and let $\vec{a}$ be a plan for $\Pi_s$. Then, $\vec{a}^+$ is a relaxed plan for $s$.*

**Proof.** Prove by induction over the length of $\vec{a}$ that $s[\![\vec{a}^+]\!]$ dominates $s[\![\vec{a}]\!]$. Base case is trivial, inductive case follows from (ii) above.

Ergo 2: It is now clear how to find a relaxed plan.

- Applying a relaxed action can only ever make more facts true ((i) above).
- That cannot render the task unsolvable (proposition slide 10).

$\Longrightarrow$ So?

### Greedy Relaxed Planning

**Greedy Relaxed Planning for $\Pi_s^+$**

$s^+ := s$; $\vec{a}^+ := \langle\rangle$
**while** $G \not\subseteq s^+$ **do**:
  **if** $\exists a \in A$ s.t. $pre_a \subseteq s^+$ and $s^+[\![a^+]\!] \neq s^+$ /* i.e. $add_a \not\subseteq s^+$ */ **then**
    select one such $a$
    $s^+ := s^+[\![a^+]\!]$; $\vec{a}^+ := \vec{a}^+ \circ \langle a^+ \rangle$
  **else return** "$\Pi_s^+$ is unsolvable" **endif**
**endwhile**
**return** $\vec{a}^+$

**Proposition**. *Greedy relaxed planning is sound, complete, and terminates in time polynomial in the size of $\Pi$.*

**Proof.** *Soundness*: If $\vec{a}^+$ is returned then, by construction, $G \subseteq s[\![\vec{a}^+]\!]$. *Completeness:* If "$\Pi_s^+$ is unsolvable" is returned, then no relaxed plan exists for $s^+$ at that point. As $s^+$ dominates $s$, by the dominance proposition (slide 10), this implies that no relaxed plan can exist for $s$. *Termination:* Every $a \in A$ can be selected at most once because afterwards $s^+[\![a^+]\!] = s^+$.

$\Longrightarrow$ It is easy to decide whether a relaxed plan exists!

### Greedy Relaxed Planning to Generate a Heuristic Function?

**Using greedy relaxed planning to generate $h$**

- In search state $s$ during forward search, run greedy relaxed planning on $\Pi_s^+$
- Set $h(s)$ to the cost of $\vec{a}^+$, or $\infty$ if "$\Pi_s^+$ is unsolvable" is returned.

→ Is this $h$ accurate? NO! Greedy relaxed planning may select arbitrary actions that aren't relevant at all, over-estimating dramatically (cf. previous slide).

→ To be accurate, a heuristic needs to approximate the *minimum effort* needed to reach the goal.

- When we talk about "the distance to Moscow", we don't mean "via Madrid" ...
- There also is an issue of "brittleness": Greedy relaxed planning may give drastically different values for very similar states. This is bound to be detrimental for search guidance.
- To the rescue: $h^+$.

### h+: The Optimal Delete Relaxation Heuristic

**Definition ($h^+$).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task with state space $\Theta_\Pi = (S, A, c, T, I, G)$. The optimal delete relaxation heuristic $h^+$ for $\Pi$ is the function $h^+ : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ where $h^+(s)$ is defined as the cost of an optimal relaxed plan for $s$.*

→ $h^+$= minimum effort to reach the goal under delete relaxation.

→ But won't $h^+$ usually under-estimate $h^*$? Yes, but that's just the effect of considering a relaxed problem. Arbitrarily adding actions useless within the relaxation (e.g., going to Moscow via Madrid) does not help to address it.

**Proposition ($h^+$ is Consistent).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. Then $h^+$ is consistent, and thus admissible, safe, and goal-aware.*

**Proof.** Let $s' = s[\![a]\!]$. We need to show that $h^+(s) \leq h^+(s') + c(a)$. Let $\pi'$ be an optimal relaxed plan for $s'$. Construct $\pi := \langle a \rangle \circ \pi'$. It suffices to show that $\pi$ is a relaxed plan for $s$. That is so because, by Proposition slide 11 (ii), $s[\![a^+]\!]$ dominates $s[\![a]\!] = s'$, from which the claim follows by Proposition slide 10 (ii).

### Finding a Path in a Graph

**Finding a Path in a Graph, STRIPS:** From $x$ to $y$ in graph $(N, E)$

- $P = \{at(n) \mid n \in N\}$.
- $A = \{move(n, n') \mid (n, n') \in E\}$ where $move(n, n') = (\{at(n)\}, \{at(n')\}, \{at(n)\})$.
- $I = \{at(x)\}$; $G = \{at(y)\}$.

**Proposition.** *In the above STRIPS task $(P, A, c, I, G)$, $h^+(I) = h^*(I)$.*

**Proof.** Say that $\vec{p} := \langle move(x, n_1), move(n_1, n_2), \ldots, move(n_k, y) \rangle$ is an optimal relaxed plan for $(P, A, c, I, G)$. Then $\vec{p}$ is a plan for $(P, A, c, I, G)$ because $x, n_1, \ldots, n_k, y$ is a shortest path from $x$ to $y$. This traverses each node at most once, hence the deleted facts are not needed later on.

→ "Shortest paths never walk back", hence deleted facts are never needed again later on, hence delete relaxation is exact here.

### How to Compute h+?

**Definition (PlanOpt$^+$).** *By PlanOpt$^+$, we denote the problem of deciding, given a STRIPS planning task $\Pi = (P, A, c, I, G)$ and $B \in \mathbb{R}_0^+$, whether there exists a relaxed plan for $\Pi$ whose cost is at most $B$.*

→ By computing $h^+$, we would solve PlanOpt$^+$.

**Theorem (Optimal Relaxed Planning is Hard).** *PlanOpt$^+$ is* **NP***-complete.*

**Proof.** Membership: Easy (guess action sequences of length $|A|$).
Hardness by reduction from SAT. Example: $\{C_1 = \{A\}, C_2 = \{\neg A\}\}$

- Actions setting variable to true, e.g.: pre empty, add $\{Atrue, Aset\}$.
- Actions setting variable to false, e.g.: pre empty, add $\{Afalse, Aset\}$.
- Actions satisfying clauses, e.g.: pre $Atrue$, add $C_1 sat$; pre $Afalse$, add $C_2 sat$.
- Goal: "$X_i set$" for all variables $X_i$, "$C_j sat$" for all clauses $C_j$.
- $B :=$ number of variables + number of clauses ($= 3$ here).

The delete relaxation heuristic we want is h+. Unfortunately, this is hard to compute so the computational overhead is very likely to be prohibitive. All implemented systems using the delete relaxation approximate h+ in one or the other way.

→ We will look at the most wide-spread approaches to do so.

### The Additive and Max Heuristics

**Definition ($h^{\mathsf{add}}$).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The additive heuristic $h^{\mathsf{add}}$ for $\Pi$ is the function $h^{\mathsf{add}}(s) := h^{\mathsf{add}}(s, G)$ where $h^{\mathsf{add}}(s, g)$ is the function that satisfies*

$$h^{\mathsf{add}}(s,g) = \begin{cases} 0 & g \subseteq s \\ min_{a \in A, g' \in add_a} c(a) + h^{\mathsf{add}}(s, pre_a) & g = \{g'\} \\ \sum_{g' \in g} h^{\mathsf{add}}(s, \{g'\}) & |g| > 1 \end{cases}$$

**Definition ($h^{\mathsf{max}}$).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. The max heuristic $h^{\mathsf{max}}$ for $\Pi$ is the function $h^{\mathsf{max}}(s) := h^{\mathsf{max}}(s, G)$ where $h^{\mathsf{max}}(s, g)$ is the function that satisfies*

$$h^{\mathsf{max}}(s,g) = \begin{cases} 0 & g \subseteq s \\ min_{a \in A, g' \in add_a} c(a) + h^{\mathsf{max}}(s, pre_a) & g = \{g'\} \\ \max_{g' \in g} h^{\mathsf{max}}(s, \{g'\}) & |g| > 1 \end{cases}$$

## The Additive and Max Heuristics: Properties

**Proposition ($h^{\max}$ is Optimistic).** *$h^{\max} \leq h^+$, and thus $h^{\max} \leq h^*$.*

**Intuition.** $h^{\max}$ simplifies relaxed planning by assuming that, to achieve a set $g$ of subgoals, it suffices to achieve the single most costly $g' \in g$. Actual relaxed planning, i.e. $h^+$, can only be more expensive.

**Proposition ($h^{\text{add}}$ is Pessimistic).** *For all STRIPS planning tasks $\Pi$, $h^{\text{add}} \geq h^+$. There exist $\Pi$ and $s$ so that $h^{\text{add}}(s) > h^*(s)$.*

**Intuition.** $h^{\text{add}}$ simplifies relaxed planning by assuming that, to achieve a set $g$ of subgoals, we must achieve every $g' \in g$ separately. Actual relaxed planning, i.e. $h^+$, can only be less expensive. Proof for inadmissiblity: see example on slide 34.

→ Both $h^{\max}$ and $h^{\text{add}}$ approximate $h^+$ by assuming that singleton subgoal facts are achieved independently. $h^{\max}$ estimates *optimistically* by the most costly singleton subgoal, $h^{\text{add}}$ estimates *pessimistically* by summing over all singleton subgoals.

**Proposition ($h^{\max}$ and $h^{\text{add}}$ agree with $h^+$ on $\infty$).** *For all STRIPS planning tasks $\Pi$ and states $s$ in $\Pi$, $h^+(s) = \infty$ if and only if $h^{\max}(s) = \infty$ if and only if $h^{\text{add}}(s) = \infty$.*

**Proof.** $h^{\max}$ and $h^{\text{add}}$ agree on states with infinite heuristic value simply because their only difference lies in the use of the $\max$ vs. $\sum$ operations which does not affect this property.

$h^+(s) < \infty$ implies $h^{\max}(s) < \infty$ because $h^{\max} \leq h^+$. Vice versa, $h^{\max}(s) < \infty$ implies $h^+(s) < \infty$ because $h^{\max}$ can then be used to generate a closed well-founded best-supporter function, from which a relaxed plan can be extracted, cf. the next section.

→ States for which no relaxed plan exists are easy to recognize, and that is done by both $h^{\max}$ and $h^{\text{add}}$. Approximation is needed only for the cost of an optimal relaxed plan, if it exists.

**Proposition.** $h^{\max} = h^1$.

**Proof.** Say $g = \{g'\}$. $regr(\{g'\}, a) \neq \bot$ if $add_a \cap \{g'\} \neq \emptyset$ and $del_a \cap \{g'\} = \emptyset$; then, $regr(g, a) = (\{g'\} \setminus add_a) \cup pre_a$. Because $add_a \cap del_a = \emptyset$, this is the same as saying "$g' \in add_a$, and $regr(g, a) = pre_a$".

**Dynamic Programming algorithm computing $h^{\text{add}}$ for state $s$**

**new** table $T_0^{\text{add}}(g)$, for $g \in P$

For all $g \in P$: $T_0^{\text{add}}(g) := \begin{cases} 0 & g \in s \\ \infty & \text{otherwise} \end{cases}$

**fn** $Cost_i(g) := \begin{cases} T_i^{\text{add}}(g) & |g| = 1 \\ \sum_{g' \in g} T_i^{\text{add}}(g') & |g| > 1 \end{cases}$

**fn** $Next_i(g) := \min[Cost_i(g), \min_{a \in A, g' \in add_a} c(a) + Cost_i(pre_a)]$

**do forever**:

  **new** table $T_{i+1}^{\text{add}}(g)$, for $g \in P$

  For all $g \in P$: $T_{i+1}^{\text{add}}(g) := Next_i(g)$

  **if** $T_{i+1}^{\text{add}} = T_i^{\text{add}}$ **then** stop **endif**

  $i := i + 1$

**enddo**

**Proposition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task. Then the series $\{T_i^{\text{add}}(g)\}_{i=0,\ldots}$ converges to $h^{\text{add}}(s, g)$, for all $g$.* (Proof omitted.)

**Summary of typical issues in practice with $h^{\text{add}}$ and $h^{\max}$:**

- Both $h^{\text{add}}$ and $h^{\max}$ can be computed reasonably quickly. (Well, compared to $h^2$ anyhow, never mind $h^m$ for even larger $m$.)
- $h^{\max}$ is admissible, but is typically far too optimistic. (slide 33)
- $h^{\text{add}}$ is not admissible, but is typically a lot more informed than $h^{\max}$. (slide 34)
- $h^{\text{add}}$ is sometimes better informed than $h^+$, but "for the wrong reasons" (slide 34): Rather than accounting for deletes, it overcounts by ignoring positive interactions, i.e., sub-plans shared between subgoals.

  → Such overcounting can result in dramatic over-estimates of $h^*$!

→ Recall: To be accurate, a heuristic needs to approximate the *minimum effort* needed to reach the goal.

→ Relaxed plans (up next) keep $h^{\text{add}}$'s informativity but avoid over-counting.

### Relaxed Plans, Basic Idea

→ First compute a best-supporter function $bs$, which for every fact $p \in P$ returns an action that is deemed to be the cheapest achiever of $p$ (within the relaxation). Then extract a relaxed plan from that function, by applying it to singleton subgoals and collecting all the actions.

→ The best-supporter function can be based directly on $h^{\max}$ or $h^{\text{add}}$, simply selecting an action $a$ achieving $p$ that minimizes $[c(a)$ plus the cost estimate for $pre_a]$.

### The Best-Supporter Functions We Will Use

**Definition (Best-Supporters from $h^{\max}$ and $h^{\text{add}}$).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task and let $s$ be a state.*

*The $h^{\max}$ supporter function $bs_s^{\max} : \{p \in P \mid 0 < h^{\max}(s, \{p\}) < \infty\} \mapsto A$ is defined by $bs_s^{\max}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\max}(s, pre_a)$.*

*The $h^{\text{add}}$ supporter function $bs_s^{\text{add}} : \{p \in P \mid 0 < h^{\text{add}}(s, \{p\}) < \infty\} \mapsto A$ is defined by $bs_s^{\text{add}}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\text{add}}(s, pre_a)$.*

### Relaxed Plan Extraction

**Relaxed Plan Extraction for state $s$ and best-supporter function $bs$**

$Open := G \setminus s$; $Closed := \emptyset$; $RPlan := \emptyset$
**while** $Open \neq \emptyset$ **do**:
  select $g \in Open$
  $Open := Open \setminus \{g\}$; $Closed := Closed \cup \{g\}$;
  $RPlan := RPlan \cup \{bs(g)\}$; $Open := Open \cup (pre_{bs(g)} \setminus (s \cup Closed))$
**endwhile**
**return** $RPlan$

→ Starting with the top-level goals, iteratively close open singleton subgoals by selecting the best supporter.

**This is fast!** Number of iterations bounded by $|P|$, each near-constant time.

**But is it correct?**

→ What if $g \notin add_{bs(g)}$? Doesn't make sense. → Condition (A).
→ What if $bs(g)$ is undefined? Segmentation fault. → Condition (B).
→ What if the support for $g$ eventually requires $g$ itself (then already in $Closed$) as a precondition? Then this does not yield a relaxed plan. → Condition (C).

### Best-Supporter Functions

→ For relaxed plan extraction to make sense, it requires a *closed well-founded* best-supporter function:

**Definition (Best-Supporter Function).** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, and let $s$ be a state. A best-supporter function for $s$ is a partial function $bs : (P \setminus s) \mapsto A$ such that $p \in add_a$ whenever $a = bs(p)$.*

*The support graph of $bs$ is the directed graph with vertices $(P \setminus s) \cup A$ and arcs $\{(a, p) \mid a = bs(p)\} \cup \{(p, a) \mid p \in pre_a\}$. We say that $bs$ is closed if $bs(p)$ is defined for every $p \in (P \setminus s)$ that has a path to a goal $g \in G$ in the support graph. We say that $bs$ is well-founded if the support graph is acyclic.*

- "$p \in add_a$ whenever $a = bs(p)$": Condition (A).
- $bs$ is closed: Condition (B). ("$bs$ will be defined wherever it takes us to")
- $bs$ is well-founded: Condition (C). (Relaxed plan extraction starts at the goals, and chains backwards in the support graph. If there are cycles, then this backchaining may not reach the currently true state $s$, and thus not yield a relaxed plan.)

### h max and h add Supporter Functions: Correctness

**Proposition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task such that, for all $a \in A$, $c(a) > 0$. Let $s$ be a state where $h^+(s) < \infty$. Then both $bs_s^{\max}$ and $bs_s^{\text{add}}$ are closed well-founded supporter functions for $s$.*

**Proof [for reference].** Since $h^+(s) < \infty$ implies $h^{\max}(s) < \infty$, it is easy to see that $bs_s^{\max}$ is closed ($h^{\max}(s, G) < \infty$, and recursively $h^{\max}(s, pre_a) < \infty$ for the best supporters).

If $a = bs_s^{\max}(p)$, then $a$ is the action yielding $0 < h^{\max}(s, \{p\}) < \infty$ in the $h^{\max}$ equation.

Since $c(a) > 0$, we have $h^{\max}(s, pre_a) < h^{\max}(s, \{p\})$ and thus, for all $q \in pre_a$, $h^{\max}(s, \{q\}) < h^{\max}(s, \{p\})$.

Transitively, if the support graph contains a path from fact vertex $r$ to fact vertex $t$, then $h^{\max}(s, \{r\}) < h^{\max}(s, \{t\})$. Thus there can't be cycles in the support graph and $bs_s^{\max}$ is well-founded. Similar for $bs_s^{\text{add}}$.

### Relaxed Plan Extraction: Correctness

**Proposition.** *Let $\Pi = (P, A, c, I, G)$ be a STRIPS planning task, let $s$ be a state, and let $bs$ be a closed well-founded best-supporter function for $s$. Then the action set $RPlan$ returned by relaxed plan extraction can be sequenced into a relaxed plan $\vec{a}^+$ for $s$.*

**Proof [for reference].** Order $a$ before $a'$ whenever the support graph contains a path from $a$ to $a'$. Since the support graph is acyclic, such a sequencing $\vec{a} := \langle a_1, \ldots, a_n \rangle$ exists.

We have $p \in s$ for all $p \in pre_{a_1}$, because otherwise $RPlan$ would contain the action $bs(p)$, necessarily ordered before $a_1$.

We have $p \in s \cup add_{a_1}$ for all $p \in pre_{a_2}$, because otherwise $RPlan$ would contain the action $bs(p) \neq a_1$, necessarily ordered before $a_2$.

Iterating the argument, over $p \in pre_{a_{i+1}}$ and $s \cup add_{a_1} \cup \cdots \cup add_{a_i}$, shows that $\vec{a}^+$ is a relaxed plan for $s$.

### The Relaxed Plan Heuristic

**Definition (Relaxed Plan Heuristic).** *A heuristic function is called a relaxed plan heuristic, denoted $h^{\mathsf{FF}}$, if, given a state $s$, it returns $\infty$ if no relaxed plan exists, and otherwise returns $\sum_{a \in RPlan} c(a)$ where $RPlan$ is the action set returned by relaxed plan extraction on a closed well-founded best-supporter function for $s$.*

**Recall:** (that this makes sense because)

- If a relaxed plan exists, then there exists a closed well-founded best-supporter function $bs$ (cf. slide 44).
- Relaxed plan extraction on $bs$ yields a relaxed plan (previous slide).

### The Relaxed Plan Heuristic: Properties

**Proposition ($h^{\mathsf{FF}}$ is Pessimistic and Agrees with $h^+$ on $\infty$).** *For all STRIPS planning tasks $\Pi$, $h^{\mathsf{FF}} \geq h^+$; for all states $s$, $h^+(s) = \infty$ if and only if $h^{\mathsf{FF}}(s) = \infty$. There exist $\Pi$ and $s$ so that $h^{\mathsf{FF}}(s) > h^*(s)$.*

**Proof.** $h^{\mathsf{FF}} \geq h^+$ follows directly from the previous slide. Agrees with $h^+$ on $\infty$: Direct from definition. Inadmissiblity: Whenever $bs$ makes sub-optimal choices.

→ Relaxed plan heuristics have the same theoretical properties as $h^{\mathsf{add}}$.

**So what's the point?**

- In practice, $h^{\mathsf{FF}}$ typically does not over-estimate $h^*$ (or not by a large amount, anyway).

  → $h^{\mathsf{FF}}$ *may* be inadmissible, just like $h^{\mathsf{add}}$, but for more subtle reasons.
- Can $h^{\mathsf{FF}}$ over-count, i.e., count sub-plans shared between subgoals more than once?

### Helpful Actions Pruning

**Definition (Helpful Actions).** *Let $h^{\mathsf{FF}}$ be a relaxed plan heuristic, let $s$ be a state, and let $RPlan$ be the action set returned by relaxed plan extraction on the closed well-founded best-supporter function for $s$ which underlies $h^{\mathsf{FF}}$. Then an action $a$ applicable to $s$ is called helpful if it is contained in $RPlan$.*

### Delete Relaxed FDR Planning

**Definition (Delete Relaxed FDR).** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task. Denote by $P_V := \{v = d \mid v \in V, d \in D_v\}$ the set of (FDR) facts. The relaxed state space of $\Pi$ is the labeled transition system $\Theta^+_\Pi = (S^+, L, c, T, I, S^{+G})$ where:*

- *The states (also relaxed states) $S^+ = 2^{P_V}$ are the subsets $s^+$ of $P_V$.*
- *The labels $L = A$ are $\Pi$'s actions; the cost function $c$ is that of $\Pi$.*
- *The transitions are $T = \{s^+ \xrightarrow{a} s'^+ \mid pre_a \subseteq s^+, s'^+ = s^+ \cup eff_a\}$.*
- *The initial state $I$ is identical to that of $\Pi$.*
- *The goal states are $S^{+G} = \{s^+ \in S^+ \mid G \subseteq s^+\}$.*

*An (optimal) relaxed plan for $s^+ \in S^+$ is an (optimal) solution for $s^+$ in $\Theta^+_\Pi$. A relaxed plan for $I$ is also called a relaxed plan for $\Pi$.*
*Let $\Theta_\Pi = (S, A, c, T, I, G)$ be the state space of $\Pi$. The optimal delete relaxation heuristic $h^+$ for $\Pi$ is the function $h^+ : S \mapsto \mathbb{R}^+_0 \cup \{\infty\}$ where $h^+(s)$ is defined as the cost of an optimal relaxed plan for $s$.*

→ FDR states contain exactly one fact for each variable $v \in V$. There is no such restriction on FDR relaxed states.

### Done With FDR-2-STRIPS

**Reminder:** → Chapter 14

**Proposition.** *Let $\Pi = (V, A, c, I, G)$ be an FDR planning task, and let $\Pi^{\mathsf{STR}}$ be its STRIPS translation. Then $\Theta_\Pi$ is isomorphic to the sub-system of $\Theta_{\Pi^{\mathsf{STR}}}$ induced by those $s \subseteq P_V$ where, for each $v \in V$, $s$ contains exactly one fact of the form $v = d$. All other states in $\Theta_{\Pi^{\mathsf{STR}}}$ are unreachable.*

**Observe:** $\Theta^+_\Pi$ has transition $s^+ \xrightarrow{a} s'^+$ if and only if $s^+[\![a^{\mathsf{STR}+}]\!] = s'^+$ in $\Pi^{\mathsf{STR}}$. (Because $s^+[\![a^{\mathsf{STR}+}]\!] = s^+ \cup \mathit{eff}_a$)

**Proposition.** *Denote by $h^*_\Pi$ and $h^+_\Pi$ the perfect heuristic and the optimal delete relaxation heuristic in $\Pi$, and denote by $h^*_{\Pi^{\mathsf{STR}}}$ and $h^+_{\Pi^{\mathsf{STR}}}$ these heuristics in $\Pi^{\mathsf{STR}}$. Then, for all states $s$ of $\Pi$, $h^*_\Pi(s) = h^*_{\Pi^{\mathsf{STR}}}(s)$ and $h^+_\Pi(s) = h^+_{\Pi^{\mathsf{STR}}}(s)$.*

→ Given an FDR task $\Pi$, *everything* we have done here can be done for $\Pi$ by doing it within $\Pi^{\mathsf{STR}}$.

### Summary

The delete relaxation simplifies STRIPS by removing all delete effects of the actions.
The cost of optimal relaxed plans yields the heuristic function h+, which is admissible but hard to compute.
We can approximate h + optimistically by h max, and pessimistically by h add . h max is admissible, h add is not. h add is typically much more informative, but can suffer from over-counting.
Either of h max or h add can be used to generate a closed well-founded best-supporter function, from which we can extract a relaxed plan.
The resulting relaxed plan heuristic h FF does not do over-counting, but otherwise has the same theoretical properties as h add; in practice, it typically does not overestimate h∗ .
The delete relaxation can be applied to FDR simply by accumulating variable values, rather than overwriting them. This is formally equivalent to treating variable/value pairs like STRIPS facts.