Lecture 4: Search in Structured States (Constraint Satisfaction Programming and Planning)

Viliam Lisý & **Branislav Bošanský**

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

bosansky@fel.cvut.cz

March, 2025

Overview

What we have covered so far:

- formal state representation
- uninformed search
- informed search and heuristics

We have not assumed that states of the world have some specific structure.

Question

What if we restrict the structure of the states?

Overview

Question

What if we restrict the structure of the states?

- – we lose generality (not every problem could be represented)
- + we gain performance (we will be able to solve much larger problems)
- + we will not have to come up with domain-specific heuristics

We can identify and solve (exactly!) instances of a subclass of problems and improve scalability by several orders of magnitude compared to standard search algorithms.

We start with Constraint Satisfaction Programming (CSPs) and then move to Logic and Classical Domain-Independent Planning.

Constraint Satisfaction Problems (CSPs)

CSPs are defined by 3 finite sets:

- **variables** $(x_1, x_2, \ldots, x_n)$
- **domains** (D_i for each variable x_i)
- **constraints** $(c_1, c_2, \ldots, c_m)$

A constraint is specified as a tuple of

- subset of variables $x_{j_1}, \ldots, x_{j_l}$
- all allowed joint assignments (l-tuples from $D_{j_1}, \ldots, D_{j_l}$)

State: (partial) assignment of values to variables

Action: assigning particular values to some variable

Goal: find such an assignment values to variables that satisfy all the constraints

Constraint Satisfaction Problems – Examples

Many problems can be represented as CPSs. These include known puzzles:

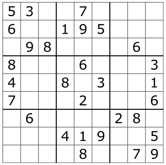
- Sudoku,
- Cryptaithmetic,

essential NP problems:

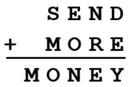
- SAT,
- Graph Coloring,

and many practical problems:

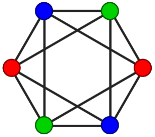
- Scheduling

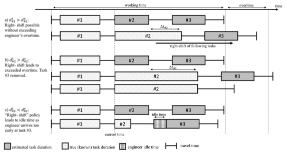


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



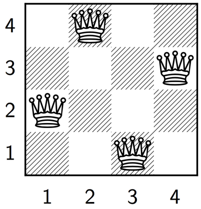
```
    S E N D
+   M O R E
-----------
  M O N E Y
```




working time
overtime
time
a) $d^t_{\#2} > d^e_{\#2}$:
Right- shift possible without exceeding engineer's overtime.
#1
#2
#3
$\Delta t_{\#2}$
right-shift of following tasks
b) $d^t_{\#2} > d^e_{\#2}$:
Right- shift leads to exceeded overtime. Task #3 removed.
c) $d^t_{\#2} < d^e_{\#2}$:
"Right- shift" policy leads to idle time as engineer arrives too early at task #3.
idle time
current time
estimated task duration
true (known) task duration
engineer idle time
travel time

N-Queens Example

Consider an N-Queens problem: Place on a chessboard of size $N \times N$ squares N queens so that no two queens threaten each other. For $N = 4$:



What would be the state representation?

- N coordinates (one tuple of coordinates for each queen)
- N numbers (every queen has to be in a different column, we can only represent rows)

Action changes position of one (or more) queen.

Small differences from previous (general) problems:

- there is no start state (we can start from any state), hence
- the path to the goal state is not interesting, only the goal state itself

N-Queens Example as a CSP

We can formulate N-Queens problem as a CSP:

- **variables**: $x_1, \ldots, x_N$ (one variable for each queen, queen i is placed in the i-th column)
- **domains**: $D_i = \{1, \ldots, N\}$ (the row in which the queen is placed)
- **constraints**:
  - $x_i \neq x_j \qquad \forall i, j \in \{1, \ldots, N\}, i \neq j$
    (some solvers support global constraint **alldifferent**($x_1, \ldots, x_N$))
  - $|x_i - x_j| \neq |i - j| \quad \forall i, j \in \{1, \ldots, N\}, i \neq j$

Question

How do we search for a solution?

Search Tree for CSPs

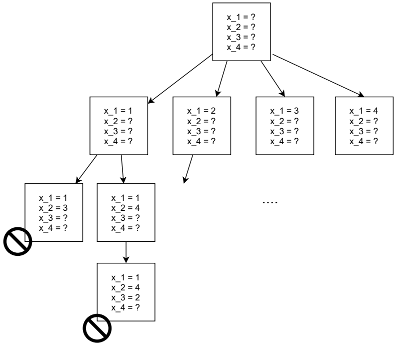
We use (uninformed) search as we know it (for now) and represent the search space as a search tree.

What are the nodes and actions in the search tree for a CSP?

- **Nodes** in the search tree – (partial) assignment of values to variables,
- **Edges** – choosing an unassigned variable and assign a value to this variable.

During the assignment, the algorithm must check whether the assignment does not violate constraints.

If there is no satisfying assignment, the algorithm backtracks.



Standard Representation of CSPs

We now move to specific CSP algorithms. Many of them assume only **binary constraints**.

Question

Is it a problem? Is it a subclass of CSP problems?

Not really, we can reformulate any k-ary constraint as a set of binary constraints:

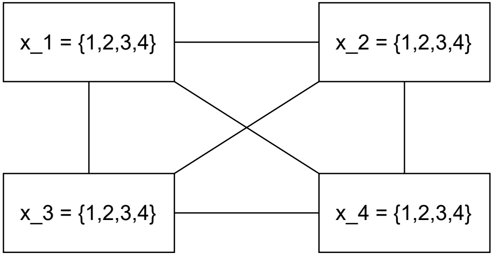
- Assume there is a constraint c involving k variables. Let Γ be the set of all k-tuples that satisfy this constraint.
- Create a new variable x_c with the domain Γ and create k binary constraints with involved k variables, such that i-th item of the value of x_c equals to value of the variable i.

Standard Representation of CSPs – Visualization

Having only binary constraints, we can visualize CSPs as graphs:

- variables are vertices in the graph,
- constraints are edges in the graph.

There is an edge connecting two vertices if there is a constraint between these variables.

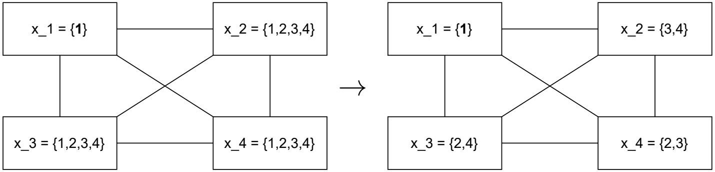


CSP Search and Propagating Constraints

Can we utilize the fact that we have a specific structure of the problem?

The simple search checks the constraints only in a passive way.

We can propagate the values to other variables. Every time we set a value for some variable, we can filter out values of other variables that do not satisfy constraints $\rightarrow$ **forward checking**:



CSP Search and Propagating Constraints

Assume the search algorithm selects a value for variable x_i. Now:

- for every other variable x_j such that there is a constraint c_{ij} between x_i and x_j, we evaluate all available values from D_j and keep only those that satisfy c_{ij}

How is the forward checking integrated into the search algorithm?

- the algorithm keeps available values for every variable
- if for any variable its domain is empty after the forward checking, the algorithm immediately backtracks

First heuristic $\rightarrow$ **minimal remaining value (MRV)**.

So far, there was no rule which variable to choose next in the search tree. MRV heuristics is a fail-fast heuristic that can quickly prune out dead-ends.

Search with Forward Checking

pseudocode of the search algorithm:

```
• if all variables are assigned then return current assignment (solution)
• $x_i \leftarrow \text{ChooseVariable}(X, D)$
• for each $v \in D_i$
    • assign $x_i = v$
    • valid $= \text{ForwardChecking}(X, D, i, v)$
    • if valid then search($X$,$D$)
    • undo local assignments
• return false
```

Towards a Better Use of Constraints

Forward checking ensures that there are supporting values in domains of other involved constraints.

The algorithm removes those values that do not satisfy the constraints.

But this can violate some other constraints ... Is there a way we can ensure that every constraint **can be satisfied** (termed **consistent)**?

Yes! We can have an algorithm that makes every edge (constraint) consistent.

Arc Consistency

Making one edge (arc) c_{ij} consistent:

```
• deleted = false
• for each $v \in D_i$
    • supported = false
    • for each $v' \in D_j$
        if $c_{ij}(v, v')$ then supported = true
    • if not supported then
        remove $v$ from $D_i$
        deleted = true
• return deleted
```

The procedure checks one constraint (in a directed manner) and returns true if some value was removed from domain D_i.

Arc Consistency – AC-3

Assume the algorithm has set value for variable x_i. We need to make consistent all incoming edges to node i (constraints that depend on this selected value). Next, if some value is removed from any variable x_j, we need to do the same for node j.

We will have a queue Q of all edges to make consistent:

```
• $Q = \{(j, i) \mid c_{ji} \in C,\ i \neq j\}$
• **while** Q is **not empty**
    • $(a, b) = \text{pop}(Q)$
    • **if** MakeConsistent$(a, b)$ **then**
          append$(Q, \{(k, a) \mid c_{ka} \in C,\ k \neq a\})$
```

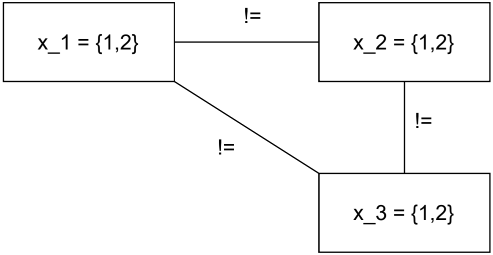
This algorithm is known as **AC-3**.

AC3 Algorithm

Question

Does AC3 solve everything? Do we still need search?

Unfortunately, AC3 is not able to guarantee there exists a solution. If AC3 prunes out some domain, the search algorithm can safely backtrack. Otherwise, the search needs to continue.



Other Interesting Search Improvements in CSP

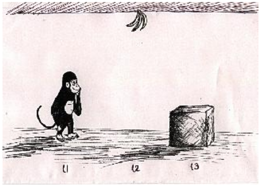
Least Constraining Value
Another heuristic for CSPs – among all the values to be assigned to a variable, choose such that supports the most other values.

Backjumping
Inability of choosing valid value for one variable can be caused by a choice of a variable up in the search tree. $\rightarrow$ The algorithm can identify which variables cause the conflict and can backtrack immediately to this conflicting variable (jumping back).

Dynamic Backtracking
In backjumping, the assignment between two conflicting variables is lost if we jump (even if it was a good one) $\rightarrow$ dynamic backtracking can dynamically choose which variable to assign (or re-assign) so that partially valid solutions are not lost.

From CSP to Planning

- Formulating a problem as a CSP can be too restrictive



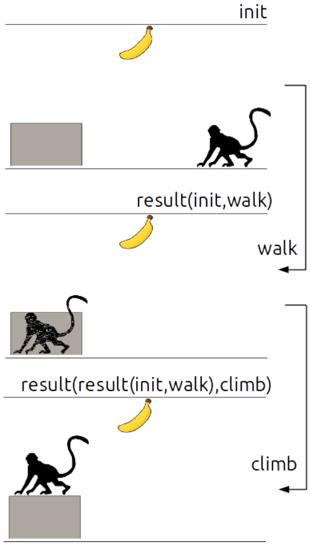
Vladimir Lifschitz: Planning course, The University of Texas at Austin.

What about a more general state representation? First-Order Logic (Situation Calculus):

- facts hold in particular situations ($\approx$ world state histories)
- predicates either rigid (eternal) or fluent (changing)
- fluent predicates include a situation argument
  e.g., $agent(monkey, at\_ban, now)$, term now denotes a situation
- rigid predicates hold regardless of a situation
  e.g., $walks(monkey)$, $moveable(box)$
- situations are connected by the *result* function
  if s is a situation than $result(s, a)$ is also a situation

Keeping track of evolving situations

agent(*agent name*, *agent position*, *stands on*, *situation*)
object(*object name*, *object position*, *who stands*, *situation*)



agent(monkey, right, ground, init).
object(box, left, none, init).

agent(monkey, left, ground, result(init,walk)).
object(box, left, none, result(init,walk)).

agent(monkey, left, box, result(result(init,walk),climb)).
object(box, left, monkey, result(result(init,walk),climb)).

Description and application of actions

$agent(agent\ name, agent\ position, stands\ on, situation)$
$object(object\ name, object\ position, who\ stands, situation)$

Action "effect" axiom for $walk(X, P_1, P_2)$:

$$\forall X, P_1, P_2, Z\ (agent(X, P_1, ground, Z) \land walks(X) \\ \rightarrow agent(X, P_2, ground, result(Z, walk(X, P_1, P_2)))$$

Action "effect" axiom for $climb(X)$:

$$\forall X, P, Z\ (agent(X, P, ground, Z) \land object(box, P, none, Z) \\ \rightarrow agent(X, P, box, result(Z, climb(X))) \\ \land object(box, P, X, result(Z, climb(X)))$$

Goal of planning: logical representation of the desired state

$$\mathcal{G} \equiv \quad \exists Z\ \ agent(monkey, middle, box, Z)$$

Domain Independent Automated Planning

While logic reasoning can work, the scalability is limited.

Is there a formal state representation specifically designed for solving deterministic sequential single-agent problems?

Yes, domain-independent (classical) planning – a subfield of AI dealing (mainly) with

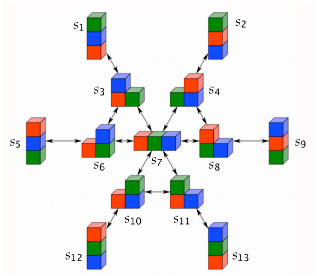
- representation languages with reasonable tradeoffs of expressivity and efficiency
- algorithms for finding plans for problems expressed in these languages

(The following slides are heavily based on Carmel Domshlak's slides)

Planning problems

What is in common?

- All these problems deal with action selection or control
- Some notion of problem state
- (Often) specification of initial state and/or goal state
- Legal moves or actions that transform states into other state



Succinct representation of transition systems

- More compact representation of actions than as relations is often
  - possible because of symmetries and other regularities,
  - unavoidable because the relations are too big.
- Represent different aspects of the world in terms of different state variables. $\rightsquigarrow$ A state is a valuation of state variables.
- Represent actions in terms of changes to the state variables.

Planning Languages

Key issue

Models represented **implicitly** in a **declarative language**

Play two roles

- **specification**: concise model description
- **computation**: reveal useful info about problem's *structure*

The STRIPS language

A problem in STRIPS is a tuple $\langle P, A, I, G \rangle$

- P stands for a finite set of **atoms** (boolean vars)
- $I \subseteq P$ stands for **initial situation**
- $G \subseteq P$ stands for **goal situation**
- A is a finite set of **actions** a specified via pre(a), add(a), and del(a), all subsets of P

- States are collections of atoms
- An action a is applicable in a state s iff pre$(a) \subseteq s$
- Applying an applicable action a at s results in $s' = (s \setminus \text{del}(a)) \cup \text{add}(a)$

Why STRIPS is interesting?

- STRIPS operators are **particularly simple**, yet expressive enough to capture general planning problems.
- In particular, STRIPS planning is **no easier** than general planning problems.
- Many algorithms in the planning literature are **easier to present in terms of STRIPS**.

(The following example is based on Antonin Komanda's slides)

Sokoban - Example planning domain

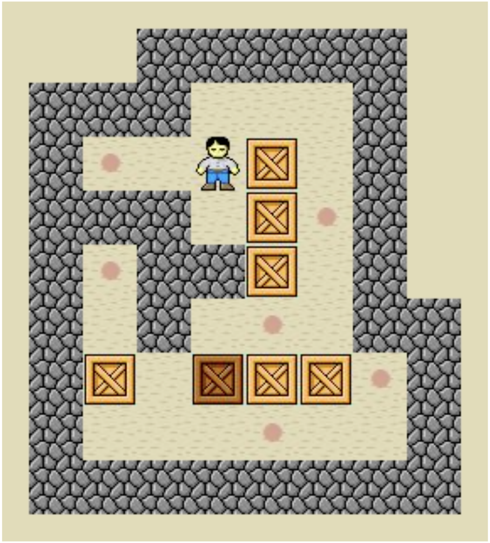
State representation:

```
positions: a1, ... a6,...
           f1, ..., f2
box_at(P), free(P)
player_at(P)
adjacent(P1,P2)
adjacent2(P1,P2)
```

Operators (Actions):

```
move(X,Y):
  pre: player_at(X)
       adjacent(X,Y)
       free(Y)
  add: player_at(Y)
  del: player_at(X)

push(X, Y, Z):
  pre:  player_at(X)
        box_at(Y)
        free(Z)
        adjacent(X,Y)
        adjacent(Y,Z)
        adjacent2(X,Z)
   ...
```



Grounding of Actions

Operators (Actions):

```
move(X,Y):
  pre: player_at(X)
       adjacent(X,Y)
       free(Y)
  add: player_at(Y)
  del: player_at(X)

push(X, Y, Z):
  pre:  player_at(X)
        box_at(Y)
        free(Z)
        adjacent(X,Y)
        adjacent(Y,Z)
        adjacent2(X,Z)
  add: player_at(Y)
        box_at(Z)
        free(Y)
  del: player_at(X)
        box_at(Y)
        free(Z)
```

Grounding:

```
move_a1_a2
  pre: player_at_a1, adjacent_a1_a2, free_a2
  add: player_at_a2
  del: player_at_a1

move_a2_a3
  pre: player_at_a2, adjacent_a2_a3, free_a3
  add: player_at_a3
  del: player_at_a2

...


push_a1_a2_a3
  pre: player_at_a1, box_at_a2, free_a3
         adjacent_a1_a2, adjacent_a2_a3,
         adjacent_a1_a3
  add: player_at_a2, box_at_a3, free_a2
  del: player_at_a1, box_at_a2, free_a3
...
```

STRIPS Representation of Sokoban

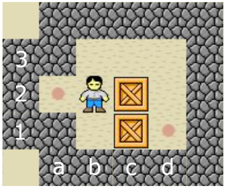
A problem in STRIPS is a tuple $\langle P, A, I, G \rangle$

- P stands for a finite set of **atoms** (boolean vars)
- $I \subseteq P$ stands for **initial situation**
- $G \subseteq P$ stands for **goal situation**
- A is a finite set of **actions** a specified via pre(a), add(a), and del(a), all subsets of P

```
P = {player_at_a2, ..., player_at_d3,
     box_at_a2, ..., box_at_d3,
     free_a2, ..., free_d3,
     adjacent_a2_b2, ..., adjacent_d2_d3,
     adjacent2_a2_c2, ..., adjacent2_d1_d3 }

I = {player_at_b2, box_at_c1, box_at_c2,
     free_a2, free_b1, ..., free_d3,
     adjacent_a2_b2,..., adjacent_d2_d3, adjacent2_a2_c2,..., adjacent2_d1_d3}

G = {box_at_a2, box_at_d1}
```



Planning in Strips

We can just use A*:

- State: a set of true atoms
- Applicable actions: based on preconditions
- Action application: add the “add” atoms and delete the “del” atoms

  (No need for separate simulator implementation)

Problem structure allows **automated** construction of **heuristics**!

- Allows exploring general heuristics domain independently
- Simple heuristic: $h(s) = |G \setminus s|$
- Solve a suitable **simpler** version of the problem
- Abstraction: solve a smaller problem

  e.g., completely remove a predicate from the problem
- Landmarks
- **Relaxation**: solve a less constraint problem

Relaxation heuristics

Whole sub-field of planning in STRIPs and beyond

- Relaxation is a general technique for heuristic design:
    - Straight-line heuristic (route planning): Ignore the fact that one must stay on roads.
    - Manhattan heuristic (15-puzzle): Ignore the fact that one cannot move through occupied tiles.
- We want to apply the idea of relaxations to planning.
- Informally, we want to ignore bad side effects of applying actions.

Example (8-puzzle)

If we move a tile from x to y, then the good effect is (in particular) that x is now free.
The bad effect is that y is not free anymore, preventing us for moving tiles through it.

Relaxed planning tasks in STRIPS

In STRIPS, good and bad effects are easy to distinguish:

- Effects that make atoms true are good (add effects).
- Effects that make atoms false are bad (delete effects).

Idea for the heuristic: Ignore all delete effects.

Relaxed planning tasks in STRIPS

Definition (relaxation of actions)

The relaxation a^+ of a STRIPS action $a = \langle \mathsf{pre}(a), \mathsf{add}(a), \mathsf{del}(a) \rangle$ is the action $a^+ = \langle \mathsf{pre}(a), \mathsf{add}(a), \emptyset \rangle$.

Definition (relaxation of planning tasks)

The relaxation Π^+ of a STRIPS planning task $\Pi = \langle P, A, I, G \rangle$ is the planning task $\Pi^+ := \langle P, \{a^+ \mid a \in A\}, I, G \rangle$.

Definition (relaxation of action sequences)

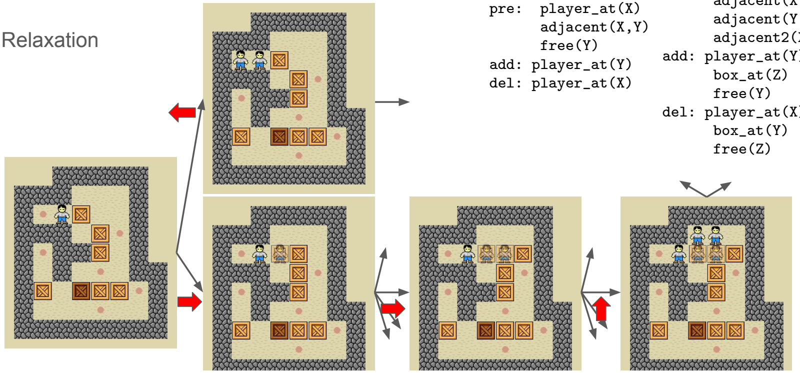
The relaxation of an action sequence $\pi = a_1 \dots a_n$ is the action sequence $\pi^+ := {a_1}^+ \dots {a_n}^+$.

Relaxation of actions in Sokoban

Relaxation

```
move(X,Y):
  pre:   player_at(X)
         adjacent(X,Y)
         free(Y)
  add: player_at(Y)
  del: player_at(X)
```

```
push(X, Y, Z):
  pre:   player_at(X)
         box_at(Y)
         free(Z)
         adjacent(X,Y)
         adjacent(Y,Z)
         adjacent2(X,Z)
  add: player_at(Y)
         box_at(Z)
         free(Y)
  del: player_at(X)
         box_at(Y)
         free(Z)
```



Building Relaxed Planning Graph

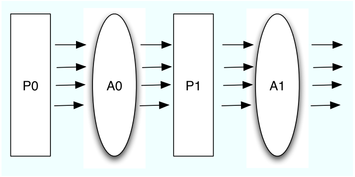
Solving a relaxed problem gives us a heuristic estimate h^+ of the original problem.

However, computing the optimal relaxed plan is still NP hard

But we can do something simpler

- Build a layered reachability graph $P_0, A_0, P_1, A_1, \dots$

$$\begin{aligned} P_0 &= \{p \in I\} \\ A_i &= \{a \in A \mid \mathsf{pre}(a) \subseteq P_i\} \\ P_{i+1} &= P_i \cup \{p \in \mathsf{add}(a) \mid a \in A_i\} \end{aligned}$$



- Terminate when $G \subseteq P_i$

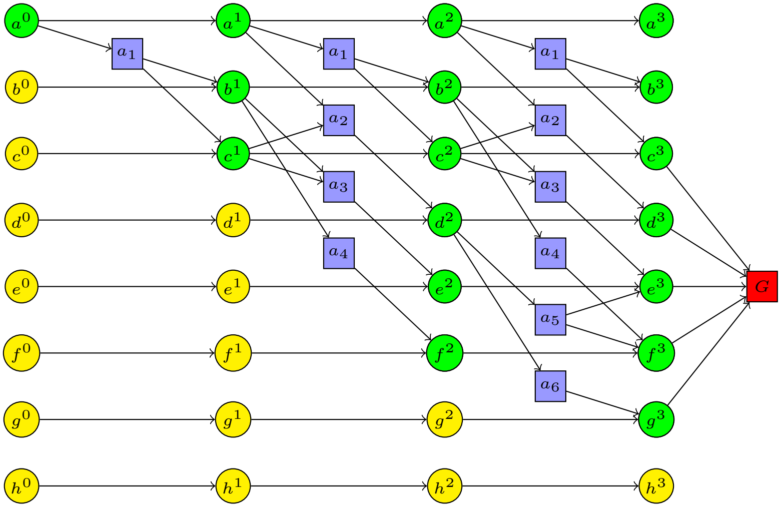
Example

$$I = \{a = 1, b = 0, c = 0, d = 0, e = 0, f = 0, g = 0, h = 0\}$$

$$a_1 = \langle\{a\}, \{b, c\}, \emptyset\rangle$$
$$a_2 = \langle\{a, c\}, \{d\}, \emptyset\rangle$$
$$a_3 = \langle\{b, c\}, \{e\}, \emptyset\rangle$$
$$a_4 = \langle\{b\}, \{f\}, \emptyset\rangle$$
$$a_5 = \langle\{d\}, \{g\}, \emptyset\rangle$$

$$G = \{c = 1, d = 1, e = 1, f = 1, g = 1\}$$

Relaxed Planning Graph

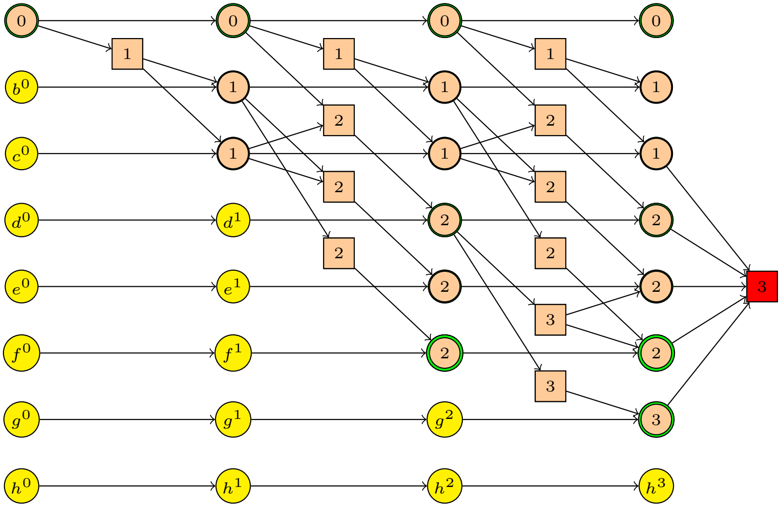


Domain Independent Automated Planning

Forward cost heuristic h_{max}

- Propagate cost layer by layer from start to goal
- At actions, take maximum cost of achieving preconditions $+1$
- At propositions, take the cheapest action to achieve it

Computing heuristic h_{max}



Summary

This concludes search in single-agent deterministic problems.

Next-up: reinforcement learning, games, dealing with uncertainty.