AI Classical and Non-deterministic Planning: Model-based Autonomous Behavior

Sebastian Sardiña

School of Computing Technologies
RMIT University

Julio 28 - Agosto 1 2025

RMIT UNIVERSITY

ECI38
Escuela de Ciencias Informáticas

Part I

Classical Planning: Languages

Part 1: Classical Planning: Languages

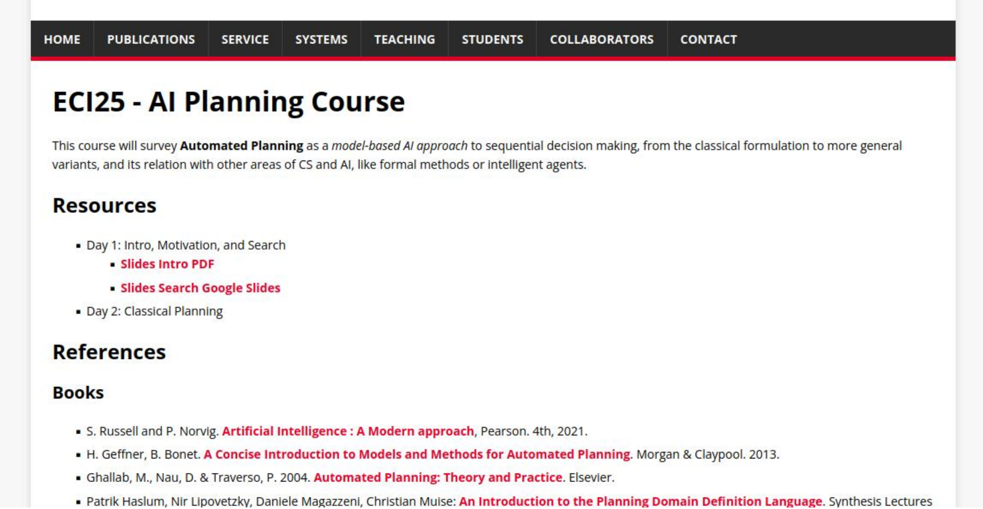
Part 1: Classical Planning: Languages

1 Motivation

2 State Models and Search

3 Planning Languages

Course Web Page



HOME | PUBLICATIONS | SERVICE | SYSTEMS | TEACHING | STUDENTS | COLLABORATORS | CONTACT

ECI25 - AI Planning Course

This course will survey **Automated Planning** as a *model-based AI approach* to sequential decision making, from the classical formulation to more general variants, and its relation with other areas of CS and AI, like formal methods or intelligent agents.

Resources

- Day 1: Intro, Motivation, and Search
  - **Slides Intro PDF**
  - **Slides Search Google Slides**
- Day 2: Classical Planning

References

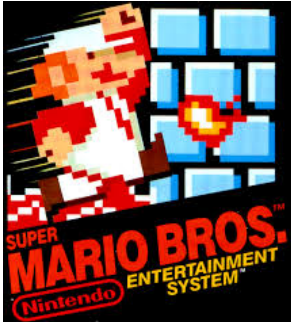
Books

- S. Russell and P. Norvig. **Artificial Intelligence : A Modern approach**, Pearson. 4th, 2021.
- H. Geffner, B. Bonet. **A Concise Introduction to Models and Methods for Automated Planning**. Morgan & Claypool. 2013.
- Ghallab, M., Nau, D. & Traverso, P. 2004. **Automated Planning: Theory and Practice**. Elsevier.
- Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, Christian Muise: **An Introduction to the Planning Domain Definition Language**. Synthesis Lectures

https://ssardina.github.io/courses/eci25/

Beating Kasparov is great...

Beating Kasparov is great . . . but how to play Mario?



- You (and your brother/sister/little nephew) are better than Deep Blue at **everything** - except playing Chess.

? Is that (artificial) 'Intelligence'?

➠ How to build machines that automatically solve **new** problems?

How to develop systems or "agents" that can make decisions on their own?

Autonomous Behavior in AI

💡 Key problem is to select **the action to do next**. This is the so-called "control problem".

Three mainstream approaches to action selection

1. **Behavior-based:** Set of independent simple reactive modules.
   - ✏ *Brook's subsumption architecture (80')*
2. **Programming-based:** Specify control by hand
   - ✏ *Agent-oriented programming (e.g., PRS, JACK, 3APL, SARL)*
3. **Learning-based:** Learn control from experience
   - ✏ *Reinforcement Learning; Evolutionary algorithms*
4. **Model-based:** Specify problem by hand, derive control automatically
   - ✏ *Automated Planning, Model Predictive Control*

Note:

- Approaches not orthogonal; successes and limitations in each ...
- Different models yield different types of controllers ...

Programming-Based Approach

Control specified by programmer, e.g.:

- If Mario finds no danger, then run...
- If danger appears and Mario is big, jump and kill ...
- ...

✓ Advantage: domain-knowledge easy to express.

✖ Disadvantage: cannot deal with situations not anticipated by programmer.

Learning-Based Approach

Learns a controller from experience or through simulation:

- **Unsupervised** (Reinforcement Learning):
  - penalize Mario each time that 'dies'
  - reward agent each time oponent 'dies' and level is finished, ...
- **Supervised** (Classification)
  - learn to classify actions into good or bad from info provided by teacher
- **Evolutionary**:
  - from pool of possible controllers: try them out, select the ones that do best, and mutate and recombine for a number of iterations, keeping best

✓ Advantage: does not require much knowledge in principle.

✖ Disadvantage: in practice, hard to know which features to learn, and is slow.

General Problem Solving

Ambition: Write one program that can solve all problems.

- Write $X \in \{\text{“algorithms”}\} : \text{for all } Y \in \{\text{“problems”}\} : X \text{ “solves” } Y$
- What is a “problem”? What does it mean to “solve” it?

Ambition 2.0: Write one program that can solve a large class of problems.

Ambition 3.0: Write one program that can solve a large class of problems effectively.

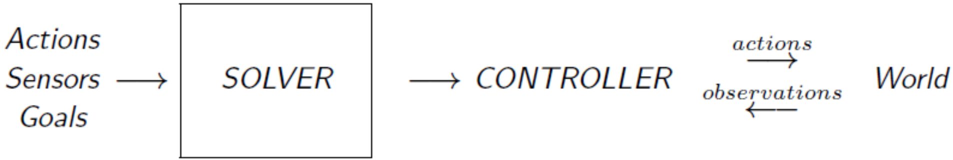
(some new problem) $\leadsto$ (describe problem $\rightarrow$ use off-the-shelf solver) $\leadsto$ (solution competitive with a human-made specialized program)

💪 Beat humans at coming up with clever solution methods!

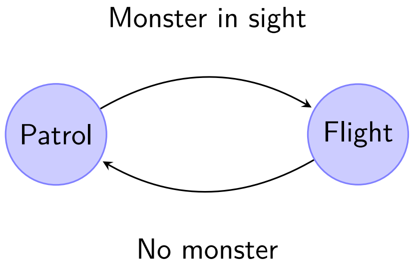
(Link: GPS started on 1959)

Model-Based Approach / General Problem Solving

1. specify model for problem: actions, initial situation, goals, and sensors; and
2. let a solver compute controller automatically.



Programming *vs.* Planning



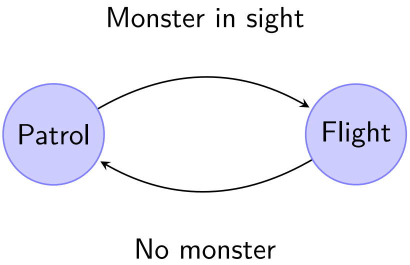
vs

Actions available:

1. **Patrol**:
   - Preconditions: No Monster
   - Effects: patrolled
2. **Fight**:
   - Preconditions: Monster in sight
   - Effects: No Monster

Goal: area patrolled

Programming *vs.* Planning



vs

Actions available:

1. **Patrol**:
   - Preconditions: No Monster
   - Effects: patrolled
2. **Fight**:
   - Preconditions: Monster in sight
   - Effects: No Monster

Goal: area patrolled

Model-Based Approach / General Problem Solving

✓ Advantages

- Powerful: In some applications generality is absolutely necessary.
- Quick: Rapid prototyping. 10s lines of problem description vs. 1000s lines of C++ code. (Language generation!)
- Flexible & Clear: Adapt/maintain the description.
- Intelligent & domain-independent: Determines automatically how to solve a complex problem effectively!

Model-Based Approach / General Problem Solving

✓ Advantages

- Powerful: In some applications generality is absolutely necessary.
- Quick: Rapid prototyping. 10s lines of problem description vs. 1000s lines of C++ code. (Language generation!)
- Flexible & Clear: Adapt/maintain the description.
- Intelligent & domain-independent: Determines automatically how to solve a complex problem effectively!

✖ Disadvantages

- Need a model: Without knowledge about Chess, you don't beat Kasparov ...
- Computationally intractable: at leat NP-hard!

Model-Based Approach / General Problem Solving

✓ Advantages

- Powerful: In some applications generality is absolutely necessary.
- Quick: Rapid prototyping. 10s lines of problem description vs. 1000s lines of C++ code. (Language generation!)
- Flexible & Clear: Adapt/maintain the description.
- Intelligent & domain-independent: Determines automatically how to solve a complex problem effectively!

✖ Disadvantages

- Need a model: Without knowledge about Chess, you don't beat Kasparov …
- Computationally intractable: at leat NP-hard!

🤝 **Trade-off** between "automatic and general" vs. "manual work but effective".

Model-based approach to intelligent behavior called "**Planning**" in AI.

? **How to make fully automatic algorithms effective?**

What is "planning"?

Patrik Haslum

"Planning is the art and practice of thinking before acting: of reviewing the courses of action one has available and predicting their expected (and unexpected) results to be able to choose the course of action most beneficial with respect to one's goals."

What is "planning"?

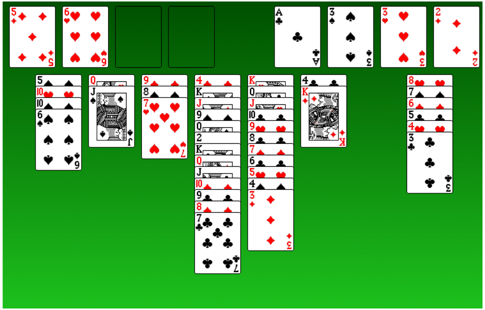
Patrik Haslum

"Planning is the art and practice of thinking before acting: of reviewing the courses of action one has available and predicting their expected (and unexpected) results to be able to choose the course of action most beneficial with respect to one's goals."

Belief-Desire-Intention (BDI) model of agency - (Bratman '87)

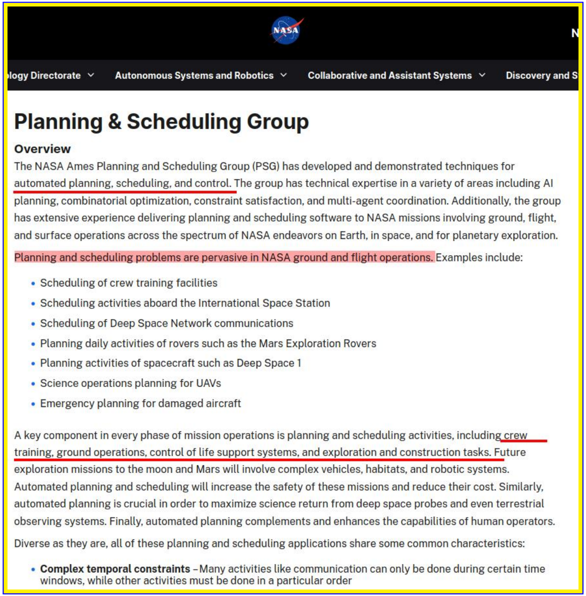
Rational behavior arises due to the agent committing to ***some of its desires****, and selecting actions that achieve its intentions given its* ***beliefs****.*

Example: Classical Search Problem



- States: Card positions (position Jspades=Qhearts).
- Actions: Card moves (move Jspades Qhearts freecell4).
- Initial state: Start configuration.
- Goal states: All cards 'home'.
- Solution: Card moves solving this game.

Applications of Planning: Space



logy Directorate | Autonomous Systems and Robotics | Collaborative and Assistant Systems | Discovery and S

Planning & Scheduling Group

Overview

The NASA Ames Planning and Scheduling Group (PSG) has developed and demonstrated techniques for automated planning, scheduling, and control. The group has technical expertise in a variety of areas including AI planning, combinatorial optimization, constraint satisfaction, and multi-agent coordination. Additionally, the group has extensive experience delivering planning and scheduling software to NASA missions involving ground, flight, and surface operations across the spectrum of NASA endeavors on Earth, in space, and for planetary exploration.

Planning and scheduling problems are pervasive in NASA ground and flight operations. Examples include:

- Scheduling of crew training facilities
- Scheduling activities aboard the International Space Station
- Scheduling of Deep Space Network communications
- Planning daily activities of rovers such as the Mars Exploration Rovers
- Planning activities of spacecraft such as Deep Space 1
- Science operations planning for UAVs
- Emergency planning for damaged aircraft

A key component in every phase of mission operations is planning and scheduling activities, including crew training, ground operations, control of life support systems, and exploration and construction tasks. Future exploration missions to the moon and Mars will involve complex vehicles, habitats, and robotic systems. Automated planning and scheduling will increase the safety of these missions and reduce their cost. Similarly, automated planning is crucial in order to maximize science return from deep space probes and even terrestrial observing systems. Finally, automated planning complements and enhances the capabilities of human operators.

Diverse as they are, all of these planning and scheduling applications share some common characteristics:

- **Complex temporal constraints** – Many activities like communication can only be done during certain time windows, while other activities must be done in a particular order

AI in Space

Editor: Richard Doyle
Jet Propulsion Lab
rdoyle@jpl.nasa.gov

MAPGEN: Mixed-Initiative Planning and Scheduling for the Mars Exploration Rover Mission

Mitchell Ai-Chang, John Bresina, Len Charest, Adam Chase, Jennifer Cheng-jung Hsu, Ari Jonsson, Bob Kanefsky, Paul Morris, Kanna Rajan, Jeffrey Yglesias, Brian G. Chafin, William C. Dias, and Pierre F. Maldague, *NASA Ames Research Center and the Jet Propulsion Laboratory*

The Mars Exploration Rover mission is one of NASA's most ambitious science missions to date. Launched in the summer of 2003, each rover carries instruments for conducting remote and in situ observations to elucidate the planet's past climate, water activity, and habitability.

Science is MER's primary driver, so making best use of the scientific instruments, within the available resources, is a crucial aspect of the mission. To address this criticality, the MER project team selected MAPGEN (*M*ixed Initiative Activity *P*lan *Gen*erator) as an activity-planning tool. MAPGEN combines two existing systems, each with a strong heritage: the APGEN activity-planning tool[1] from the Jet Propulsion Laboratory and the Europa planning and scheduling system[2] from NASA Ames Research Center. This article discusses the issues arising from combining these tools in this mission's context.

Combining systems

In a most exciting development, two NASA rovers—named Spirit and Opportunity—were slated to arrive at the Red Planet in January, at two scientifically distinct sites. (Spirit arrived successfully on 3 January, with Opportunity scheduled to arrive 24 January—see Figures 1 and 2.) Each rover will have an operational lifetime of 90 sols (Martian days) or more and can traverse an integrated distance of one kilometer or more, although the maximum range from the landing site might be less. Scientifically, MER seeks to

- Determine the aqueous, climatic, and geologic history of a site where on Mars conditions might have been

Applications of Planning: Machine Control

On-line Planning and Scheduling: An Application to Controlling Modular Printers

Wheeler Ruml RUML AT CS.UNH.EDU
Department of Computer Science
University of New Hampshire
33 Academic Way
Durham, NH 03824 USA

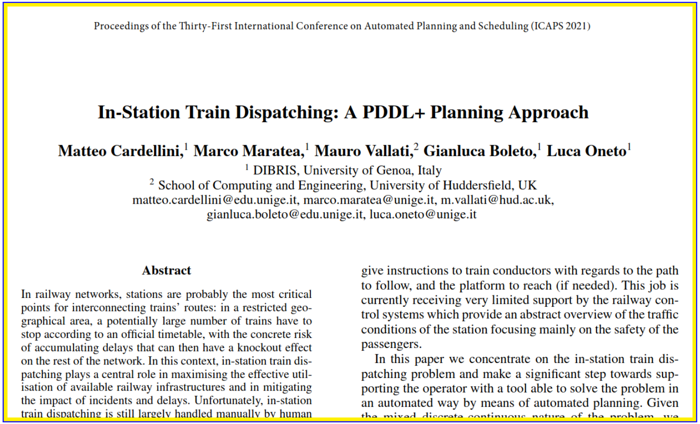
Minh Binh Do MINHDO AT PARC.COM
Rong Zhou RZHOU AT PARC.COM
Markus P. J. Fromherz FROMHERZ AT PARC.COM
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA

Abstract

We present a case study of artificial intelligence techniques applied to the control of production printing equipment. Like many other real-world applications, this complex domain requires high-speed autonomous decision-making and robust continual operation. To our knowledge, this work represents the first successful industrial application of embedded domain-independent temporal planning. Our system handles execution failures and multi-objective preferences. At its heart is an on-line algorithm that combines techniques from state-space planning and partial-order scheduling. We suggest that this general architecture may prove useful in other applications as more intelligent systems operate in continual, on-line settings. Our system has been used to drive several commercial prototypes and has enabled a new product architecture for our industrial partner. When compared with state-of-the-art off-line planners, our system is hundreds of times faster and often finds better plans. Our experience demonstrates that domain-independent AI planning based on heuristic search can flexibly handle time, resources, replanning, and multiple objectives in a high-speed practical application without requiring hand-coded control knowledge.

Figure 1: A prototype modular printer built at PARC. The system is composed of approximately 170 individually controlled modules, including four print engines.

Applications of Planning: Train Dispatching


Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)

In-Station Train Dispatching: A PDDL+ Planning Approach

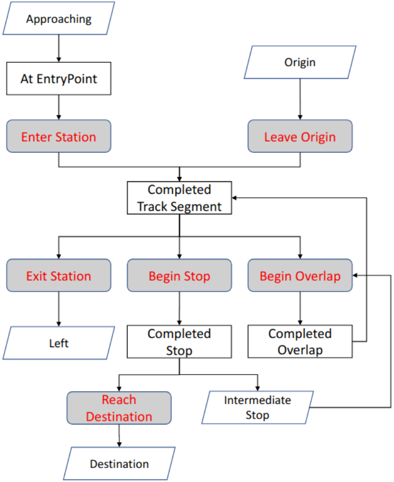
Matteo Cardellini,[1] Marco Maratea,[1] Mauro Vallati,[2] Gianluca Boleto,[1] Luca Oneto[1]
[1] DIBRIS, University of Genoa, Italy
[2] School of Computing and Engineering, University of Huddersfield, UK
matteo.cardellini@edu.unige.it, marco.maratea@unige.it, m.vallati@hud.ac.uk,
gianluca.boleto@edu.unige.it, luca.oneto@unige.it

Abstract

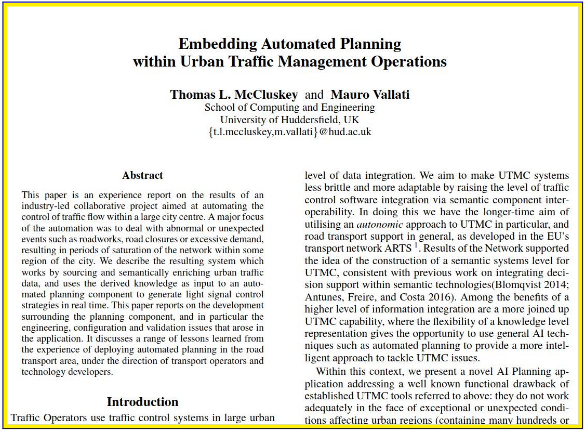
In railway networks, stations are probably the most critical points for interconnecting trains' routes: in a restricted geographical area, a potentially large number of trains have to stop according to an official timetable, with the concrete risk of accumulating delays that can then have a knockout effect on the rest of the network. In this context, in-station train dispatching plays a central role in maximising the effective utilisation of available railway infrastructures and in mitigating the impact of incidents and delays. Unfortunately, in-station train dispatching is still largely handled manually by human

give instructions to train conductors with regards to the path to follow, and the platform to reach (if needed). This job is currently receiving very limited support by the railway control systems which provide an abstract overview of the traffic conditions of the station focusing mainly on the safety of the passengers.

In this paper we concentrate on the in-station train dispatching problem and make a significant step towards supporting the operator with a tool able to solve the problem in an automated way by means of automated planning. Given



Applications of Planning: Traffic Light Control



Embedding Automated Planning within Urban Traffic Management Operations

Thomas L. McCluskey and **Mauro Vallati**
School of Computing and Engineering
University of Huddersfield, UK
{t.l.mccluskey,m.vallati}@hud.ac.uk

Abstract

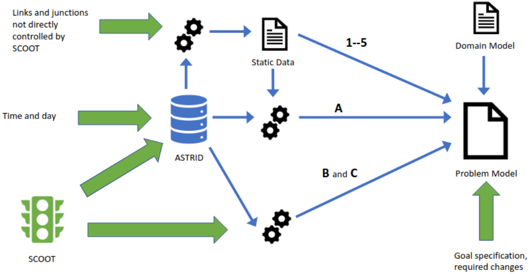
This paper is an experience report on the results of an industry-led collaborative project aimed at automating the control of traffic flow within a large city centre. A major focus of the automation was to deal with abnormal or unexpected events such as roadworks, road closures or excessive demand, resulting in periods of saturation of the network within some region of the city. We describe the resulting system which works by sourcing and semantically enriching urban traffic data, and uses the derived knowledge as input to an automated planning component to generate light signal control strategies in real time. This paper reports on the development surrounding the planning component, and in particular the engineering, configuration and validation issues that arose in the application. It discusses a range of lessons learned from the experience of deploying automated planning in the road transport area, under the direction of transport operators and technology developers.

Introduction

Traffic Operators use traffic control systems in large urban

level of data integration. We aim to make UTMC systems less brittle and more adaptable by raising the level of traffic control software integration via semantic component interoperability. In doing this we have the longer-time aim of utilising an *autonomic* approach to UTMC in particular, and road transport support in general, as developed in the EU's transport network ARTS [1]. Results of the Network supported the idea of the construction of a semantic systems level for UTMC, consistent with previous work on integrating decision support within semantic technologies(Blomqvist 2014; Antunes, Freire, and Costa 2016). Among the benefits of a higher level of information integration are a more joined up UTMC capability, where the flexibility of a knowledge level representation gives the opportunity to use general AI techniques such as automated planning to provide a more intelligent approach to tackle UTMC issues.

Within this context, we present a novel AI Planning application addressing a well known functional drawback of established UTMC tools referred to above: they do not work adequately in the face of exceptional or unexpected conditions affecting urban regions (containing many hundreds or



Applications of Planning: UAVs and UGVs


Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

ELSEVIER

IFAC PapersOnLine 56-2 (2023) 7873–7879

Automated Planning for Inspection and Maintenance operations using Unmanned Ground Vehicles ⋆

M.A. Hinostroza * Anastasios M. Lekkas *
Aksel A. Transeth ** Bjørnar Luteberget **
Christian de Jonge *** Svein Ivar Sagatun ***

* Department of Engineering Cybernetics, Norwegian University of Science and Technology, Trondheim, Norway.
(e-mail: {miguel.hinostroza, anastasios.lekkas} @ntnu.no)
** SINTEF Digital, Mathematics and Cybernetics, Trondheim, Norway.
(e-mail: {aksel.a.transeth, bjornar.luteberget} @sintef.no)
*** Equinor, Norway.
(e-mail: {chjo; svesa} @equinor.com)

Abstract: Offshore oil and gas industry has a strong incentive to improve its traditional operations and move towards more remote controlled and automated installations. This allows for improved efficiency, reduced cost and improved quality, and safety by removing personnel out of harm's way. The use of Unmanned Ground Vehicles (UGVs) in these upcoming platforms, is relevant for Inspection and Maintenance (I&M) operations. Traditionally, UGVs are used only for pre-defined tasks and have no capabilities for replanning, if a new task is required or any unexpected event occurs. This paper presents a novel concept for I&M operations using automated planning for UGVs. The automated planner is based on a temporal planning algorithm, and considers actions related to, for example, visiting a specific waypoint, inspect a sensor or manipulate an actuator. Also, the proposed system allows to perform replanning in case of any specific location needs to be revisited or a path is blocked. In addition, we couple the mission planner with a UGV guidance, navigation and control system, which has path planning, path following and control capabilities. To assess the performance of the proposed system, an use case for I&M operations on board of an oil and gas platform was simulated and promising results were obtained.

Keywords: Automated planning, maintenance and inspection, oil and gas platform, unmanned ground vehicle.

1. INTRODUCTION

Offshore oil and gas platforms are often located in remote and distant places and may pose a challenging environment for personnel due to the exposure to potential hazardous or harmful chemicals, work in areas exposed for weather and on smaller installations with hydrocarbons

- periodic or on-demand acoustic inspection using directional sound looking for anomalies or vibrations;
- thermal (using infrared) inspection of electrical equipment, process equipment and heated surfaces to look for leaks, anomalies in temperature;
- thermal (using infrared) for detection of small (fugitive) gas leaks and monitoring of these;

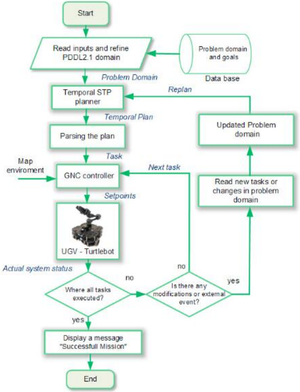


Fig. 2. Algorithm flow chart of proposed system

The 3D model and plant description was recently released under open-source license by Equinor[1] for research and innovation developments. In order to perform numerical simulations, the plant was simplified as can be seen in Fig. 3b, additionally a Gazebo map was created in Fig. 3c to perform simulations in ROS, where 1 grid map is equal to 1m.

3.2 Vehicle: Turtlebot3 UGV

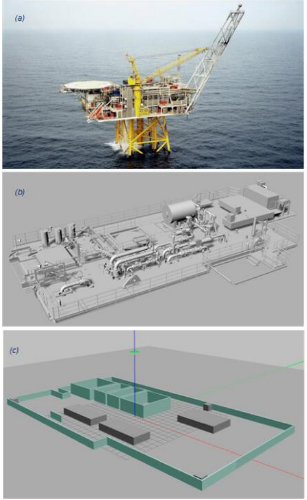
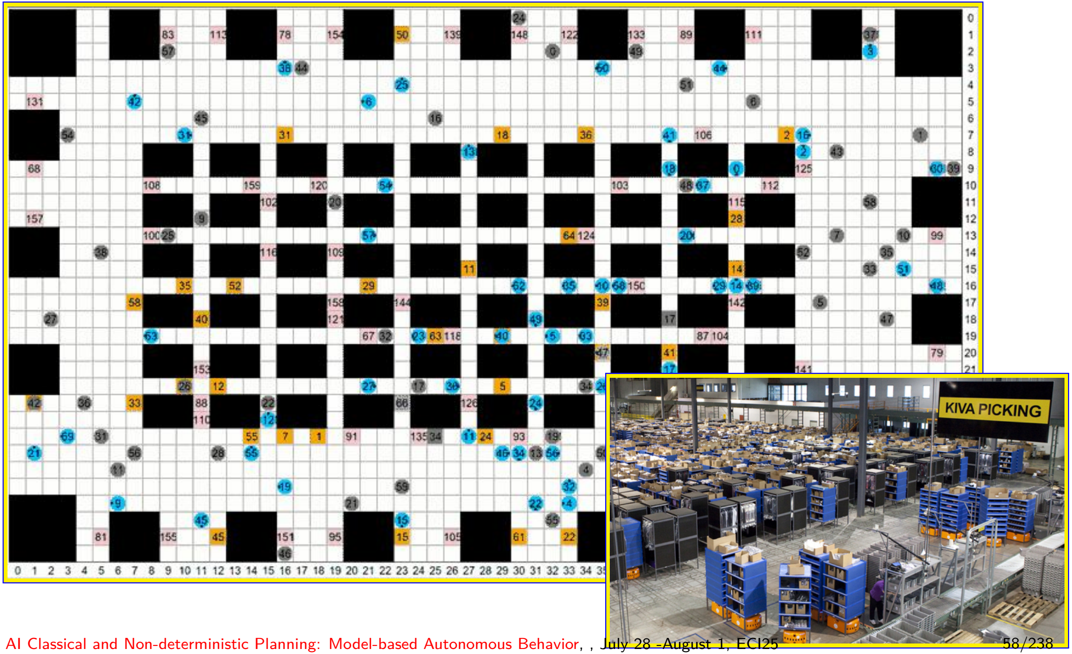


Fig. 3. (a) Huldra oil and gas offshore platform (Courtesy of Equinor), (b) Upper-layer of Huldra, (c) Simplified ROS gazebo map.

Applications of Planning: MAPF



Applications of Planning: Others...

Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)

Combining Heuristic Search and Linear Programming to Compute Realistic Financial Plans

Alberto Pozanco, Kassiani Papasotiriou, Daniel Borrajo*, Manuela Veloso
J.P. Morgan AI Research
{alberto.pozancolancho, kassiani.papasotiriou, daniel.borrajo, manuela.veloso}@jpmorgan.com

Abstract

Defining financial goals and formulating actionable plans to achieve them are essential components for ensuring financial health. This task is computationally challenging, given the abundance of factors that can influence one's financial situation. In this paper, we present the Personal Finance Planner (PFP), which can generate personalized financial plans that consider a person's context and the likelihood of taking financially related actions to help them achieve their goals. PFP solves the problem in two stages. First, it uses heuristic search to find a high-level sequence of actions that increase the income and reduce spending to help users achieve their financial goals. Next, it uses integer linear programming to determine the best low-level actions to implement the high-level plan. Results show that PFP is able to scale on generating realistic financial plans for complex tasks involving many low level actions and long planning horizons.

Introduction

ietting financial goals and planning ahead are crucial for ıchieving financial health whether for individuals, houseıolds or companies. For individuals, financial planning in-

do not provide detailed solutions (i.e., plans with monthly actions). They also do not consider the feasibility of the recommended plans based on the user financial habits.

In this paper we present the Personal Finance Planner (PFP), which generates realistic plans that achieve users' financial goals. Due to the large action space, (i.e., there is a potentially great number of income and expenses sources), PFP solves the problem hierarchically in two stages, by exploiting the task's structure. First, it uses heuristic search to find a high-level sequence of income increase and spending decrease actions at each month that achieve the financial goal. Then, it uses integer linear programming (ILP) to decide how to implement the prescribed high-level plan by composing the right low-level actions to be applied at each month. In this paper, we primarily focus on personal finance planning. But our framework can also be applied to assist with financial planning tasks for households and companies.

Financial Planning Tasks

We aim to find realistic plans that allow users to transit from their current financial state to a state that fulfills their

Applications of Planning: Others...

Proceedings of the Thirty-Th

Combining Heuristic S

Alberto Pozanco,

{alberto.pozancolancho

Abstract

Defining financial goals and formulatin
achieve them are essential components f
health. This task is computationally ch
abundance of factors that can influence
ation. In this paper, we present the Pers
(PFP), which can generate personalized
consider a person's context and the f
financially related actions to help them
PFP solves the problem in two stages. F
search to find a high-level sequence of
the income and reduce spending to help
financial goals. Next, it uses integer lin
determine the best low-level actions to
level plan. Results show that PFP is able
ing realistic financial plans for complex t
low level actions and long planning hori

Introduction

etting financial goals and planning a
chieving financial health whether fo
olds or companies. For individuals, f

Scaling Web API Integrations

Guido Chari, Brandon Sheffer, S.R.K Branavan, Nicolás D'ippolito
ASAPP

***Abstract*—In ASAPP, a company that offers AI solutions to enterprise customers, internal services consume data from our customers' web APIs. Implementing and maintaining integrations between our customers' APIs and internal services is a major effort for the company. In this paper, we present a scalable approach for integrating web APIs in enterprise software that is lightweight and semi-automatic. It leverages a combination of Ontology-Based Data Access architectures (OBDA), a Domain Specific Language (DSL) called IBL, Natural Language Processing (NLP) models, and Automated Planning techniques. The OBDA architecture decouples our platform from our customers' APIs via an ontology that acts as a single internal data access point. IBL is a functional and graphical DSL that enables domain experts to implement integrations, even if they don't have software development expertise. To reduce the effort of manually writing the IBL code, an NLP model suggests correspondences from each web API to the ontology. Given the API, ontology, and selected mappings for a set of desired fields from the ontology, we define an Automated Planning problem. The resulting policy is finally fed to a code synthesizer that generates the appropriate IBL method implementing the desired integration.**

This approach has been in production in ASAPP for 2 years with more than 300 integrations already implemented. Results indicate a ≈ 50% reduction in effort due to implementing integrations with IBL. Preliminary results on the IBL automatic code generation show an encouraging further ≈ 25% reduction so far.

I. Introduction

The process of exchanging heterogeneous data between multiple systems is known as integration [29]. The exchange consists of consuming structured data under a `source` schema and instantiating a `target` schema that reflects the

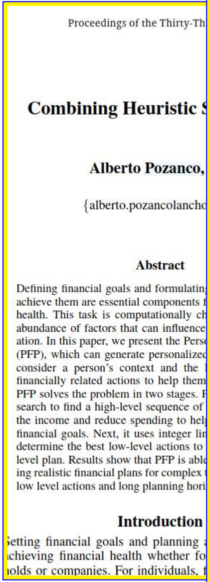
In this paper, we present a lightweight and semi-automated approach to integrating web APIs, with a focus on reducing the time and effort required. The approach was designed based on constraints observed at ASAPP, an AI company that sells products and services to enterprise customers. We model our approach to meet the following desired attributes:

a) The approach should enable complete decoupling between internal systems and customers' APIs
b) It should enable domain experts, who may not be professional software developers, to specify the mapping and allow for editing of high-level source code when necessary
c) It should allow for integrations to be exhaustively tested or proven correct before deployment.

To honor these constraints, we first design our approach around an Ontology-Based Data Access (OBDA) architecture. Ontology-Based Data Access (OBDA) is a common strategy for integrating data stored in databases [36]. OBDA provides access to heterogeneous data through the mediation of a single ontology that end users can query. A mapping specifies how to reconstruct the data stored in the sources in terms of this ontology. Leveraging on the mapping, OBDA implementations can automatically rewrite a query issued on the ontology into queries against the respective source table(s). We adapted the approach to the web API domain.

We then leverage a machine-learning model that suggests candidate mappings between $\mathcal{S}$ (the web API) and $\mathcal{T}$ (the ontology). In addition, we introduce the Integrations Block

Applications of Planning: Others...


Proceedings of the Thirty-Th

Combining Heuristic S

Alberto Pozanco,

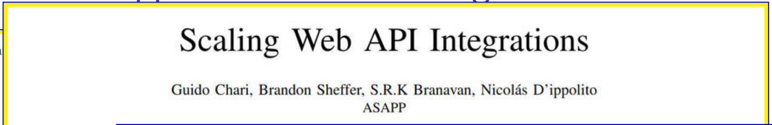
{alberto.pozancolancho

Abstract

Defining financial goals and formulatin
achieve them are essential components f
health. This task is computationally ch
abundance of factors that can influence
ation. In this paper, we present the Pers
(PFP), which can generate personalized
consider a person's context and the f
financially related actions to help them
PFP solves the problem in two stages. F
search to find a high-level sequence of
the income and reduce spending to help
financial goals. Next, it uses integer lin
determine the best low-level actions to
level plan. Results show that PFP is able
ing realistic financial plans for complex t
low level actions and long planning hori

Introduction

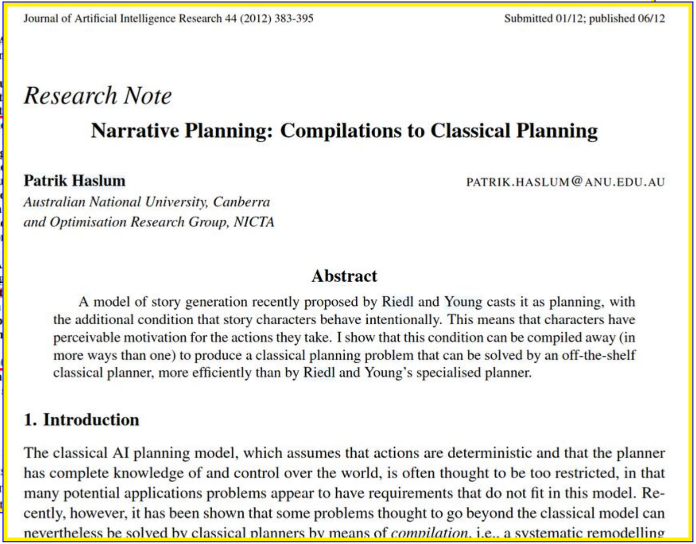
etting financial goals and planning a
chieving financial health whether fo
olds or companies. For individuals, f


Scaling Web API Integrations

Guido Chari, Brandon Sheffer, S.R.K Branavan, Nicolás D'ippolito
ASAPP

Abstract—In A
enterprise custor
customers' web
tions between ou
major effort for t
approach for int
is lightweight and
Ontology-Based
Specific Languag
cessing (NLP) m
OBDA architectu
APIs via an onto
point. IBL is a
domain experts t
software develop
writing the IBL
from each web A
selected mapping
we define an Aut
is finally fed to a
IBL method imp
This approach
with more than
indicate a ≈ 5
integrations with
code generation
so far.

The process
multiple system
consists of cor
schema and inst


Journal of Artificial Intelligence Research 44 (2012) 383-395 Submitted 01/12; published 06/12

Research Note

Narrative Planning: Compilations to Classical Planning

Patrik Haslum PATRIK.HASLUM@ANU.EDU.AU
Australian National University, Canberra
and Optimisation Research Group, NICTA

Abstract

A model of story generation recently proposed by Riedl and Young casts it as planning, with the additional condition that story characters behave intentionally. This means that characters have perceivable motivation for the actions they take. I show that this condition can be compiled away (in more ways than one) to produce a classical planning problem that can be solved by an off-the-shelf classical planner, more efficiently than by Riedl and Young's specialised planner.

1. Introduction

The classical AI planning model, which assumes that actions are deterministic and that the planner has complete knowledge of and control over the world, is often thought to be too restricted, in that many potential applications problems appear to have requirements that do not fit in this model. Recently, however, it has been shown that some problems thought to go beyond the classical model can nevertheless be solved by classical planners by means of *compilation*, i.e., a systematic remodelling

Applications of Planning: Others...

Proceedings of the Thirty-Th

Combining Heuristic S

{a

Defining financi
achieve them are
health. This tas
abundance of fa
ation. In this pap
(PFP), which ca
consider a pers
financially relat
PFP solves the p
search to find a
the income and
financial goals.
determine the b
level plan. Resul
ing realistic fina
low level actions

Setting financial
achieving financi
holds or compan

Scaling Web API Integrations

Guido Chari, Brandon Sheffer, S.R.K Branavan, Nicolás D'ippolito
ASAPP

Journal of Artificial Intelligence Research 44 (2012) 383-395 Submitted 01/12; published 06/12

mpilations to Classical Planning

PATRIK.HASLUM@ANU.EDU.AU

Abstract

roposed by Riedl and Young casts it as planning, with
behave intentionally. This means that characters have
ke. I show that this condition can be compiled away (in
planning problem that can be solved by an off-the-shelf
iedl and Young's specialised planner.

mes that actions are deterministic and that the planner
the world, is often thought to be too restricted, in that
to have requirements that do not fit in this model. Re-
problems thought to go beyond the classical model can
y means of *compilation*, i.e., a systematic remodelling

SPRINGER NATURE Link

Find a journal | Publish with us | Track your research | Search

Home > Knowledge Engineering Tools and Techniques for AI Planning > Chapter

Planning in a Real-World Application: An AUV Case Study

Chapter | First Online: 26 March 2020
pp 249–259 | Cite this chapter

Lukáš Chrpa

Abstract

Automated planning deals with the problem of finding a (partially ordered) action
sequence of plan transforming the environment from a given initial state to some required
goal state.

Applications of Planning: Others...

Proceedings of the Thirty-Th

Combining Heuristic S

Defining financi achieve them are health. This tasl abundance of fa ation. In this pap (PFP), which ca consider a pers financially relat PFP solves the p search to find a the income and financial goals. determine the be level plan. Resul ing realistic fina low level actions

Setting financial chieving financi olds or compan

Scaling Web API Integrations

Guido Chari, Brandon Sheffer, S.R.K Branavan, Nicolás D'ippolito
ASAPP

Journal of Artificial Intelligence Research 4

SPRINGER NATURE Link

Find a journal | Publish with us | Track your research | Search

Home > Knowledge Engineering Tools and Techniques for AI Planning > Chapter

Planning in a Real-World Application: A AUV Case Study

Chapter | First Online: 26 March 2020
pp 249–259 | Cite this chapter

Lukáš Chrpa

Abstract

Automated planning deals with the problem of finding a (partially ordered) acti

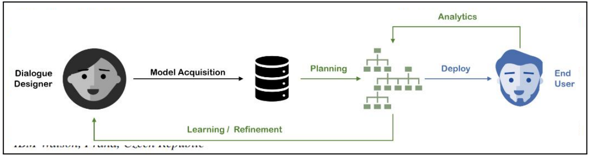
Planning for Goal-Oriented Dialogue Systems

Christian Muise CHRISTIAN.MUISE@IBM.COM
IBM Research AI, Cambridge, USA

Tathagata Chakraborti TCHAKRA2@IBM.COM
IBM Research AI, Cambridge, USA

Shubham Agarwal SHUBHAM.AGARWAL@IBM.COM
IBM Research AI, Cambridge, USA

Ondrej Bajgar* ONDREJ@BAJGAR.ORG
Future of Humanity Institute, University of Oxford, UK



Miroslav Vodolán MVODOLAN@CZ.IBM.COM
IBM Watson, Praha, Czech Republic

Charlie Wiecha WIECHA@US.IBM.COM
Watson Data and AI, Yorktown Heights, USA

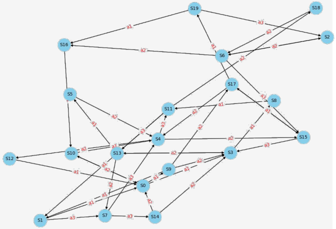
Part 1: Classical Planning: Languages

Part 1: Classical Planning: Languages

State Models & Plans

State Models $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$

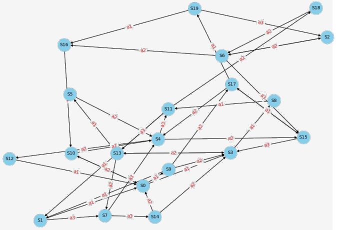
- finite and discrete state space S
- a known **initial state** $s_0 \in S$
- a set $S_G \subseteq S$ of **goal** states
- a set Act of **actions**
- subsets of actions $A(s) \subseteq Act$ **applicable** in each $s \in S$
- a (deterministic) **transition function** $s' = f(a, s)$, $a \in A(s)$
- positive **action costs** $c(a, s)$



State Models & Plans

State Models $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$

- finite and discrete state space S
- a known **initial state** $s_0 \in S$
- a set $S_G \subseteq S$ of **goal** states
- a set Act of **actions**
- subsets of actions $A(s) \subseteq Act$ **applicable** in each $s \in S$
- a (deterministic) **transition function** $s' = f(a, s)$, $a \in A(s)$
- positive **action costs** $c(a, s)$



Solution Plan σ: sequence of applicable actions $a_0, \ldots, a_n$ that reaches S_G

There must be states $s_0, \ldots, s_{n+1}$ such that:

1. s_0 is the initial state and $s_{n+1} \in S_G$ is a goal state; and
2. $s_{i+1} = f(a_i, s_i)$, $a_i \in A(s_i)$, for $i = 0, \ldots, n$:

A plan is **optimal** if it minimizes the **sum of action costs** $\sum_{i=0,n} c(a_i, s_i)$.

🔅 If costs are all 1, plan cost is plan **length**.

Classical Planning: Assumptions

Classical planning makes several assumptions about state models (underlined):

1. **Static** vs **Dynamic**: agent is the only actor in the world.
2. **Deterministic** vs **Stochastic**: actions have deterministic effects.
3. **Instantaneous** vs **temporal**: actions happy instantaneous.
4. **Fully Observable** vs **Partially Observable**: agent knows the state of the world.
5. **Discrete** vs **Numeric**: state space is finite and discrete.

State Models: Variations

Other types of state models obtained by relaxing restriction:

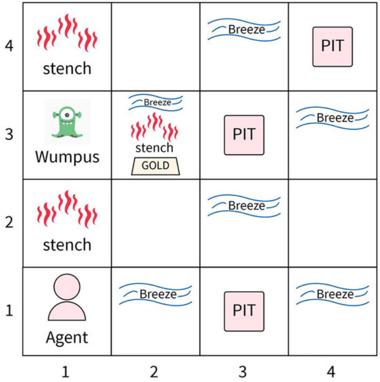
- **Markov Decision Processes:** state transition **probabilities** $P_a(s' \mid s)$ and **full obs**
- **Partially Observable MDPs (POMDPs):** $P_a(s' \mid s$ and **sensor model** $P_a(o \mid s)$, $o \in \Omega$
- **Fully Observable Non-Det (FOND) Models:** set of successor states $s' \in F(a, s)$
- **Partially Observable Non-Det (POND) Models:** $F(a, s)$ and sensor model $o(s) \in \Omega$
- **Conformant Models:** uncertain S_0 and $F(a, s)$, and no feedback,
- **Continuous Models:** infinite state space; e.g., represent velocity and continous processes like filling a bucket.
- ...

– In presence of **uncertainty**, **feedback** is critical.
– **Solution form** depends on feedback: **open loop** vs **closed-loop** control.

☼ **Our classical state models $\mathcal{S}$ are the simplest:** s_0 known, deterministic, known dynamics $f(a, s)$, no feedback; **solutions** are action sequences (open loop).

State Model Variations: Example

- **Agent**, at lower-left corner, aims to find the **gold**, while avoiding falling in a **pit** or meeting the **wumpus**.
- Positions of pits, gold, and wumpus, however, **not known**, but agent can **sense** presence of pit or Wumpus when at distance 1
- How to **model** problem?
- What's a **solution**? How to **find** it?



By Eshika Shah - "Wumpus World in AI"

Examples of our basic, deterministic state models

Model these problems as state models; i.e. fill the 7 bullets of definition

- **Navigation:** agent moves in $n \times m$ grid with some cells blocked.
- **15-puzzle:** sliding tiles in empty slot to get tiles 1 to 15 ordered.
- **Blocks world:** arm picks "clear" blocks from table or other blocks; reach target config.
- **Delivery:** n packages in grid must be picked & delivered to target cell.; one at a time.
- **Missionaries and Cannibals:** 3 Ms + 3 Cs to cross river using boat for 2; cannibals can't be outnumbered in either bench at risk of being converted.
- **TSP:** travelling salesman problem; min-cost tour that visits each node of a graph once
- **Applications:** GPS, Video Games, ...; matrix multiplication algorithms that minimize # of operations wrt standard algorithms (Deep Mind 2022; Speck *et al.* 2023)

➠ States models sometimes called also **search models**, **problem spaces**, ...

➠ In general, S given by **state variables** x_1, ..., x_N and their **domains** D_1, ..., D_N.

➠ Number of states $|S|$ bounded by cross-product $|D_1| \times |D_2| \times \cdots \times |D_n|$; not all states **reachable** with actions from s_0.

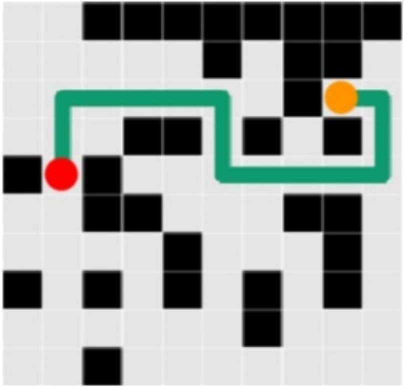
➠ Model adequate if (opt) solutions to model represent (opt) solutions to problem.

Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$

- Agent moves in $n \times m$ grid.
- Some cells blocked.



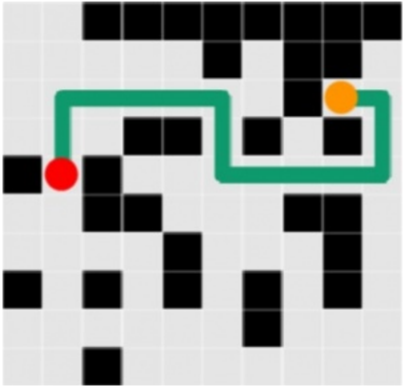
Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$
2. s_0: initial location $(x_0, y_0) = (0, 0)$

- Agent moves in $n \times m$ grid.
- Some cells blocked.



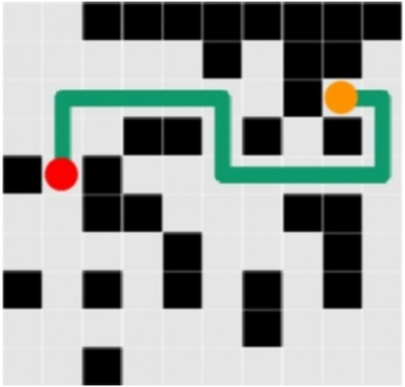
Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c\rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$
2. s_0: initial location $(x_0, y_0) = (0, 0)$
3. S_G: set of target locations

- Agent moves in $n \times m$ grid.
- Some cells blocked.



Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

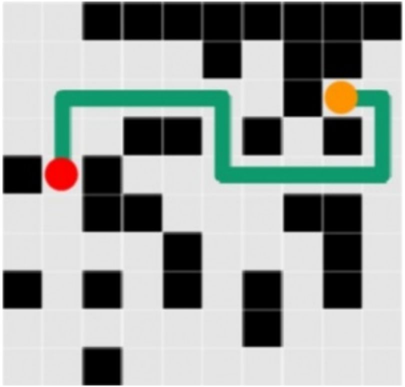
Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c\rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$
2. s_0: initial location $(x_0, y_0) = (0, 0)$
3. S_G: set of target locations
4. Act: *up, down, right, left*

Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

- Agent moves in $n \times m$ grid.
- Some cells blocked.

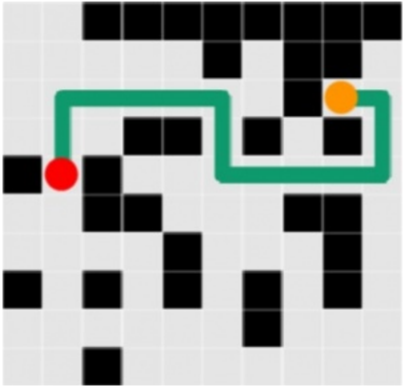


Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$
2. s_0: initial location $(x_0, y_0) = (0, 0)$
3. S_G: set of target locations
4. Act: *up, down, right, left*
5. $A(s)$ includes *up* if cell $(x, y + 1)$ for $s = (x, y)$ is traversable; it includes *left* if ...

- Agent moves in $n \times m$ grid.
- Some cells blocked.



Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

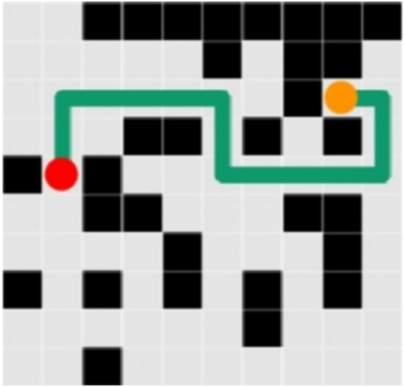
Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$
2. s_0: initial location $(x_0, y_0) = (0, 0)$
3. S_G: set of target locations
4. Act: *up, down, right, left*
5. $A(s)$ includes *up* if cell $(x, y + 1)$ for $s = (x, y)$ is traversable; it includes *left* if ...
6. $s' = f(up, s)$ if $s' = (x, y + 1)$ and $s = (x, y)$, ...

Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

- Agent moves in $n \times m$ grid.
- Some cells blocked.



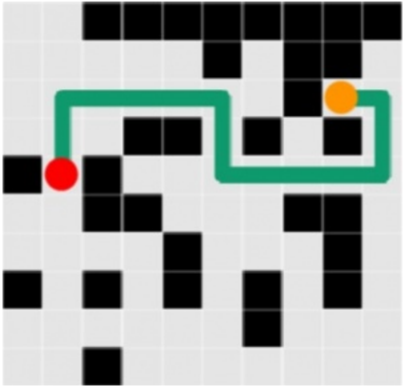
Examples: Navigation

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: agent locations $s = (x, y)$; bottom left is $(0, 0)$
2. s_0: initial location $(x_0, y_0) = (0, 0)$
3. S_G: set of target locations
4. Act: *up, down, right, left*
5. $A(s)$ includes *up* if cell $(x, y + 1)$ for $s = (x, y)$ is traversable; it includes *left* if ...
6. $s' = f(up, s)$ if $s' = (x, y + 1)$ and $s = (x, y)$, ...
7. $c(a, s) = 1$

Single state variable, x_1, representing **agent location** with $n \times m$ values (x, y) in D_1.

- Agent moves in $n \times m$ grid.
- Some cells blocked.



Example: 15-puzzle

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: a 16-tuple of unique values $0, \ldots, 15$ (0 is "blank").
2. s_0: $(15, 2, 1, 12, 8, \ldots)$; entry l at pos. t encodes $loc(t) = l$
3. S_G: singleton state $(1, 2, 3, 4, 5, \ldots, 0)$
4. Act: *up, down, right, left* (moving the "blank")
5. $A(s)$ includes *up* if location above blank in s, $loc(0)$, in board
6. $s' = f(up, s)$ is s' is like s but with positions of blank and tile above blank, swapped; similar for $down$, $left$, ...
7. $c(a, s) = 1$

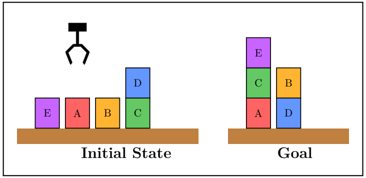
Reach ordered configuration (1,2,3,4,..)
Can move the "blank" tile up, down, left, right.

- The **state variables** x_t are $loc(t)$, $t = 0, \ldots, 16$; domain $D_t = \{0, \ldots, 15\}$

❓ $|S|$ not $|D_0| \times |D_1| \times \cdots \times |D_{15}|$ but $16!$ (16 Factorial). ***Why?***

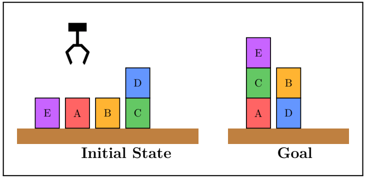
❓ ***Alternative state model?***

Example: (Oh no it's) The Blocksworld 😀



Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

Example: (Oh no it's) The Blocksworld 😀



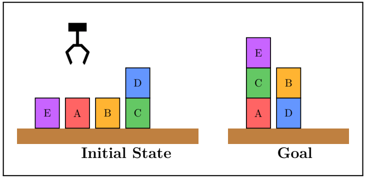
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.

Example: (Oh no it's) The Blocksworld 😀



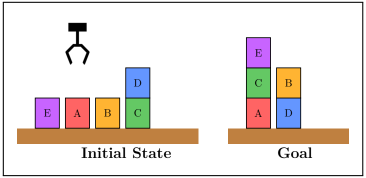
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.

Example: (Oh no it's) The Blocksworld 😀



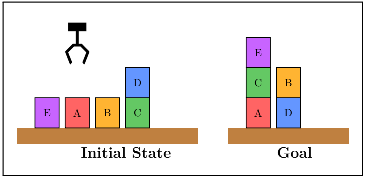
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$

Example: (Oh no it's) The Blocksworld 😀



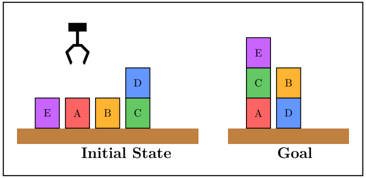
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$
4. Act: pick block b, place block being held onto block b or table

Example: (Oh no it's) The Blocksworld 😀



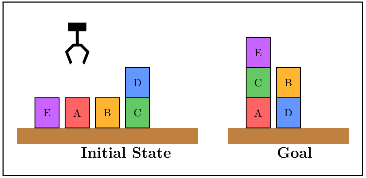
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$
4. Act: pick block b, place block being held onto block b or table
5. $A(s)$ includes $pick(B)$ if $loc(x) \neq B$ and $loc(x) \neq gripper$ for all blocks $x \neq B$

Example: (Oh no it's) The Blocksworld 😀



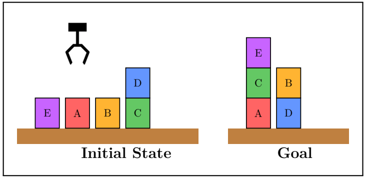
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$
4. Act: pick block b, place block being held onto block b or table
5. $A(s)$ includes $pick(B)$ if $loc(x) \neq B$ and $loc(x) \neq gripper$ for all blocks $x \neq B$
6. $s' = f(pickup(x), s)$ is like s but with $loc(x)$ set to $gripper$.

Example: (Oh no it's) The Blocksworld 😀



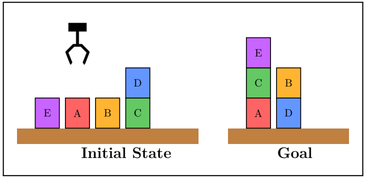
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c\rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c\rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$
4. Act: pick block b, place block being held onto block b or table
5. $A(s)$ includes $pick(B)$ if $loc(x) \neq B$ and $loc(x) \neq gripper$ for all blocks $x \neq B$
6. $s' = f(pickup(x), s)$ is like s but with $loc(x)$ set to $gripper$.
7. $c(a, s) = 1$

Example: (Oh no it's) The Blocksworld 😀



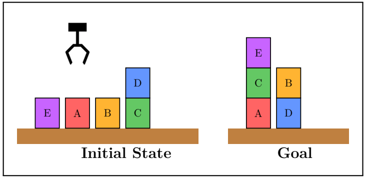
Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$
4. Act: pick block b, place block being held onto block b or table
5. $A(s)$ includes $pick(B)$ if $loc(x) \neq B$ and $loc(x) \neq gripper$ for all blocks $x \neq B$
6. $s' = f(pickup(x), s)$ is like s but with $loc(x)$ set to $gripper$.
7. $c(a, s) = 1$

Example: (Oh no it's) The Blocksworld 😀



Robot arm picks "clear" blocks from table or from other blocks, and place them on table or on other blocks. Each block has a **unique ID**.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: assigns location to each block b: $loc(b)$ can be another block, table, gripper.
2. s_0: given initial state such that $loc(A) = loc(B) = loc(C) = table$; $loc(D) = C$.
3. S_G: where $loc(A) = loc(D) = table$, $loc(C) = A$, $loc(E) = C$, $loc(B) = D$
4. Act: pick block b, place block being held onto block b or table
5. $A(s)$ includes $pick(B)$ if $loc(x) \neq B$ and $loc(x) \neq gripper$ for all blocks $x \neq B$
6. $s' = f(pickup(x), s)$ is like s but with $loc(x)$ set to $gripper$.
7. $c(a, s) = 1$

❓ ***How many states?*** Not all assignments $loc(b) = v$ reachable; **state invariants** (which?)

Example: Delivery/Driverlog

Agent must move and pick packages spread in an $n \times m$ grid, and take them one by one, to the target cells.

What is the state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$?

1. $s \in S$: location of agent and packages; $loc(a)$, $loc(pkg)$
2. s_0: given
3. S_G: $loc(pkg) = target$ for all packages pkg
4. Act: $pick(pkg)$, $drop(pkg)$, moves up, $down$, $left$, $right$
5. $A(s)$ includes $pick(pkg)$ if $loc(pkg) = loc(a)$, and agent hand empty, ...
6. $s' = f(pick(pkg), s)$ is like s but $loc(pkg)$ changes to $agent$, ...
7. $c(a, s) = 1$

❓ Number of states is exponential, but exponential on *what?*

Example: River crossing puzzle

A farmer needs to cross a river with a goat, a wolf, and a cabbage. His boat can only carry one item at a time. The goat cannot be left alone with the cabbage (the goat will eat the cabbage!). The goat cannot be left alone with the wolf (the wolf will eat the goat!)

Model problem as a state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$.

- $s \in S$: contains $x_l, x_r \in \{0, 1\}$, for $x \in \{cabbage, goat, boat, wolf\}$
- s_0, S_G, Act, ...

Example: River crossing puzzle

A farmer needs to cross a river with a goat, a wolf, and a cabbage. His boat can only carry one item at a time. The goat cannot be left alone with the cabbage (the goat will eat the cabbage!). The goat cannot be left alone with the wolf (the wolf will eat the goat!)

Model problem as a state model $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$.

- $s \in S$: contains $x_l, x_r \in \{0, 1\}$, for $x \in \{cabbage, goat, boat, wolf\}$
- s_0, S_G, Act, ...

☛ Constraint that "cabbage should not be left alone with the goat" is not a **state invariant** (true no matter what actions are taken); but a **constraint to be enforced**!

❓ What about make $A(s)$ **empty** if s does not satisfy the constraint (making s a **dead-end**)?

Computation: How to solve (deterministic) state models?

- State model $\mathcal{S}$ defines **directed graph** $G(\mathcal{S})$ with nodes n that represent states $s = s(n)$, and labeled edges that represent state transitions:
  - root node n_0 in $G(\mathcal{S})$ represents initial state $s(n_0) = s_0$
  - target nodes n_G represent the goal states $s(n) \subseteq S_G$
  - labeled edge $n \rightarrow_a n'$ if $s(n') = f(a, s)$ for $a \in A(s)$, $s = s(n)$.

Computation: How to solve (deterministic) state models?

- State model $\mathcal{S}$ defines **directed graph** $G(\mathcal{S})$ with nodes n that represent states $s = s(n)$, and labeled edges that represent state transitions:
  - root node n_0 in $G(\mathcal{S})$ represents initial state $s(n_0) = s_0$
  - target nodes n_G represent the goal states $s(n) \subseteq S_G$
  - labeled edge $n \rightarrow_a n'$ if $s(n') = f(a, s)$ for $a \in A(s)$, $s = s(n)$.
- Finding a solution to **state model** $\mathcal{S}$ becomes **finding a path in graph** $G(\mathcal{S})$ connecting nodes representing initial states and goal states.
- While any path-finding algorithms for graphs could be used for solving state models, **few scale up** to very large graphs (billions of nodes!).

⚠ Size of state models and graphs is **exponential** in the number of **state variables**.
  - *Models and graphs not given **explicitly** but **implicitly**.*

Search Algorithms for Path Finding in Directed Graphs

Blind search/Brute force algorithms

Goal plays **passive** role in the search.

Informed/Heuristic Search Algorithms

Goals plays **active** role in the search through **heuristic function** $h(s)$ that estimates cost from s to the goal.

- Heuristic h is said **admissible** if $h(s) \leq h^*(s)$ for all s where h^* is **optimal cost** from s to goal. That is, h is an **optimistic estimate**, or alternatively, a **lower bound** over cost.

Search Algorithms for Path Finding in Directed Graphs

Blind search/Brute force algorithms

Goal plays **passive** role in the search.

✏ *e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID), Iterative Width (IW)*

Informed/Heuristic Search Algorithms

Goals plays **active** role in the search through **heuristic function** $h(s)$ that estimates cost from s to the goal.

- Heuristic h is said **admissible** if $h(s) \leq h^*(s)$ for all s where h^* is **optimal cost** from s to goal. That is, h is an **optimistic estimate**, or alternatively, a **lower bound** over cost.

✏ *e.g., A*, IDA*, Hill Climbing, Best First, DFS B&B, LRTA*, ...*

Basic General Search Scheme (reviwe)

Solve(G: Graph, Init: State; Goals: Set Nodes)

```
Open := {(Init, g:0, f:0, p:None)}; Closed := {}
WHILE Open is not empty DO
  Node := *Select-Node* from Open and move it to Closed
  IF Node is Goal THEN RETURN Solution
  IF s(Node) is not in Closed THEN
    FOR EVERY Child in *Expand-Node* Node DO  // Child = (s, g, f, p)
      *Add-node* Child node to Open
RETURN Fail
```

- Nodes n are data structures that track state $s(n)$ + bookkeeping info.
- Bookkeeping for n includes labeled pointer to parent and **accummulated cost** $g(n)$
  - $g(n) = c(a, n') + g(n')$ where n' is parent of n, a is action label

Basic General Search Scheme (reviwe)

Solve(G: Graph, Init: State; Goals: Set Nodes)

```
Open := {(Init, g:0, f:0, p:None)}; Closed := {}
WHILE Open is not empty DO
  Node := *Select-Node* from Open and move it to Closed
  IF Node is Goal THEN RETURN Solution
  IF s(Node) is not in Closed THEN
    FOR EVERY Child in *Expand-Node* Node DO  // Child = (s, g, f, p)
      *Add-node* Child node to Open
RETURN Fail
```

- Nodes n are data structures that track state $s(n)$ + bookkeeping info.
- Bookkeeping for n includes labeled pointer to parent and **accummulated cost** $g(n)$
  - $g(n) = c(a, n') + g(n')$ where n' is parent of n, a is action label
- **Duplicate nodes** are nodes n and n' that represent the same state $s(n) = s(n')$
  - They are avoided, except in **depth-first search** and **tree-search algorithms**
  - For this, newly generated node n **pruned** if duplicate of n' and $g(n') \leq g(n)$
  - Yet if duplicate and $g(n) < g(n')$, n' **pruned** instead (important! *why?*)

One basic schema, many different search algorithms

- **Different search algorithms** obtained by different choices of **node to expand** from $Open$ given by:
  - `Select-Node` *Open*
  - `Add-Nodes` *New Old Open*
- **Why to consider different algorithms?** Because different properties:
  - Completeness: **guaranteed** to find a solution if one exists.
  - Optimality: **guaranteed** to find an optimal solution if one exists.
  - Space complexity: **memory** used by algorithm.
  - Time complexity: **time** used by algorithm.

Some instances of general search scheme

- **Depth-First Search** expands 'deepest' nodes n first
  - `Select-Node` $Open$: Select **First** Node in $Open$
  - `Add-Nodes` New Old: Puts New **before** Old
  - Implementation: $Open$ as a **Stack** (LIFO)
  - **Cycle checking:** prune Child in New if duplicate of ancestor

- **Breadth-First Search** expands 'shallowest' nodes n first
  - `Select-Node` $Open$: Selects **First** Node in $Open$
  - `Add-Nodes` New Old: Puts New **after** Old
  - Implementation: $Open$ as a **Queue** (FIFO)

Heuristic search and heuristic functions

- Heuristic search algorithms use two functions:
  - $g(n)$: **accumulated cost** from root to node n in OPEN
  - $h(n)$: **estimated cost** from state $s(n)$ represented by n to goal
- Heuristic function $h(n)$ provides the search with a **sense of direction**
  - **Quick** and **rough** approximation of cost from $s(n)$ to goal

Heuristic search and heuristic functions

- Heuristic search algorithms use two functions:
  - $g(n)$: **accumulated cost** from root to node n in OPEN
  - $h(n)$: **estimated cost** from state $s(n)$ represented by n to goal
- Heuristic function $h(n)$ provides the search with a **sense of direction**
  - **Quick** and **rough** approximation of cost from $s(n)$ to goal
- Simple but useful **heuristic functions** $h(n)$:
  - **Navigation:** Manhattan distance (ignores blocked cells)
  - **15-puzzle:** Sum of Manhattan distances (ignores interactions)
  - **Blocks:** Twice number of blocks sitting on different block in goal
  - **Delivery:** Sum of Manhattan distances, ...
- A **heuristic** h is **admissible** if $h(n) \leq h^*(n)$ for all nodes n (states)
- Which heuristics above are **admissible**? Why?

Simplest heuristic search algorithm (not too good though)

Greedy search or **Hill climbing (descending)** search

1. **Starting** with $s = s_0$,
2. **Evaluate** each action $a \in A(s)$ as: $Q(a,s) = c(a,s) + h(s')$, where $s' = f(a,s)$
3. **Apply** action **a** that minimizes $Q(\mathbf{a}, s)$
4. **Exit** if s' is goal, else go to 1 with $s := s'$

Greedy search is **incomplete**, even if extended with **cycle checking**. Yet:

- ✓ It uses constant memory (if no cycle checks); or linear memory (cycle checks)
- ✓ It's a "real-time" algorithm; i.e., there is notion of **current state**
- ✓ There is a **simple way** to fix **incompleteness** and **non-optimality** (!)
  - **Update** the heuristic function h of parent when moving to child
  - Resulting algorithm is **Learning Real Time A* (LRTA*)**
  - LRTA* generalizes nicely to MDPs! (RTDP)

Back to the general search scheme

Best First Search expands best nodes n with $\min f(n)$ ($f(n)$ is the **evaluation function**)

- `Select-Node` $Open$: Returns node n in $Open$ with min $f(n)$
- `Add-Nodes` $New\ Old$: Performs ordered merge
- Implementation: Open as **Priority Queue**
- Special cases
  - **Uniform cost/Dijkstra**: $f(n) = g(n)$
  - **A***: $f(n) = g(n) + h(n)$
  - **WA*:** $f(n) = g(n) + Wh(n)$, $W \geq 1$
  - **Greedy Best First:** $f(n) = h(n)$ (different than greedy search)

Memory. Properties. Consistency

- All algorithms **except** DFS and its variants (below) store **all nodes** in memory.
- When nodes expanded, children looked up in **Open** and **Closed** "lists".
- **Duplicates prevented; only cheapest "copy" kept.**
  - Newly generated node n pruned, if there is a node n' in OPEN or CLOSED that represents same state s as n such that $g(n) \not< g(n')$.
  - Yet, n' pruned instead if $g(n) < g(n')$ ("reopened" if n' CLOSED)

A* Good Properties

- ✓ A* is **optimal**, yields cheapest solutions, if h **admissible**.
- ✓ A* is **optimal** also in following sense: no other algorithm expands less # of nodes than A* with same heuristic function *(this doesn't mean that A* is fastest!)*.
- ✓ A* expands 'less' # of nodes with **more informed heuristic**: h_2 more informed that h_1 if $0 < h_1(s) < h_2(s) \leq h^*(s)$, for all s.
- ✓ A* won't re-open nodes if heuristic is **consistent** (**monotonic**); i.e., $h(n) \leq c(n, n') + h(n')$ for child n' of n (f doesn't decrease along any path).

Variants of Depth-First Search (DFS)

Bounded DFS

- Like normal DFS but uses a **bound** B on solution cost
- Node n **pruned** (not added to OPEN), if $g(n) > B$
- Incomplete if no solution with cost $< B$

Iterative Deepening (ID)

- Calls **Bounded DFS** with bound $B_1 = 0$ in first iteration
- Node n **pruned** in iteration i if $g(n) > B_i$
- If no solution found in iteration i, **Bounded DFS** called with bound $B_{i+1} = \min_k g(n_k)$, over nodes n_k **pruned** in iteration i

Iterative Deepening A* (IDA*)

- Like ID but uses **evaluation function** $f(n) = g(n) + h(n)$ instead of $g(n)$
- Node n **pruned** in iteration i if $f(n) = g(n) + h(n) > B_i$
- $B_0 = h(n_0)$ and $B_{i+1} = \min_k f(n_k)$, over nodes n_k **pruned** in iteration i

Properties of Algorithms

- **Completeness**: whether guaranteed to find solution
- **Optimality**: whether solution guaranteed optimal
- **Time Complexity**: how time increases with size
- **Space Complexity:** how space increases with size

	DFS	BrFS	ID	A*	HC	IDA*	B&B
Complete	Yes*	Yes	Yes	Yes	No	Yes	Yes
Optimal	No	Yes*	Yes	Yes	No	Yes	Yes
Time	b^D	b^d	b^d	b^d	∞	b^d	b^D
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$	$b \cdot d$

- Parameters: d is optimal solution depth; b is branching factor; $D >> d$
- BrFS **optimal** when costs are uniform; DFS **complete** with cyclic checking
- A*/IDA* optimal when h is **admissible**; $h \leq h^*$
- B&B refers to Depth-first search Branch-and-Bound ...

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.
2. **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA* – Weighted A*).

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.
2. **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA* – Weighted A*).
   - *Solutions **no longer optimal** but at most W times from optimal (if h admissible).*

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.
2. **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA* – Weighted A*).
   - *Solutions **no longer optimal** but at most W times from optimal (if h admissible).*
3. For very large spaces, only feasible optimal algorithms are **linear-memory** algorithms such as IDA* and B&B.

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.
2. **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA* – Weighted A*).
   - *Solutions **no longer optimal** but at most W times from optimal (if h admissible).*
3. For very large spaces, only feasible optimal algorithms are **linear-memory** algorithms such as IDA* and B&B.
4. Optimal solutions have been reported to problems with **huge state spaces** such 24-puzzle, Rubik's cube, and Sokoban (Korf, Schaeffer); e.g. $|S| > 10^{20}$, using IDA* and **pattern-database heuristics**.

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.
2. **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA* – Weighted A*).
   - *Solutions **no longer optimal** but at most W times from optimal (if h admissible).*
3. For very large spaces, only feasible optimal algorithms are **linear-memory** algorithms such as IDA* and B&B.
4. Optimal solutions have been reported to problems with **huge state spaces** such 24-puzzle, Rubik's cube, and Sokoban (Korf, Schaeffer); e.g. $|S| > 10^{20}$, using IDA* and **pattern-database heuristics**.
5. Recent developments combine **deep reinforcement learning** with search: learn value/heuristic functions, learn policies, learn general policies, ...

Practical Issues: Search in Large Spaces

1. Exponential-memory algorithms like A* **not feasible** in very large spaces.
2. **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA* – Weighted A*).
   - *Solutions **no longer optimal** but at most W times from optimal (if h admissible).*
3. For very large spaces, only feasible optimal algorithms are **linear-memory** algorithms such as IDA* and B&B.
4. Optimal solutions have been reported to problems with **huge state spaces** such 24-puzzle, Rubik's cube, and Sokoban (Korf, Schaeffer); e.g. $|S| > 10^{20}$, using IDA* and **pattern-database heuristics**.
5. Recent developments combine **deep reinforcement learning** with search: learn value/heuristic functions, learn policies, learn general policies, ...
6. Resulting solutions not necessarily optimal though (or not easy to prove so).

Learning Real Time A* (LRTA*)

- LRTA* is a very interesting **real-time** search algorithm (Korf 90)
- It's like a **hill-descending** or **greedy search**, but it **updates** the heuristic V as it moves, starting with $V = h$.

1. **Evaluate** each action a in s as: $Q(a, s) = c(a, s) + V(s')$
2. **Apply** action **a** that minimizes $Q(\mathbf{a}, s)$
3. **Update** $V(s)$ to $Q(\mathbf{a}, s)$
4. **Exit** if s' is goal, else go to 1 with $s := s'$

- Two remarkable **properties**
  - **Each trial** of LRTA gets eventually to the goal if space connected
  - **Repeated trials** eventually get to the goal **optimally**, if h **admissible**!
- Generalizes well to **stochastic actions** (MDPs): RTDP

Iterative Width: IW

- IW(k) and IW are **exploration algorithms** (no heuristic h) that make use of the **state structure** as given by set of **Boolean state features** $F = \{f_1, \ldots, f_N\}$
  - IW(1) is just **breadth-first search** that **prunes** states s that don't make a **feature** f_i true for first time in the search
  - IW(k) is IW(1) but over set F^k made up of conjunctions of k features from F
  - IW(k) expands up to N^k nodes and runs in **polytime** $\exp(2k)$
  - **IW** runs IW(1), IW(2), ..., IW(k) sequentially until problem solved ...
- IW is blind like DFS, BrFS, and ID but **enumerates** state space differently
- Many domains with **exponential state space** provably solved in **polynomial time** by IW when using "natural" features
  - Goals like $on(b1, b2)$ in Blocks solvable by IW(2) if F captures **locations** and **clear** status of blocks (Lipovetzky and G. 2012)
  - Idea, **width-based search**, used in state-of-the-art **classical planning algorithms**

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.

⚠️ But:

1. how to get and solve suitable relaxations?
2. how to get heuristics automatically?

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.

⚠️ But:

1. how to get and solve suitable relaxations?
2. how to get heuristics automatically?

⛑️ This is where (classical) planning comes to the rescue!

- ***state models*** $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$ *expressed in compact form by means of* ***planning languages***

.

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.

⚠️ But:

1. how to get and solve suitable
2. how to get heuristi

AI Planning = Search + KR

⛑️ Thi cal) planning comes to the rescue!

odels $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$ *expressed in compact form by means of* ***planning languages***

.

Part 1: Classical Planning: Languages

Part 1: Classical Planning: Languages

1 Motivation

2 State Models and Search

3 Planning Languages

Planning

- Planning is one of the oldest areas in AI; many ideas have been tried.
  - *A bit of **history**: first AI planners from late 50s: **GPS** (Simon and Newell)*
- A **planner** is a general solver that accepts a **problem description** of a dynamic system and computes a **solution** plan.

$$\textit{Problem} \Longrightarrow \boxed{\textit{Planner}} \Longrightarrow \textit{Plan}$$

- Problem description encodes **state model** in a compact (and accessible) form.
- **Planning Languages** for encoding state models based on **fragment of FOL**
  - Make room for **objects** and **relations**: STRIPS, ADL, PDDL, ...

Planning

- Planning is one of the oldest areas in AI; many ideas have been tried.
  - *A bit of **history**: first AI planners from late 50s: **GPS** (Simon and Newell)*
- A **planning** is a general solver that accepts a **problem description** of a dynamic system and computes a **solution** plan.

$$Problem \Longrightarrow \boxed{Planner} \Longrightarrow Plan$$

- Problem description encodes **state model** in a compact (and accessible) form.
- **Planning Languages** for encoding state models based on **fragment of FOL**
  - Make room for **objects** and **relations**: STRIPS, ADL, PDDL, ...
- **Classical planning** is "vanilla" planning:
  - Known initial state and deterministic actions; discrete time, no other changes.
- Other **planning models** relax these assumptions:
  - Incomplete information on the state; non-deterministic actions; multi-agent, etc.

State Model for Classical AI Planning

State model underlying classical planning: $\mathcal{S} = \langle S, s_0, S_G, Act, A, f, c \rangle$ where:

- S is finite and discrete **state space**
- s_0 is known **initial state** $s_0 \in S$
- S_G is subset of **goal states**, $S_G \subseteq S$
- Act is finite set of **actions**
- $A(s)$ is subset of actions **applicable** in each $s \in S$, $A(s) \subseteq Act$
- f is a deterministic **transition function**; successors $s' = f(a, s)$, $a \in A(s)$
- c is a positive **action cost** function; $c(a, s) > 0$

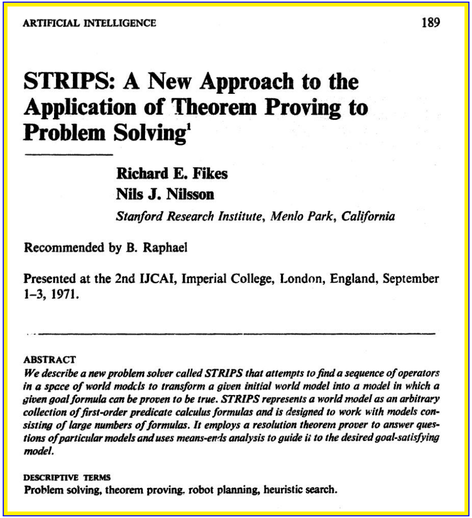
A **solution** or **plan** is a sequence of applicable actions $a_0, \ldots, a_n$ that maps s_0 into S_G; i.e. there is a state sequence $s_0, \ldots, s_{n+1}$ such that $a_i \in A(s_i)$, $s_{i+1} = f(a_i, s_i)$, and $s_{n+1} \in S_G$, $i = 0, \ldots, n$.

A plan is **optimal** if it minimizes **sum of action costs** $\sum_{i=0,n} c(a_i, s_i)$

Basic Language for Classical Planning: STRIPS

- A (grounded) planning problem in STRIPS is a tuple $P = \langle F, O, I, G \rangle$:
  - F stands for set of all **atoms** (boolean variables)
  - O stands for set of all operators (or **actions**)
  - $I \subseteq F$ stands for **initial situation**
  - $G \subseteq F$ stands for **goal situation**

- Actions or operators $o \in O$ **represented** by:
  - the Add list $\text{Add}(o) \subseteq F$: atoms that become true
  - the Delete list $\text{Del}(o) \subseteq F$: atoms that stop being true (i.e., become false)
  - the Precondition list $\text{Pre}(o) \subseteq F$: atoms that must be true for action to apply/execute



ARTIFICIAL INTELLIGENCE 189

STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving[1]

Richard E. Fikes
Nils J. Nilsson
Stanford Research Institute, Menlo Park, California

Recommended by B. Raphael

Presented at the 2nd IJCAI, Imperial College, London, England, September 1–3, 1971.

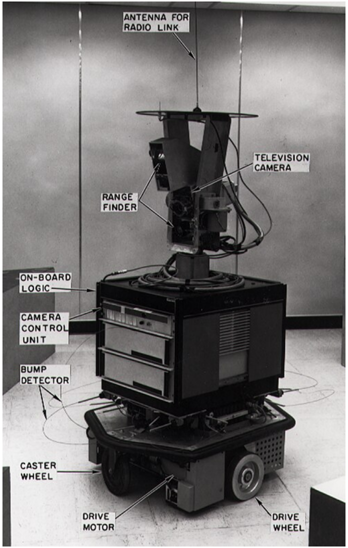
ABSTRACT

We describe a new problem solver called STRIPS that attempts to find a sequence of operators in a space of world models to transform a given initial world model into a model in which a given goal formula can be proven to be true. STRIPS represents a world model as an arbitrary collection of first-order predicate calculus formulas and is designed to work with models consisting of large numbers of formulas. It employs a resolution theorem prover to answer questions of particular models and uses means-ends analysis to guide it to the desired goal-satisfying model.

DESCRIPTIVE TERMS
Problem solving, theorem proving, robot planning, heuristic search.

Stanford Research Institute Problem Solver

STRIPS for SRI Shakey (1966-1972)



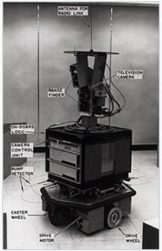
Check this video for a demo of Shakey's capabilities.

Software [edit]

Main article: Stanford Research Institute Problem Solver

The robot's programming was primarily done in LISP. The Stanford Research Institute Problem Solver (STRIPS) planner it used was conceived as the main planning component for the software it utilized. As the first robot that was a logical, goal-based agent, Shakey experienced a limited world. A version of Shakey's world could contain a number of rooms connected by corridors, with doors and light switches available for the robot to interact with.[9]

Shakey had a short list of available actions within its planner. These actions involved traveling from one location to another, turning the light switches on and off, opening and closing the doors, climbing up and down from rigid objects, and pushing movable objects around.[10] The STRIPS automated planner could devise a plan to enact all the available actions, even though Shakey himself did not have the capability to execute all the actions within the plan personally.



Shakey in 1972

An example mission for Shakey might be something like, an operator types the command "push the block off the platform" at a computer console. Shakey looks around, identifies a platform with a block on it, and locates a ramp in order to reach the platform. Shakey then pushes the ramp over to the platform, rolls up the ramp onto the platform, and pushes the block off the platform.

☼ Shakey was inducted into Carnegie Mellon University's Robot Hall of Fame in 2004 alongside such notables as ASIMO and C-3PO.

From Language to Models

$\mathcal{S}(P)$: state model of planning problem P

Problem $P = \langle F, O, I, G \rangle$ determines/induces model $\mathcal{S}(P) = \langle S, s_0, S_G, Act, A, f, c \rangle$:

1. the states $s \in S$ are collections of atoms from F (what is $|S|$?)
2. the initial state s_0 is I
3. the set S_G of goal states s are those that $G \subseteq s$
4. the set of actions Act is $Act = O$,
5. the actions a in $A(s)$ are those such that $\text{Pre}(a) \subseteq s$
6. the transition function f is such that $s' = f(a, s) = (s \setminus \text{Del}(a)) \cup \text{Add}(a)$
7. action costs $c(a, s)$ are all 1

ℹ️ Note:

- (Optimal) **Solution** of P is (optimal) **solution** of $\mathcal{S}(P)$
- Language extensions often convenient (e.g., **negation** and **conditional effects**)
  - *some required for describing richer models (costs, probabilities, duration, ...).*

Example: Simple Problem in STRIPS

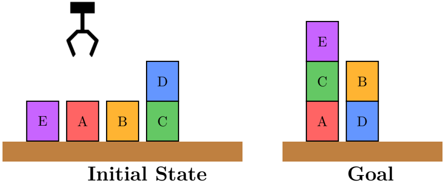
Problem $P = \langle F, I, O, G \rangle$ where:

- $F = \{p, q, r\}$
- $I = \{p\}$
- $G = \{q, r\}$
- O has two actions a and b such that:
  - Pre$(a) = \{p\}$, Add$(a) = \{q\}$, Del$(a) = \{\}$
  - Pre$(b) = \{q\}$, Add$(b) = \{r\}$, Del$(b) = \{q\}$

Questions

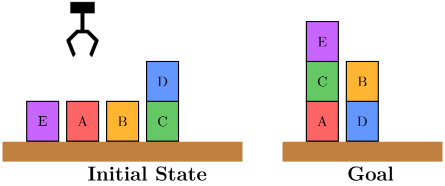
1. How many states?
2. What is $\mathcal{S}(P)$?
3. How many states are **reachable** from the initial state?

(Oh no it's) The Blocksworld (again!)



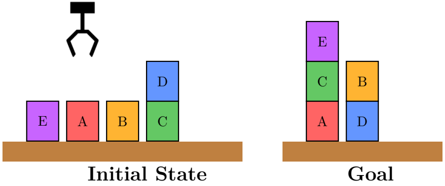
- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.

(Oh no it's) The Blocksworld (again!)



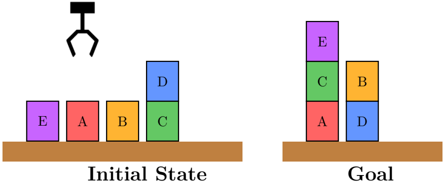
- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state:
  $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.

(Oh no it's) The Blocksworld (again!)



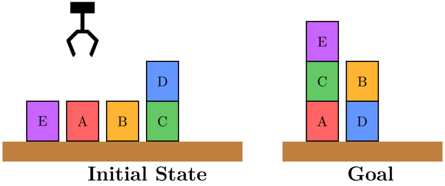
- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state:
  $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C)$, $on(C, A), on(B, D)\}$.

(Oh no it's) The Blocksworld (again!)



- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state:
  $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C), on(C, A), on(B, D)\}$.
- Actions: $stack(x, y)$, $unstack(x, y)$, $putdown(x)$, $pickup(x)$.

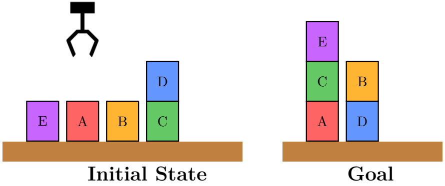
(Oh no it's) The Blocksworld (again!)



- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state: $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C), on(C, A), on(B, D)\}$.
- Actions: $stack(x, y)$, $unstack(x, y)$, $putdown(x)$, $pickup(x)$.

🤔 $pickup(x)$? - (pickup block from table)

(Oh no it's) The Blocksworld (again!)

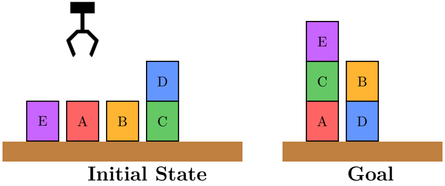


- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state: $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C),\ on(C, A), on(B, D)\}$.
- Actions: $stack(x, y)$, $unstack(x, y)$, $putdown(x)$, $pickup(x)$.

🤔 $pickup(x)$? - (pickup block from table)

Pre: $\{armEmpty(), clear(x), onTable(x)\}$

(Oh no it's) The Blocksworld (again!)

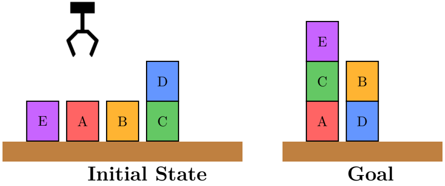


- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state:
  $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C), on(C, A), on(B, D)\}$.
- Actions: $stack(x, y)$, $unstack(x, y)$, $putdown(x)$, $pickup(x)$.

🤔 $pickup(x)$? - (pickup block from table)

Pre: $\{armEmpty(), clear(x), onTable(x)\}$
Add $\{holding(x)\}$

(Oh no it's) The Blocksworld (again!)

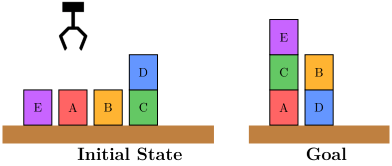


- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state:
  $\{onTable(E), clear(E), \ldots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C), on(C, A), on(B, D)\}$.
- Actions: $stack(x, y)$, $unstack(x, y)$, $putdown(x)$, $pickup(x)$.

🤔 $pickup(x)$? - (pickup block from table)

Pre: $\{armEmpty(), clear(x), onTable(x)\}$
Add $\{holding(x)\}$
Del $\{armEmpty(), clear(x), onTable(x)\}$

(Oh no it's) The Blocksworld (operators)

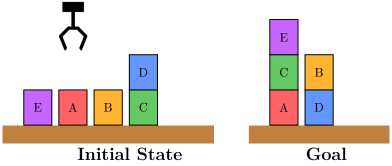


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$			
$unstack(x, y)$			
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)

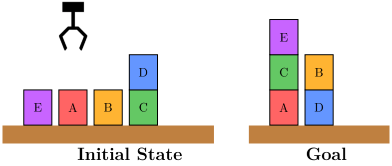


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$		
$unstack(x, y)$			
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)

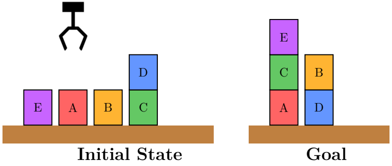


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	
$unstack(x, y)$			
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)



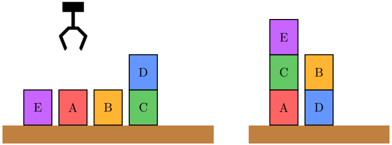
Propositions:

$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$			
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)



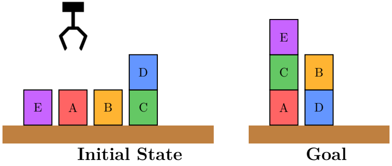
Initial State **Goal**

Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	**Precondition**	**Add**	**Delete**
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$		
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)

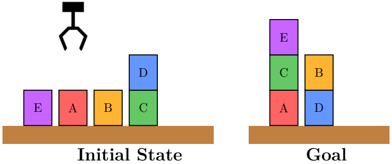


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	**Precondition**	**Add**	**Delete**
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$	$\{holding(x), clear(x)\}$	
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)

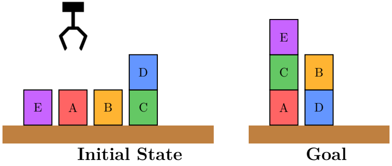


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$	$\{holding(x), clear(x)\}$	$\{armEmpty(), on(x, y), clear(x)\}$
$stack(x, y)$			

(Oh no it's) The Blocksworld (operators)

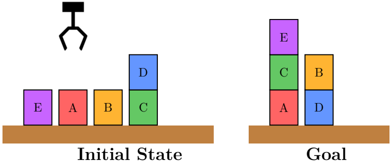


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$	$\{holding(x), clear(x)\}$	$\{armEmpty(), on(x, y), clear(x)\}$
$stack(x, y)$	$\{holding(x), clear(y)\}$		

(Oh no it's) The Blocksworld (operators)



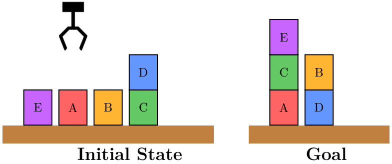
Propositions:

$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$	$\{holding(x), clear(x)\}$	$\{armEmpty(), on(x, y), clear(x)\}$
$stack(x, y)$	$\{holding(x), clear(y)\}$	$\{on(x, y), armEmpty(), clear(x)\}$	

(Oh no it's) The Blocksworld (operators)

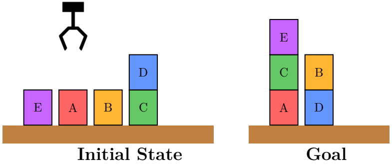


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$	$\{holding(x), clear(x)\}$	$\{armEmpty(), on(x, y), clear(x)\}$
$stack(x, y)$	$\{holding(x), clear(y)\}$	$\{on(x, y), armEmpty(), clear(x)\}$	$\{holding(x), clear(y)\}$

(Oh no it's) The Blocksworld (operators)



Propositions:

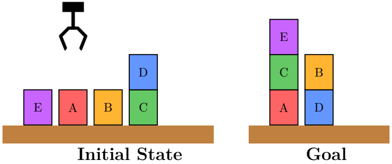
$on(x, y),\ onTable(x),\ clear(x),\ holding(x),\ armEmpty()$

Goal: $\{on(E, C),\ on(C, A), on(B, D)\}$

Action	Precondition	Add	Delete
$pickup(x)$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$
$putdown(x)$	$\{holding(x)\}$	$\{armEmpty(), clear(x), onTable(x)\}$	$\{holding(x)\}$
$unstack(x, y)$	$\{armEmpty(x), clear(x), on(x, y)\}$	$\{holding(x), clear(x)\}$	$\{armEmpty(), on(x, y), clear(x)\}$
$stack(x, y)$	$\{holding(x), clear(y)\}$	$\{on(x, y), armEmpty(), clear(x)\}$	$\{holding(x), clear(y)\}$

❓ What is a successful plan for the above problem?

(Oh no it's) The Blocksworld (plans)

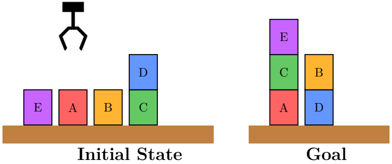


Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

❓ What is a successful plan for the above problem?

(Oh no it's) The Blocksworld (plans)



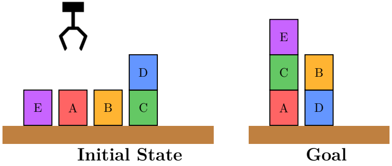
Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

❓ What is a successful plan for the above problem?

$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(B), stack(B, D), pickup(E), stack(E, C)$
✓

(Oh no it's) The Blocksworld (plans)



Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), \ on(C, A), on(B, D)\}$

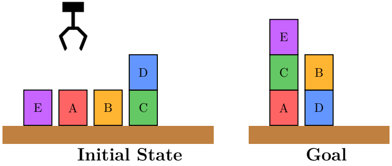
❓ What is a successful plan for the above problem?

$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(B), stack(B, D), pickup(E), stack(E, C)$
✓

❓ What about this plan?

$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(E),$
$\quad stack(E, C), pickup(D), stack(D, E), pickup(B), stack(B, D)$

(Oh no it's) The Blocksworld (plans)



Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

Goal: $\{on(E, C), on(C, A), on(B, D)\}$

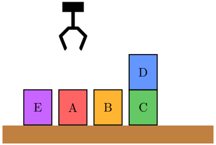
❓ What is a successful plan for the above problem?

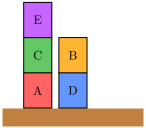
$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(B), stack(B, D), pickup(E), stack(E, C)$
✓

❓ What about this plan?

$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(E),$
$stack(E, C), pickup(D), stack(D, E), pickup(B), stack(B, D)$
✓

(Oh no it's) The Blocksworld (plans)





Propositions:
$on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$

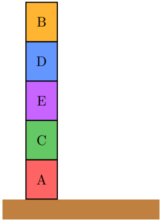
Goal: $\{on(E, C), on(C, A), on(B, D)\}$

❓ What is a successful plan for the above problem?

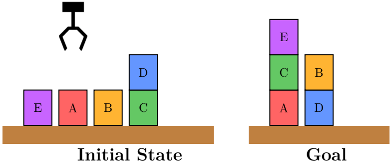
$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(B), stack(B, D), pickup(E), stack(E, C)$
✓

❓ What about this plan?

$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(E),$
$stack(E, C), pickup(D), stack(D, E), pickup(B), stack(B, D)$
✓



(Oh no it's) The Blocksworld (fixed!)



Propositions:

$on(x, y),\ onTable(x),\ clear(x),\ holding(x),\ armEmpty()$

Goal: $\{on(E, C),\ on(C, A), on(B, D), \underline{onTable(A), onTable(D)}$

❓ What is a successful plan for the above problem?

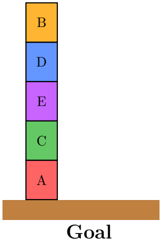
$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(B), stack(B, D), pickup(E), stack(E, C)$

✓

❓ What about this plan?

$unstack(D, C), putdown(D), pickup(C), stack(C, A), pickup(E),$
$\quad stack(E, C), pickup(D), stack(D, E), pickup(B), stack(B, D)$

✗



How to “write” STRIPS planning problems?

PDDL: A Standard Syntax for Classical Planning Problems

- **PDDL** stands for **Planning Domain Description Language**
- Developed for **International Planning Competetion (IPC)**; evolving since 1998.
- PDDL specifies syntax for problems $P = \langle F, I, O, G \rangle$ supporting **STRIPS**, **variables**, **types**, and much more...

$$\textit{Problem in PDDL} \Longrightarrow \boxed{\textsc{Planner}} \Longrightarrow \textit{Plan}$$

PDDL: A Standard Syntax for Classical Planning Problems

- **PDDL** stands for **P**lanning **D**omain **D**escription **L**anguage
- Developed for **International Planning Competetion (IPC)**; evolving since 1998.
- PDDL specifies syntax for problems $P = \langle F, I, O, G \rangle$ supporting **STRIPS**, **variables**, **types**, and much more...

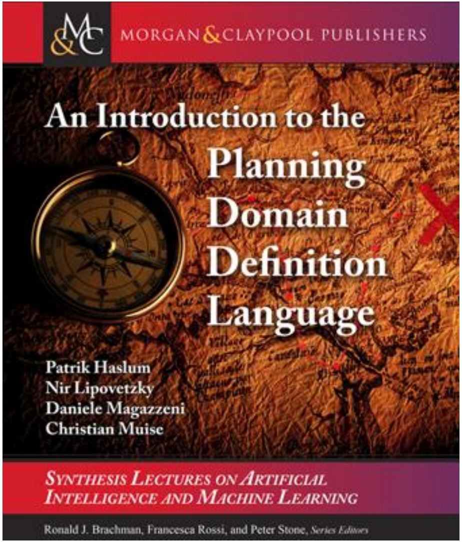
$$\textit{Problem in PDDL} \Longrightarrow \boxed{\text{PLANNER}} \Longrightarrow \textit{Plan}$$

- Problems in PDDL specified in two parts:
  1. **Domain:** general info on the system (e.g., features, actions).
  2. **Instance:** specifics of a problem (e.g., exact blocks).
- Many problem instances for the same domain.
- In IPC, planners are evaluated over unseen problems encoded in **PDDL**.

PDDL Quick Facts

PDDL is not a propositional language:

- Representation is lifted: using object variables to be instantiated from a finite set of objects. (Similar to predicate logic)
- Predicates to be instantiated with objects.
  ✏ at(?p, ?r): package ?p is at room ?r
- Action schemas parameterized by objects.
  ✏ pickup(?x): pickup block ?x



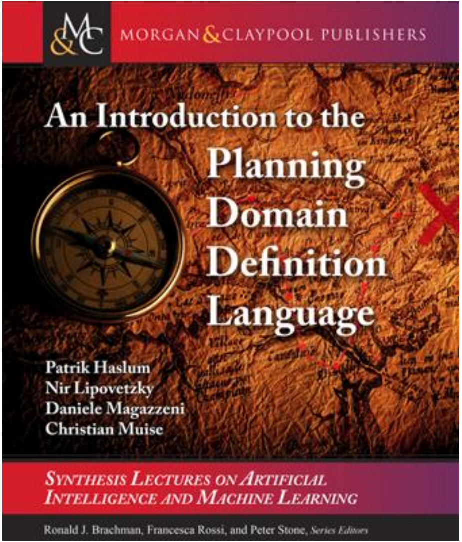
PDDL Quick Facts

PDDL is not a propositional language:

- Representation is <u>lifted</u>: using object variables to be instantiated from a finite set of objects. (Similar to predicate logic)
- Predicates to be instantiated with objects.
  ✏ at(?p, ?r): package ?p is at room ?r
- Action schemas parameterized by objects.
  ✏ pickup(?x): pickup block ?x

A PDDL planning task comes in two parts:

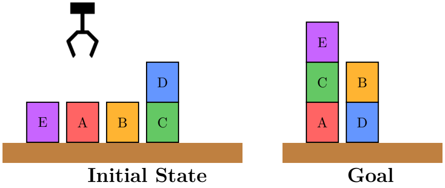
1. Domain: predicates, operators, types.
2. Problem: objects, initial state, goal condition.



Example: Blocks World Domain in STRIPS (PDDL Syntax)

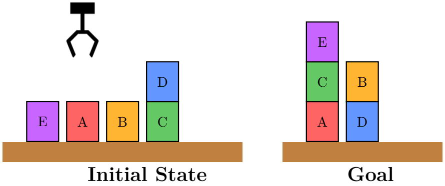
```
(define (domain blocks)
  (:requirements :strips)
  (:action pick_up
     :parameters (?x)
     :precondition (and (clear ?x) (ontable ?x) (handempty))
     :effect (and (not (ontable ?x)) (not (clear ?x)) (not (handempty)) (holding ?x)))
  (:action put_down
     :parameters (?x)
     :precondition (holding ?x)
     :effect  (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action stack
     :parameters (?x ?y)
     :precondition (and (holding ?x) (clear ?y))
     :effect  (and (not (holding ?x)) (not (clear ?y)) (clear ?x) (handempty) (on ?x ?y)))
  (:action unstack
     :parameters (?x ?y)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect (and (clear ?y) (holding ?x) (not (on ?x ?y))
                  (not (clear ?x)) (not (handempty))))
```

An instance of blocks world in PDDL




```
(define (problem blocks-example)
    (:domain blocks)
    (:objects A B C D E)
    (:init (clear E) (clear A) (clear B) (clear D) (handempty)
           (ontable E) (ontable A) (ontable B)  (ontable C) (on D C))
    (:goal (and (on C A) (on E C) (on B D))))
```

An instance of blocks world in PDDL

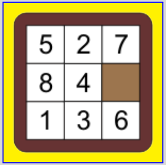



```
(define (problem blocks-example)
    (:domain blocks)
    (:objects A B C D E)
    (:init (clear E) (clear A) (clear B) (clear D) (handempty)
           (ontable E) (ontable A) (ontable B)  (ontable C) (on D C))
    (:goal (and (on C A) (on E C) (on B D))))
```

or better: 😉

```
(define (problem blocks-example)
    (:domain blocks)
    (:objects A B C D E)
    (:init (clear E) (clear A) (clear B) (clear D) (handempty)
           (ontable E) (ontable A) (ontable B) (ontable C) (on D C))
    (:goal (and (on C A) (on E C) (on B D) (ontable A) (ontable D))))
```

Example: 8-Puzzle in PDDL




```
(define (domain tile)
  (:requirements :strips :typing :equality)
  (:types tile position)
  (:constants blank - tile)
  (:predicates (at ?t - tile ?x - position ?y - position)
        (inc ?p - position ?pp - position)
        (dec ?p - position ?pp - position))
  (:action move-up
    :parameters (?t - tile ?px - position ?py - position  ?bx - position ?by - position)
    :precondition (and (= ?px ?bx) (dec ?by ?py) (not (= ?t blank)) ...)
    :effect (and (not (at blank ?bx ?by)) (not (at ?t ?px ?py))
                 (at blank ?px ?py) (at ?t ?bx ?by)))
  (:action move-down
    :parameters ...  )
  (:action move-left
    :parameters ...  )
  ...
```

```
(define (problem eight_tile)
  (:domain tile)
  (:objects t1 t2 t3 t4 t5 t6 t7 t8 - tile    p1 p2 p3 - position)
  (:init (inc p1 p2) (inc p2 p3) (dec p3 p2) (dec p2 p1)
         (at blank p1 p1) (at t1 p2 p1) (at t2 p3 p1) (at t3 p1 p2) ..)
  (:goal (and (at t8 p1 p1) (at t7 p2 p1) (at t6 p3 p1) ..)))
```

Example: 2-Gripper Problem in PDDL

An autonomous robot moves picks/drops the balls in two rooms with its arms. Check post.

```
(define (domain gripper)
   (:requirements :typing)
   (:types room ball gripper)
   (:constants left right - gripper)
   (:predicates (at-robot ?r - room)(at ?b - ball ?r - room)(free ?g - gripper)
        (carry ?o - ball ?g - gripper))
   (:action move
       :parameters  (?from ?to - room)
       :precondition (at-robot ?from)
       :effect (and  (at-robot ?to) (not (at-robot ?from))))
   (:action pick
       :parameters (?obj - ball ?room - room ?gripper - gripper)
       :precondition  (and  (at ?obj ?room) (at-robot ?room) (free ?gripper))
       :effect (and (carry ?obj ?gripper) (not (at ?obj ?room)) (not (free ?gripper))))
   (:action drop
       :parameters  (?obj - ball ?room - room ?gripper - gripper)
       :precondition  (and  (carry ?obj ?gripper) (at-robot ?room))
       :effect (and (at ?obj ?room) (free ?gripper) (not (carry ?obj ?gripper)))))

(define (problem gripper2)
    (:domain gripper)
    (:objects roomA roomB - room Ball1 Ball2 - ball)
    (:init  (at-robot roomA) (free left) (free right)  (at Ball1 roomA)(at Ball2 roomA))
    (:goal (and (at Ball1 roomB) (at Ball2 roomB))))
```

Example: Visitall Domain in PDDL

```
(define (domain grid-visit-all)  ;;;   Visit all cells in a grid
 (:requirements :strips)
 (:predicates (connected ?x ?y) (at-robot ?x) (visited ?x))

 (:action move
    :parameters (?curpos ?nextpos)
    :precondition (and (at-robot ?curpos) (connected ?curpos ?nextpos))
    :effect (and (at-robot ?nextpos) (not (at-robot ?curpos)) (visited ?nextpos))))
```

```
(define (problem grid-2)
  (:domain grid-visit-all)
  (:objects loc-x0-y0 loc-x0-y1 loc-x1-y0 loc-x1-y1)
  (:init (at-robot loc-x0-y0) (visited loc-x0-y0) (connected loc-x0-y0 loc-x1-y0)
         (connected loc-x0-y0 loc-x0-y1) (connected loc-x0-y1 loc-x0-y0)
         (connected loc-x0-y1 loc-x1-y1) (connected loc-x1-y0 loc-x1-y1)
         (connected loc-x1-y0 loc-x0-y0) (connected loc-x1-y1 loc-x1-y0)
         (connected loc-x1-y1 loc-x0-y1))
  (:goal (and (visited loc-x0-y0) (visited loc-x0-y1)
              (visited loc-x0-y2) (visited loc-x0-y3))))
```

⚠ The grid needs to be represented in PDDL:

- one object per cell (e.g., loc-x0-y0, loc-x0-y1, etc.)
- adjacency relations between cells (e.g., (connected loc-x0-y0 loc-x1-y0))

Example: Logistics in STRIPS PDDL

There are trucks and airplanes that can move packages between different citites and airports. The goal is to deliver packages to their destinations.

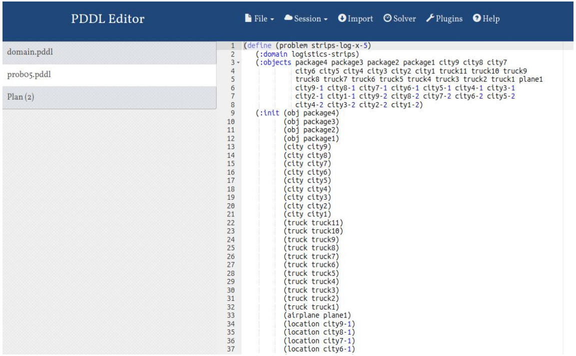
More info here; planning domain here

```
(define (domain logistics)
(:requirements :strips :typing :equality)
(:types airport - location  truck airplane - vehicle  vehicle packet - thing  ..)
(:predicates (loc-at ?x - location ?y - city) (at ?x - thing ?y - location) ...)
(:action load
    :parameters (?x - packet ?y - vehicle ?z - location)
    :precondition (and (at ?x ?z) (at ?y ?z))
    :effect (and (not (at ?x ?z)) (in ?x ?y)))
(:action unload ..)
(:action drive
    :parameters (?x - truck ?y - location ?z - location ?c - city)
    :precondition (and (loc-at ?z ?c) (loc-at ?y ?c) (not (= ?z ?y)) (at ?x ?z))
    :effect (and (not (at ?x ?z)) (at ?x ?y)))
...
```

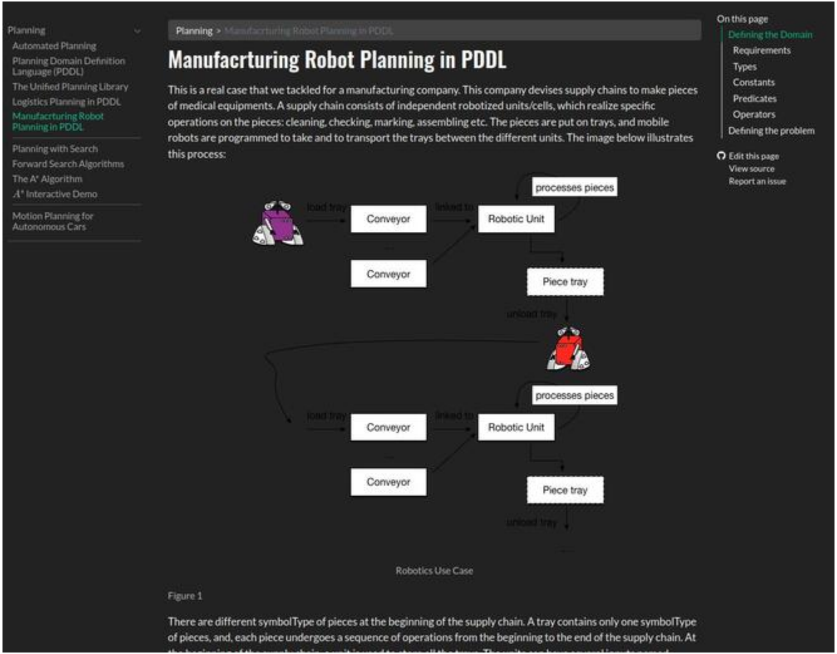
Example: Logistics in STRIPS PDDL

There are trucks and airplanes that can move packages between different citites and airports. The goal is to deliver packages to their destinations.
More info here; planning domain here

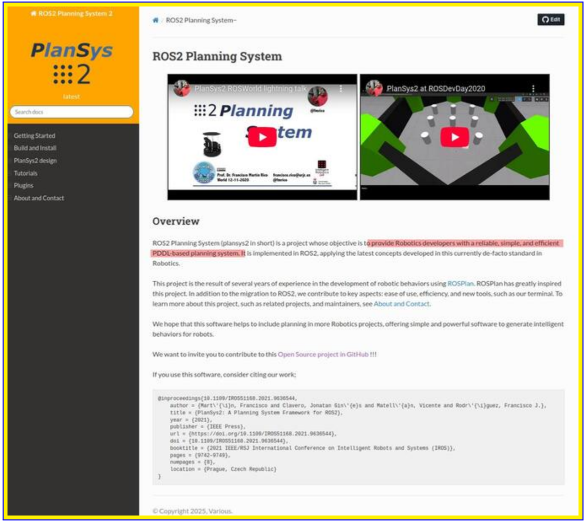
```
(define (problem log3_2)
  (:domain logistics)
  (:objects packet1 packet2 ... - packet
            truck1 truck2 truck3 ... - truck
            city1 city2 ... - city ...)
  (:init (at packet1 office1)
         (at packet2 office3)
         (at truck9 city7-1) ...)
  (:goal (and (at packet1 office2)
              (at packet2 office2)
              ...)))
```



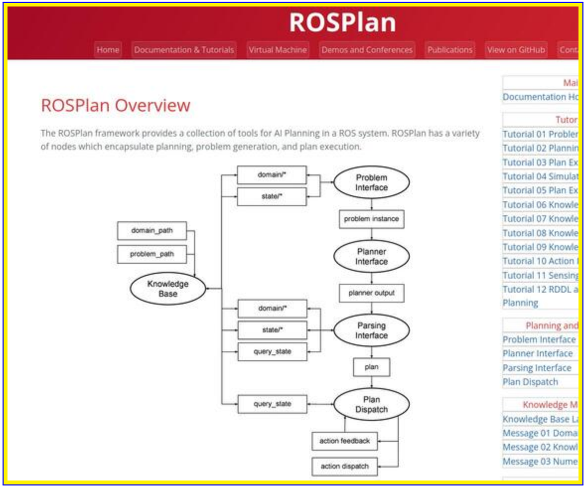
Manufactoring Robot Planning in PDDL



PDDL @ ROS Robotics



https://plansys2.github.io/



https://kcl-planning.github.io/ROSPlan/

Grounding

PDDL encoding uses variables on **predicates** and **action schemas**.

- variables replaced by **constants** of given **types** – avoids repetition
- name start with $?$, e.g., $?p$ for package, $?r$ for room, etc.

🔅 Process of replacing variables by constants, called "**instantiation**" or "**grounding**".

- **Grounded** $on(?x, ?y)$: $on(A, A)$, $on(A, B)$, $on(B, A)$, $on(A, C)$, ...

- **Grounding actions** obtained by replacing variables by constants of corresponding **type**

Grounding

PDDL encoding uses variables on **predicates** and **action schemas**.

- variables replaced by **constants** of given **types** – avoids repetition
- name start with $?$, e.g., $?p$ for package, $?r$ for room, etc.

☼ Process of replacing variables by constants, called “**instantiation**” or “**grounding**”.

- **Grounded** $on(?x, ?y)$: $on(A, A)$, $on(A, B)$, $on(B, A)$, $on(A, C)$, ...
- **Grounding actions** obtained by replacing variables by constants of corresponding **type**
- Note that instantiation above yields actions like $stack(A, A)$ and $unstack(C, C)$
  - *To avoid such instances, one can add **equality** or **inequality** preconditions such as $?r1 \neq ?r2$ that would avoid instantiations where variables $?r1$ and $?r2$ replaced by **same** constant.*

Grounding

PDDL encoding uses variables on **predicates** and **action schemas**.

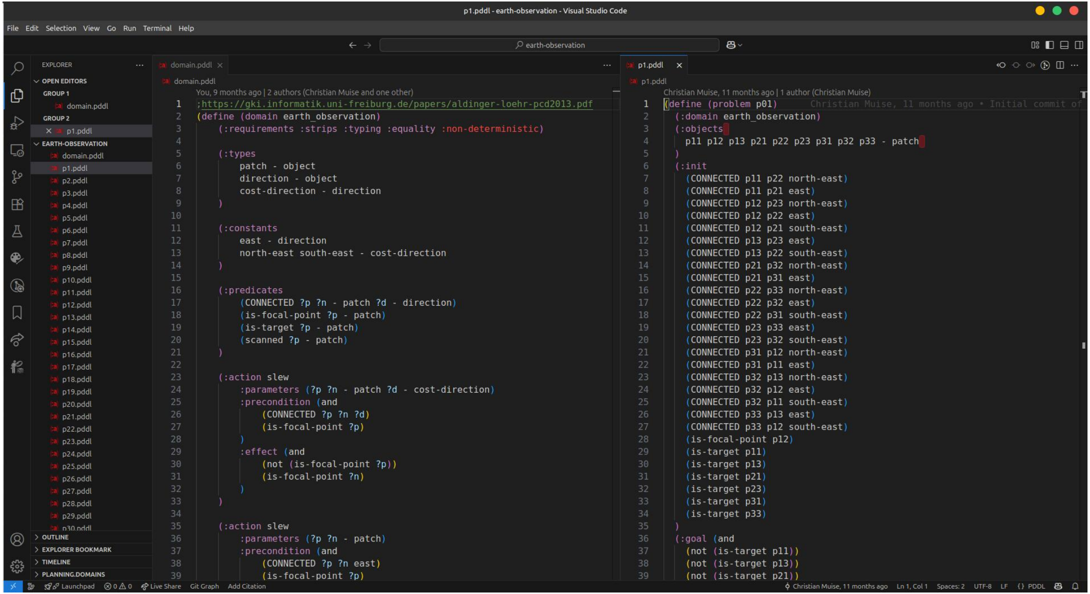
- variables replaced by **constants** of given **types** – avoids repetition
- name start with $?$, e.g., $?p$ for package, $?r$ for room, etc.

☼ Process of replacing variables by constants, called "**instantiation**" or "**grounding**".

- **Grounded** $on(?x, ?y)$: $on(A, A)$, $on(A, B)$, $on(B, A)$, $on(A, C)$, ...
- **Grounding actions** obtained by replacing variables by constants of corresponding **type**
- Note that instantiation above yields actions like $stack(A, A)$ and $unstack(C, C)$
  - *To avoid such instances, one can add* ***equality*** *or* ***inequality*** *preconditions such as* $?r1 \neq ?r2$ *that would avoid instantiations where variables* $?r1$ *and* $?r2$ *replaced by* ***same*** *constant.*
- Specialized "**grounding systems**" are used.
- Grounded instance is (much) larger than original one (but easier to solve!).
  - ❓ *How large? What does it depends on?*

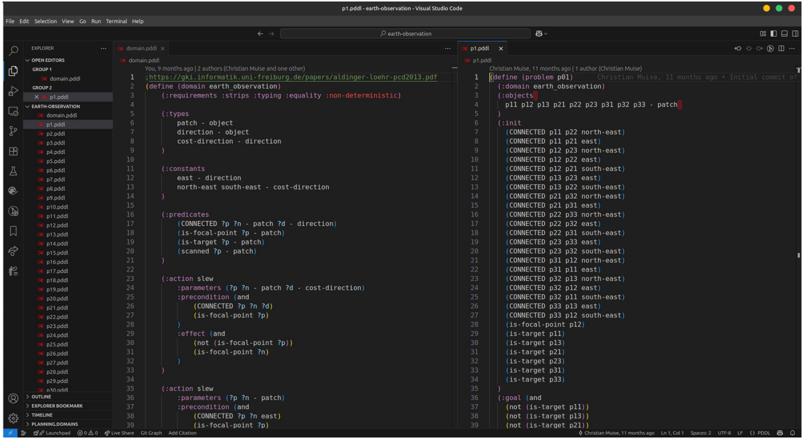
PDDL in VSCode!

Install PDDL Extension by Jan Dolejsi (Extension Id: `jan-dolejsi.pddl`)

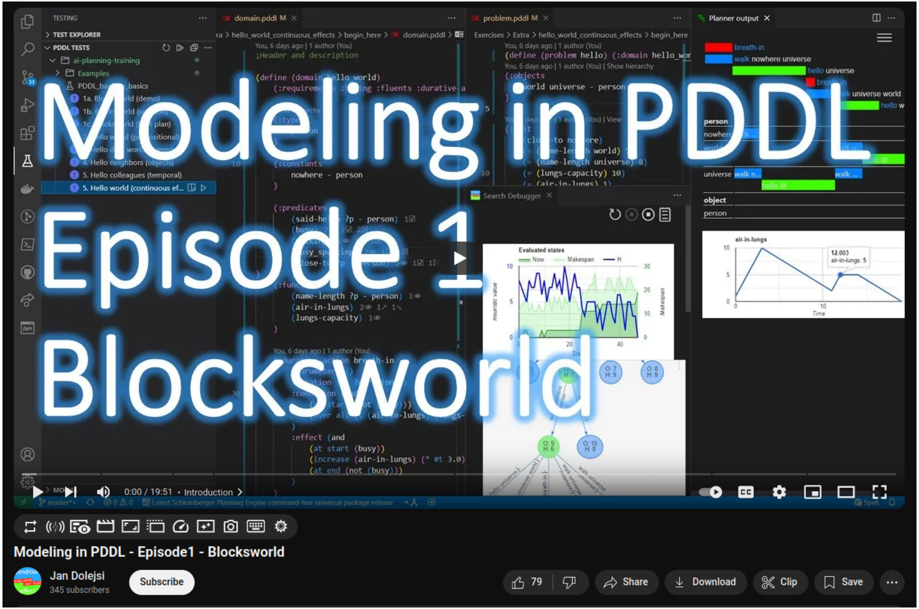


Main Selling Points...

1. Generality.
2. Accessibility.
3. Explainable.
4. Elaboration tolerant.
5. Flexibility.
6. Autonomy.
7. Rapid prototyping.
8. Declarative.



Blocks World tutorial in VSCODE



Challenge: Smart Home Planning

An intelligent robot can perform basic actions in a smart house such as **turning on lights**, **setting room thermostats**, and **opening/locking doors**. Each device (e.g., lights, thermostats, doors) is associated with a specific **room**, and **actions are conditioned on the type and locations of the device and robot**. The domain includes predicates to represent the state of the environment (e.g., whether a light is on or a door is open or locked) and enables planning agents to achieve goals like preparing a room for occupancy or securing the house before bedtime.

```
(define (domain smart-home)
  (:requirements :strips :typing)
  (:types room device)
  (:predicates
    (robotAt ?x)
    (light-on ?r - room)
    (thermostat-set ?r - room)
    (door-locked ?d - device)
    (door-open ?d - device)
    (in-room ?d - device ?r - room)
    (is-light ?d - device)
    (is-thermostat ?d - device)
    (is-door ?d - device))
```

Complete this action: ✍

```
(:action open-door
    :parameters (?d - device)
    :precondition ...
    :effect ...
  )
```

Challenge: Smart Home Planning

An intelligent robot can perform basic actions in a smart house such as **turning on lights**, **setting room thermostats**, and **opening/locking doors**. Each device (e.g., lights, thermostats, doors) is associated with a specific **room**, and **actions are conditioned on the type and locations of the device and robot**. The domain includes predicates to represent the state of the environment (e.g., whether a light is on or a door is open or locked) and enables planning agents to achieve goals like preparing a room for occupancy or securing the house before bedtime.

```
(define (domain smart-home)
  (:requirements :strips :typing)
  (:types room device)
  (:predicates
    (robotAt ?x)
    (light-on ?r - room)
    (thermostat-set ?r - room)
    (door-locked ?d - device)
    (door-open ?d - device)
    (in-room ?d - device ?r - room)
    (is-light ?d - device)
    (is-thermostat ?d - device)
    (is-door ?d - device))
```

Complete this action: ✍

```
(:action open-door
    :parameters (?d - device)
    :precondition (and (is-door ?d) (at ?d)
                       (not (door-locked ?d)))
    :effect (and (door-open ?d)))
```

✓

Challenge: Smart Home Planning

An intelligent robot can perform basic actions in a smart house such as **turning on lights**, **setting room thermostats**, and **opening/locking doors**. Each device (e.g., lights, thermostats, doors) is associated with a specific **room**, and **actions are conditioned on the type and locations of the device and robot**. The domain includes predicates to represent the state of the environment (e.g., whether a light is on or a door is open or locked) and enables planning agents to achieve goals like preparing a room for occupancy or securing the house before bedtime.

```
(define (domain smart-home)
  (:requirements :strips :typing)
  (:types room device)
  (:predicates
    (robotAt ?x)
    (light-on ?r - room)
    (thermostat-set ?r - room)
    (door-locked ?d - device)
    (door-open ?d - device)
    (in-room ?d - device ?r - room)
    (is-light ?d - device)
    (is-thermostat ?d - device)
    (is-door ?d - device))
```

Complete this action: ✍

```
(:action toggle-light
    :parameters ...
    :precondition ...
    :effect ...
  )
```

Challenge: Smart Home Planning

An intelligent robot can perform basic actions in a smart house such as **turning on lights**, **setting room thermostats**, and **opening/locking doors**. Each device (e.g., lights, thermostats, doors) is associated with a specific **room**, and **actions are conditioned on the type and locations of the device and robot**. The domain includes predicates to represent the state of the environment (e.g., whether a light is on or a door is open or locked) and enables planning agents to achieve goals like preparing a room for occupancy or securing the house before bedtime.

```
(define (domain smart-home)
  (:requirements :strips :typing)
  (:types room device)
  (:predicates
    (robotAt ?x)
    (light-on ?r - room)
    (thermostat-set ?r - room)
    (door-locked ?d - device)
    (door-open ?d - device)
    (in-room ?d - device ?r - room)
    (is-light ?d - device)
    (is-thermostat ?d - device)
    (is-door ?d - device))
```

Complete this action: ✍

```
(:action toggle-light
    :parameters (?d - device)
    :precondition (and (is-light ?d) (in-room ?d ?r))
    :effect (and (when (light-on ?r)
                       (not (light-on ?r)))
                 (when (not (light-on ?r))
                       (light-on ?r))))
```

✖ Conditional effects not part of STRIPS!

Smart-house by ChatGPT! 😉

Here's a **PDDL domain and problem** for a **smart house**. This example models simple actions such as turning lights on/off, adjusting the thermostat, and locking doors.

🏠 PDDL Domain: Smart House

lisp Copy Edit

```lisp
(define (domain smart-home)
  (:requirements :strips :typing)

  (:types
    room device door
  )

  (:predicates
    (light-on ?r - room)
    (thermostat-set ?r - room)
    (door-locked ?d - door)
    (in-room ?d - device ?r - room)
    (is-light ?d - device)
    (is-thermostat ?d - device)
    (is-door ?d - door)
  )

  ;; Action: turn on a light
  (:action turn-on-light
    :parameters (?l - device ?r - room)
    :precondition (and (in-room ?l ?r) (is-light ?l))
    :effect (light-on ?r)
  )

  ;; Action: set thermostat
  (:action set-thermostat
```

📋 PDDL Problem: Secure and Prepare Living Room

lisp Copy Edit

```lisp
(define (problem smart-home-problem)
  (:domain smart-home)

  (:objects
    living-room bedroom - room
    light1 thermo1 - device
    door1 - door
  )

  (:init
    (in-room light1 living-room)
    (in-room thermo1 living-room)
    (is-light light1)
    (is-thermostat thermo1)
    (is-door door1)
  )

  (:goal
    (and
      (light-on living-room)
      (thermostat-set living-room)
      (door-locked door1)
    )
  )
)
```

The International Planning Competition (IPC)

Competition?

"Run competing planners on a set of benchmarks devised by the IPC organizers. Give awards to the most effective planners."

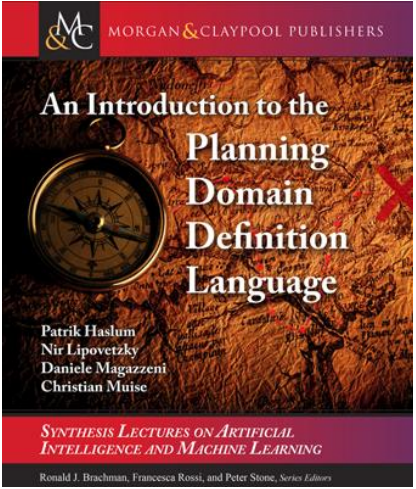
- 1998, 2000, 2002, 2004, 2006, 2008, 2011, 2014, 2018, 2019, 2020, 2023, ...
- PDDL [McDermott and others (1998); Fox and Long (2003); Hoffmann and Edelkamp (2005)]
- $\approx$ 40 domains, $\gg$ 1000 instances, 74 (!!) planners in 2011
- Optimal track vs. satisficing track
- Various others: uncertainty, learning, . . .

http://ipc.icaps-conference.org/

PDDL beyond STRIPS 👍

PDDL can express significantly more than what STRIPS can model, including:

1. Conditional effects (ADL)
2. Universal quantification
3. Typed variables
4. Functions
5. Durative actions
6. Numeric fluents
7. Temporal planning
8. Planning with preferences
9. Axioms (derived predicates)
10. Continous processes PDDL+
11. Non-deterministic actions! 👉 **later**...



First PDDL @ IPC 1998

PDDL — The Planning Domain Definition Language

Version 1.2

This manual was produced by the AIPS-98 Planning Competition Committee:

Malik Ghallab, Ecole Nationale Superieure D'ingenieur des Constructions Aeronautiques
Adele Howe (Colorado State University)
Craig Knoblock, ISI
Drew McDermott (chair) (Yale University)
Ashwin Ram (Georgia Tech University)
Manuela Veloso (Carnegie Mellon University)
Daniel Weld (University of Washington)
David Wilkins (SRI)

It was based on the UCPOP language manual, written by the following researchers from the University of Washington:

Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David E Smith, Ying Sun, & Daniel Weld

Contact Drew McDermott (drew.mcdermott@yale.edu) with comments.

Yale Center for Computational Vision and Control
Tech Report CVC TR-98-003/DCS TR-1165
October, 1998

PDDL 2.1 @ IPC 2002

In the 2002 Competition, planners were set the challenge of considering more complicated domains and problems which feature both temporal and numeric considerations (scheduling and resources). As a result, additions the language were necessary to facilitate modelling time and numbers:

- Level 1: STRIPS fragment.
- Level 2: numeric fluents, functions.
- Level 3: durative actions.
- Level 4: continuous effects/changes.

Journal of Artificial Intelligence Research 20 (2003) 61-124 Submitted 09/02; published 12/03

PDDL2.1 : An Extension to PDDL for Expressing Temporal Planning Domains

Maria Fox MARIA.FOX@CIS.STRATH.AC.UK
Derek Long DEREK.LONG@CIS.STRATH.AC.UK
Department of Computer and Information Sciences
University of Strathclyde, Glasgow, UK

Abstract

In recent years research in the planning community has moved increasingly towards application of planners to realistic problems involving both time and many types of resources. For example, interest in planning demonstrated by the space research community has inspired work in observation scheduling, planetary rover exploration and spacecraft control domains. Other temporal and resource-intensive domains including logistics planning, plant control and manufacturing have also helped to focus the community on the modelling and reasoning issues that must be confronted to make planning technology meet the challenges of application.

The International Planning Competitions have acted as an important motivating force behind the progress that has been made in planning since 1998. The third competition (held in 2002) set the planning community the challenge of handling time and numeric resources. This necessitated the development of a modelling language capable of expressing temporal and numeric properties of planning domains. In this paper we describe the language, PDDL2.1, that was used in the competition. We describe the syntax of the language, its formal semantics and the validation of concurrent plans. We observe that PDDL2.1 has considerable modelling power — exceeding the capabilities of current planning technology — and presents a number of important challenges to the research community.

PDDL+ for Continous Processes and Events

Related to Hybrid Automata!

Journal of Artificial Intelligence Research 27 (2006) 235–297 Submitted 03/06; published 10/06

Modelling Mixed Discrete-Continuous Domains for Planning

Maria Fox MARIA.FOX@CIS.STRATH.AC.UK
Derek Long DEREK.LONG@CIS.STRATH.AC.UK
Department of Computer and Information Sciences
University of Strathclyde,
26 Richmond Street, Glasgow, G1 1XH, UK

Abstract

In this paper we present PDDL+, a planning domain description language for modelling mixed discrete-continuous planning domains. We describe the syntax and modelling style of PDDL+, showing that the language makes convenient the modelling of complex time-dependent effects. We provide a formal semantics for PDDL+ by mapping planning instances into constructs of hybrid automata. Using the syntax of HAs as our semantic model we construct a semantic mapping to labelled transition systems to complete the formal interpretation of PDDL+ planning instances.

An advantage of building a mapping from PDDL+ to HA theory is that it forms a bridge between the Planning and Real Time Systems research communities. One consequence is that we can expect to make use of some of the theoretical properties of HAs. For example, for a restricted class of HAs the Reachability problem (which is equivalent to Plan Existence) is decidable.

PDDL+ provides an alternative to the continuous durative action model of PDDL2.1, adding a more flexible and robust model of time-dependent behaviour.

1. Introduction

This paper describes PDDL+, an extension of the PDDL (McDermott & the AIPS'98 Plan-

Planning Wiki

https://planning.wiki/

PDDL beyond STRIPS 👍

PDDL Version	Year	Features
PDDL 1.0	1998	STRIPS, typing
PDDL 2.1	2003	Numeric fluents, durative actions, functions
PDDL 2.2	2004	Derived predicates, timed initial literals
PDDL 3.0	2005	Trajectory constraints, preferences
PDDL 3.1	2008	Functional fluents
PDDL+	2006	Continuous processes/events (HAs)
PPDDL	2004	Probabilistic effects
FOND-PDDL	2006	Like PPDDL but also non-deterministic effects

Table: PDDL versions and their main features.

Part II

Classical Planning: Methods

Part 2: Classical Planning: Methods

4 Complexity of Planning

5 Planning as heuristic search
- Relaxations
- Delete-relaxation h^+
- From h^+ to h_{max}, h_{add} and h_{FF}
- State of the art classical planners

6 Planning as SAT

Part 2: Classical Planning: Methods

Algorithmic Problems in Planning

Satisficing Planning ✔️

Input: A planning task $P = \langle F, O, I, G \rangle$.
Output: A plan for P, or 'unsolvable' if no plan for P exists.

Optimal Planning 💯

Input: A planning task $P = \langle F, O, I, G \rangle$.
Output: An optimal plan for P, or 'unsolvable' if no plan for P exists.

Algorithmic Problems in Planning

Satisficing Planning ✔️

Input: A planning task $P = \langle F, O, I, G \rangle$.
Output: A plan for P, or 'unsolvable' if no plan for P exists.

Optimal Planning 💯

Input: A planning task $P = \langle F, O, I, G \rangle$.
Output: An optimal plan for P, or 'unsolvable' if no plan for P exists.

🔅 Observations:

- The successful techniques for either one of these are almost disjoint!
- Satisficing planning is much more effective in practice.
- Programs solving these problems are called (optimal) planners, planning systems, or planning tools.

Decision Problems in Planning

PlanEx: Satisficing Planning ✔️

The problem of deciding, given a planning task P, whether or not there exists a plan for P.

PlanLen: Optimal Planning 💯

The problem of deciding, given a planning task P and an integer B (bound), whether or not there exists a plan for P of length at most B.

Review of Complexity: **P**, **NP** and **PSPACE**

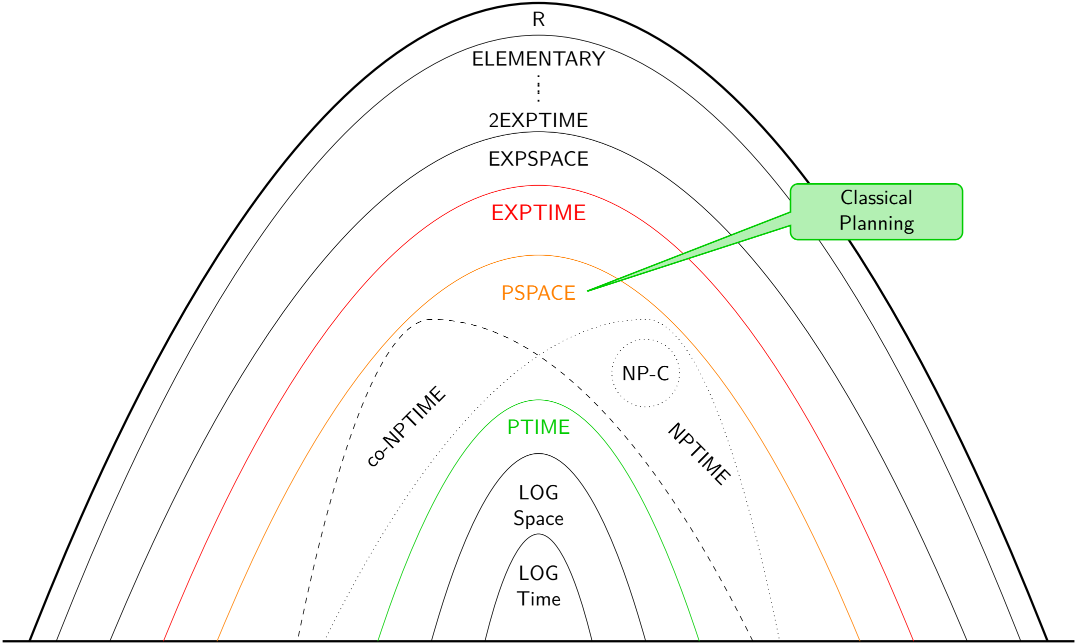
Turing Machine (TM)

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, and whether the R/W head moves left or right or remains where it is. Some internal states are accepting ('yes'; else 'no').

Thre Complexity Classes for Decision Problems

1. **P**: Decision problems for which there exists a deterministic TM that runs in *time* polynomial (in the size of its input).
2. **NP**: Decision problems for which there exists a non-deterministic TM that runs in *time* polynomial. Accepts if at least one of the possible runs accepts.
3. **PSPACE**: Decision problems for which there exists a deterministic TM that runs in *space* polynomial in the size of its input.

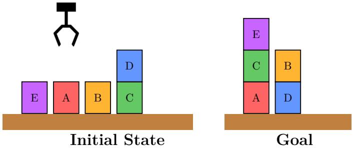
Planning is hard!



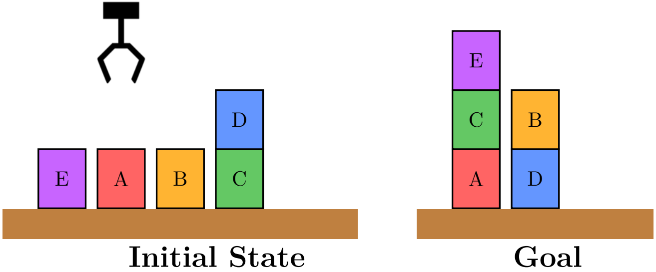
Domain-Specific: **PlanEx** vs. **PlanLen**

- In general, both have the same complexity (PSPACE-complete).
- Within particular applications, bounded length plan existence (i.e., optimal planning) is often harder than plan existence.
- This happens in many IPC benchmark domains.
- PlanLen is **NP**-complete while PlanEx is in **P**.
  - For example: Blocksworld and Logistics.

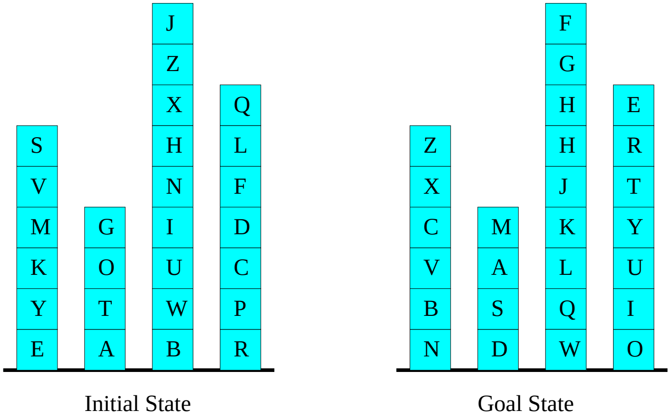
⚠ In practice, optimal planning is (almost) never "easy".



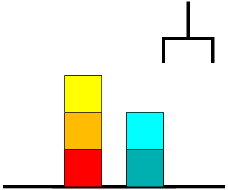
The Blocksworld is Hard?



The Blocksworld is Hard!

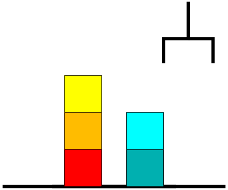


So, why all the fuss?



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

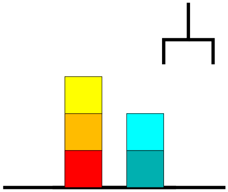
So, why all the fuss?



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

blocks	states	blocks	states
1	1	9	4596553
2	3	10	58941091
3	13	11	824073141
4	73	12	12470162233
5	501	13	202976401213
6	4051	14	3535017524403
7	37633	15	65573803186921
8	394353	16	1290434218669921

So, why all the fuss?

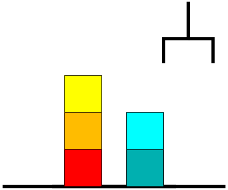


- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

blocks	states	blocks	states
1	1	9	4596553
2	3	10	58941091
3	13	11	824073141
4	73	12	12470162233
5	501	13	202976401213
6	4051	14	3535017524403
7	37633	15	65573803186921
8	394353	16	1290434218669921

State spaces may be huge. In particular, the state space is typically exponentially large in the size of the factored (compact) specification of the problem.

So, why all the fuss?



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

blocks	states	blocks	states
1	1	9	4596553
2	3	10	58941091
3	13	11	824073141
4	73	12	12470162233
5	501	13	202976401213
6	4051	14	3535017524403
7	37633	15	65573803186921
8	394353	16	1290434218669921

State spaces may be huge. In particular, the state space is typically exponentially large in the size of the factored (compact) specification of the problem.

☼ In other words: Search problems typically are computationally hard (e.g., optimal Blocksworld solving is NP-complete).

Computation: how to solve STRIPS planning problems?

Key idea

Exploit two roles of **language**:

1. specification: concise and accessible model description.
2. computation: reveal useful heuristic information (structure).

Two traditional approaches: search vs. decomposition

1. explicit search of the state model $S(P)$ direct but not effective until "recently".
2. near decomposition of the planning problem thought a better idea.

Computational Approaches to Classical Planning

- **General Problem Solver (GPS) and Strips** (50's-70's): mean-ends analysis, decomposition, regression, ...
- **Partial Order (POCL) Planning** (80's): work on any open subgoal, resolve threats; UCPOP 1992.
- **Graphplan (1995 – 2000)**: build graph containing all possible **parallel** plans up to certain length; then extract plan by searching the graph backward from Goal.
- **SATPlan** (1996 – ...): map planning problem given horizon into SAT problem; use state-of-the-art SAT solver.
- **Heuristic Search Planning** (1996 – ...): search state space $\mathcal{S}(P)$ with heuristic function h extracted from problem P.
- **Model Checking Planning** (1998 – ...): search state space $\mathcal{S}(P)$ with 'symbolic' Breadth first search where sets of states represented by formulas implemented by BDDs ...

State of the Art in Classical Planning

- Significant **progress** since Graphplan.
- **Empirical methodology**:
  1. standard PDDL language
  2. planners and benchmarks available; competitions
  3. focus on performance and scalability
- **Large problems solved** (non-optimally).
- Different **formulations** and **ideas**
  1. Planning as **Heuristic Search**. 👈
  2. Planning as **SAT**. 👈
  3. **Other:** Local Search (LPG), Monte-Carlo Search (Arvand), ...

We'll focus on **1** mainly, and partially on **2**.

Part 2: Classical Planning: Methods

4 Complexity of Planning

5 Planning as heuristic search
- Relaxations
- Delete-relaxation h^+
- From h^+ to $h_{\max}$, h_{add} and h_{FF}
- State of the art classical planners

6 Planning as SAT

Part 2: Classical Planning: Methods

Computation: How to Solve Classical Planning Problems?

- Planning is one of the oldest areas in AI; many ideas have been tried
  - *A bit of **history**: first AI planners from late 50s: **GPS** (Simon and Newell)*

$$Problem \Longrightarrow \boxed{Planner} \Longrightarrow Plan$$

- We focus on two of the ideas that scale up best **computationally**:
  1. Planning as **Heuristic Search**.
  2. Planning as **SAT**.

- These methods are able to solve problems over huge state spaces.

! But some domains are inherently hard, and for them, **general, domain-independent planners** unlikely to approach **specialized methods**.

Planning as Heuristic Search

- STRIPS $P = \langle F, O, I, G \rangle$ encodes model $\mathcal{S}(P) = \langle S, s_0, S_G, Act, A, f, c \rangle$
- Finding a **plan** in $\mathcal{S}(P)$ reduces to **finding a path/reachability** in a graph where:
  - **Nodes** represent the **states** s in the model
  - **Edges** (s, s') capture corresponding transitions $s' = f(a, s)$, $a \in A(s)$
- State models and graphs given **implicitly** by P.

Planning as Heuristic Search

- STRIPS $P = \langle F, O, I, G \rangle$ encodes model $\mathcal{S}(P) = \langle S, s_0, S_G, Act, A, f, c \rangle$
- Finding a **plan** in $\mathcal{S}(P)$ reduces to **finding a path/reachability** in a graph where:
  - **Nodes** represent the **states** s in the model
  - **Edges** (s, s') capture corresponding transitions $s' = f(a, s)$, $a \in A(s)$
- State models and graphs given **implicitly** by P.
- Their sizes are **exponential** in number of atoms in F.

‼️ It's critical to use **heuristic functions** to guide the search.

⚠️ If the user had to supply the heuristic function by hand, then we would lose some of the selling points: generality + autonomy + flexibility + rapid prototyping.

Planning as Heuristic Search

- STRIPS $P = \langle F, O, I, G \rangle$ encodes model $\mathcal{S}(P) = \langle S, s_0, S_G, Act, A, f, c \rangle$
- Finding a **plan** in $\mathcal{S}(P)$ reduces to **finding a path/reachability** in a graph where:
  - **Nodes** represent the **states** s in the model
  - **Edges** (s, s') capture corresponding transitions $s' = f(a, s)$, $a \in A(s)$
- State models and graphs given **implicitly** by P.
- Their sizes are **exponential** in number of atoms in F.

‼️ It's critical to use **heuristic functions** to guide the search.

⚠️ If the user had to supply the heuristic function by hand, then we would lose some of the selling points: generality + autonomy + flexibility + rapid prototyping.

Question

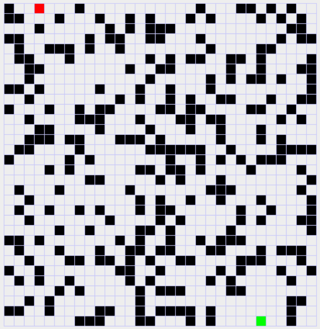
How to get heuristic functions **automatically** from P itself?

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.



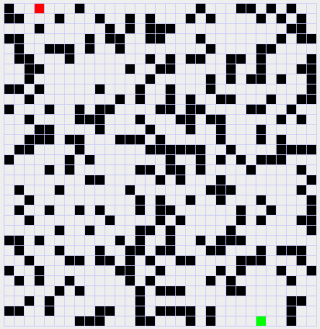
Why is navigation hard?

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.



Why is navigation hard?

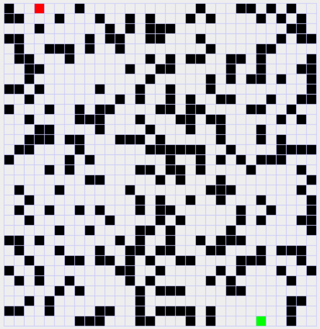
Because of obstacles!

Heuristics: where they come from? 🤔

General idea for obtaining heuristics

Heuristic functions obtained as **optimal cost functions** of **relaxed problems**.

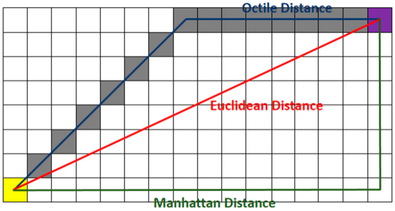
- Routing Finding: Manhattan distance or straight line.
- N-puzzle: # misplaced tiles or sum of Manhattan distances.
- Travelling Salesman Problem: Spanning Tree.



Why is navigation hard?

Because of obstacles!

So, suppose you can flight or walk through walls!



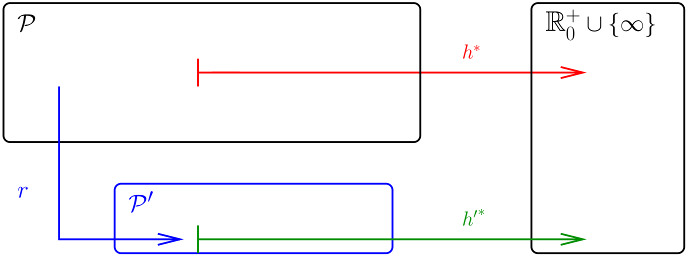
How to Relax Informally

☼ Relaxation means to **simplify** the problem, and take the **solution to the simpler problem as the heuristic estimate** for the solution to the actual problem.

How to Relax Informally

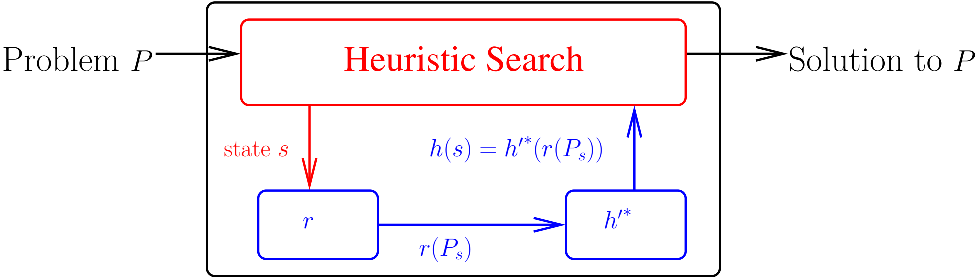
Relaxation means to **simplify** the problem, and take the **solution to the simpler problem as the heuristic estimate** for the solution to the actual problem.

- You have a problem, $P \in \mathcal{P}$, whose perfect heuristic h^* you wish to estimate.
- You define a simpler problem, $P' \in \mathcal{P}'$, whose perfect heuristic h'^* can be used to estimate h^*.
- You define a transformation, r, that simplifies instances from $\mathcal{P}$ into instances $\mathcal{P}'$.
- Given problem instance $P \in \mathcal{P}$, you estimate $h^*(P)$ by $h'^*(r(P))$.



How to Relax During Search: Diagram

Using a relaxation $\mathcal{R} = (\mathcal{P}', r, h'^*)$ during search:



- Π_s: Π with initial state replaced by s, i.e., $\Pi = (F, A, c, I, G)$ changed to (F, A, c, s, G).
  ➠ That is, the task of finding a plan for state s.

☼ So, during search, the relaxation is used only inside the computation of the heuristic function on each state; the relaxation does not affect anything else. 👍

Relaxations: Navigation

Navigation in 4-connected grid with obstacles:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     (not (obstacle ?nextpos)))
   :effect (and (at ?nextpos)
                (not (at ?curpos))))
```

P': can go through walls, drop obstacle preconditions:

Relaxations: Navigation

Navigation in 4-connected grid with obstacles:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     (not (obstacle ?nextpos)))
   :effect (and (at ?nextpos)
                (not (at ?curpos))))
```

P': can go through walls, drop obstacle preconditions:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     ;; drop obstacle precondition
                     )
   :effect (and (at ?nextpos)
                (not (at-robot ?curpos))))
```

What is h'^* for the **relaxed problem**?

Relaxations: Navigation

Navigation in 4-connected grid with obstacles:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     (not (obstacle ?nextpos)))
   :effect (and (at ?nextpos)
                (not (at ?curpos))))
```

P': can go through walls, drop obstacle preconditions:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     ;; drop obstacle precondition
                     )
   :effect (and (at ?nextpos)
                (not (at-robot ?curpos))))
```

What is h'^* for the **relaxed problem**?
Manhattan Distance! $(|x - goal.x| + |y - goal.y|)$

Relaxations: Navigation

Navigation in 4-connected grid with obstacles:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     (not (obstacle ?nextpos)))
   :effect (and (at ?nextpos)
                (not (at ?curpos))))
```

P': can go through walls, drop obstacle preconditions:

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     ;; drop obstacle precondition
                     )
   :effect (and (at ?nextpos)
                (not (at-robot ?curpos))))
```

What is h'^* for the **relaxed problem**?

Manhattan Distance! $(|x - goal.x| + |y - goal.y|)$

⚠ But, *how do we know which predicate to drop?*

Relaxations: N-Puzzle


```
(:action slide
   :parameters (?t ?s1 ?s2)
   :precondition (and (at ?t ?s1) (blank ?s2)
                      (connected ?s1 ?s2))
   :effect (and (at ?t ?s2) (blank ?s1)
                (not (at ?t ?s1)) (not (blank ?s2))))
```

Proposal 1: P': ignore blanks; can overlap tiles

Relaxations: N-Puzzle


```
(:action slide
   :parameters (?t ?s1 ?s2)
   :precondition (and (at ?t ?s1) (blank ?s2)
                      (connected ?s1 ?s2))
   :effect (and (at ?t ?s2) (blank ?s1)
                (not (at ?t ?s1)) (not (blank ?s2))))
```

Proposal 1: P': ignore blanks; can overlap tiles

```
(:action slide
   :parameters (?t ?s1 ?s2)
   :precondition (and (at ?t ?s1) ;; drop blank
                      (connected ?s1 ?s2))
   :effect (and (at ?t ?s2)
                (not (at ?t ?s1))))
```

h'^*: **Manhattan Distance!**

In the example: $h'^* = 2 + 0 + 5 + \cdots + 2 + 0 + 5$

Relaxations: N-Puzzle


```
(:action slide
    :parameters (?t ?s1 ?s2)
    :precondition (and (at ?t ?s1) (blank ?s2)
                       (connected ?s1 ?s2))
    :effect (and (at ?t ?s2) (blank ?s1)
                 (not (at ?t ?s1)) (not (blank ?s2))))
```

Proposal 2: P': can lift and move tiles together

Relaxations: N-Puzzle


```
(:action slide
    :parameters (?t ?s1 ?s2)
    :precondition (and (at ?t ?s1) (blank ?s2)
                       (connected ?s1 ?s2))
    :effect (and (at ?t ?s2) (blank ?s1)
                 (not (at ?t ?s1)) (not (blank ?s2))))
```

Proposal 2: P': can lift and move tiles together

```
 (:action slide
    :parameters (?t ?s1 ?s2)
    :precondition (and (at ?t ?s1)) ;; drop blank
    :effect (and (at ?t ?s2)          ;;  and connected
                 (not (at ?t ?s1))))
```

h'^*: **Misplaced tiles**

In the example: $h'^* = 15$

Relaxations: N-Puzzle


```
(:action slide
    :parameters (?t ?s1 ?s2)
    :precondition (and (at ?t ?s1) (blank ?s2)
                       (connected ?s1 ?s2))
    :effect (and (at ?t ?s2) (blank ?s1)
                 (not (at ?t ?s1)) (not (blank ?s2))))
```

<u>**Proposal 2:**</u> P': can lift and move tiles together

```
 (:action slide
    :parameters (?t ?s1 ?s2)
    :precondition (and (at ?t ?s1)) ;; drop blank
    :effect (and (at ?t ?s2)         ;;  and connected
                 (not (at ?t ?s1))))
```

h'^*: **Misplaced tiles**

In the example: $h'^* = 15$

⚠️ Again, *how do we know which predicate to drop?*

Goal Counting Relaxation

Let's act as if every action is possible and no 'undos':

1. Drop all preconditions — all is executable.
2. Drop all negative effects — no undos.

Goal Counting Relaxation

Let's act as if every action is possible and no 'undos':

1. Drop all preconditions — all is executable.
2. Drop all negative effects — no undos.

- **Problem** P: All STRIPS planning tasks.
- **Simpler problem** P': All STRIPS planning tasks with empty preconditions and deletes.
- **Perfect heuristic** h'^* **for** P': Optimal plan cost wrt P'.
- **Transformation** r**:** Drop the preconditions and deletes.

Goal Counting Relaxation

Let's act as if every action is possible and no 'undos':

1. Drop all preconditions — all is executable.
2. Drop all negative effects — no undos.

- **Problem** P: All STRIPS planning tasks.
- **Simpler problem** P': All STRIPS planning tasks with empty preconditions and deletes.
- **Perfect heuristic** h'^* **for** P': Optimal plan cost wrt P'.
- **Transformation** r**:** Drop the preconditions and deletes.

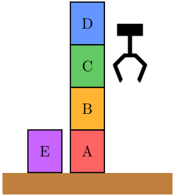
```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at ?curpos)
                      (connected ?curpos ?nextpos)
                     (not (obstacle ?nextpos)))
   :effect (and (at ?nextpos) (not (at ?curpos))
                (visited ?nextpos)))
(:goal (and (visited loc-x0-y0)
            (visited loc-x0-y1)
            (visited loc-x0-y3 )))
```

Relaxation P':

```
(:action move
   :parameters (?curpos ?nextpos)
   :precondition ()
   :effect (and (at-robot ?nextpos)
                (visited ?nextpos)))
```

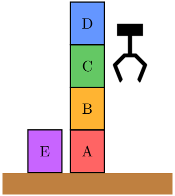
What is h'^* for P'?

Precondition + Delete Relaxation in Blocksworld




```
(:action put_down
    :parameters (?x)
    :precondition (holding ?x)
    :effect (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action unstack
     :parameters (?x ?y)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect (and (clear ?y) (holding ?x) (not (on ?x ?y))
                 (not (clear ?x)) (not (handempty))))

(:goal (and (holding d) (clear b)))
```

Precondition + Delete Relaxation in Blocksworld




```
(:action put_down
    :parameters (?x)
    :precondition (holding ?x)
    :effect (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action unstack
     :parameters (?x ?y)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect (and (clear ?y) (holding ?x) (not (on ?x ?y))
                  (not (clear ?x)) (not (handempty))))

(:goal (and (holding d) (clear b)))
```

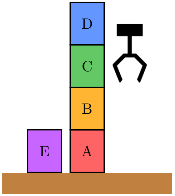
Relaxation P':

```
(:action put_down
    :parameters (?x)
    :precondition ()
    :effect  (and (clear ?x) (handempty) (ontable ?x)))
(:action unstack
     :parameters (?x ?y)
     :precondition ()
     :effect (and (clear ?y) (holding ?x)))
```

Plan $pickup(d), putdown(b)$ works for P'.

? *Is then $h'^* = 2$?*

Precondition + Delete Relaxation in Blocksworld




```
(:action put_down
     :parameters (?x)
     :precondition (holding ?x)
     :effect (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
  (:action unstack
      :parameters (?x ?y)
      :precondition (and (on ?x ?y) (clear ?x) (handempty))
      :effect (and (clear ?y) (holding ?x) (not (on ?x ?y))
                   (not (clear ?x)) (not (handempty))))

(:goal (and (holding d) (clear b)))
```

Relaxation P':

```
(:action put_down
     :parameters (?x)
     :precondition ()
     :effect  (and (clear ?x) (handempty) (ontable ?x)))
(:action unstack
      :parameters (?x ?y)
      :precondition ()
      :effect (and (clear ?y) (holding ?x)))
```

Plan $pickup(d), putdown(b)$ works for P'.

Is then $h'^* = 2$? **No!** $h'^* = 1$! Optimal plan is $unstack(d, b)$ 😉

Precondition + Delete Relaxation vs. Goal Counting

Let's act "as if every action is possible and no 'undos"':

1. Drop all preconditions — all is executable.
2. Drop all negative effects — no undos.

Yet:

Precondition + Delete Relaxation vs. Goal Counting

Let's act "as if every action is possible and no 'undos"':

1. Drop all preconditions — all is executable.
2. Drop all negative effects — no undos.

Yet:

⚠️ Optimal STRIPS planning with empty preconditions and deletes is still NP-hard!

👉 (Reduction from MINIMUM COVER, of goal set by add lists.)

Precondition + Delete Relaxation vs. Goal Counting

Let's act "as if every action is possible and no 'undos"':

1. Drop all preconditions — all is executable.
2. Drop all negative effects — no undos.

Yet:

⚠️ Optimal STRIPS planning with empty preconditions and deletes is still NP-hard!

☞ (Reduction from MINIMUM COVER, of goal set by add lists.)

Need to approximate the perfect heuristic h'^* for $\mathcal{P}'$.

Hence **goal counting**: just approximate h'^* by $h^\sharp$ = number-of-false-goals.

Challenge!

Question

We have a robot with one gripper, two rooms A and B, and n balls to be transported from A to B. The actions available are $move$, $pickBall$ and $dropBall$; say $h =$ "number of balls not yet in room B". Can h be derived as $h^{\mathcal{R}}$ for a relaxation $\mathcal{R}$?

1. No.
2. Yes, just drop the deletes.
3. Sure, *every* admissible h can be derived via a relaxation.
4. I'd rather relax at the beach. 🏖️

Challenge!

Question

We have a robot with one gripper, two rooms A and B, and n balls to be transported from A to B. The actions available are $move$, $pickBall$ and $dropBall$; say $h =$ "number of balls not yet in room B". Can h be derived as $h^{\mathcal{R}}$ for a relaxation $\mathcal{R}$?

1. No.
2. Yes, just drop the deletes.
3. Sure, *every* admissible h can be derived via a relaxation.
4. I'd rather relax at the beach. 🏖️

1. Incorrect. We can define $\mathcal{P}'$ as the problem of computing the cardinality of a finite set, and define r as the function that maps a state to the set of balls not yet in room B.

Challenge!

Question

We have a robot with one gripper, two rooms A and B, and n balls to be transported from A to B. The actions available are $move$, $pickBall$ and $dropBall$; say $h =$ "number of balls not yet in room B". Can h be derived as $h^{\mathcal{R}}$ for a relaxation $\mathcal{R}$?

1. No.
2. Yes, just drop the deletes.
3. Sure, *every* admissible h can be derived via a relaxation.
4. I'd rather relax at the beach. 🏖

1. Incorrect. We can define $\mathcal{P}'$ as the problem of computing the cardinality of a finite set, and define r as the function that maps a state to the set of balls not yet in room B.
2. Incorrect, should drop *preconditions* (and deletes).

Challenge!

Question

We have a robot with one gripper, two rooms A and B, and n balls to be transported from A to B. The actions available are $move$, $pickBall$ and $dropBall$; say $h =$ "number of balls not yet in room B". Can h be derived as $h^{\mathcal{R}}$ for a relaxation $\mathcal{R}$?

1. No.
2. Yes, just drop the deletes.
3. Sure, *every* admissible h can be derived via a relaxation.
4. I'd rather relax at the beach. 🏖️

1. Incorrect. We can define $\mathcal{P}'$ as the problem of computing the cardinality of a finite set, and define r as the function that maps a state to the set of balls not yet in room B.
2. Incorrect, should drop *preconditions* (and deletes).
3. Yes. Given admissible $h : \mathcal{P} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$, we can simply define $\mathcal{P}' := \mathcal{P}$ and take r to be the identity function $id_{\mathcal{P}}$. In other words, $\mathcal{R} := (\mathcal{P}, id_{\mathcal{P}}, h)$ is a relaxation with $h^{\mathcal{R}} = h$.

Challenge!

Question

We have a robot with one gripper, two rooms A and B, and n balls to be transported from A to B. The actions available are $move$, $pickBall$ and $dropBall$; say $h =$ "number of balls not yet in room B". Can h be derived as $h^{\mathcal{R}}$ for a relaxation $\mathcal{R}$?

1. No.
2. Yes, just drop the deletes.
3. Sure, *every* admissible h can be derived via a relaxation.
4. I'd rather relax at the beach. 🏖

1. Incorrect. We can define $\mathcal{P}'$ as the problem of computing the cardinality of a finite set, and define r as the function that maps a state to the set of balls not yet in room B.
2. Incorrect, should drop *preconditions* (and deletes).
3. Yes. Given admissible $h : \mathcal{P} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$, we can simply define $\mathcal{P}' := \mathcal{P}$ and take r to be the identity function $id_{\mathcal{P}}$. In other words, $\mathcal{R} := (\mathcal{P}, id_{\mathcal{P}}, h)$ is a relaxation with $h^{\mathcal{R}} = h$.
4. **Me, too!** 😉

Remarks

⚠️ **Is Goal Counting any good?**

The goal-counting approximation $h^\sharp =$ "count the number of goals currently not true" is **a very** **uninformative** **heuristic function:** 😢

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp =$ "count the number of goals currently not true" is **a very uninformative heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp$ = "count the number of goals currently not true" is **a very uninformative heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.
2. We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. *How?*

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp =$ "count the number of goals currently not true" is **a very** **uninformative** **heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.
2. We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. *How?*
   - *Replace goal by new fact g and add a new action achieving g with precondition G.*

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp =$ "count the number of goals currently not true" is **a very** **uninformative** **heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.
2. We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. *How?*
   - *Replace goal by new fact g and add a new action achieving g with precondition G.*
3. Ignores almost all structure: Heuristic value does not depend on the actions at all!

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp =$ "count the number of goals currently not true" is **a very** **uninformative** **heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.
2. We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. *How?*
   - *Replace goal by new fact g and add a new action achieving g with precondition G.*
3. Ignores almost all structure: Heuristic value does not depend on the actions at all!
   - *Dropping preconditions is "too much".*

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp$ = "count the number of goals currently not true" is **a very** **uninformative** **heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.
2. We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. *How?*
   - *Replace goal by new fact g and add a new action achieving g with precondition G.*
3. Ignores almost all structure: Heuristic value does not depend on the actions at all!
   - *Dropping preconditions is "too much".*

Remarks

⚠️ Is Goal Counting any good?

The goal-counting approximation $h^\sharp$ = "count the number of goals currently not true" is **a very** **uninformative** **heuristic function:** 😢

1. Range of heuristic values is small $(0 \ldots |G|)$.
2. We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. *How?*
   - *Replace goal by new fact g and add a new action achieving g with precondition G.*
3. Ignores almost all structure: Heuristic value does not depend on the actions at all!
   - *Dropping preconditions is "too much".*

💡 Let's next see how to compute **much** better (more informed) heuristic functions (still automatically from the PDDL description!).

Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Real world: (before)

Reminder: Relaxing the World by Ignoring Delete Lists

“What was once true remains true forever.”

Real world: (after)

Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Relaxed world: (before)

Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Relaxed world: (after)

Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Real world: (before)

Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Real world: (after)

Reminder: Relaxing the World by Ignoring Delete Lists

“What was once true remains true forever.”

Relaxed world: (before)

Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Relaxed world: (after)

Heuristics for Classical Planning

- Heuristics derived from **relaxation** where **delete-lists** of actions are **dropped**.
  - *That is, delete all* `(not ...)` *clauses in the each action's* `:effect` *in the PDDL*

- This simplification is called the **delete-relaxation**.

- Define delete-relaxation heuristic $h^+(s)$ as:

$$h^+(s) \stackrel{\text{def}}{=} h^*_{P'}(s)$$

where P' **is delete-relaxation of** P, $P(s)$ is P but with s as initial state, and $h^*_P(s)$ is optimal cost of $P(s)$.

Heuristics for Classical Planning

- Heuristics derived from **relaxation** where **delete-lists** of actions are **dropped**.
  - *That is, delete all* `(not ...)` *clauses in the each action's* `:effect` *in the PDDL*

- This simplification is called the **delete-relaxation**.

- Define delete-relaxation heuristic $h^+(s)$ as:

$$h^+(s) \stackrel{\text{def}}{=} h^*_{P'}(s)$$

where P' **is delete-relaxation of** P, $P(s)$ is P but with s as initial state, and $h^*_P(s)$ is optimal cost of $P(s)$.

✓ Delete relaxation is **admissible** (i.e., optimistic):
  - Applying a relaxed action can only ever make more facts true.
  - That can only be good, i.e., cannot render the task unsolvable

✓ Keeps actions' preconditions, and thus the causal "structure"

? **... but what does it "mean"?**

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x,y)) = \begin{cases} 1 & x, y \in \{Syd, Bri\} \\ 1.5 & x, y \in \{Syd, Ade\} \\ 3.5 & x, y \in \{Ade, Per\} \\ 4 & x, y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

Planning vs. Relaxed Planning:

- Optimal plan:

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x,y)) = \begin{cases} 1 & x, y \in \{Syd, Bri\} \\ 1.5 & x, y \in \{Syd, Ade\} \\ 3.5 & x, y \in \{Ade, Per\} \\ 4 & x, y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x,y)) = \begin{cases} 1 & x,y \in \{Syd, Bri\} \\ 1.5 & x,y \in \{Syd, Ade\} \\ 3.5 & x,y \in \{Ade, Per\} \\ 4 & x,y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

Planning vs. Relaxed Planning:

- Optimal plan: $drive(Syd, Bri)$, $drive(Bri, Syd)$, $drive(Syd, Ade)$, $drive(Ade, Per)$, $drive(Per, Ade)$, $drive(Ade, Dar)$, $drive(Dar, Ade)$, $drive(Ade, Syd)$.
- Optimal relaxed:

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x,y)) = \begin{cases} 1 & x, y \in \{Syd, Bri\} \\ 1.5 & x, y \in \{Syd, Ade\} \\ 3.5 & x, y \in \{Ade, Per\} \\ 4 & x, y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

Planning vs. Relaxed Planning:

- Optimal plan: $drive(Syd, Bri)$, $drive(Bri, Syd)$, $drive(Syd, Ade)$, $drive(Ade, Per)$, $drive(Per, Ade)$, $drive(Ade, Dar)$, $drive(Dar, Ade)$, $drive(Ade, Syd)$.
- Optimal relaxed: $drive(Syd, Bri), drive(Syd, Ade), drive(Ade, Per), drive(Ade, Dar)$
- So, $h^*(I) =$

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x,y)) = \begin{cases} 1 & x, y \in \{Syd, Bri\} \\ 1.5 & x, y \in \{Syd, Ade\} \\ 3.5 & x, y \in \{Ade, Per\} \\ 4 & x, y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

Planning vs. Relaxed Planning:

- Optimal plan: $drive(Syd, Bri)$, $drive(Bri, Syd)$, $drive(Syd, Ade)$, $drive(Ade, Per)$, $drive(Per, Ade)$, $drive(Ade, Dar)$, $drive(Dar, Ade)$, $drive(Ade, Syd)$.
- Optimal relaxed: $drive(Syd, Bri), drive(Syd, Ade), drive(Ade, Per), drive(Ade, Dar)$
- So, $h^*(I) = 20$

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x, y)) = \begin{cases} 1 & x, y \in \{Syd, Bri\} \\ 1.5 & x, y \in \{Syd, Ade\} \\ 3.5 & x, y \in \{Ade, Per\} \\ 4 & x, y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

Planning vs. Relaxed Planning:

- Optimal plan: $drive(Syd, Bri)$, $drive(Bri, Syd)$, $drive(Syd, Ade)$, $drive(Ade, Per)$, $drive(Per, Ade)$, $drive(Ade, Dar)$, $drive(Dar, Ade)$, $drive(Ade, Syd)$.
- Optimal relaxed: $drive(Syd, Bri), drive(Syd, Ade), drive(Ade, Per), drive(Ade, Dar)$
- So, $h^*(I) = 20$ and $h^+(I) =$

Visiting Australia Cities with h^+

Problem: starting from Sydney, visit Brisbane, Adelaide, Perth, and Darwin. Can only use highways. Take set of cities $C = \{Syd, Ade, Bri, Per, Ade, Dar\}$.

- P: $at(x)$ and $visited(x)$, for $x \in C$.
- A: $drive(x, y)$ where $x \neq y$ have a high-way.

$$c(drive(x,y)) = \begin{cases} 1 & x, y \in \{Syd, Bri\} \\ 1.5 & x, y \in \{Syd, Ade\} \\ 3.5 & x, y \in \{Ade, Per\} \\ 4 & x, y \in \{Ade, Dar\} \end{cases}$$

- $I = \{at(Syd), visited(Syd)\}$;
- $G = \{at(Syd)\} \cup \{visited(x) \mid x \in C\}$.

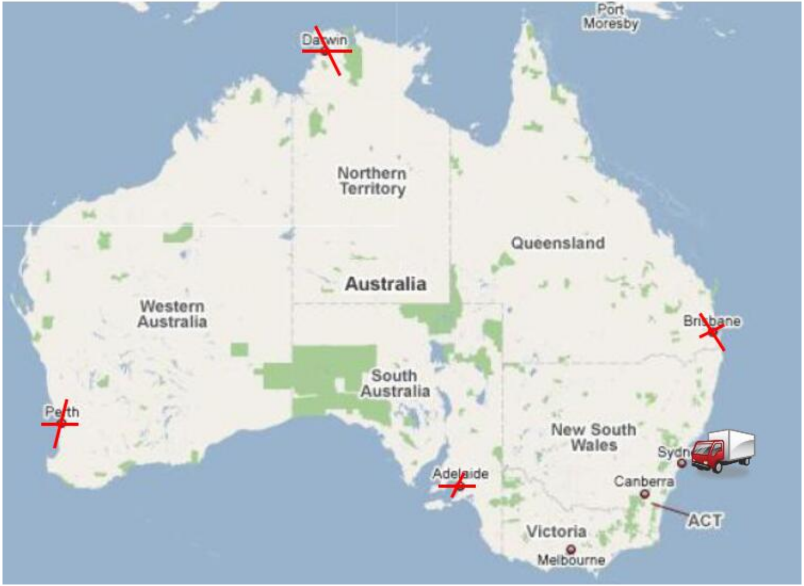
Planning vs. Relaxed Planning:

- Optimal plan: $drive(Syd, Bri)$, $drive(Bri, Syd)$, $drive(Syd, Ade)$, $drive(Ade, Per)$, $drive(Per, Ade)$, $drive(Ade, Dar)$, $drive(Dar, Ade)$, $drive(Ade, Syd)$.
- Optimal relaxed: $drive(Syd, Bri), drive(Syd, Ade), drive(Ade, Per), drive(Ade, Dar)$
- So, $h^*(I) = 20$ and $h^+(I) = 10$.

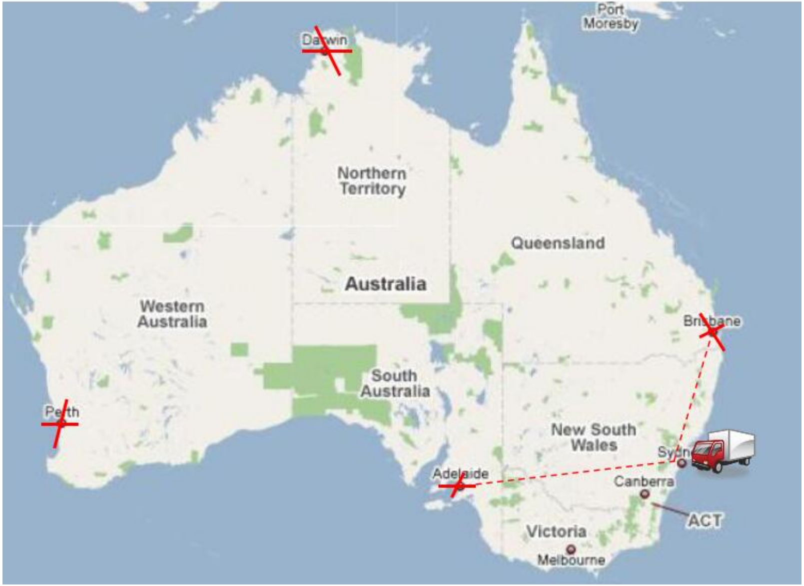
What does h^+ give us?

What does h^+ give us?

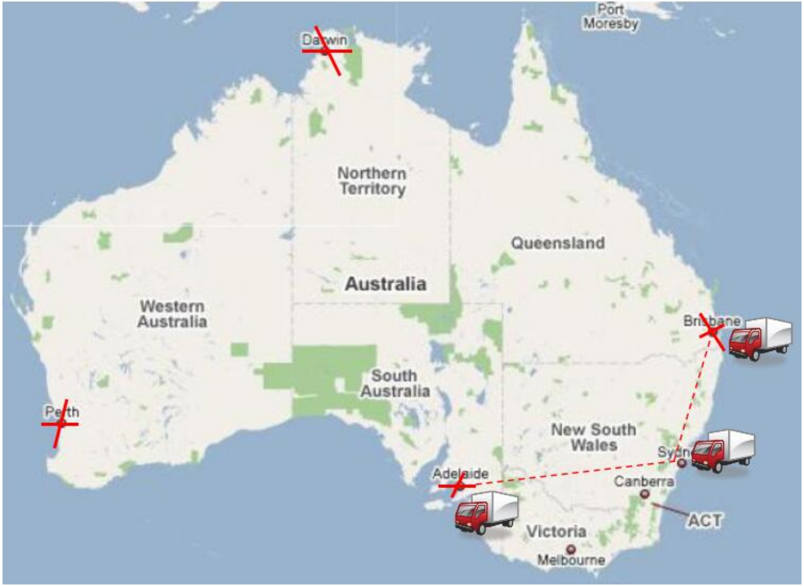
What does h^+ give us?



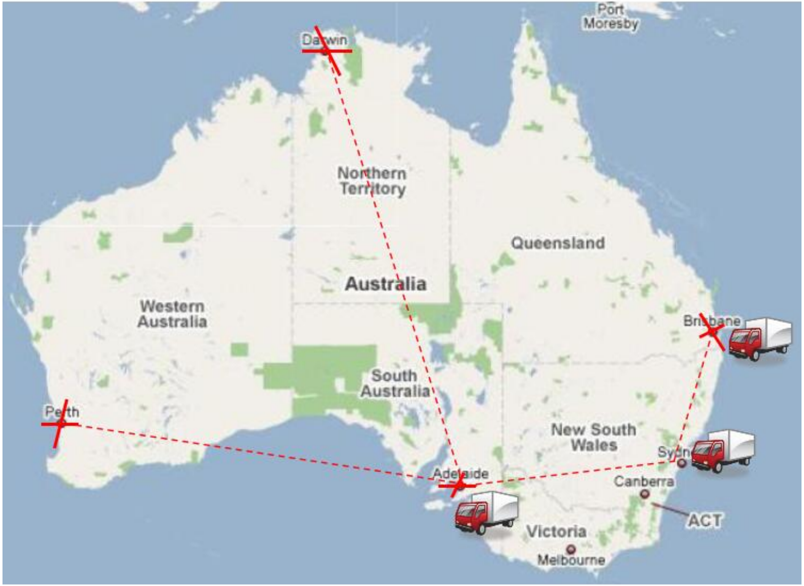
What does h^+ give us?



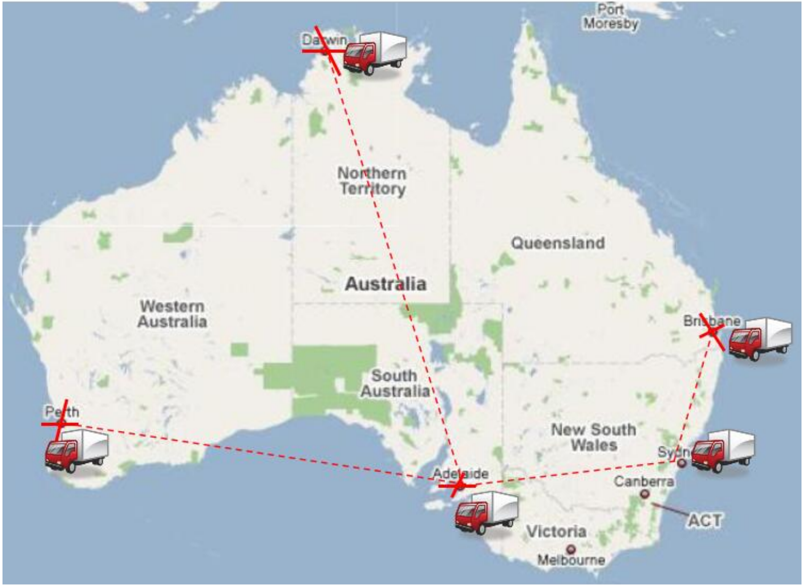
What does h^+ give us?



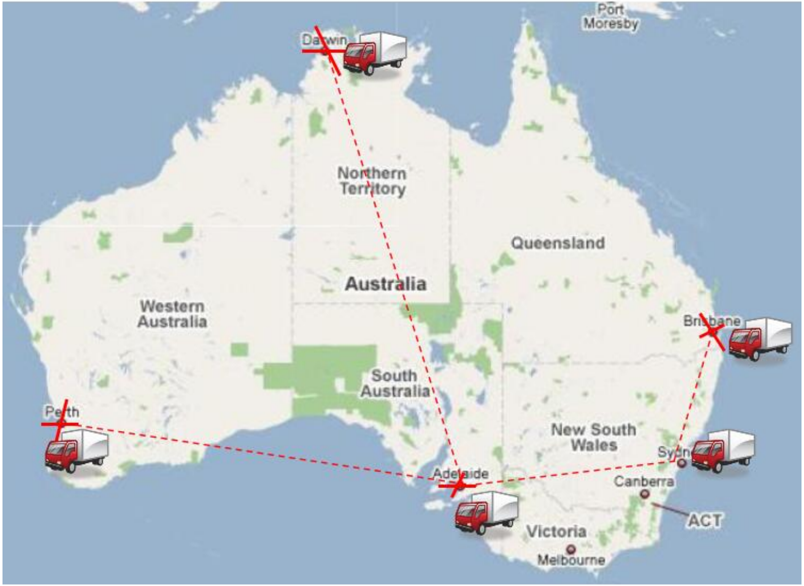
What does h^+ give us?



What does h^+ give us?

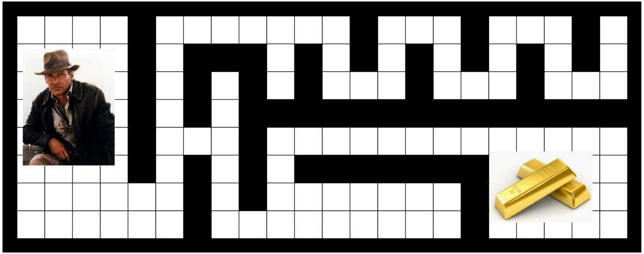


What does h^+ give us?



h^+(**Visit Autralia**) = **Minimum Spanning Tree!**

Challenge!

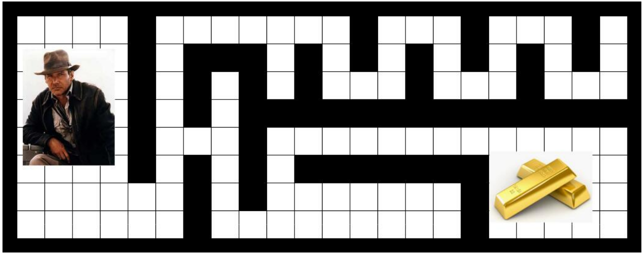


3805 8489 @ menti.com

Question: What is h^+ for this domain?

1. Manhattan Distance.
2. h^*.
3. Horizontal distance.
4. Vertical distance.

Challenge!

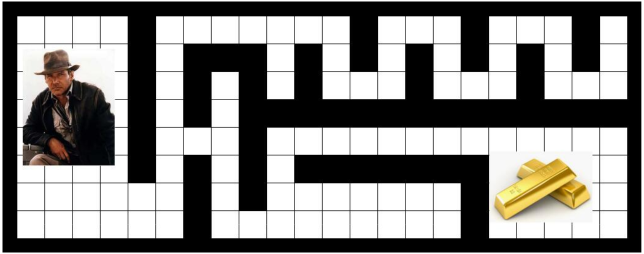


3805 8489 @ menti.com

Question: What is h^+ for this domain?

1. Manhattan Distance. **No**, relaxed plans can't walk through walls.
2. h^*.
3. Horizontal distance.
4. Vertical distance.

Challenge!

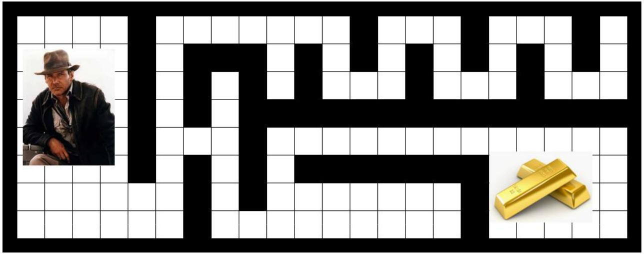


3805 8489 @ menti.com

Question: What is h^+ for this domain?

1. Manhattan Distance. **No**, relaxed plans can't walk through walls.
2. h^*. **Yes**, optimal plan = shortest path = relaxed plan (deletes do not matter because "shortest paths never walk back").
3. Horizontal distance.
4. Vertical distance.

Challenge!

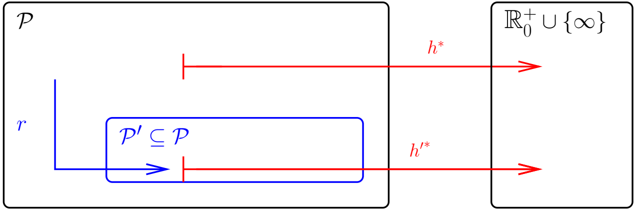


3805 8489 @ menti.com

Question: What is h^+ for this domain?

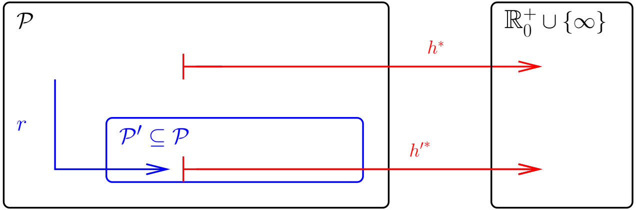
1. Manhattan Distance. **No**, relaxed plans can't walk through walls.
2. h^*. **Yes**, optimal plan = shortest path = relaxed plan (deletes do not matter because "shortest paths never walk back").
3. Horizontal distance. **No**, relaxed plans must move both horizontally and vertically.
4. Vertical distance. **No**, relaxed plans must move both horizontally and vertically.

h^+ as a Relaxation Heuristic



where, for all $P \in \mathcal{P}$:
$h'^*(r(P)) \leq h^*(P)$.

h^+ as a Relaxation Heuristic

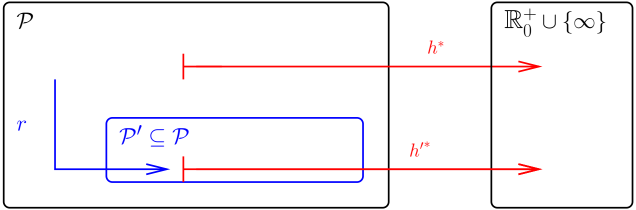


where, for all $P \in \mathcal{P}$: $h'^*(r(P)) \leq h^*(P)$.

For $h^+ = h^* \circ r$:

- Problem $P \in \mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $P \in \mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic h'^* for P': Optimal plan cost on P'.
- Transformation r: Drop the deletes; drop all `(not ...)` terms in `:effects`

h^+ as a Relaxation Heuristic



where, for all $P \in \mathcal{P}$: $h'^*(r(P)) \leq h^*(P)$.

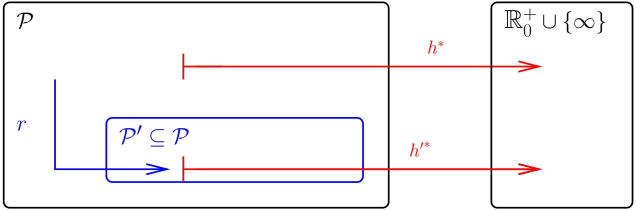
For $h^+ = h^* \circ r$:

- Problem $P \in \mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $P \in \mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic h'^* for P': Optimal plan cost on P'.
- Transformation r: Drop the deletes; drop all `(not ...)` terms in `:effects`

Questions

1. Is this a native relaxation?
2. Is this relaxation efficiently constructible?
3. Is this relaxation efficiently computable?

h^+ as a Relaxation Heuristic



where, for all $P \in \mathcal{P}$:
$h'^*(r(P)) \leq h^*(P)$.

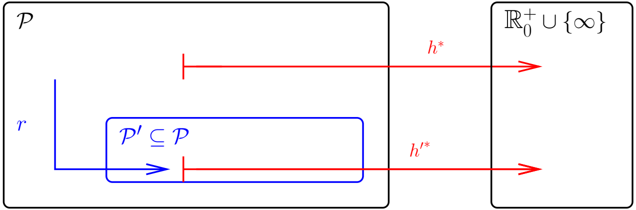
For $h^+ = h^* \circ r$:

- Problem $P \in \mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $P \in \mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic h'^* for P': Optimal plan cost on P'.
- Transformation r: Drop the deletes; drop all `(not ...)` terms in `:effects`

Questions

1. Is this a native relaxation? Yes!
2. Is this relaxation efficiently constructible?
3. Is this relaxation efficiently computable?

h^+ as a Relaxation Heuristic



where, for all $P \in \mathcal{P}$: $h'^*(r(P)) \leq h^*(P)$.

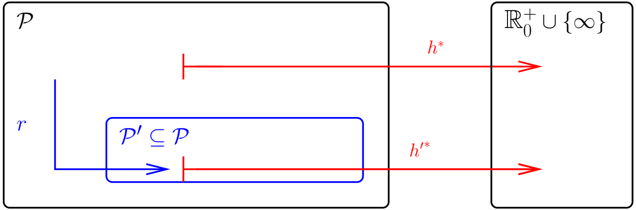
For $h^+ = h^* \circ r$:

- Problem $P \in \mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $P \in \mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic h'^* for P': Optimal plan cost on P'.
- Transformation r: Drop the deletes; drop all `(not ...)` terms in `:effects`

Questions

1. Is this a native relaxation? Yes!
2. Is this relaxation efficiently constructible? Yes!
3. Is this relaxation efficiently computable?

h^+ as a Relaxation Heuristic



where, for all $P \in \mathcal{P}$:
$h'^*(r(P)) \leq h^*(P)$.

For $h^+ = h^* \circ r$**:**

- Problem $P \in \mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $P \in \mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic h'^* for P': Optimal plan cost on P'.
- Transformation r: Drop the deletes; drop all `(not ...)` terms in `:effects`

Questions

1. Is this a native relaxation? Yes!
2. Is this relaxation efficiently constructible? Yes!
3. Is this relaxation efficiently computable? **No!** 😞

Perfect delete-relaxation h^+ is hard!

Unfortunately, definition $h^+(s) = h^*_{P'}(s)$ **not** suitable **computationally**:

- Solving $P'(s)$ **optimally** as difficult as solving $P(s)$ **optimally** (NP-hard).
- Hardness proved by reduction from SAT:

  "When operators are restricted to one positive precondition and one positive postcondition, PLANMIN remains intractable." (Bylander'94)

- Remember, heuristics need to be computed fast!

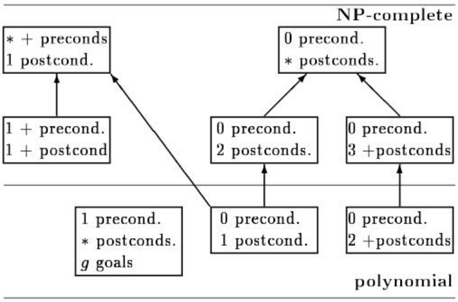


Figure 2: Complexity Results for PLANMIN

Perfect delete-relaxation h^+ is hard!

Unfortunately, definition $h^+(s) = h^*_{P'}(s)$ **not** suitable **computationally**:

- Solving $P'(s)$ **optimally** as difficult as solving $P(s)$ **optimally** (NP-hard).
- Hardness proved by reduction from SAT:

  > *"When operators are restricted to one positive precondition and one positive postcondition, PLANMIN remains intractable."* *(Bylander'94)*

- Remember, heuristics need to be computed fast!

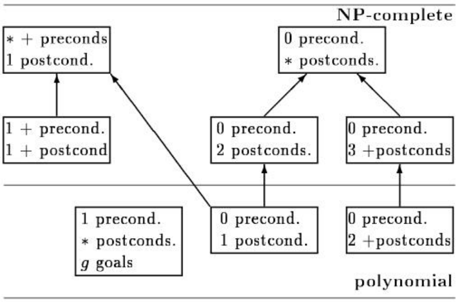


Figure 2: Complexity Results for PLANMIN

! Yet, **finding one plan** for $P'(s)$, not necessarily optimal, is **easy**. **Why?** Next slide!

Perfect delete-relaxation h^+ is hard!

Unfortunately, definition $h^+(s) = h^*_{P'}(s)$ **not** suitable **computationally**:

- Solving $P'(s)$ **optimally** as difficult as solving $P(s)$ **optimally** (NP-hard).
- Hardness proved by reduction from SAT:
  "When operators are restricted to one positive precondition and one positive postcondition, PLANMIN remains intractable." *(Bylander'94)*
- Remember, heuristics need to be computed fast!

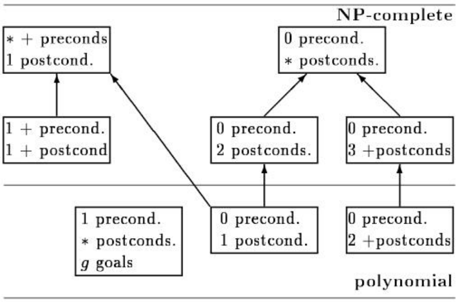


Figure 2: Complexity Results for PLANMIN

! Yet, **finding one plan** for $P'(s)$, not necessarily optimal, is **easy**. **Why?** Next slide!

- All implemented systems using the delete relaxation **approximate** h^+ in one or the other way. We now look at the the most wide-spread approaches to do so...

- (not , vi,)

Why solving $P'(s)$ is "easy"?

Key Idea: **Delete-free** STRIPS problems like $P'(s)$ are **fully decomposable**

If plan π_1 achieves G_1 and plan π_2 achieves G_2, then plan $\pi_1 \cdot \pi_2$ achieves G_1 **and** G_2.

➠ So, plans π_p for each atom p yield plans for **any goal** G (with lots of "redundancy").

Why solving $P'(s)$ is "easy"?

💡 Key Idea: **Delete-free** STRIPS problems like $P'(s)$ are **fully decomposable**

If plan π_1 achieves G_1 and plan π_2 achieves G_2, then plan $\pi_1 \cdot \pi_2$ achieves G_1 **and** G_2.

➠ So, plans π_p for each atom p yield plans for **any goal** G (with lots of "redundancy").

Let's compute how many steps are needed to reach each atom p:

▶ **Procedure:** Atom p reachable in k steps with support a_p from state s

1. Atom p **reachable** in 0 steps with no action **support** if $p \in s$.
2. Atom p **reachable** in $i + 1$ steps with **support** a_p, if not reachable in i steps or less, and **preconditions** p_i of a_p **reachable** in i steps or less.

Why solving $P'(s)$ is "easy"?

Key Idea: Delete-free STRIPS problems like $P'(s)$ are **fully decomposable**

If plan π_1 achieves G_1 and plan π_2 achieves G_2, then plan $\pi_1 \cdot \pi_2$ achieves G_1 **and** G_2.
➠ So, plans π_p for each atom p yield plans for **any goal** G (with lots of "redundancy").

Let's compute how many steps are needed to reach each atom p:

Procedure: Atom p reachable in k steps with support a_p from state s

1. Atom p **reachable** in 0 steps with no action **support** if $p \in s$.
2. Atom p **reachable** in $i+1$ steps with **support** a_p, if not reachable in i steps or less, and **preconditions** p_i of a_p **reachable** in i steps or less.

- Procedure terminates in $\#$ of steps bounded by number of atoms
  - *... and if p not reachable, there is no plan for p in either $P'(s)$ or $P(s)$*
- Supporters a_p needed to get to goal G of P yield (relaxed) plan $\pi'(s)$ for $P'(s)$

Max and Additive Heuristics

For all **atoms** p:

$$h(p;s) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p \in s \\ \min_{a \in \mathsf{Add}(p)} [cost(a) + h(\mathsf{Pre}(a);s)] & \text{otherwise} \end{cases}$$

Observe: $h(\mathsf{Pre}(a);s)$ is on set of propositions — $\mathsf{Pre}(a)$ may contain many atoms.

Max and Additive Heuristics

For all **atoms** p:

$$h(p;s) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p \in s \\ \min_{a \in \mathsf{Add}(p)}[cost(a) + h(\mathsf{Pre}(a);s)] & \text{otherwise} \end{cases}$$

Observe: $h(\mathsf{Pre}(a);s)$ is on set of propositions — $\mathsf{Pre}(a)$ may contain many atoms.

The Max Heuristic $h_{\max}$

For **sets** of atoms C, define:

$$h(C;s) \stackrel{\text{def}}{=} \max_{r \in C} h(r;s)$$

Resulting **heuristic function**:

$$h_{\max}(s) \stackrel{\text{def}}{=} h(G;s)$$

- $\#$ of steps to reach all atoms in G.
- Admissible, but often too optimistic.

Max and Additive Heuristics

For all **atoms** p:

$$h(p;s) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p \in s \\ \min_{a \in \mathsf{Add}(p)} [cost(a) + h(\mathsf{Pre}(a);s)] & \text{otherwise} \end{cases}$$

Observe: $h(\mathsf{Pre}(a);s)$ is on set of propositions — $\mathsf{Pre}(a)$ may contain many atoms.

The Max Heuristic $h_{\max}$

For **sets** of atoms C, define:

$$h(C;s) \stackrel{\text{def}}{=} \max_{r \in C} h(r;s)$$

Resulting **heuristic function**:

$$h_{\max}(s) \stackrel{\text{def}}{=} h(G;s)$$

- $\#$ of steps to reach all atoms in G.
- Admissible, but often too optimistic.

The Additive Heuristic h_{add}

For **sets** of atoms C, define:

$$h(C;s) \stackrel{\text{def}}{=} \sum_{r \in C} h(r;s)$$

Resulting **heuristic function**:

$$h_{\text{add}}(s) \stackrel{\text{def}}{=} h(G;s)$$

- **sum** of steps to reach each atom in G.
- Not admissible, but often informative.

Example

Problem $P = \langle F, I, O, G \rangle$ where:

- $F = \{p_i, q_i \mid i \in \{0, \ldots, n\}\}$
- $I = \{p_0, q_0\}$
- $G = \{p_n, q_n\}$
- O contains actions a_i and b_i, for $i\{0, \ldots, n-1\}$:
  - Pre$(a_i) = \{p_i\}$, Add$(a_i) = \{p_{i+1}\}$, Del$(a_i) = \{p_i\}$
  - Pre$(b_i) = \{q_i\}$, Add$(b_i) = \{q_{i+1}\}$, Del$(b_i) = \{q_i\}$

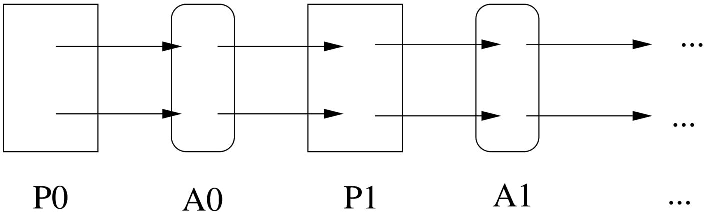
Questions

For the initial state I:

1. What is $h_{\max}(I)$?
2. What is $h_{\text{add}}(I)$?
3. What is relaxed plan obtained from $h_{\max}$?
4. What is **optimal cost** $h^*_P(I)$?

Alternative Graphic Procedure to Compute Max Heuristic

Procedure builds propositional and action **layers** P_i and A_i ignoring deletes from state s:



$$
\begin{aligned}
P_0 &= \{p \mid p \in s\} \\
A_i &= \{a \mid a \in O, \mathsf{Pre}(a) \subseteq P_i\} \\
P_{i+1} &= P_i \cup \{p \mid a \in A_i, p \in \mathsf{Add}(a)\} \qquad \text{(ignore deletes!)}
\end{aligned}
$$

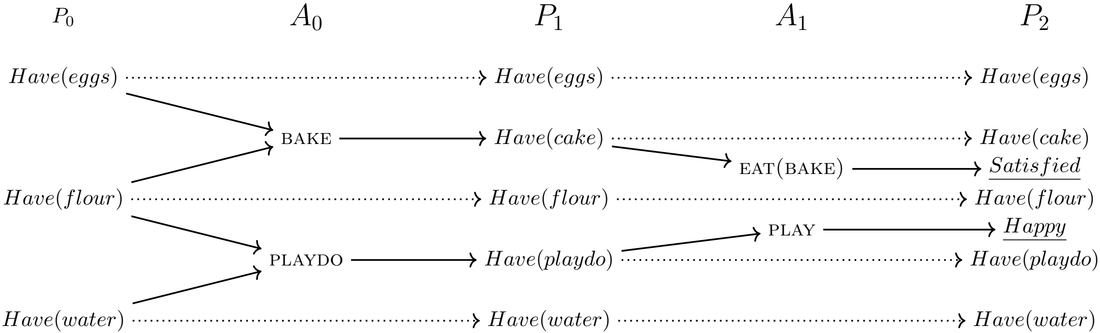
Max Heuristic $h_{\max}$

The max heuristic is implicitly **represented** in this layered graph:

$$h_{\max}(s) = \text{smallest } i \text{ such that each } p \in G \text{ is in some layer } P_k, \text{ with } k \leq i$$

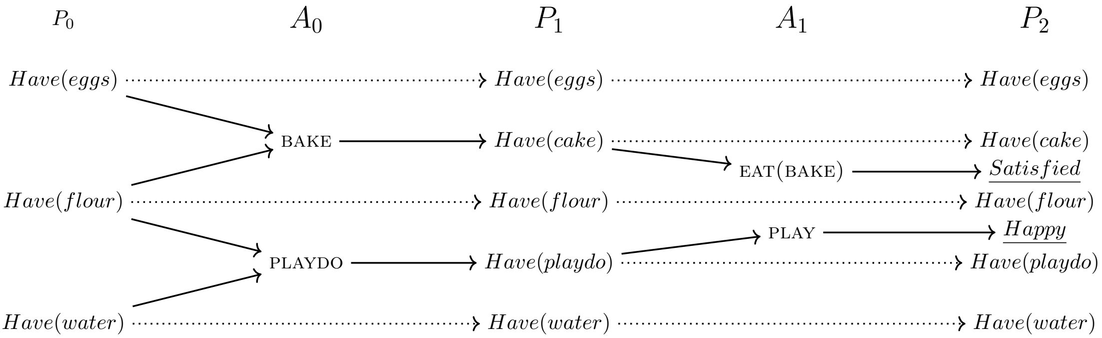
Planning Graph to Compute $h_{\max}$

Eggs, flour, and water are needed to bake (and eat) a cake, and to make playdo, have fun, and be happy! Goal is to be happy 🥳 and feel satisfied 🍰



Planning Graph to Compute $h_{\max}$

Eggs, flour, and water are needed to bake (and eat) a cake, and to make playdo, have fun, and be happy! Goal is to be happy 🥳 and feel satisfied 🍰



✳ $h_{\max} = \max\{h(Happy), h(Satisfied)\} = \max\{2, 2\} = 2$ (G appears first in level 2!)

$$h(Happy) = 1 + h(Have(playdo)) = 1 + (1 + h(Have(water))) = 1 + (1 + 0) = 2$$

The Additive and Max Heuristics: So What?

Summary of typical issues in practice with h_{add} and $h_{\max}$:

1. Both h_{add} and $h_{\max}$ can be computed reasonably quickly.
2. $h_{\max}$ is **admissible**, but is typically far too optimistic.
3. h_{add} is **not admissible**, but is typically a lot more informed than $h_{\max}$.
4. But h_{add} may overcount by **ignoring positive interactions**, i.e., sub-plans shared between sub-goals.
5. Such overcounting can result in dramatic over-estimates of h^*!!

The Additive and Max Heuristics: So What?

Summary of typical issues in practice with h_{add} and $h_{\max}$:

1. Both h_{add} and $h_{\max}$ can be computed reasonably quickly.
2. $h_{\max}$ is **admissible**, but is typically far too optimistic.
3. h_{add} is **not admissible**, but is typically a lot more informed than $h_{\max}$.
4. But h_{add} may overcount by **ignoring positive interactions**, i.e., sub-plans shared between sub-goals.
5. Such overcounting can result in dramatic over-estimates of h^*!!

☼ **Relaxed plans** (next) is a way to reduce this kind of over-counting.

- Similar to h_{add}, but can account for positive interactions and are much less prone to overcounting.
- They achieve this by adding another technology layer – relaxed plan extraction – on top of $h_{\max}$ or h_{add}.

Relaxed Plans and Best Supporters

Basic Idea for relaxed plans

1. First compute a best-supporter action a_p for every fact $p \in F$: action that is deemed to be the cheapest achiever of p (within the relaxation).
2. Then extract a **relaxed plan** from best supporters of all goal atoms.

The **best-supporter** can be based directly on $h_{\max}$ or h_{add} heuristics by **recursively collecting best supporters backwards** from the goal, where a_p is **best support** for $p \notin s$:

$$a_p = \underset{a \in \mathsf{Add}(p)}{\text{argmin}}[cost(a) + h(\mathsf{Pre}(a))]$$

Relaxed Plans and Best Supporters

Basic Idea for relaxed plans

1. First compute a best-supporter action a_p for every fact $p \in F$: action that is deemed to be the cheapest achiever of p (within the relaxation).
2. Then extract a **relaxed plan** from best supporters of all goal atoms.

The **best-supporter** can be based directly on $h_{\max}$ or h_{add} heuristics by **recursively collecting best supporters backwards** from the goal, where a_p is **best support** for $p \notin s$:

$$a_p = \operatorname*{argmin}_{a \in \mathsf{Add}(p)} [cost(a) + h(\mathsf{Pre}(a))]$$

A **plan** $\pi(p; s)$ **for** p in delete-relaxation can be computed backwards as:

$$\pi(p; s) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p \in s \\ a_p \cup \bigcup_{q \in \mathsf{Pre}(a_p)} \pi(q; s) & \text{otherwise} \end{cases}$$

Relaxed Plans and h_{FF}

The **best-supporter** wrt $h_{\max}$ (cheapest achiever of p based on $h_{\max}$):

$$a_p = \operatorname*{argmin}_{a \in \mathsf{Add}(p)} [cost(a) + h_{\max}(\mathsf{Pre}(a))]$$

A **plan** $\pi(p; s) = O_k \cdot O_{k-1} \cdots O_1$ **for** p in delete-relaxation can be computed backwards as:

$$\pi(p; s) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } p \in s \\ \{a_p\} \cup \bigcup_{q \in \mathsf{Pre}(a_p)} \pi(q; s) & \text{otherwise} \end{cases}$$

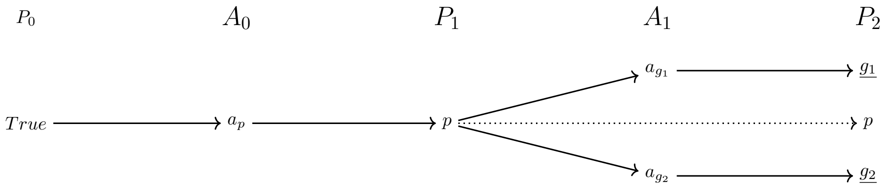
h_{FF}: $\#$ of different a_p-supporters needed to get to G:

$$h_{\text{FF}}(s) = |\bigcup_{p \in G} \pi(p; s)|$$

using $h = h_{\max}$ for the best supporters.

Planning Graphs for Relaxed Plans

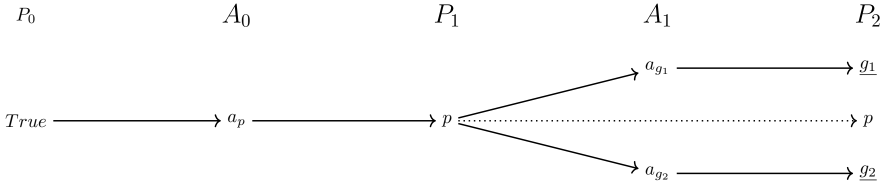
Consider three atoms p, g_1, and g_2, and three actions a_p, a_{g_1}, and a_{g_2}, that make them true, respectively. Precondition of a_p is empty, but both a_{g_1} and $= a_{g_2}$ require atom p to be true. Goal is $\{g_1, g_2\}$ and initial state $I = \emptyset$ (nothing is true).



- $h^*(I) = 3$ (optimal plan is $a_p \cdot a_{g_1} \cdot a_{g_2}$).

Planning Graphs for Relaxed Plans

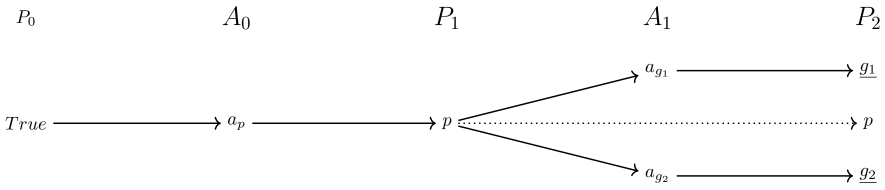
Consider three atoms p, g_1, and g_2, and three actions a_p, a_{g_1}, and a_{g_2}, that make them true, respectively. Precondition of a_p is empty, but both a_{g_1} and $= a_{g_2}$ require atom p to be true. Goal is $\{g_1, g_2\}$ and initial state $I = \emptyset$ (nothing is true).



- $h^*(I) = 3$ (optimal plan is $a_p \cdot a_{g_1} \cdot a_{g_2}$).
- $h_{\max}(I) = \max\{h(g_1; I), h(g_1; I)\} = 2$ (goal appears at level 2 - optimistic!)

Planning Graphs for Relaxed Plans

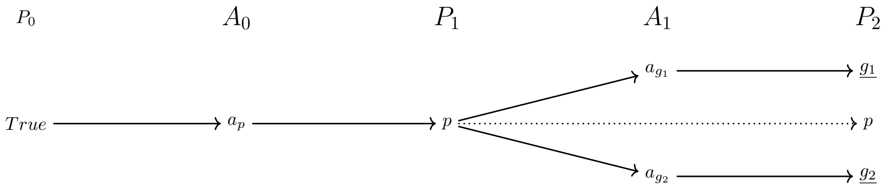
Consider three atoms p, g_1, and g_2, and three actions a_p, a_{g_1}, and a_{g_2}, that make them true, respectively. Precondition of a_p is empty, but both a_{g_1} and $= a_{g_2}$ require atom p to be true. Goal is $\{g_1, g_2\}$ and initial state $I = \emptyset$ (nothing is true).



- $h^*(I) = 3$ (optimal plan is $a_p \cdot a_{g_1} \cdot a_{g_2}$).
- $h_{\text{max}}(I) = \max\{h(g_1; I), h(g_1; I)\} = 2$ (goal appears at level 2 - optimistic!)
- $h_{\text{add}}(I) = h(g_1; I) + h(g_1; I) = 2 + 2 = 4$ (pessimistic, counts a_p twice!)

Planning Graphs for Relaxed Plans

Consider three atoms p, g_1, and g_2, and three actions a_p, a_{g_1}, and a_{g_2}, that make them true, respectively. Precondition of a_p is empty, but both a_{g_1} and $= a_{g_2}$ require atom p to be true. Goal is $\{g_1, g_2\}$ and initial state $I = \emptyset$ (nothing is true).



- $h^*(I) = 3$ (optimal plan is $a_p \cdot a_{g_1} \cdot a_{g_2}$).
- $h_{\max}(I) = \max\{h(g_1; I), h(g_1; I)\} = 2$ (goal appears at level 2 - optimistic!)
- $h_{\text{add}}(I) = h(g_1; I) + h(g_1; I) = 2 + 2 = 4$ (pessimistic, counts a_p twice!)
- $h_{\text{FF}}(I) = |\langle\{a_p\} \cup \{a_{g_1}, a_{g_2}\}\rangle| = 1 + 2 = 3$ perfect!

Other heuristics...

Key development in planning in the 90's...

Relaxations

- h^+ (Hoffmann & Nebel, '01)
- $h_{\max}$ and h_{add} (Bonet & Geffner, '01)
- h_{FF} (Hoffmann & Nebel, '01)
- h^{pmax} (Mirkis & Domshlak, '07)
- h^{sa} (Keyder & Geffner, '08

Critical paths

- h^m (Haslum & Geffner, '00) with $h^1 = h_{\max}$

Abstractions

- PDBs (Edelkamp, '01; Haslum et al., '05, '07)
- Merge & Shrink (Helmert et al., '07,'14; Katz et al, '12; Sievers et al., '14)

Landmarks

- Landmark count (Hoffmann et al., '04)
- h^L and h^{LA} (Karpas & Domshlak, '09)
- LM-cut (Helmert & Domshlak, '10)

Example

Problem $P = \langle F, I, O, G \rangle$ where:

- $F = \{p_i, q_i \mid i = 0, \ldots, n\}$
- $I = \{p_0, q_0\}$
- $G = \{p_n, q_n\}$
- O contains actions a_i and b_i, $i = 0, \ldots, n-1$:
  - Pre$(a_i) = \{p_i\}$, Add$(a_i) = \{p_{i+1}\}$, Del$(a_i) = \{p_i\}$
  - Pre$(b_i) = \{q_i\}$, Add$(b_i) = \{q_{i+1}\}$, Del$(b_i) = \{q_i\}$

Questions

For the initial state I:

1. What is relaxed plan obtained for $h_{\text{FF}}(I)$?
2. What is $h_{\text{FF}}(I)$?

Example

Problem $P = \langle F, I, O, G \rangle$ where:

- $F = \{p_i, q_i \mid i = 0, \ldots, n\}$
- $I = \{p_0, q_0\}$
- $G = \{p_n, q_n\}$
- O contains actions a_i and b_i, $i = 0, \ldots, n-1$:
  - Pre$(a_i) = \{p_i\}$, Add$(a_i) = \{p_{i+1}\}$, Del$(a_i) = \{p_i\}$
  - Pre$(b_i) = \{q_i\}$, Add$(b_i) = \{q_{i+1}\}$, Del$(b_i) = \{q_i\}$

Questions

For the initial state I:

1. What is relaxed plan obtained for $h_{\text{FF}}(I)$?
2. What is $h_{\text{FF}}(I)$?
3. What happens if we have actions c_i for i even:
   - Pre$(c_i) = \{p_i, q_i\}$, Add$(c_i) = \{p_{i+1}, q_{i+1}\}$, Del$(c_i) = \{p_i, q_i\}$

Exercise

Problem $P = \langle F, I, O, G \rangle$ where:

- $F = \{p_i, q_i \mid i = 0, \ldots, n\}$
- $I = \{p_0, q_0\}$
- $G = \{p_n, q_n\}$
- O contains actions a_i, b_i, and c_i:
  - $\text{Pre}(a_i) = \{p_i\}$, $\text{Add}(a_i) = \{p_{i+1}\}$, $\text{Del}(a_i) = \{p_i\}$, for $i = 0, \ldots, n-1$.
  - $\text{Pre}(b_i) = \{q_i\}$, $\text{Add}(b_i) = \{q_{i+1}\}$, $\text{Del}(b_i) = \{q_i\}$, for $i = 0, \ldots, n-1$.
  - $\text{Pre}(c_i) = \{p_i, q_i\}$, $\text{Add}(c_i) = \{p_{i+1}, q_{i+1}\}$, $\text{Del}(c_i) = \{p_i, q_i\}$, for $i = 0, \ldots, n-1$ such that $i \mod 2 = 0$ (that is, action c_i exists when i is even).

Questions

1. Calculate $h^+(I)$.
2. Calculate $h_{\text{add}}(I)$.
3. Calculate $h_{\max}(I)$.
4. Calculate $h_{\text{FF}}(I)$. What is relaxed plan obtained for $h_{\text{FF}}(I)$?
5. Calculate $h^*(I)$.

Example Systems

HSP [*Bonet and Geffner, AI-01*]

1. Search algorithm: Greedy best-first search.
2. Search control: h_{add}.

FF [*Hoffmann and Nebel ,JAIR-01*]

1. Search algorithm: Enforced hill-climbing.
2. Search control: h_{FF} extracted from $h_{\max}$ supporter function; helpful actions pruning (basically expand only those actions contained in the relaxed plan).

LAMA [*Richter and Westphal, JAIR-10*]

1. Search algorithm: Multiple-queue greedy best-first search.
2. Search control: h_{FF} + a landmarks heuristic (similar to goal counting); for each, one search queue all actions, one search queue only helpful actions.

BFWS [*Lipovetzky and Geffner, AAAI-17*]

1. Search algorithm: best-first width search.
2. Search control: novelty + variant of h_{FF} + goal counting.

Modern Planners: EHC Search, Helpful Actions, Landmarks

- First generation of **heuristic search planners** like **HSP**, searched the graph defined by state model $\mathcal{S}(P)$ using standard search algorithms like **Greedy Best-First** or WA*, and **heuristics** like h_{add}.
- Second generation planners like **FF** and **LAMA** beyond this in two ways:
  1. They exploit the structure of the heuristic and/or problem further:
     - **Helpful Actions:** actions most relevant in relaxation.
     - **Landmarks:** implicit problem subgoals.
  2. They use novel search algorithms:
     - **Enforced Hill Climbing (EHC)**.
     - **Multi-Queue Best First Search**.
- The result is that they can solve **huge problems**, **very fast**. Not always though...

Modern Planners: EHC Search, Helpful Actions, Landmarks

- First generation of **heuristic search planners** like **HSP**, searched the graph defined by state model $\mathcal{S}(P)$ using standard search algorithms like **Greedy Best-First** or WA*, and **heuristics** like h_{add}.
- Second generation planners like **FF** and **LAMA** beyond this in two ways:
  1. They exploit the structure of the heuristic and/or problem further:
     - **Helpful Actions:** actions most relevant in relaxation.
     - **Landmarks:** implicit problem subgoals.
  2. They use novel search algorithms:
     - **Enforced Hill Climbing (EHC)**.
     - **Multi-Queue Best First Search**.
- The result is that they can solve **huge problems**, **very fast**. Not always though...
- The **delete relaxation** is still used at large, specially since the wins of LAMA in the satisficing planning tracks of IPC'08 and IPC'11.
- More generally, the relaxation principle is very generic and applicable in many contexts.
  ☼ This is where all started: Planning as Heuristic Search [Bonet and Geffner, AI-01].

Search in the FF Planner

- **Heuristic** in FF is $h_{\text{FF}}(s)$ given by size $|\pi'(s)|$ of **relaxed plan** $\pi'(s)$ for $P'(s)$.
- The **search** in FF split in **two phases**:
  1. First phase, called **EHC (Enforced Hill Climbing)** is **incomplete** but **fast**:
     - Starting with $s = s_0$, **EHC** does a **breadth-first search** from s using only **"helpful actions"** until a state s' is found such that $h_{\text{FF}}(s') < h_{\text{FF}}(s)$.
     - If such a state s' is found, the process is **repeated** starting with $s = s'$. Else, the EHC **fails**, and the second phase is triggered.
  2. Second phase is a **Greedy Best-First** search guided by h_{FF}: **complete** but **slow**.
- Action deemed **helpful** in s if applicable in s and adds a goal or precondition of action in "relaxed plan" $\pi'(s)$.

Part 2: Classical Planning: Methods

Part 2: Classical Planning: Methods

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \lor \neg y \lor \neg z) \land (\neg x \lor y \lor z) \land (y \lor z) \land \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*

- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \ldots, p_n$ and $a_0, a_1, \ldots, a_{n-1}$ for each $p \in F$ and $a \in O$.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*

- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \ldots, p_n$ and $a_0, a_1, \ldots, a_{n-1}$ for each $p \in F$ and $a \in O$.
  - p_i: atom p is true at time step i.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*

- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \ldots, p_n$ and $a_0, a_1, \ldots, a_{n-1}$ for each $p \in F$ and $a \in O$.
  - p_i: atom p is true at time step i.
  - a_i: action a is executed/selected at time step i.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*

- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \ldots, p_n$ and $a_0, a_1, \ldots, a_{n-1}$ for each $p \in F$ and $a \in O$.
  - p_i: atom p is true at time step i.
  - a_i: action a is executed/selected at time step i.

- $C(P, n)$ satisfiable **iff** there is a plan of length no greater than n.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \lor \neg y \lor \neg z) \land (\neg x \lor y \lor z) \land (y \lor z) \land \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*
- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \ldots, p_n$ and $a_0, a_1, \ldots, a_{n-1}$ for each $p \in F$ and $a \in O$.
  - p_i: atom p is true at time step i.
  - a_i: action a is executed/selected at time step i.
- $C(P, n)$ satisfiable **iff** there is a plan of length no greater than n.
- Such a **plan** can be read from **truth valuation** that satisfies $C(P, n)$.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (y \vee z) \wedge \ldots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*
- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \ldots, p_n$ and $a_0, a_1, \ldots, a_{n-1}$ for each $p \in F$ and $a \in O$.
  - p_i: atom p is true at time step i.
  - a_i: action a is executed/selected at time step i.
- $C(P, n)$ satisfiable **iff** there is a plan of length no greater than n.
- Such a **plan** can be read from **truth valuation** that satisfies $C(P, n)$.
- SAT-based planners like **SATPLAN** or **Madagascar** use this encoding.

Planning as SAT

- SAT: determine if there is a truth assignment that satisfies a set of clauses:

$$(x \lor \neg y \lor \neg z) \land (\neg x \lor y \lor z) \land (y \lor z) \land \dots$$

- Maps planning problem $P = \langle F, O, I, G \rangle$ with horizon n into a **set of clauses** $C(P, n)$, solved by **SAT solvers**.
  - *Use conflict-driven clause learning algorithms (CDCL), an optimisation of DPLL.*
- Formula/theory $C(P, n)$ includes variables $p_0, p_1, \dots, p_n$ and $a_0, a_1, \dots, a_{n-1}$ for each $p \in F$ and $a \in O$.
  - p_i: atom p is true at time step i.
  - a_i: action a is executed/selected at time step i.
- $C(P, n)$ satisfiable **iff** there is a plan of length no greater than n.
- Such a **plan** can be read from **truth valuation** that satisfies $C(P, n)$.
- SAT-based planners like **SATPLAN** or **Madagascar** use this encoding.
  - *Winners of the 2004 and 2006 IPCs optimal track; 2nd in 2014 agile track; part of top portfolio planners in 2023.*

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \text{Del}(a)$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \text{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \text{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:
  - $p_i \wedge \bigwedge_{a \in O(p^-)} \neg a_i \supset p_{i+1}$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \text{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:
  - $p_i \wedge \bigwedge_{a \in O(p^-)} \neg a_i \supset p_{i+1}$
  - $\neg p_i \wedge \bigwedge_{a \in O(p^+)} \neg a_i \supset \neg p_{i+1}$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \mathsf{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \mathsf{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:
  - $p_i \wedge \bigwedge_{a \in O(p^-)} \neg a_i \supset p_{i+1}$
  - $\neg p_i \wedge \bigwedge_{a \in O(p^+)} \neg a_i \supset \neg p_{i+1}$
- **Seriality:** For each $i = 0, \ldots, n-1$, if $a \neq a'$, $\neg(a_i \wedge a'_i)$

Theory $C(P, n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \text{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:
  - $p_i \wedge \bigwedge_{a \in O(p^-)} \neg a_i \supset p_{i+1}$
  - $\neg p_i \wedge \bigwedge_{a \in O(p^+)} \neg a_i \supset \neg p_{i+1}$
- **Seriality:** For each $i = 0, \ldots, n-1$, if $a \neq a'$, $\neg(a_i \wedge a'_i)$

Theory $C(P,n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \mathsf{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \mathsf{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:
  - $p_i \wedge \bigwedge_{a \in O(p^-)} \neg a_i \supset p_{i+1}$
  - $\neg p_i \wedge \bigwedge_{a \in O(p^+)} \neg a_i \supset \neg p_{i+1}$
- **Seriality:** For each $i = 0, \ldots, n-1$, if $a \neq a'$, $\neg(a_i \wedge a'_i)$

👍 **If theory $C(P,n)$ is SAT:** plan can be recovered from the truth assignment to atoms a_i.

Theory $C(P,n)$ for Problem $P = \langle F, O, I, G \rangle$

- **Init:** p_0 for $p \in I$, $\neg q_0$ for $q \in F \setminus I$
- **Goal:** p_n for $p \in G$
- **Actions:** For $i = 0, 1, \ldots, n-1$, and each action $a \in O$:
  - $a_i \supset p_i$ for $p \in Prec(a)$
  - $a_i \supset p_{i+1}$ for each $p \in \text{Add}(a)$
  - $a_i \supset \neg p_{i+1}$ for each $p \in \text{Del}(a)$
- **Persistence:** For $i = 0, \ldots, n-1$, and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p, resp.:
  - $p_i \wedge \bigwedge_{a \in O(p^-)} \neg a_i \supset p_{i+1}$
  - $\neg p_i \wedge \bigwedge_{a \in O(p^+)} \neg a_i \supset \neg p_{i+1}$
- **Seriality:** For each $i = 0, \ldots, n-1$, if $a \neq a'$, $\neg(a_i \wedge a'_i)$

👍 **If theory $C(P,n)$ is SAT:** plan can be recovered from the truth assignment to atoms a_i.

🔅 This encoding is simple but not best computationally; optimized encodings use parallelism (no seriality), NO-OPs, lower bounds, …

From SAT to Answer Set Programming (ASP)

- **ASP** is a **logic programming** paradigm for knowledge representation and reasoning.
  - More convenient representation than SAT: predicate logic (i.g., variables!)
  - Based on *stable model* semantics for logic programs with negation as failure.
  - Related to Constraint Programming and CSP.

- ASP encodings for planning similar to SAT encodings, but use rules instead of clauses:

```
{do(A, T) : action(A)} = 1 :- step(T).      % exactly one action per step
:- do(A, T), prec(A, P), not holds(P, T-1). % precondition applies!

holds(P, 0) :- init(P).                     % define init state
holds(P, T) :- do(A, T-1), add(A, P).   % add effects
holds(F, T) :- holds(F, T-1), step(T), not do(A, T-1) : del(A, F). % frame

:- goal(p), not holds(p, k).  % goal at last step k
```

  Problem instance encoded via facts `action(A)`, `prec(A,P)`, `add(A,P)`, `del(A,P)`, `init(P)`, `goal(P)`, and `step(T)` — e.g., `prec(unstack(A,B), on(A,B))`.

- ASP solvers compute **stable models** (answer sets) that represent plans.
  - *Plans extracted from atoms of the form `do(A,T)` in the stable model.*

Blocks Worlds in ASP

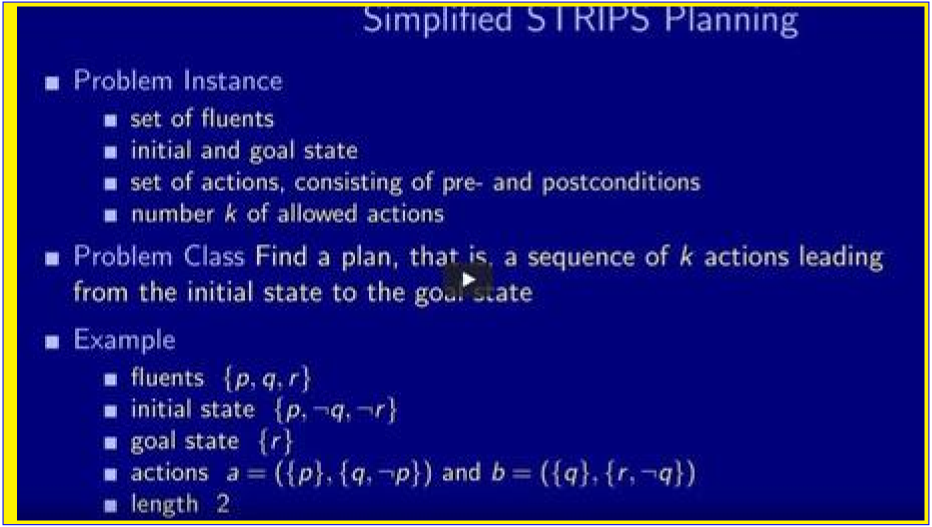
Planner is a fixed ASP program:

```
{do(A, T) : action(A)} = 1 :- step(T).        % exactly one action per step
:- do(A, T), prec(A, P), not holds(P, T-1). % precondition applies!

holds(P, 0) :- init(P).                       % define init state
holds(P, T) :- do(A, T-1), add(A, P).    % add effects
holds(F, T) :- holds(F, T-1), step(T), not do(A, T-1) : del(A, F). % frame

:- goal(p), not holds(p, k).  % goal at last step k
```

Problem instance encoding:

```
block(a;b;c;d).
init(on(a,b)). init(on(b,c)). init(ontable(c)). init(ontable(d)).
goal(on(a,d)). goal(on(d,b)). goal(on(b,c)).

action(stack(X,Y)) :- block(X), block(Y), X != Y.
prec(stack(X,Y), clear(Y)) :- block(X), block(Y), X != Y.
prec(stack(X,Y), holding(X)) :- block(X), block(Y), X != Y.
add(stack(X,Y), on(X,Y)) :- block(X), block(Y), X != Y.
del(stack(X,Y), holding(X);clear(X)) :- block(X), block(Y), X != Y.
...
step(1..10).
```

ASP for Planning youtube tutorial



Plasp: Tools for planning in ASP using Clingo

README · MIT license

plasp release v3.1.1 Build Status Build Status

> ASP planning tools for PDDL

Overview

`plasp` is a tool collection for planning in *answer set programming*. `plasp` 3 supports the input languages PDDL 3.1 (except for advanced features such as durative actions, numerical fluents, and preferences) and SAS (full support of SAS 3), which is used by Fast Downward.

The most notable tool provided by `plasp` is `plasp translate`, which translates PDDL descriptions to ASP facts.

Translating PDDL to ASP Facts

PDDL instances are translated to ASP facts as follows:

```
plasp translate domain.pddl problem.pddl
```

Alternatively, PDDL instances may first be translated to SAS, the output format of Fast Downward.

```
./fast-downward.py --translate --build=release64 domain.pddl problem.pddl
```

This creates a file called `output.sas`, which may now be translated by `plasp` as well.

```
plasp translate output.sas
```

Solving the Translated Instance

The translated instance can finally be solved with `clingo` and a meta encoding, for instance, `sequential-horizon.lp`:

Lots of planners in IPC 2023

International Planning Competition 2023 Classical Tracks

IPC 2023 Classical Tracks

International Planning Competition 2023 Classical Tracks

- Results
  - Optimal Track
  - Satisficing Track
  - Agile Track
  - Domains
- IPC 2023 Dataset
- Using IPC 2023 planners
- Calls
- Preliminary Schedule
- Tracks
  - Optimal Track
  - Satisficing Track
  - Agile Track
- PDDL Fragment
- Participants
  - Optimal Track
  - Satisficing Track
  - Agile Track
- Registration
- Planner Submission
- Apptainer Images

PDDL Fragment

IPC 2023 will use a subset of PDDL 3.1, as done since IPC 2011. Planners must support the subset of the language involving STRIPS, action costs, negative preconditions, and conditional effects (possibly in combination with forall, as in IPC 2014 and 2018). We will also consider including domains with disjunctive preconditions and existential quantifiers, in which case we provide an automatic translation compiling these features away, and we run all planners on both variants and select the best result per domain.

Most planners in previous IPCs rely on a grounding procedure to instantiate the entire planning task prior to start solving it. In IPC 2023, we will follow in the steps of the previous IPC by including domains and problems that are hard to ground.

Participants

Optimal Track

- **SymBD** (planner abstract) (code)
  Alvaro Torralba
  Symbolic Bidirectonal Blind Search

- **Hapori MIPlan Optimal All Data** (planner abstract) (code)
  Patrick Ferber, Michael Katz, Jendrik Seipp, Silvan Sievers, Daniel Borrajo, Isabel Cenamor, Tomas de la Rosa, Fernando Fernandez-Rebollo, Carlos Linares, Sergio Nunez, Alberto Pozanco, Horst Samulowitz, Shirin Sohrabi
  Sequential portfolio of optimal IPC planners computed with the MIP formulation by Nunez, Borrajo and Linares (2015).

- **Ragnarok** (planner abstract) (code)
  Dominik Drexler, Daniel Gnad, Paul Höft, Jendrik Seipp, David Speck, Simon Ståhlberg
  Sequential portfolio of optimal planners developed at Linköping University

- **Hapori Stone Soup Optimal** (planner abstract) (code)