ICAPS 2025 Tutorial
Learning for Generalised Planning

Dillon Z. Chen Felipe Trevizan Sylvie Thiébaux

https://l4p-tutorial.github.io/

Who is this tutorial for?

- planners
  - interested in "learning" for planning
- machine learners
  - interested in learning for "planning"
- knowledge representationers
  - interested in inductive reasoning in "learning for planning"

Assumed knowledge

- planning:
  - find a course of actions (i.e. a plan) or policy to achieve a goal
  - lifted representations, e.g. PDDL
- machine learning:
  - make predictions based on data and experience
  - training vs. testing pipelines

Agenda – Learning for Generalised Planning (L4P)

1. Introduction
2. Theory
   - 2.1. The L4P problem setup
   - 2.2. Methodologies using learning for L4P
     - 2.2.1. Forms of learned knowledge: policies, heuristics, transformations
     - 2.2.2. Tools for learning knowledge: graph learning, abstraction, constraint programming, policy search, language models
   - 2.3. Evaluating L4P methodologies
     - 2.3.1. Theoretical metrics: expressivity, generalisation, complexity
     - 2.3.2. Empirical metrics: training and planning costs, solution quality
3. Lab
   - 3.1. Getting used to the L4P setup: datasets and visualising data
   - 3.2. Testing and benchmarking existing L4P architectures
   - 3.3. Building a L4P architecture from scratch

Tutorial website
(contains links to related resources and these slides)

https://l4p-tutorial.github.io/

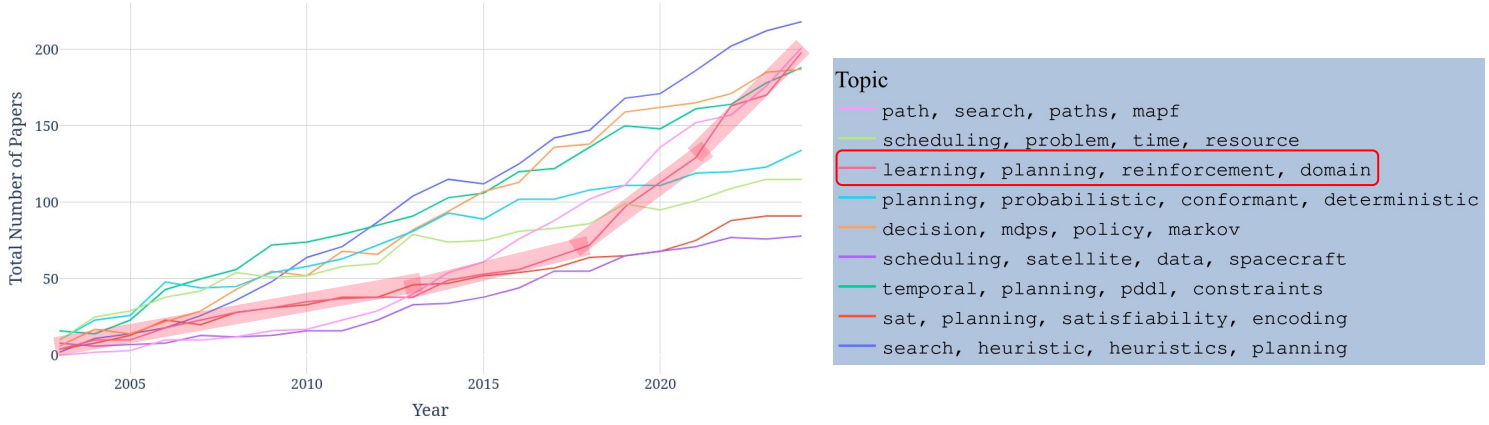
Article link
(comprehensive version of slides with more references)

Introduction

Motivation

Learning for (Generalised) Planning is a rapidly growing topic

- learn knowledge from easy-to-solve, small problems
- generalise to problems with unseen initial states/goals, and greater number of objects

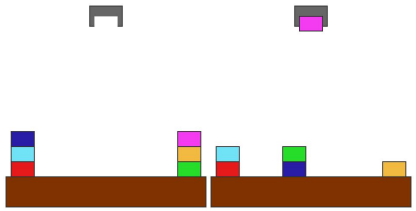


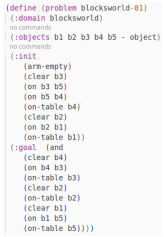
Number of papers grouped by topic at ICAPS 2003–2024

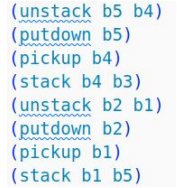
Learning for Generalised Planning?

- Q: Can we leverage learning research for planning?
- Q: Does learning for planning work?
- Q: Is there still need for domain-independent planning research?

What do we mean in this talk by Learning for Planning?







planning problem → planning model → solver → solution

Learn a model of the world

Learn to solve the model

What do we want with Learning for Generalised Planning?

"learning generalised plans from example problems with the aim of

amortising the cost of learning by solving problems more efficiently (faster) and effectively

(higher quality solutions) than domain-independent planners"

Preliminaries: Lifted Planning Problem

A lifted planning domain $\boldsymbol{D}$ is a tuple $\langle \boldsymbol{P}, \boldsymbol{A} \rangle$

- $\boldsymbol{P}$ = predicates; e.g. `(at ?obj ?loc)`
- $\boldsymbol{A}$ = action schemata; e.g. `(move ?obj ?loc_from ?loc_to)`

A lifted planning problem is a tuple $\langle \boldsymbol{D}, \boldsymbol{O}, \boldsymbol{s_0}, \boldsymbol{g} \rangle$

- $\boldsymbol{O}$ = objects; e.g. `dog, kitchen`
- $\boldsymbol{s_0}$ = initial state; e.g. `(at dog bedroom), (hungry dog)`
- $\boldsymbol{g}$ = goal condition; e.g. `(happy dog)`

Learning approaches covered in this tutorial can support more expressive planning models, e.g. numeric, probabilistic, temporal, partially observable etc.
The presentation here is done for simplicity.

ICAPS 2025 Tutorial
Learning for Generalised Planning: Theory

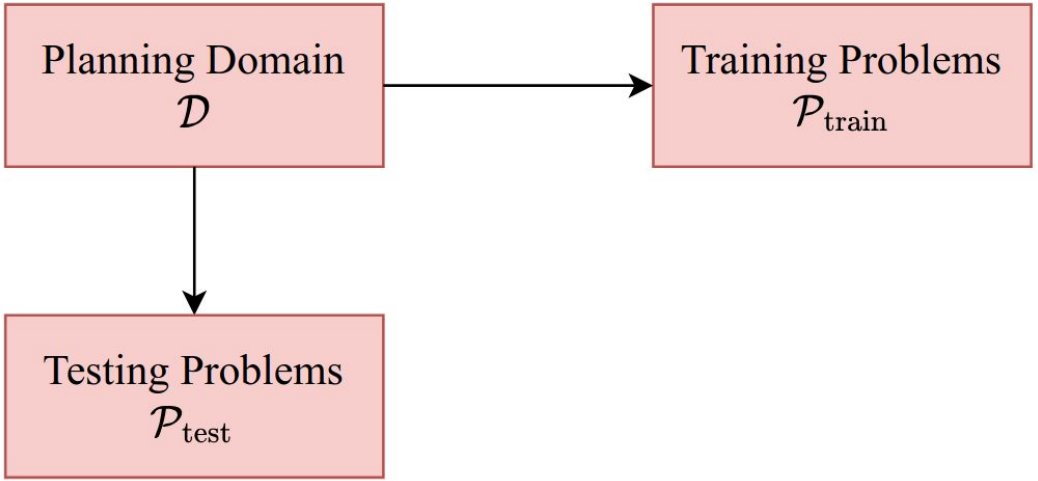
Dillon Z. Chen Felipe Trevizan Sylvie Thiébaux

https://l4p-tutorial.github.io/

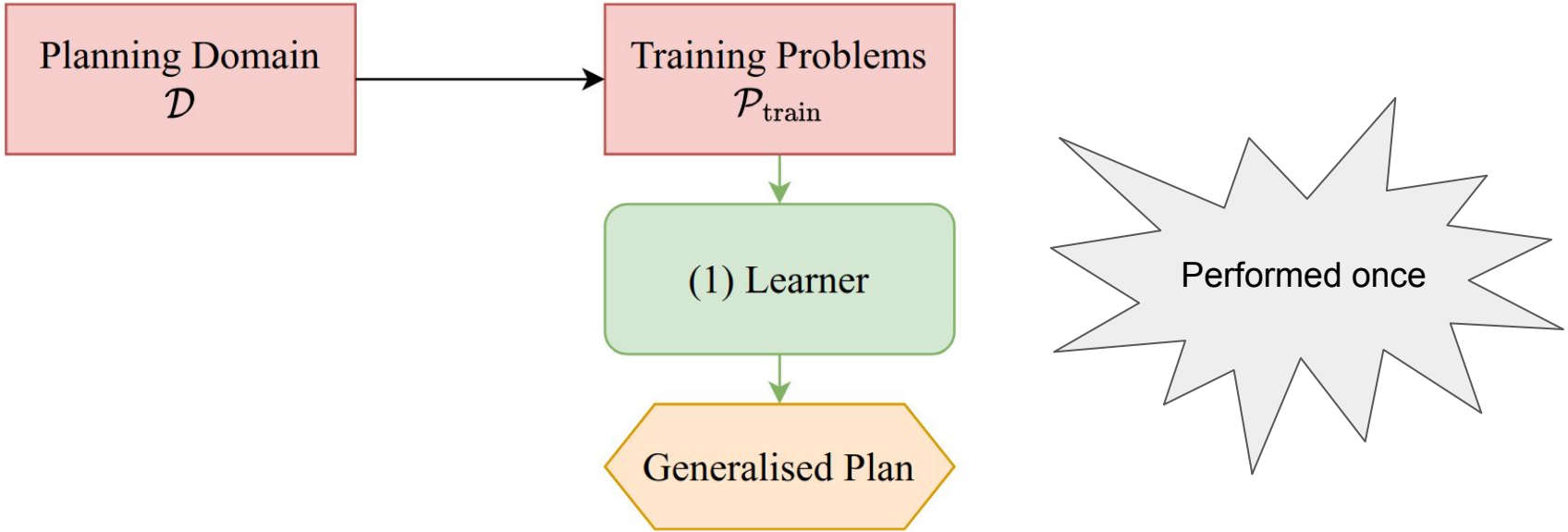
(2.1) The L4P problem setup

Learning for Generalised Planning (L4P) Setup

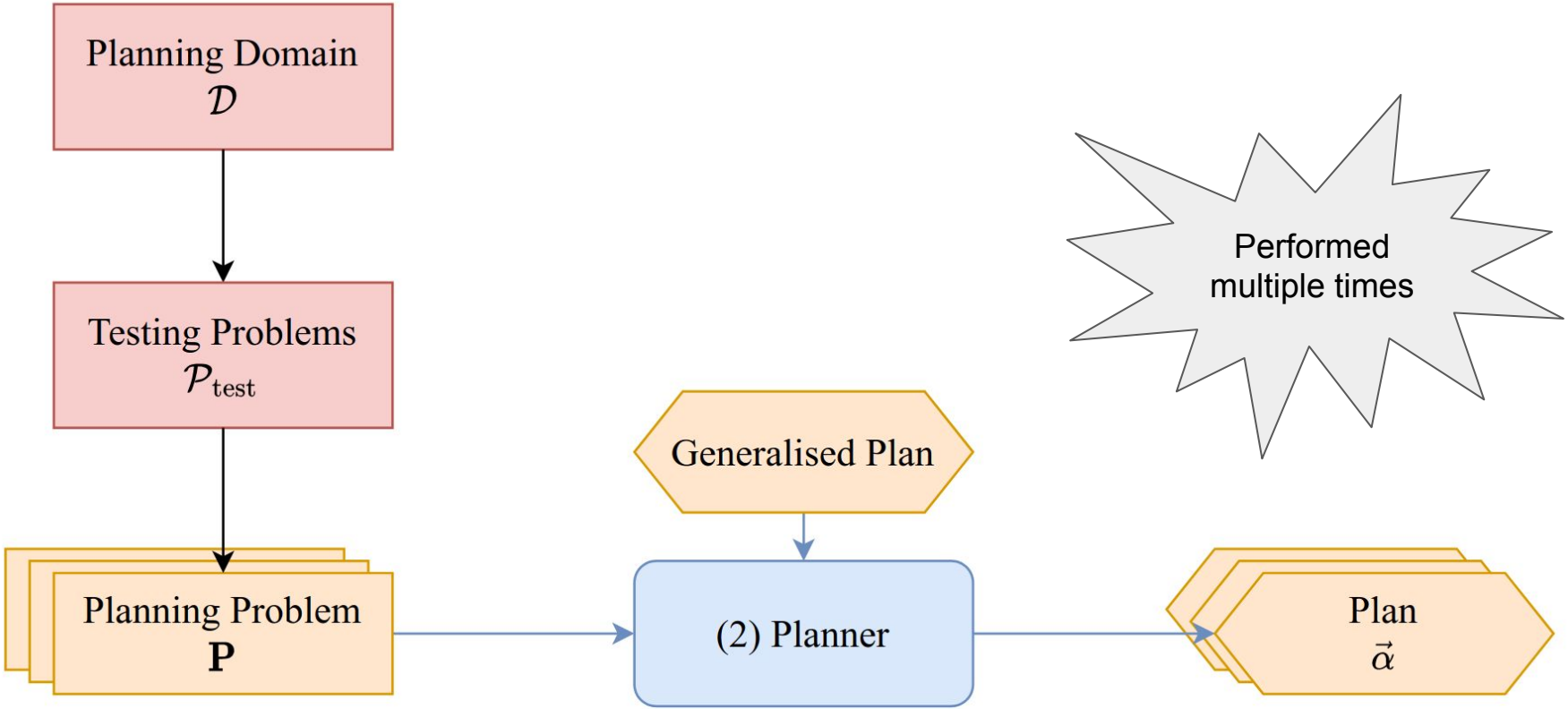
L4P Setup – Inputs



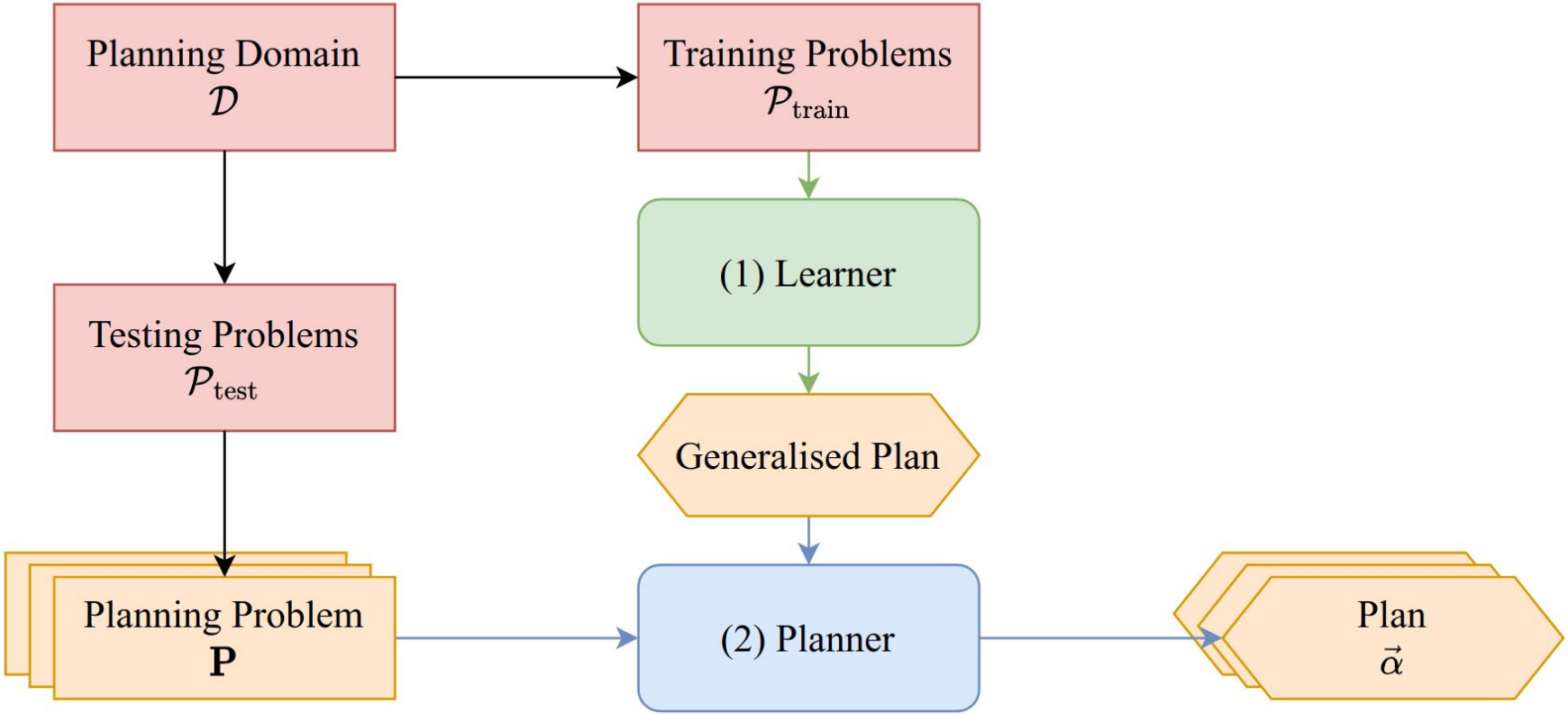
L4P Setup – Step 1: Learning



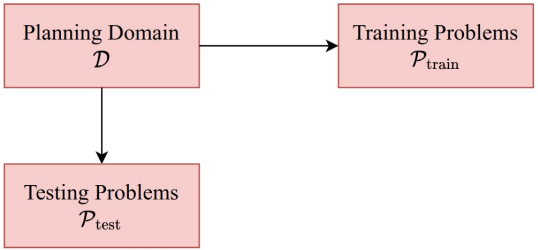
L4P Setup – Step 2: Planning



L4P Setup


Planning Domain
$\mathcal{D}$
Training Problems
$\mathcal{P}_{\text{train}}$
(1) Learner
Testing Problems
$\mathcal{P}_{\text{test}}$
Generalised Plan
Planning Problem
$\mathbf{P}$
(2) Planner
Plan
$\vec{\alpha}$

Training vs Testing distributions



Define "*f(x) = maximum number of objects and plan length of problems in x*"

1. **Interpolation; in-distribution learning:**

$$f(\mathcal{P}_{\text{test}}) \leq f(\mathcal{P}_{\text{train}})$$

2. **Extrapolation; out-of-distribution learning:**

$$f(\mathcal{P}_{\text{test}}) > f(\mathcal{P}_{\text{train}})$$

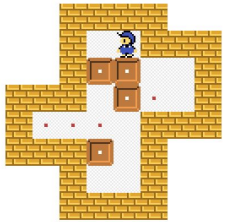
Training vs Testing distributions

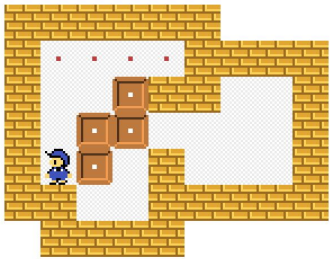
Define "*f(x) = maximum number of objects and plan length of problems in x*"

1. **Interpolation:** *f(* $\mathcal{P}_{\text{test}}$ *) ≤ f(* $\mathcal{P}_{\text{train}}$ *)*
   - for *intractable* domains
   - e.g. Sokoban, Rubik's cube; optimal planning
2. **Extrapolation:** *f(* $\mathcal{P}_{\text{test}}$ *) > f(* $\mathcal{P}_{\text{train}}$ *)*
   - for *tractable* domains
   - e.g. package transportation, Blocksworld

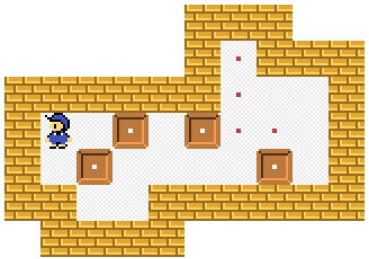
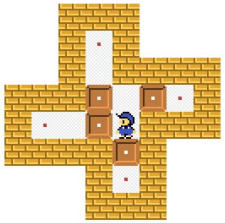
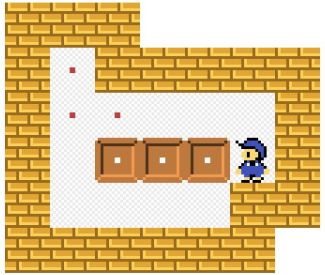
GP interpolation example; Sokoban

Training



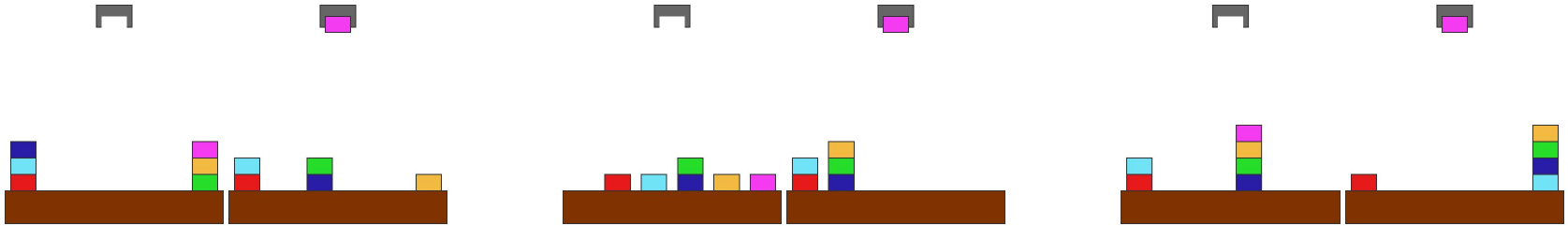


Testing: initial states and goals not seen before in training

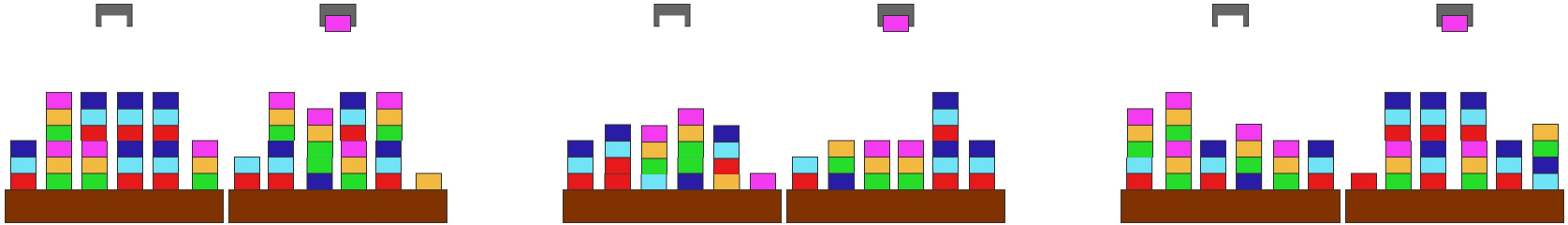




GP extrapolation example; Blocksworld

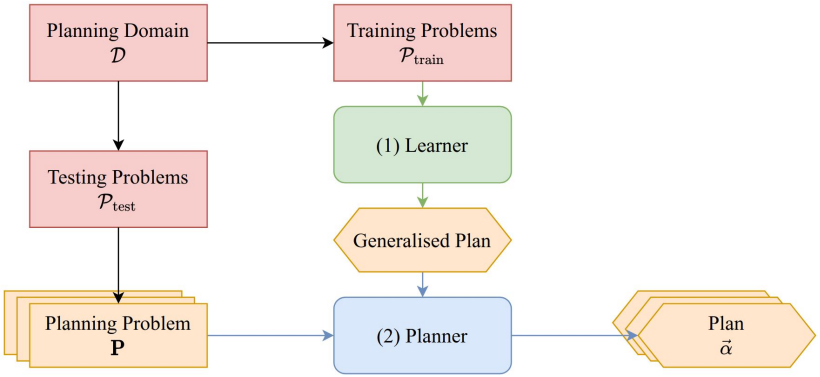
Training



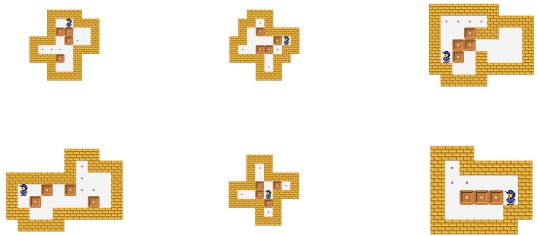
Testing: more blocks than seen in training problems



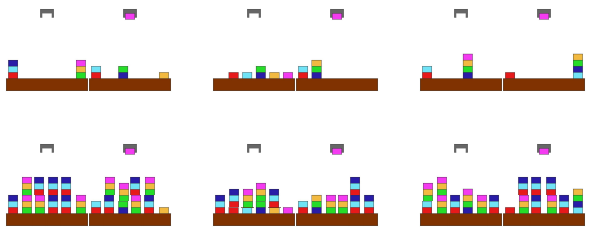
(2.1) The GP problem setup – *Summary*



1. Interpolation; in-distribution learning: $f(\mathcal{P}_{\text{test}}) \leq f(\mathcal{P}_{\text{train}})$

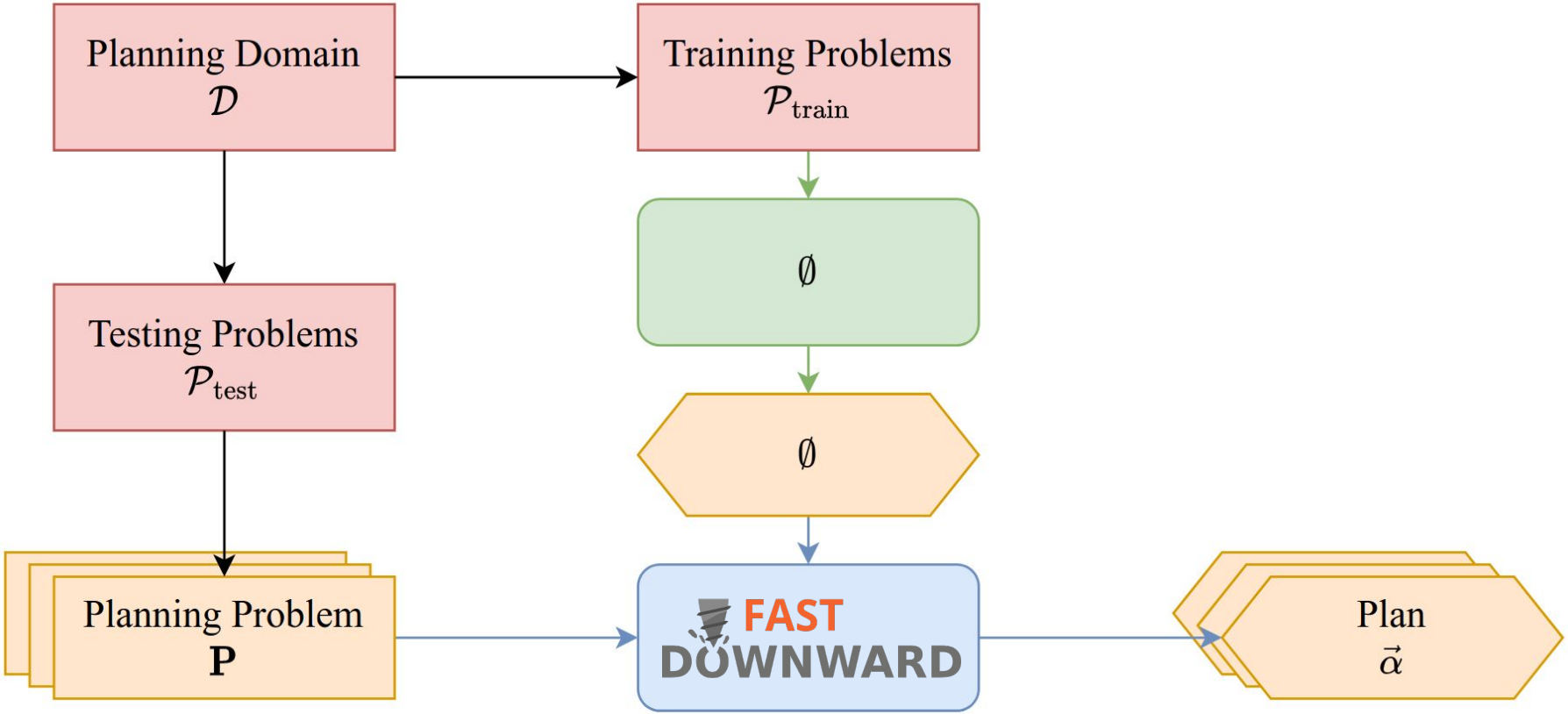


2. Extrapolation; out-of-distribution learning: $f(\mathcal{P}_{\text{test}}) > f(\mathcal{P}_{\text{train}})$



(2.2) Methodologies for solving GP problems

Trivial example: domain-independent planner with no learning

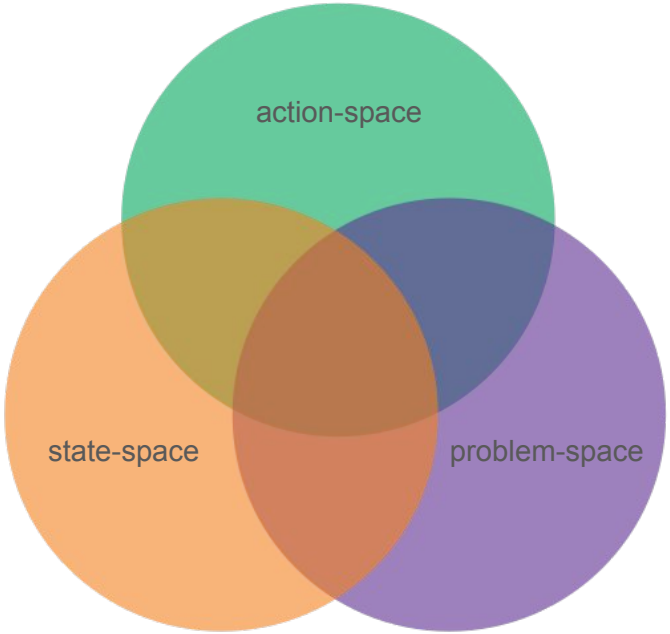


(2.2.1) Forms of learned knowledge

GP Knowledge Taxonomy

Learned knowledge can be taxonomised in one of 3 main spaces:

1. action-space; e.g. policies $\pi(a \mid s, g)$, finite state automata
2. state-space; e.g. heuristic $h(s)$, LTL constraints
3. problem-space; e.g. problem transformation



~ a method can be a combination of one or more spaces

Action-space methods: Policies

Action-space methods: Policies

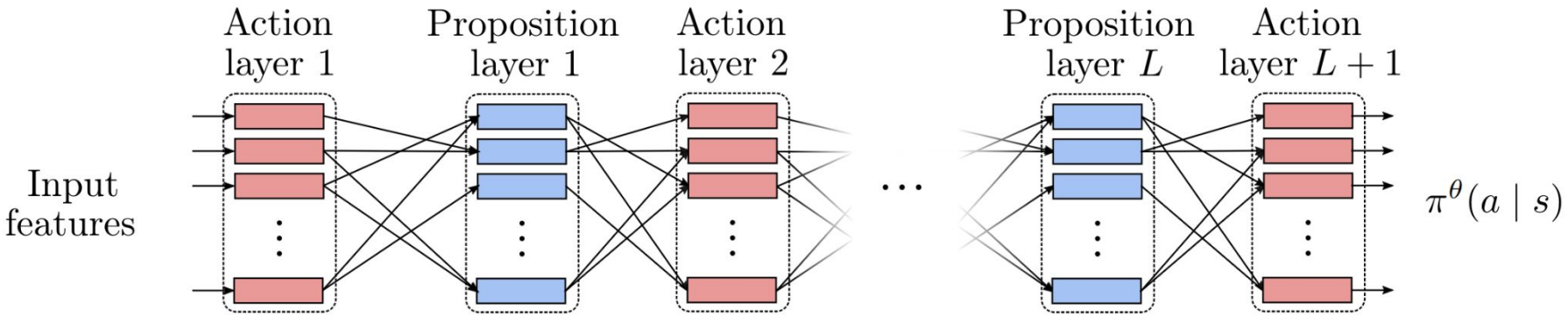
- A policy, usually denoted $\pi(a \mid s)$, is a mapping from states to distributions of actions
- Learning
  - supervised: labelled optimal actions
  - reinforcement: improve based on signal
- Planning
  - usually rollout by sampling or choosing the argmax from $\pi(a \mid s)$
  - repeat until goal is reached or a time or rollout limit is reached
- Analogous to policy gradient methods in RL, e.g. DDPG, TRPO, PPO, etc.

Action-space example: ASNets

Sam Toyer, Sylvie Thiébaux, Felipe W. Trevizan, Lexing Xie: ASNets: Deep Learning for Generalised Planning. J. Artif. Intell. Res. 68: 1-68 (2020)
Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, Lexing Xie: Action Schema Networks: Generalised Policies With Deep Learning. AAAI 2018: 6294-6301

ASNets – architecture

- *Input*: planning state, goal condition, and action applicability
- *Output*: stochastic policy (distribution of actions)
- *Backbone*: Graph Neural Network



ASNets – learning

- *Reinforcement-learning style*; Repeat the following:
  a) sample states by rolling out parameterised policy π^θ
  b) sample states by executing a teacher planner
  c) compute best actions for all sampled states via a teacher planner
  d) update π^θ based on best actions; loss = binary cross entropy
- ~ similar to RL: explore (a) and exploit (b), with reward signals computed by a teacher planner (c) for improving the incumbent policy (d)

ASNets – planning

1. Set $s \leftarrow$ the initial state of the problem
2. Repeat the following:
   a. return plan to s if goal is reached
   b. sample an action a or select action with highest probability from $\pi^{\theta}(a \mid s)$
   c. $s \leftarrow$ apply action a at s

ASNets – key attributes

- *Extrapolates* to testing problems larger than training problems
- *Sound* (returned solutions are correct)
- *Not complete* (if a solution exists, it is found)
- Plans on *grounded* representations
- Originally designed for *probabilistic planning*
- Extended to *numeric planning* [Wang and Thiébaux, ICAPS'24]

Ryan Xiao Wang, Sylvie Thiébaux: Learning Generalised Policies for Numeric Planning. ICAPS 2024: 633-642

ASNets – notable achievements

- First deep learning architecture for learning generalised policies (2018)
- Blocksworld
  a) 25 training problems with 8-10 blocks
  b) solves all testing problems with 35-50 blocks
- Outperforms (probabilistic) planners on various domains
- Sparse regularisation loss for *interpretable* models

State-space methods: Heuristics

State-space methods: Heuristics

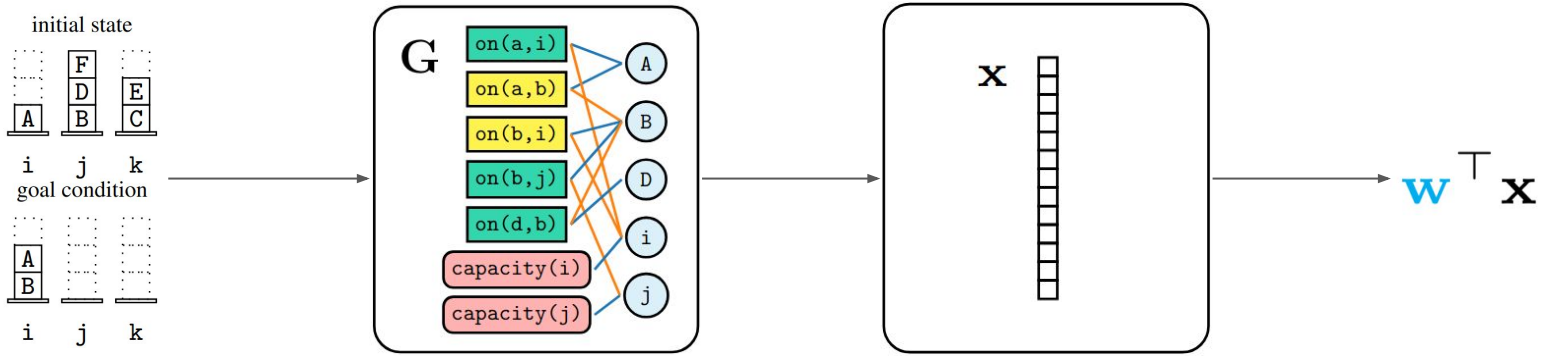
- A heuristic is a real-valued function $h: S \rightarrow \boldsymbol{R}$ estimating cost to go to a goal
- Learning
  - supervised: labelled optimal actions
  - reinforcement: improve based on signal
- Planning
  - combine with heuristic search, e.g. GBFS; but *NOT* A* as learned heuristics not guaranteed to be optimal
  - for greedy policies: select action whose successor has lowest *h* value
- Analogous to value function approximation in RL, e.g. Q-Learning, TD(ƛ), DQN, etc.

State-space example: (WL-)GOOSE

Dillon Z. Chen, Felipe W. Trevizan, Sylvie Thiébaux: Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. ICAPS 2024: 68-76
Dillon Z. Chen, Sylvie Thiébaux, Felipe W. Trevizan: Learning Domain-Independent Heuristics for Grounded and Lifted Planning. AAAI 2024: 20078-20086

GOOSE – architecture

- *Input*: planning state and goal condition
- *Output*: scalar heuristic value
- *Backbone*: Weisfeiler-Leman Graph Kernel



GOOSE – learning

- *Supervised learning*
- Generate optimal plans from training data
- Optimise weights subject to predicting optimal
  a) h^* values; loss = mean squared error
  b) state rankings; loss = cf. [Garrett et al., IJCAI'16; Hao et al., IJCAI'24]

Caelan Reed Garrett, Leslie Pack Kaelbling, Tomás Lozano-Pérez: Learning to Rank for Synthesizing Planning Heuristics. IJCAI 2016: 3089-3095
Mingyu Hao, Felipe W. Trevizan, Sylvie Thiébaux, Patrick Ferber, Jörg Hoffmann: Guiding GBFS through Learned Pairwise Rankings. IJCAI 2024: 6724-6732

GOOSE – planning

- Plug into a heuristic search algorithm
  - e.g. GBFS

GOOSE – key attributes

- *Extrapolates* to testing problems larger than training problems
- *Sound* (returned solutions are correct)
- *Complete* (if a solution exists, it is found)
- Plans on *lifted* representations
- Extended to *probabilistic planning* [Zhang and Trevizan, Tech. Rep. 24]
- Extended to *numeric planning* [Chen and Thiébaux, NeurIPS'24]

Dillon Z. Chen, Sylvie Thiébaux: Graph Learning for Numeric Planning. NeurIPS 2024

GOOSE – notable achievements

- First learning approach to outperform domain-independent heuristics on competition IPC benchmarks with hundreds of objects (2024)
- Orders of magnitude more efficient than neural networks for learning and planning
- Maximally expressive compared to graph neural network counterparts
- Symbolic linear model with *interpretable* features

Problem-space methods: Problem transformation

Problem-space methods: Problem transformation

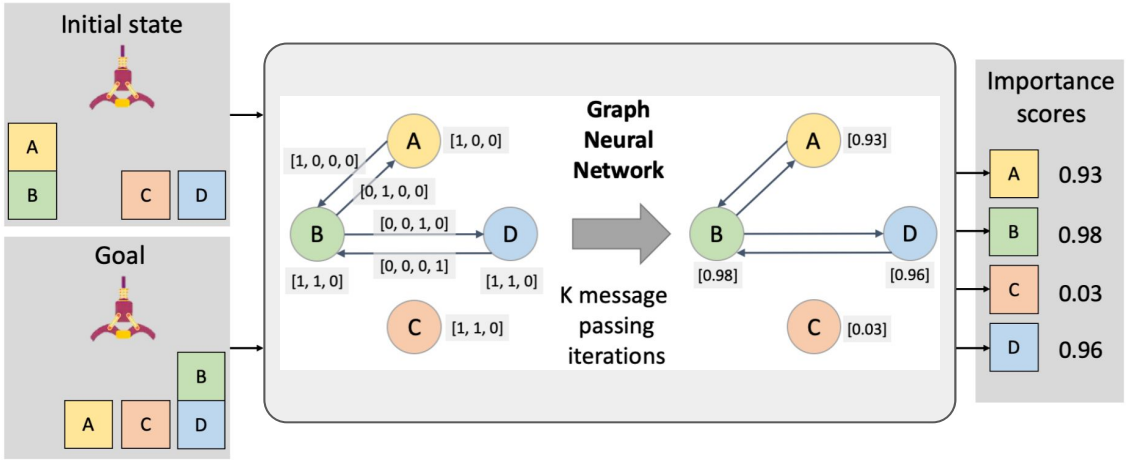
- converts *Prob* = $\langle P, O, A, s_0, G \rangle$ into *Prob'* = $\langle P', O', A', s_0', G' \rangle$, ideally such that a plan for *Prob'* can be transformed back to a plan for *Prob*
- advantage: scales with the improvement of planners over time

Problem-space example: PLOI

Tom Silver, Rohan Chitnis, Aidan Curtis, Joshua B. Tenenbaum, Tomás Lozano-Pérez, Leslie Pack Kaelbling: Planning with Learned Object Importance in Large Problem Instances using Graph Neural Networks. AAAI 2021: 11962-11971

PLOI – architecture

- *Input*: planning state and goal condition
- *Output*: scalar scores in range *[0, 1]* for each object in the problem
- *Backbone*: Graph Neural Network



PLOI – learning

- *Supervised learning*
- loss = binary cross entropy
- Greedily approximate ground truth labels of relevant objects in training data:
  a) start with full object set O
  b) incrementally remove some object from O until removing an object causes the problem to be unsolvable

PLOI – planning

1. score objects in a problem; select threshold value γ from the range *(0, 1)*
2. for N = 1, 2, 3, … until the problem is solved:
   a. keep the set of objects with score greater than γ^N
   b. try to solve the problem with the subset of objects

PLOI – key attributes

- *Extrapolates* to testing problems larger than training problems
- *Sound* (returned solutions are correct)
- *Complete* (if a solution exists, it is found)
- Learns on *lifted* representations
- Can be extended to *probabilistic and numeric planning*

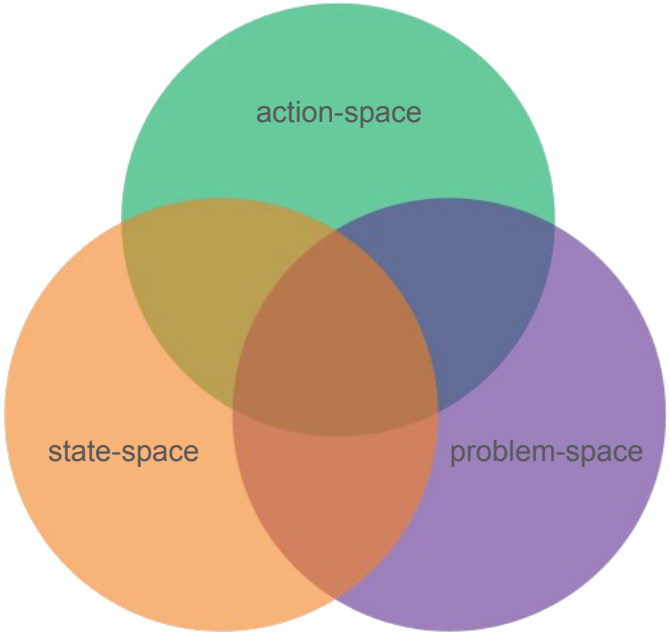
PLOI – notable achievements

- Always matches or improves upon base planner in experiments
- Extended to handle continuous task and motion planning problems
- Learns a problem transformation ⇒ can be used with any backend planner

(2.2.1) Forms of Learned Knowledge – *Summary*

Learned knowledge can be taxonomised in one of 3 main spaces:

1. action-space; e.g. policies
2. state-space; e.g. heuristic
3. problem-space; e.g. problem transformation



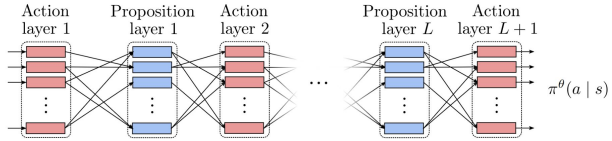
~ a method can be a combination of one or more spaces

(2.2.2) Tools for learning knowledge

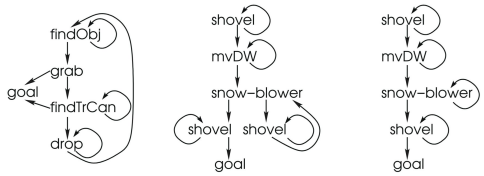
Approaches for synthesising generalised plans

Diverse body of approaches for synthesising generalised plans

- graph learning
- abstraction
- constraint programming
- program search
- goal regression
- language models



Toyer et al., AAAI'18; JAIR'20



Srivastava et al., AAAI'11

1. Policy: $\bigvee_{(s,s')} Good(s,s')$, s is non-goal state,
2. V_1.: Exactly-1 $\{V(s,d) : V^*(s) \leq d \leq \delta V^*(s)\}$,[1]
3. V_2: $Good(s,s') \rightarrow V(s,d) \wedge V(s',d')$, $d' < d$,
4. Goal: $\bigvee_{f:[\![f(s)]\!] \neq [\![f(s')]\!]} Select(f)$, one $\{s,s'\}$ is goal,
5. Bad trans: $\neg Good(s,s')$ for s solvable, and s' dead-end,
6. D2-sep: $Good(s,s') \wedge \neg Good(t,t') \rightarrow D2(s,s';t,t')$, where $D2(s,s';t,t')$ is $\bigvee_{\Delta_f(s,s') \neq \Delta_f(t,t')} Select(f)$.

Francès et al., AAAI'21

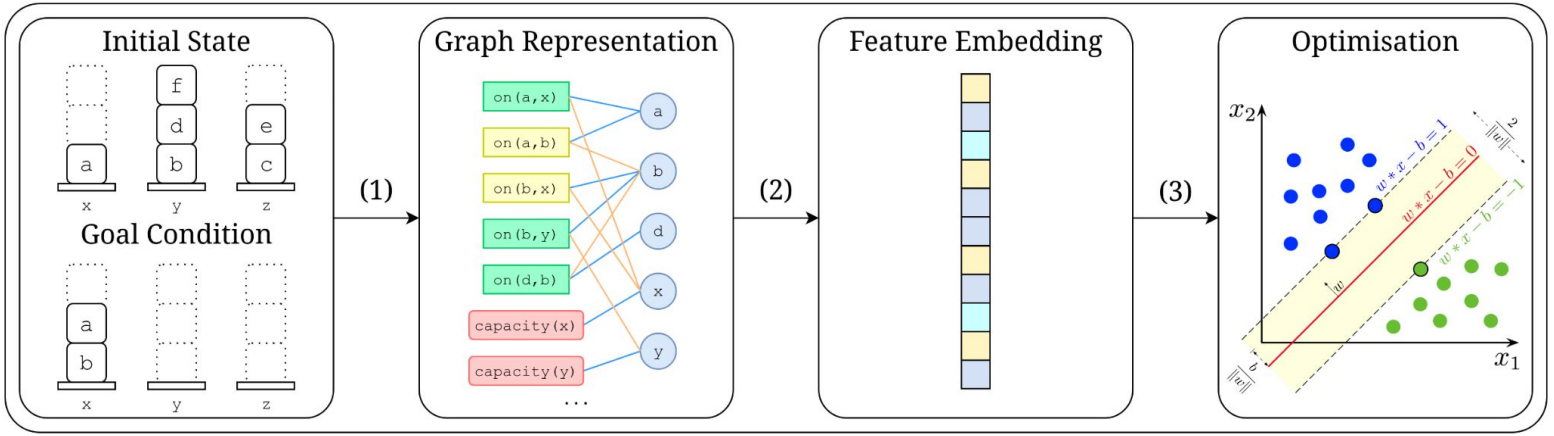
Algorithm 1: Best-First Generalized Planning (BFGP).

Data: The GP problem $\mathcal{P}$, pointers Z, program lines n, a list of evaluation functions $\mathcal{F}$
Result: A generalized plan Π that solves $\mathcal{P}$
$Open_{\mathcal{F}} \leftarrow \{\Pi_{empty}\}$;
while $Open_{\mathcal{F}} \neq \emptyset$ **do**
  $\Pi \leftarrow$ extractBestProgram($Open_{\mathcal{F}}$) ;
  $ChildrenPrograms \leftarrow$ expandProgram(Π, Z, n) ;
  for $\Pi' \in ChildrenPrograms$ **do**
    if *isDeadEnd*($\Pi', \mathcal{P}$) **then**
      continue
    if *isGoal*($\Pi', \mathcal{P}$) **then**
      return (Π')
    $Open_{\mathcal{F}} \leftarrow$ insertProgram($Open_{\mathcal{F}}, \Pi'$);
  end
end
return () // no solution found

Segovia-Aguas et al., ICAPS'21; AIJ'24

Graph Learning

e.g. ASNets, PLOI, GOOSE in Sec. 2.2.1.



representation
- domain information: predicates, schema
- problem information: objects, state, goals

architecture
- graph neural networks
- graph kernels
- transformers (GNNs + positional encodings)

optimisation
- imitation learning, supervised, unsupervised, reinforcement learning
- MSE, CE, ranking

Abstraction

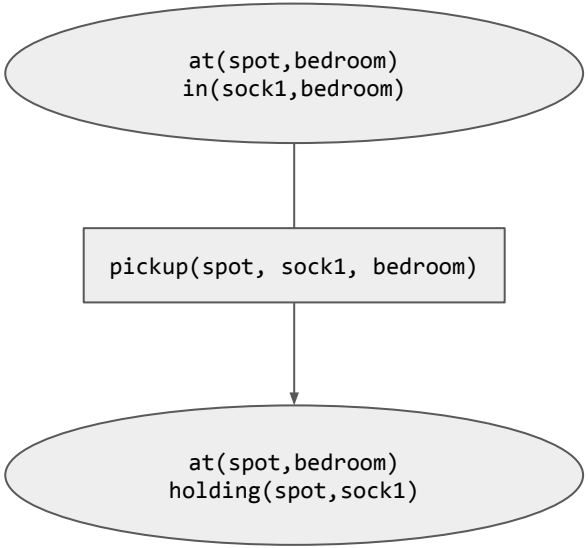
- e.g. place all the dirty objects on the floor into the laundry basket

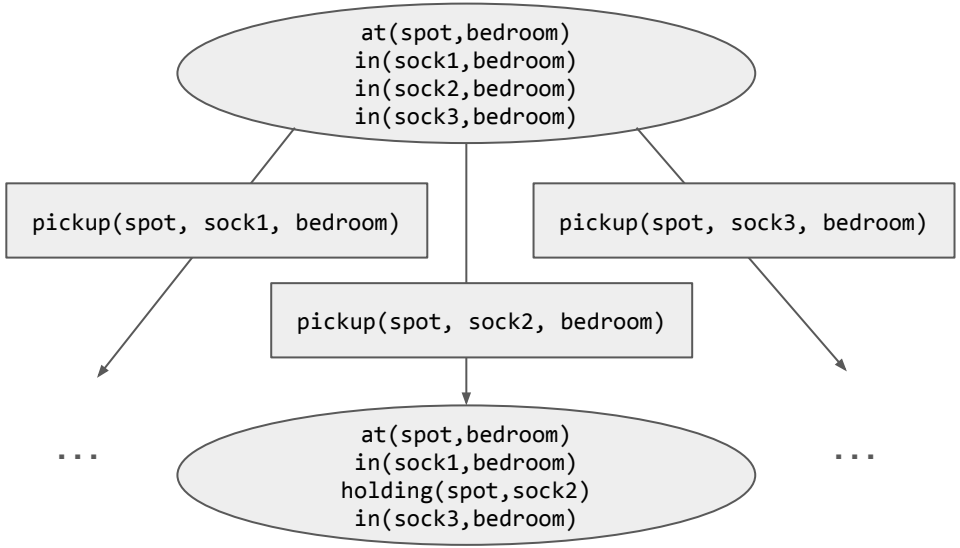
Day 1

Day 10

How to synthesise a policy across instances?

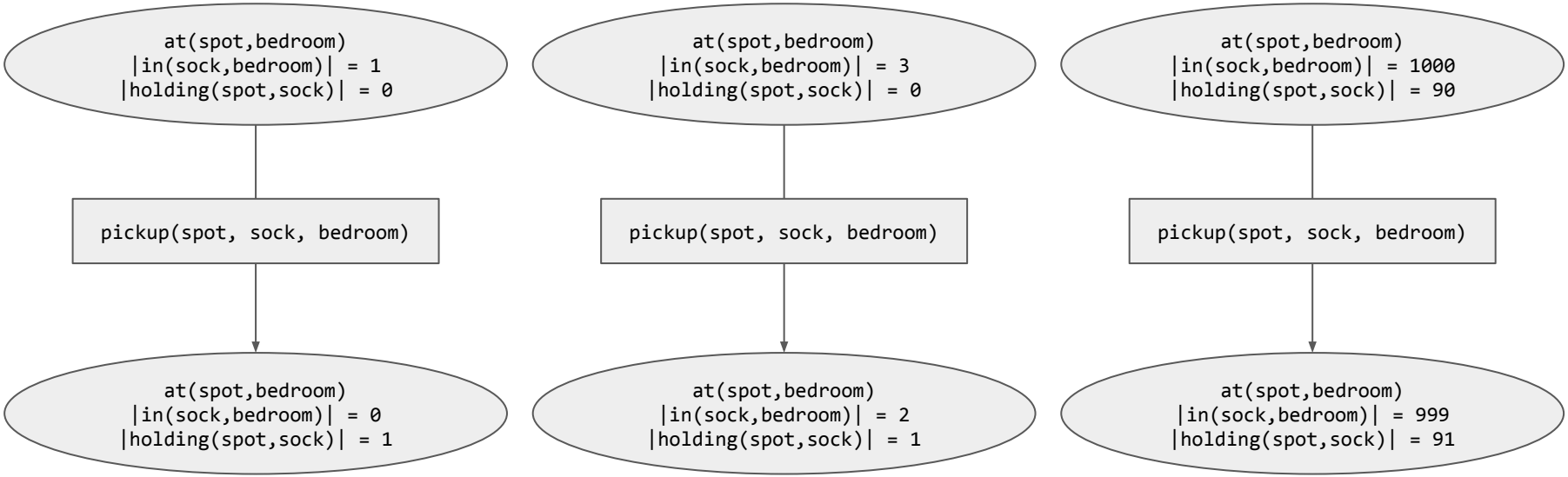
- many different instances = many different transitions, … what can we do?





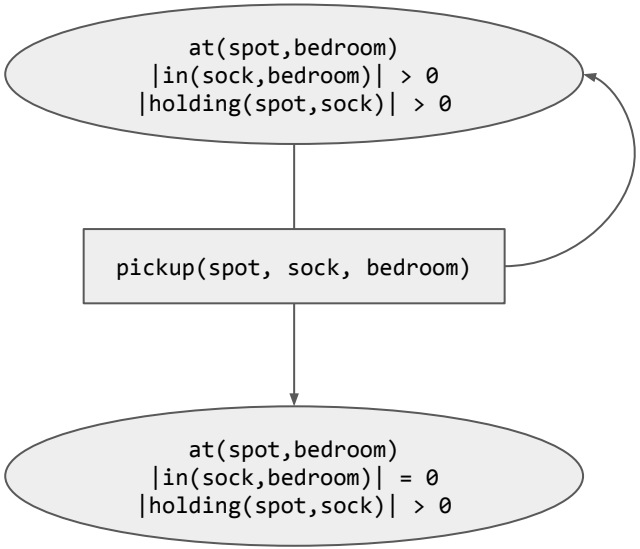
Abstraction

- we can abstract equivalent objects away:



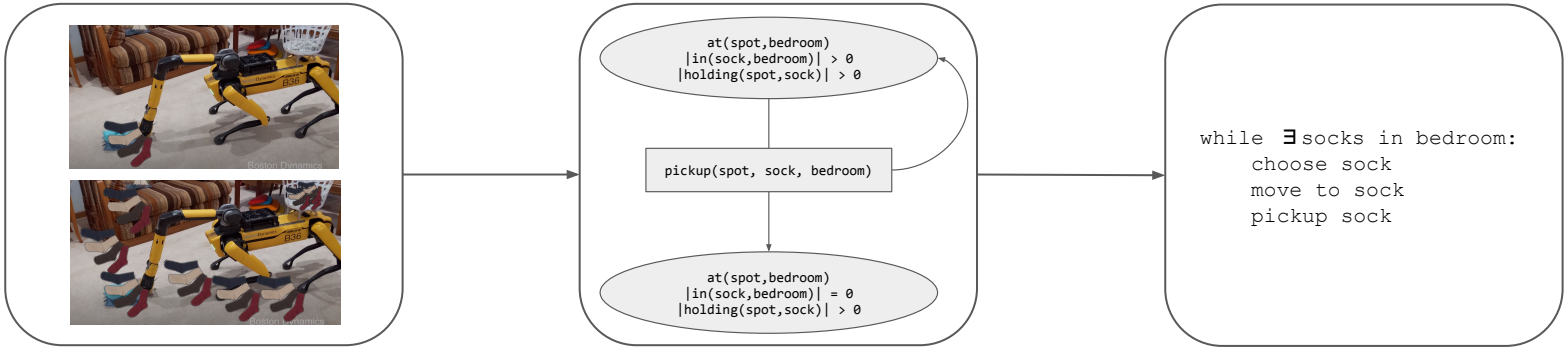
Abstraction

- we can even abstract numbers away via qualitative states [Srivastava et al., AAAI'11]



Siddharth Srivastava, Shlomo Zilberstein, Neil Immerman, Hector Geffner: *Qualitative Numeric Planning*. AAAI 2011

Abstraction



find an abstraction

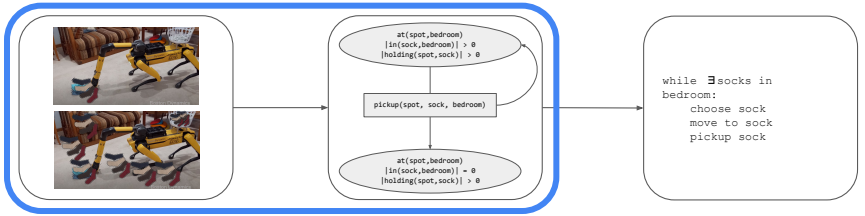
- synthesise a single nondeterministic problem representing a family of problems

solve the abstraction

- synthesise a generalised plan for the original family of problems from solving the abstraction

Finding Abstractions

- Illanes and McIlraith, AAAI'19
  - bag equivalent objects

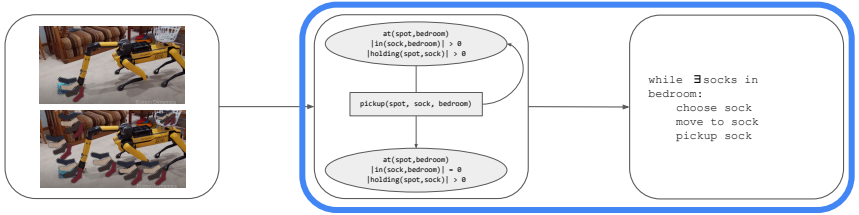


[Riddle et al., HSDIP'16, Fuentetaja and Rosa, AI Comm. 2016]

- Bonet et al., AAAI'19
  - MaxSAT over description logic features for planning

[Martín and Geffner, Appl. Intell. 2004]

Solving Abstractions



- 'generate-and-test'
  - iteratively *generate* policies and *test* for termination
  - generate: e.g. anytime FOND solver [Illanes and McIlraith, AAAI'19; Zeng et al., IJCAI'22]
  - test: e.g. Sieve algorithm [Srivastava et al., AAAI'11; Srivastava, JAIR'23]
- compilation approach
  - compile to FOND($^+$) restricted to terminating policies and solve
  - FOND: [Bonet and Geffner, JAIR'20]; FOND$^+$: [Rodriguez et al., JAIR'22]

Constraint Programming

- finding and solving abstractions 'end-to-end'
- synthesise generalised policies with SAT [Francès et al., AAAI'21]
- synthesise generalised policy sketches with ASP [Drexler et al, ICAPS'22]

1. Policy: $\bigvee_{(s,s')} Good(s, s')$, s is non-goal state,
2. V_1.: Exactly-1 $\{V(s, d) : V^*(s) \leq d \leq \delta V^*(s)\}$,[1]
3. V_2: $Good(s, s') \rightarrow V(s, d) \wedge V(s', d'), d' < d$,
4. Goal: $\bigvee_{f:[\![f(s)]\!] \neq [\![f(s')]\!]} Select(f)$, one $\{s, s'\}$ is goal,
5. Bad trans: $\neg Good(s, s')$ for s solvable, and s' dead-end,
6. D2-sep: $Good(s, s') \wedge \neg Good(t, t') \rightarrow D2(s, s'; t, t')$, where $D2(s, s'; t, t')$ is $\bigvee_{\Delta_f(s,s') \neq \Delta_f(t,t')} Select(f)$.

Francès et al., AAAI'21

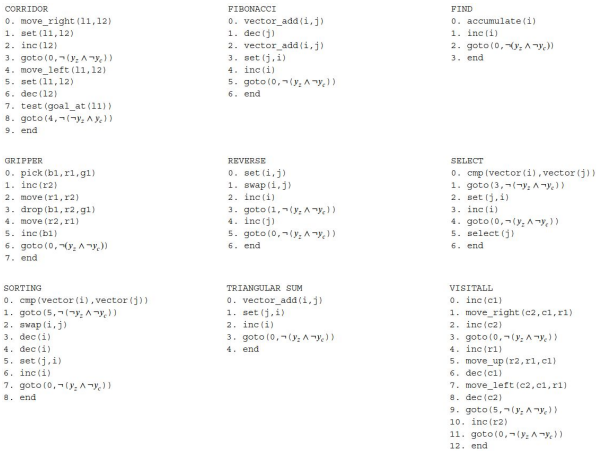
Listing 1: Full ASP code for learning sketches: constraints C1-C8 satisfied. Optimization in lines 24-25 for finding a simplest solutions measured by number of sketch rules plus sum of feature complexities.

```
1  % C1: construct rules and select features.
2  { select(F) } :- feature(F).                { rule(1..max_sketch_rules) }.
3  { c_eq(R, F); c_gt(R, F); c_unk(R, F) } = 1 :- rule(R), numerical(F).
4  { c_pos(R, F); c_neg(R, F); c_unk(R, F) } = 1 :- rule(R), boolean(F).
5  { e_dec(R, F); e_inc(R, F); e_unk(R, F); e_bot(R, F) } = 1 :- rule(R), numerical(F).
6  { e_pos(R, F); e_neg(R, F); e_unk(R, F); e_bot(R, F) } = 1 :- rule(R), boolean(F).
7  % C4 and C7: good and bad state pairs must comply with rules and selected features.
8  { good(R, I, S, S') } :- rule(R), s_distance(I, S, S', _).
9  c_satisfied(R, F, I, S) :- { c_eq(R, F) : V = 0; c_gt(R, F) : V > 0; c_pos(R, F) : V = 1;
       c_neg(R, F) : V = 0; c_unk(R, F) } = 1, rule(R), feature_valuation(F, I, S, V),
       s_distance(I, S, S', _).
10 e_satisfied(R, F, I, S, S') :- { e_dec(R, F) : V > V'; e_inc(R, F) : V < V';
       e_pos(R, F) : V' = 1; e_neg(R, F) : V' = 0; e_bot(R, F) : V = V'; e_unk(R, F) } = 1,
       rule(R), feature_valuation(F, I, S, V), feature_valuation(F, I, S', V'),
       s_distance(I, S, S', _).
11 :- { not c_satisfied(R, F, I, S); not e_satisfied(R, F, I, S, S') } != 0, select(F), good(
       R, I, S, S').
12 :- { not c_satisfied(R, F, I, S) : select(F); not e_satisfied(R, F, I, S, S') : select(F)
       } = 0, rule(R), s_distance(I, S, S', _), not good(R, I, S, S').
13 % C2: there must be one subgoal tuple for each state with unbounded width.
14 { subgoal(I, S, T) : tuple(I, S, T) } = 1 :- solvable(I, S), exceed(I, S).
15 % C3: states underlying subgoal tuples must be good.
16 :- { good(R, I, S, S') : rule(R) } = 0, subgoal(I, S, T), contain(I, S, T, S').
17 % C5: good pairs to dead-end states must be at larger distance than subgoal tuple.
18 :- D <= D', s_distance(I, S, S', D), t_distance(I, S, T, D'), subgoal(I, S, T),
       good(_, I, S, S'), solvable(I, S), unsolvable(I, S').
19 % C8: ensure acyclicity.
20 order(I, S, S') :- solvable(I, S), solvable(I, S'), good(_, I, S, S'), order(I, S').
21 order(I, S) :- solvable(I, S), order(I, S, S') : good(_, I, S, S'), solvable(I, S),
       solvable(I, S').
22 :- solvable(I, S), not order(I, S).
23 % Optimization objective: smallest number of rules plus the sum of feature complexities.
24 #minimize { C,complexity(F, C) : complexity(F, C), select(F) }.
25 #minimize { 1,rule(R) : rule(R) }.
```

Drexler et al., ICAPS'22

Program Search

- directly search over space of programs/generalised plans
- compile program search as planning problem: [Segovia-Aguas et al., AIJ'19]
- Best-First Generalised Planning: GP as heuristic search [Segovia-Aguas et al., JAIR'24]
- programs consist of
  - planning actions
  - goto instructions
  - termination instructions

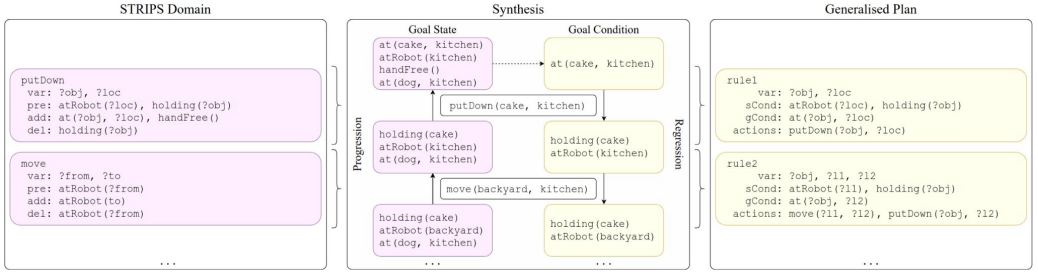


```
CORRIDOR
0. move_right(l1,l2)
1. set(l1,l2)
2. inc(l2)
3. goto(0,¬(y_z ∧ ¬y_c))
4. move_left(l1,l2)
5. set(l1,l2)
6. dec(l2)
7. test(goal_at(l1))
8. goto(4,¬(¬y_c ∧ y_z))
9. end

FIBONACCI
0. vector_add(i,j)
1. dec(j)
2. vector_add(i,j)
3. set(j,i)
4. inc(i)
5. goto(0,¬(y_z ∧ ¬y_c))
6. end

FIND
0. accumulate(i)
1. inc(i)
2. goto(0,¬(y_z ∧ ¬y_c))
3. end

GRIPPER
0. pick(b1,r1,g1)
1. inc(r2)
2. move(r1,r2)
3. drop(b1,r2,g1)
4. move(r2,r1)
5. inc(b1)
6. goto(0,¬(y_z ∧ ¬y_c))
7. end

REVERSE
0. set(i,j)
1. swap(i,j)
2. inc(i)
3. goto(1,¬(y_z ∧ ¬y_c))
4. inc(j)
5. goto(0,¬(y_z ∧ ¬y_c))
6. end

SELECT
0. cmp(vector(i),vector(j))
1. goto(3,¬(¬y_z ∧ ¬y_c))
2. set(j,i)
3. inc(i)
4. goto(0,¬(y_z ∧ ¬y_c))
5. select(j)
6. end

SORTING
0. cmp(vector(i),vector(j))
1. goto(5,¬(¬y_z ∧ ¬y_c))
2. swap(i,j)
3. dec(i)
4. dec(i)
5. set(j,i)
6. inc(i)
7. goto(0,¬(y_z ∧ ¬y_c))
8. end

TRIANGULAR SUM
0. vector_add(i,j)
1. set(j,i)
2. inc(i)
3. goto(0,¬(y_z ∧ ¬y_c))
4. end

VISITALL
0. inc(c1)
1. move_right(c2,c1,r1)
2. inc(c2)
3. goto(0,¬(y_z ∧ ¬y_c))
4. inc(r1)
5. move_up(r2,r1,c1)
6. dec(c1)
7. move_left(c2,c1,r1)
8. dec(c2)
9. goto(5,¬(y_z ∧ ¬y_c))
10. inc(r2)
11. goto(0,¬(y_z ∧ ¬y_c))
12. end
```

Segovia-Aguas et al., JAIR'24

Goal Regression

- goal regression determines minimal and sufficient condition for achieving a goal under a sequence of actions
- used to synthesise generalised, first-order policies by
  - Gretton and Thiébaux, UAI'04
  - Illanes and McIlraith, AAAI'19
  - Chen et al., AAAI'26



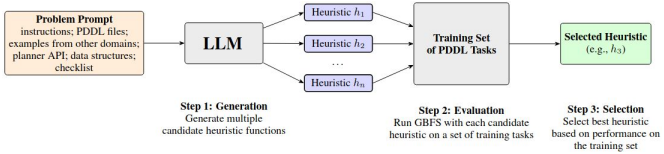
Language Models

prompt for programs representing generalised plans

```
def get_plan(objects, init, goal):
    # Helper function to extract objects of a specific type
    def get_objects_of_type(type_name):
        return [obj for obj in objects if obj.startswith(type_name)]
    # Helper function to find the initial or goal location of a ball
    def get_location(state, ball):
        for atom in state:
            if atom[0] == 'at' and atom[1] == ball:
                return atom[2]
        return None
    # Extract balls, rooms, and grippers from objects
    balls = get_objects_of_type('ball')
    rooms = get_objects_of_type('room')
    grippers = get_objects_of_type('gripper')
    ...
```

standalone solvers

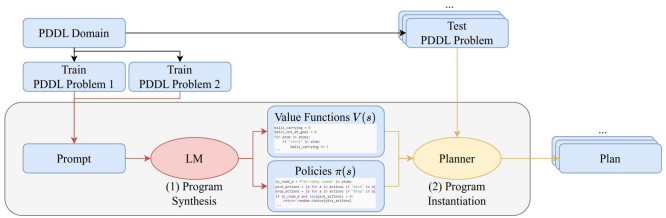
Silver et al., AAAI'24



generalised heuristic functions

Tuisov et al., 2025

figure from Corrêa et al., NeurIPS'25



generalised action policies

Chen et al., EWRL'25

(2.3) Evaluating L4P Methodologies

Theoretical and Empirical Measures

- Theory
  - expressivity
  - generalisability
  - complexity and decidability
- Practice
  - training costs
  - planning costs
  - solution quality

Theoretical Measures – Expressivity

- expressivity = range of functions or hypotheses a model can potentially learn
- ~ measures the *best* theoretical performance of a model
- e.g.
  - domain-dependent GNNs related to C_2 logic [Stålberg et al., ICAPS'22]
  - domain-independent GNNs can express h^{max}, h^{add} [Chen et al., AAAI'24]

Theoretical Measures – Generalisability

- generalisability = performance on unseen data
- ~ measures the estimated *average* performance of a model
- see e.g. https://mlstory.org/generalization.html
  - Algorithmic stability
  - Vapnik–Chervonenkis (VC) dimension
  - Rademacher complexity
  - PAC learning
- almost no theory for generalised planning

Theoretical Measures – Complexity and Decidability

- theoretical computational resources required to synthesise a generalised plan
- usually trade-off with expressivity
- e.g.
  - QNP (planning with loops with non-deterministic semantics) is decidable (EXPTIME-complete) [Srivastava et al., AAAI'11; Bonet and Geffner, JAIR'20]
  - planning with loops with deterministic semantics is undecidable [Srivastava et al., AAAI'15; Srivastava, JAIR'23]

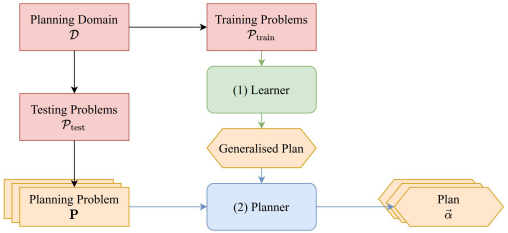
Empirical Measures

- Training Costs
  - resources for synthesising a generalised plan
- Planning Costs
  - resources for instantiating a generalised plan
- Solution Quality
  - quality (e.g. size) of a generalised plan, and of instantiated plans

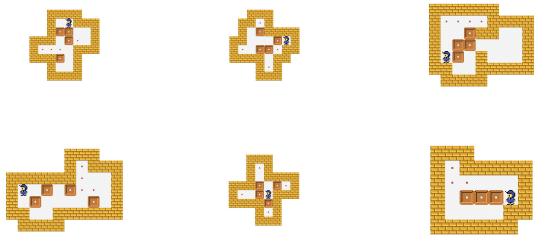
Benchmarks

- Various papers often introduce their own train and test splits
- IPC 2023 Learning Track introduced canonical splits

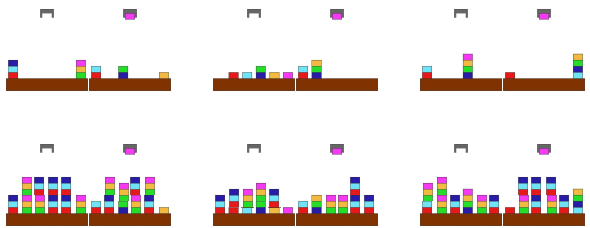
  https://github.com/ipc2023-learning/benchmarks
- Recall interpolation and extrapolation settings. Most L4P works focus on extrapolation



1. Interpolation; in-distribution learning: $f(\mathcal{P}_{\text{test}}) \leq f(\mathcal{P}_{\text{train}})$



2. Extrapolation; out-of-distribution learning: $f(\mathcal{P}_{\text{test}}) > f(\mathcal{P}_{\text{train}})$



ICAPS 2025 Tutorial
Learning for Generalised Planning: Lab

Dillon Z. Chen Felipe Trevizan Sylvie Thiébaux



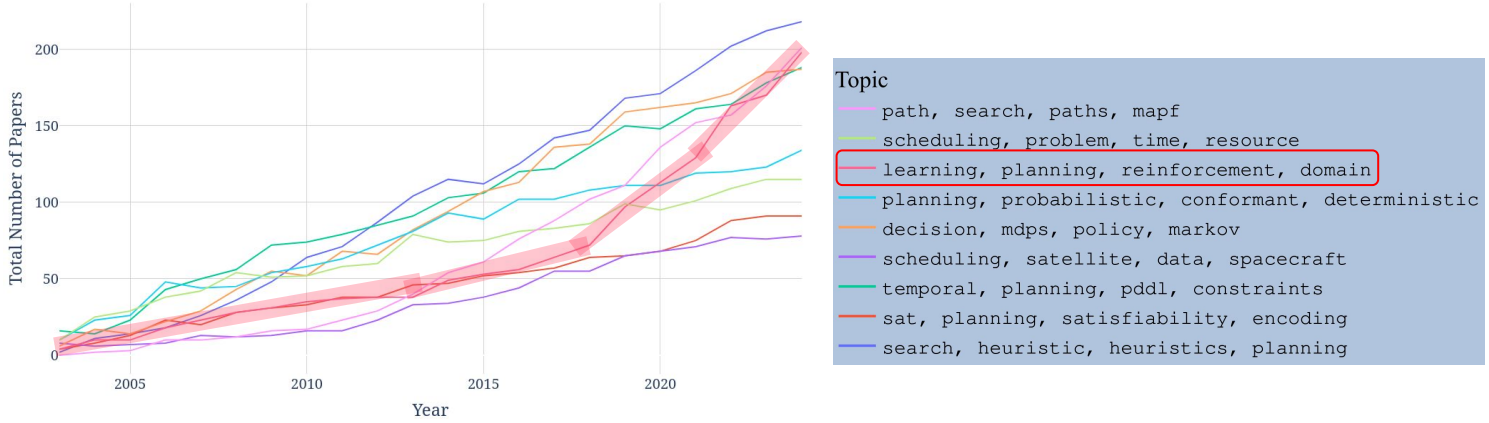
https://github.com/DillonZChen/wlplan/blob/main/docs/source/tutorials/1_introduction.ipynb

https://l4p-tutorial.github.io/

Motivation

Learning for (Generalised) Planning is a rapidly growing topic

- learn knowledge from easy-to-solve, small problems
- generalise to problems with unseen initial states/goals, and greater number of objects



Number of papers grouped by topic at ICAPS 2003–2024

Motivation

- plenty of open source planner systems and planning libraries

FAST DOWNWARD

ENHSP

pddl

pypi v0.4.3 | python 3.9 | 3.10 | 3.11 | 3.12 | status pre-alpha | implementation cpython | wheel yes | license MIT

test failing | lint passing | docs passing | codecov 88%

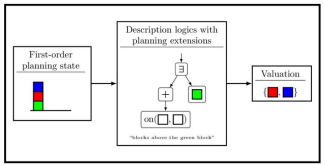
flake8 checked | mypy checked | code style black | docs mkdocs

`pddl` aims to be an unquestionable and complete parser for PDDL 3.1.

- few libraries specifically for learning for planning

Description Logics State Features for Planning Library (DLPlan)

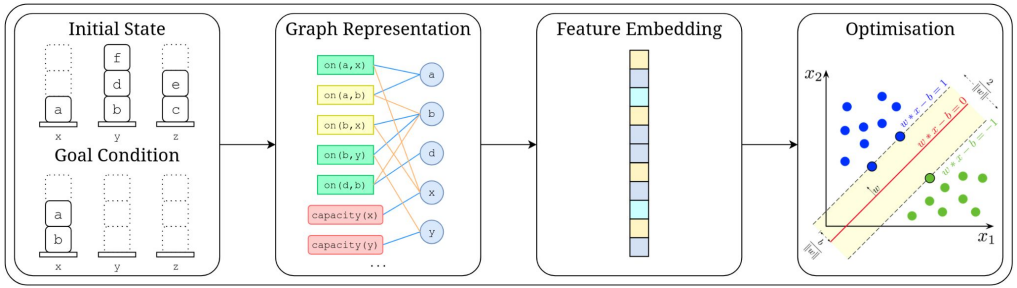
Welcome to the DLPlan-library.



WLPlan

C++ package with Python bindings that implements

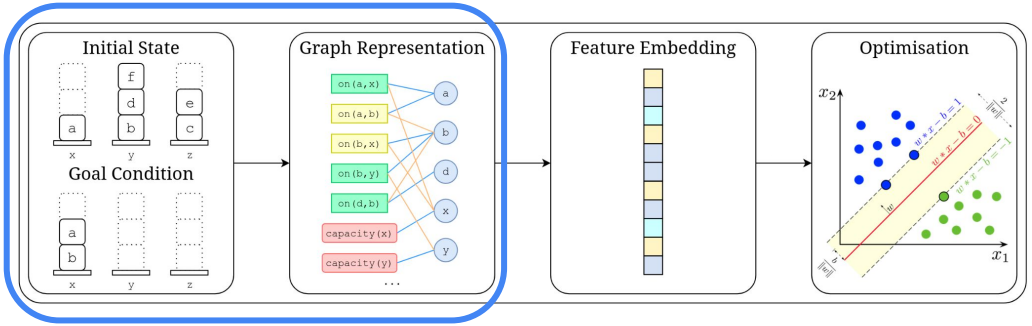
1. Graph representations of planning tasks
2. Embeddings of planning tasks and graphs
3. Serialisation of models




```
pip install wlplan
```

https://dillonzchen.github.io/wlplan/

Graph representations of planning tasks



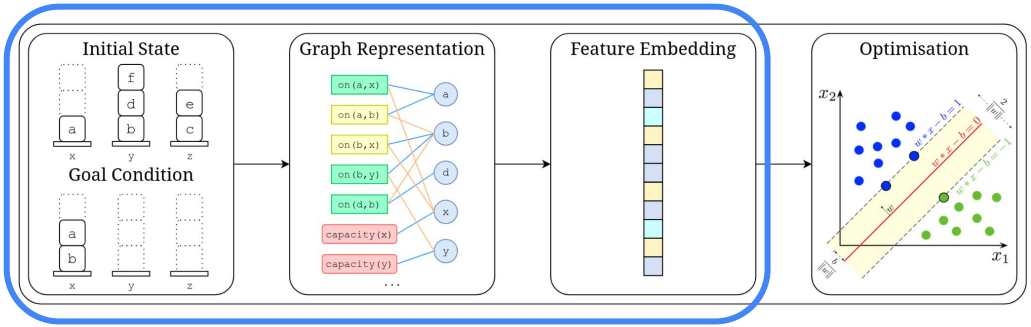
Python

```python
g_generator = init_graph_generator(
    graph_representation="ilg", domain=domain
)
graphs = g_generator.to_graphs(dataset)
```

C++

```cpp
graph_generator::ILGGenerator g_generator =
    graph_generator::ILGGenerator(domain);
std::vector<graph_generator::Graph> graphs =
   g_generator.to_graphs(dataset);
```

Embeddings of planning tasks and graphs



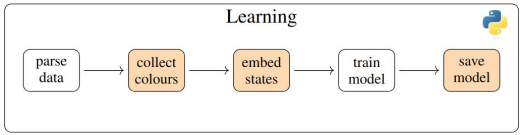
Python

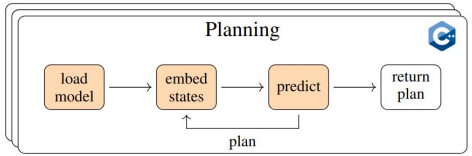
```python
f_generator = init_feature_generator(
   feature_algorithm="wl", domain=domain
)
f_generator.collect(dataset)
X = f_generator.embed(dataset)
```

C++

```cpp
feature_generator::WLFeatures f_generator =
    feature_generator::WLFeatures(domain);
...
std::vector<int> x =
    f_generator.embed_state(state);
```

Serialisation of models





Python

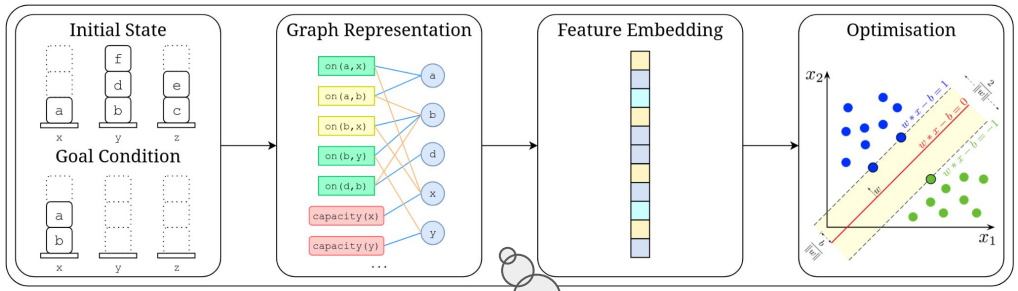
```python
f_generator.save(
    filename=x,
    weights=predictor.get_weights(),
)
```

C++

```cpp
std::shared_ptr<feature_generator::Features>
  f_generator = load_feature_generator(model_file);
```

WLPlan Methodology and Configurations

WLPlan Methodology



Algorithm 1: WL algorithm

Input: A graph $\mathbf{G} = \langle \mathbf{V}, \mathbf{E}, \mathbf{F}, \mathbf{L} \rangle$, injective HASH function, and number of iterations L.

Output: Multiset of colours.

1 $c^0(v) \leftarrow \mathbf{F}(v), \forall v \in \mathbf{V}$

2 **for** $l = 1, \ldots, L$ **do for** $v \in \mathbf{V}$ **do**

3 $c^l(v) \leftarrow$

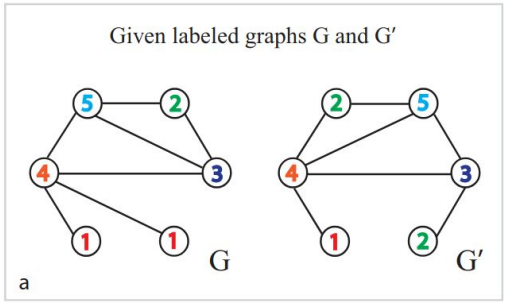
$$\text{HASH}\left(c^{l-1}(v), \bigcup_{\iota \in \Sigma_{\mathrm{E}}} \{\!\{(c^{l-1}(u), \iota) \mid u \in \mathbf{N}_\iota(v)\}\!\}\right)$$

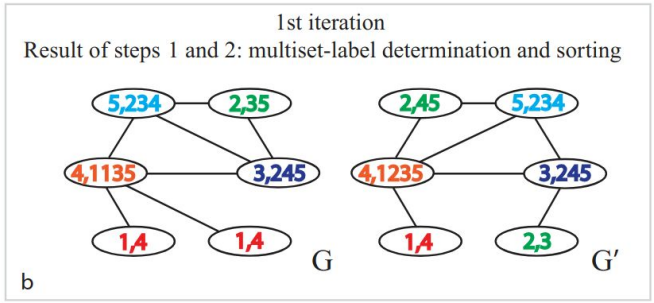
4 **return** $\bigcup_{l=0,\ldots,L} \{\!\{c^l(v) \mid v \in \mathbf{V}\}\!\}$

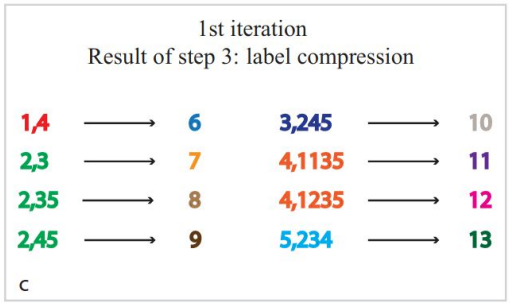
Iteratively refine node colours using neighbour information

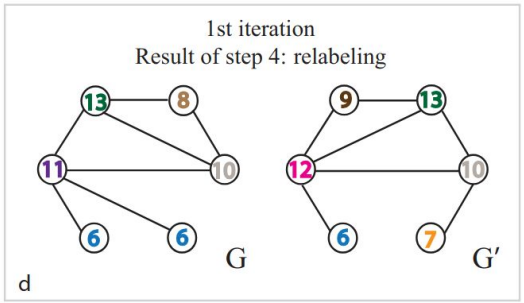
Dillon Z. Chen, Felipe W. Trevizan, Sylvie Thiébaux: Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. ICAPS 2024: 68-76

Weisfeiler-Leman Graph Kernel









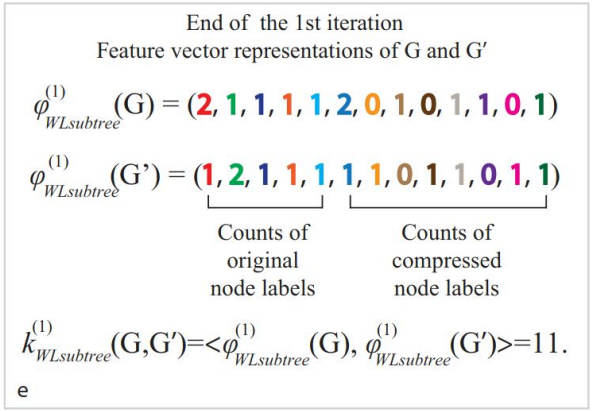
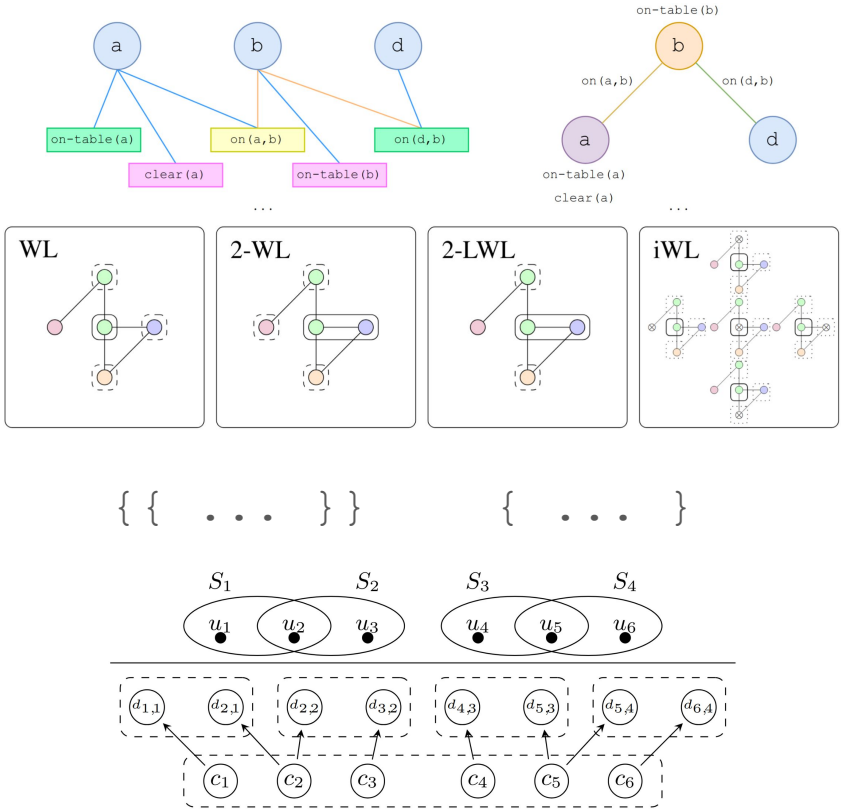


Figure from Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, Karsten M. Borgwardt: Weisfeiler-Lehman Graph Kernels. J. Mach. Learn. Res. 12: 2539-2561 (2011)

WLPlan Configurations

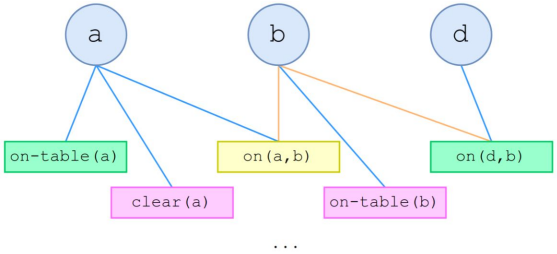
Configurations trade off between expressivity, memory and speed

Graph Representation	ILG
	PLOIG
Feature Generator	WL
	iWL
	2-LWL
	2-KWL
Hash Function	Multiset
	Set
Feature Pruner	None
	MaxSat
	iterMaxSat
	iterMaxSat+freq
Iterations	1, 2, ...

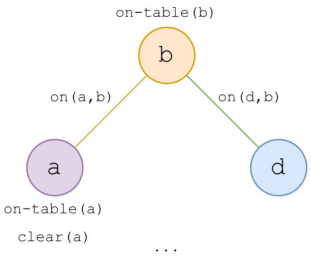


Graph Generator — Representation Options

Graph representations of planning problems



ilg

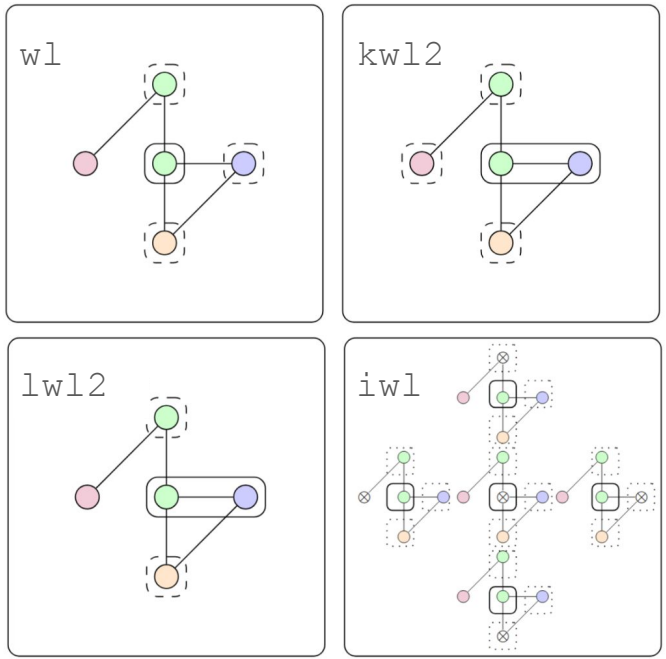


ploig

```
g_generator = init_graph_generator(
    domain : Domain,
    graph_representation : str,
)
```

Feature Generator — Algorithm Options

WL or an extension for generating features




```
f_generator = init_feature_generator(
    domain : Domain,
    feature_algorithm : str,
    multiset_hash : bool,
    pruning : str,
    iterations : int,
)
```

Feature Generator — Hash Function Options

Collect neighbour colours via multiset vs. set

Algorithm 1: WL algorithm

Input: A graph $\mathbf{G} = \langle \mathbf{V}, \mathbf{E}, \mathbf{F}, \mathbf{L} \rangle$, injective HASH function, and number of iterations L.

Output: Multiset of colours.

1 $c^0(v) \leftarrow \mathbf{F}(v), \forall v \in \mathbf{V}$

2 **for** $l = 1, \ldots, L$ **do for** $v \in \mathbf{V}$ **do**

3 $c^l(v) \leftarrow$

$$\text{HASH}\left(c^{l-1}(v), \bigcup_{\iota \in \Sigma_E} \{\!\{(c^{l-1}(u), \iota) \mid u \in \mathbf{N}_\iota(v)\}\!\}\right)$$

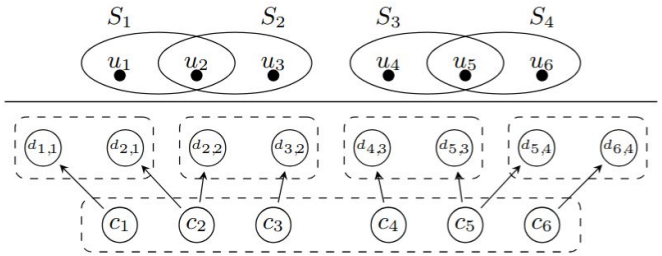
$$\text{HASH}\left(c^{l-1}(v), \bigcup_{\iota \in \Sigma_E} \{(c^{l-1}(u), \iota) \mid u \in \mathbf{N}_\iota(v)\}\right)$$

4 **return** $\bigcup_{l=0,\ldots,L} \{\!\{c^l(v) \mid v \in \mathbf{V}\}\!\}$

```
f_generator = init_feature_generator(
    domain : Domain,
    feature_algorithm : str,
    multiset_hash : bool,
    pruning : str,
    iterations : int,
)
```

Feature Generator — Pruning Options

Option to prune seemingly equivalent features




```
f_generator = init_feature_generator(
    domain : Domain,
    feature_algorithm : str,
    multiset_hash : bool,
    pruning : str,
    iterations : int,
)
```

Feature Generator — Iteration Options

Number of iterations to perform in WL

Algorithm 1: WL algorithm

Input: A graph $\mathbf{G} = \langle \mathbf{V}, \mathbf{E}, \mathbf{F}, \mathbf{L} \rangle$, injective HASH function, and number of iterations L.

Output: Multiset of colours.

1 $c^0(v) \leftarrow \mathbf{F}(v), \forall v \in \mathbf{V}$

2 **for** $l = 1, \ldots, L$ **do** **for** $v \in \mathbf{V}$ **do**

3 $c^l(v) \leftarrow$

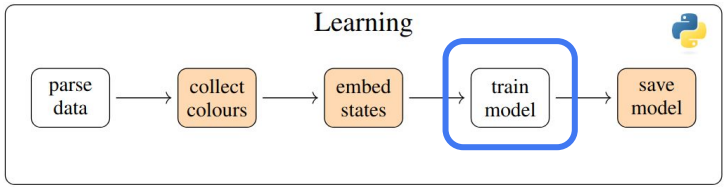
$$\text{HASH}\left(c^{l-1}(v), \bigcup_{\iota \in \Sigma_E} \{\!\{(c^{l-1}(u), \iota) \mid u \in \mathbf{N}_\iota(v)\}\!\}\right)$$

4 **return** $\bigcup_{l=0,\ldots,L} \{\!\{c^l(v) \mid v \in \mathbf{V}\}\!\}$

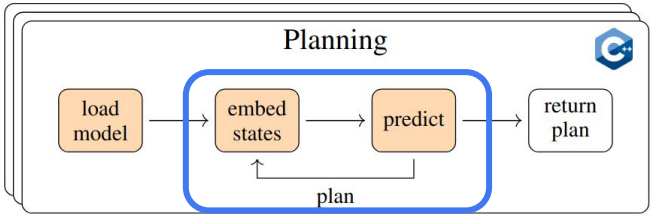
```
f_generator = init_feature_generator(
    domain : Domain,
    feature_algorithm : str,
    multiset_hash : bool,
    pruning : str,
    iterations : int,
)
```

WLPlan Use Cases

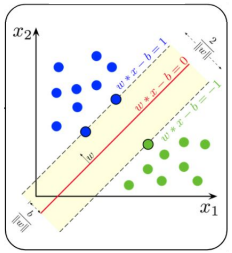
Learning Heuristics

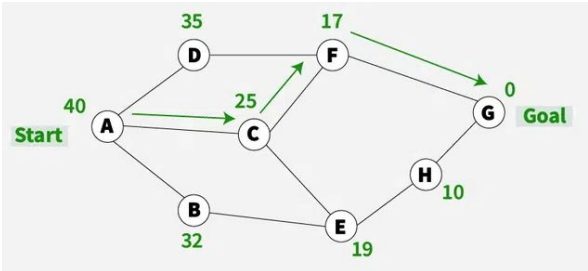


Learn a heuristic



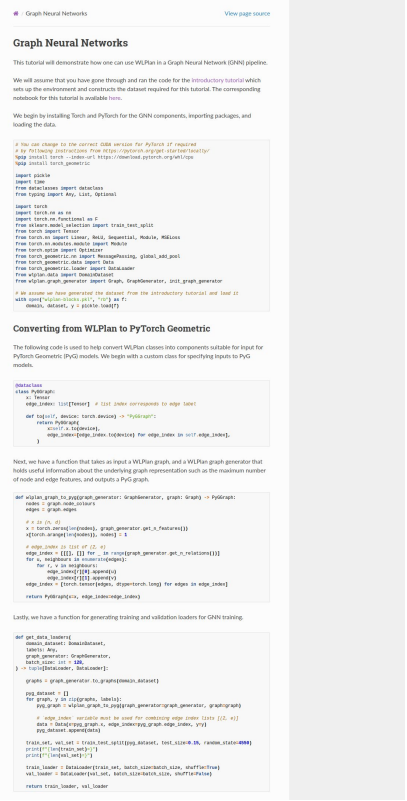
Employ the heuristic in search, e.g. Greedy Best First Search





Graph Neural Network Pipeline

https://dillonzchen.github.io/wlplan/tutorials/3_gnns.html



Graph Neural Networks

View page source

Graph Neural Networks

This tutorial will demonstrate how one can use WLPlan in a Graph Neural Network (GNN) pipeline.

We will assume that you have gone through and ran the code for the introductory tutorial which sets up the environment and constructs the dataset required for this tutorial. The corresponding notebook for this tutorial is available here.

We begin by installing Torch and PyTorch for the GNN components, importing packages, and loading the data.

```
# You can change to the correct CUDA version for PyTorch if required
# by following instructions from https://pytorch.org/get-started/locally/
!pip install torch --index-url https://download.pytorch.org/whl/cpu
!pip install torch_geometric

import pickle
import time
from dataclasses import dataclass
from typing import Any, List, Optional

import torch
import torch.nn as nn
import torch.nn.functional as F
from sklearn.model_selection import train_test_split
from torch.nn import Linear, ReLU, Sequential, Module, MSELoss
from torch_geometric.nn.models.mlp import MLP
from torch.optim import Optimizer
from torch_geometric.nn import MessagePassing, global_add_pool
from torch_geometric.data import Data
from torch_geometric.loader import DataLoader
from wlplan.data import DomainDataset
from wlplan.graph_generator import Graph, GraphGenerator, init_graph_generator

# We assume we have generated the dataset from the introductory tutorial and load it
with open("wlplan-blocks.pkl", "rb") as f:
    domain, dataset, y = pickle.load(f)
```

Converting from WLPlan to PyTorch Geometric

The following code is used to help convert WLPlan classes into components suitable for input for PyTorch Geometric (PyG) models. We begin with a custom class for specifying inputs to PyG models.

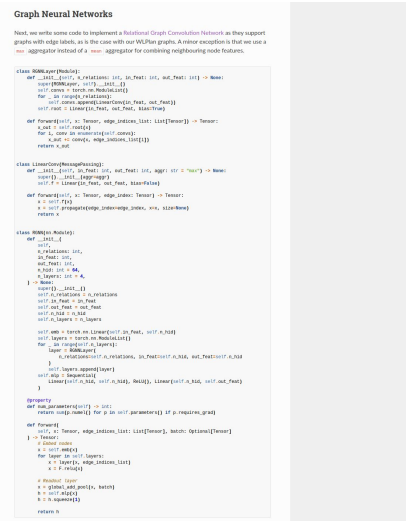
```
@dataclass
class PyGGraph:
    x: Tensor
    edge_index: List[Tensor]  # list index corresponds to edge label

    def to(self, device: torch.device) -> "PyGGraph":
        return PyGGraph(
            self.x.to(device),
            edge_index=[edge_index.to(device) for edge_index in self.edge_index],
        )
```

Next, we have a function that takes as input a WLPlan graph, and a WLPlan graph generator that holds useful information about the underlying graph representation such as the maximum number of node and edge features, and outputs a PyG graph.

```
def wlplan_graph_to_pyg(graph_generator: GraphGenerator, graph: Graph) -> PyGGraph:
    nodes = graph.node_colours
    edges = graph.edges

    # x is (n, d)
    x = torch.zeros((len(nodes), graph_generator.get_n_features()))
    x[torch.arange(len(nodes)), nodes] = 1

    # edge_index is list of (2, e)
    edge_index = [[[], []] for _ in range(graph_generator.get_n_relations())]
    for u, neighbours in enumerate(edges):
        for v, w in neighbours:
            edge_index[w][0].append(u)
            edge_index[w][1].append(v)
    edge_index = [torch.tensor(edges, dtype=torch.long) for edges in edge_index]

    return PyGGraph(x=x, edge_index=edge_index)
```

Lastly, we have a function for generating training and validation loaders for GNN training.

```
def get_data_loaders(
    domain_dataset: DomainDataset,
    labels: Any,
    graph_generator: GraphGenerator,
    batch_size: int = 128,
) -> tuple[DataLoader, DataLoader]:

    graphs = graph_generator.to_graphs(domain_dataset)

    pyg_dataset = []
    for graph, y in zip(graphs, labels):
        pyg_graph = wlplan_graph_to_pyg(graph_generator=graph_generator, graph=graph)

        # `edge_index` variable must be used for combining edge index lists [(2, e)]
        data = Data(x=pyg_graph.x, edge_index=pyg_graph.edge_index, y=y)
        pyg_dataset.append(data)

    train_set, val_set = train_test_split(pyg_dataset, test_size=0.10, random_state=4950)
    print(f"|train_set|={len(train_set)}")
    print(f"|val_set|={len(val_set)}")

    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=False)

    return train_loader, val_loader
```



Graph Neural Networks

Next, we write some code to implement a Relational Graph Convolution Network as they support graphs with edge labels, as is the case with our WLPlan graphs. A minor exception is that we use a `max` aggregator instead of a `mean` aggregator for combining neighbouring node features.

```
class RGNNLayer(Module):
    def __init__(self, n_relations: int, in_feat: int, out_feat: int) -> None:
        super(RGNNLayer, self).__init__()
        self.convs = torch.nn.ModuleList()
        for _ in range(n_relations):
            self.convs.append(Linear(in_feat, out_feat))
        self.root = Linear(in_feat, out_feat, bias=True)

    def forward(self, x: Tensor, edge_indices_list: List[Tensor]) -> Tensor:
        x_out = self.root(x)
        for i, conv in enumerate(self.convs):
            x_out += conv(x, edge_indices_list[i])
        return x_out

class LinearConv(MessagePassing):
    def __init__(self, in_feat: int, out_feat: int, aggr: str = "max") -> None:
        super().__init__(aggr=aggr)
        self.f = Linear(in_feat, out_feat, bias=False)

    def forward(self, x: Tensor, edge_index: Tensor) -> Tensor:
        x = self.f(x)
        x = self.propagate(edge_index=edge_index, x=x, size=None)
        return x

class RGNN(nn.Module):
    def __init__(
        self,
        n_relations: int,
        in_feat: int,
        out_feat: int,
        n_hid: int = 64,
        n_layers: int = 4,
    ) -> None:
        super().__init__()
        self.n_relations = n_relations
        self.in_feat = in_feat
        self.out_feat = out_feat
        self.n_hid = n_hid
        self.n_layers = n_layers

        self.emb = torch.nn.Linear(self.in_feat, self.n_hid)
        self.layers = torch.nn.ModuleList()
        for _ in range(self.n_layers):
            layer = RGNNLayer(
                self.n_relations, self.n_hid, self.n_hid
            )
            self.layers.append(layer)
        self.mlp = Sequential(
            Linear(self.n_hid, self.n_hid), ReLU(), Linear(self.n_hid, self.out_feat)
        )

    @property
    def num_parameters(self) -> int:
        return sum(p.numel() for p in self.parameters() if p.requires_grad)

    def forward(
        self, x: Tensor, edge_indices_list: List[Tensor], batch: Optional[Tensor]
    ) -> Tensor:
        # Embed nodes
        x = self.emb(x)
        for layer in self.layers:
            x = layer(x, edge_indices_list)
            x = F.relu(x)

        # Readout layer
        x = global_add_pool(x, batch)
        h = self.mlp(x)
        h = h.squeeze(1)

        return h
```

Training Pipeline

Now with the data preprocessing functions implemented, we get to the main part of the tutorial training a GNN. We begin by calling our preprocessing functions and initialising the model and optimiser.

```
# Process dataset
graph_generator = init_graph_generator(graph_representation="1lg", domain=domain)
train_loader, val_loader = get_data_loaders(
    domain_dataset=dataset, labels=y, graph_generator=graph_generator
)

# Initialise model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = RGNN(
    n_relations=graph_generator.get_n_relations(),
    in_feat=graph_generator.get_n_features(),
    out_feat=1,  # include or are preexisting heuristics
)
model = model.to(device)
print(f"|model.num_parameters|={model.num_parameters}")

# Initialise optimiser
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode="min", factor=0.1, patience=10
)
```

Now we implement our main training loop by minimising mean squared error with gradient descent, and using a scheduler to decrement our learning rate and stop training early when performance on the validation set plateaus.

```
criterion = MSELoss()
epochs = 2024
for epoch in range(epochs):
    t = time.time()

    # Train step
    model.train()
    train_loss = 0
    for data in train_loader:
        data = data.to(device)
        y_true = data.y.float().to(device)
        optimizer.zero_grad(set_to_none=True)
        y_pred = model.forward(x=data.x, edge_indices_list=data.edge_index, batch=data.batch)
        loss = criterion.forward(y_pred, y_true)
        loss.backward()
        optimizer.step()
        train_loss += loss.detach().cpu().item()
    train_loss /= len(train_loader)

    # Validation step
    model.eval()
    val_loss = 0
    for data in val_loader:
        data = data.to(device)
        h_true = data.y.float().to(device)
        h_pred = model.forward(x=data.x, edge_indices_list=data.edge_index, batch=data.batch)
        loss = criterion.forward(h_pred, h_true)
        val_loss += loss.detach().cpu().item()
    val_loss /= len(val_loader)
    scheduler.step(val_loss)

    t = time.time() - t
    lr = optimizer.param_groups[0]["lr"]

    print(
        ", ".join(
            [
                f"epoch={i}",
                f"|lr|={lr}",
                f"|train_loss|={train_loss}",
                f"|val_loss|={val_loss}",
                f"|lr|={t:.1e}",
            ]
        )
    )

    if lr < 1e-6:
        print("Early stopping due to small |lr|.")
        break
```

Depending on the seed, the model should have converged with a training and validation loss less than 1. This is not bad at all.

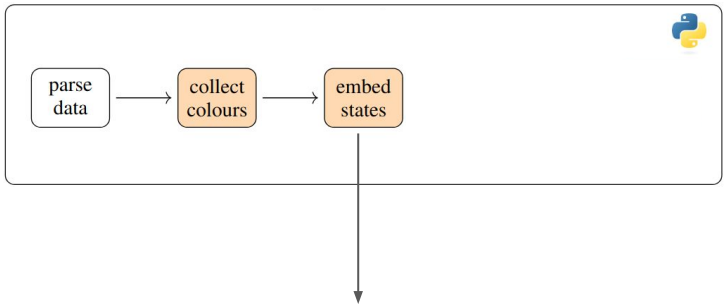
However, recall how much code was required to implement, the time to train, and the resulting mean squared error loss of using a classical ML model in the introductory tutorial. It seems that training an ML model is much more efficient and effective than a GNN for learning heuristic functions for PDDL planning.

The paper Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning shows that it is also the case that training a classical ML model with WLPlan features also leads to much more efficient planning performance compared to GNNs!

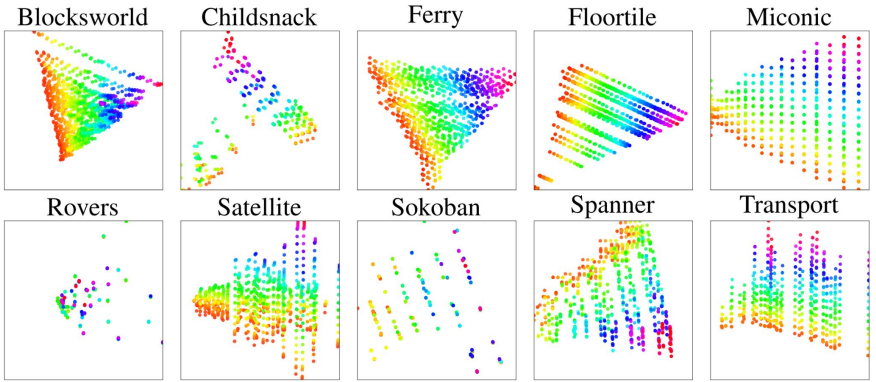
Previous

Next

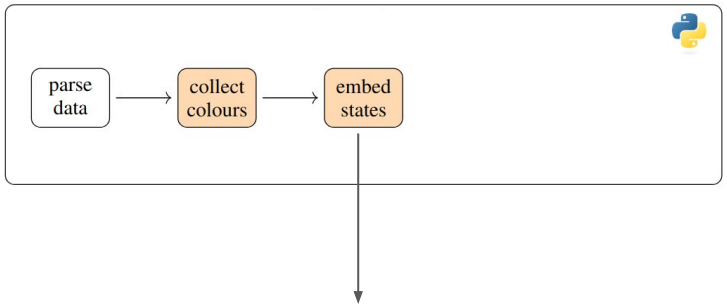
Visualising Planning Domains



perform dimensionality Reduction (e.g. PCA, t-SNE)



Expressivity Testing



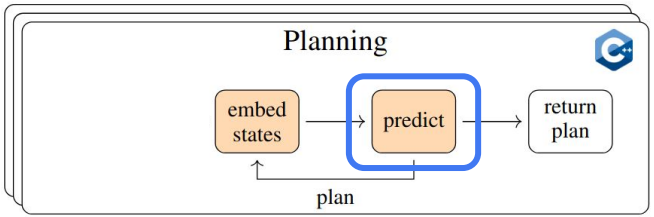
Symmetries and Expressive Requirements for Learning General Policies

Dominik Drexler[1] , **Simon Ståhlberg**[2] , **Blai Bonet**[3] , **Hector Geffner**[2]
[1]Linköping University, Sweden
[2]RWTH Aachen University, Germany
[3]Universitat Pompeu Fabra, Spain
dominik.drexler@liu.se, simon.stahlberg@gmail.com,
bonetblai@gmail.com, hector.geffner@ml.rwth-aachen.de

	Blocksworld	Childsnack	Ferry	Floortile	Miconic	Rovers	Satellite	Sokoban	Spanner	Transport
WL	0	0	0	22	0	0	0	129	68	0
2-LWL	0	0	0	–	0	–	–	–	14	0
iWL	0	0	0	22	0	–	0	110	68	0

Table 1: Number of pairs of indistinguishable states encoded as ILGs with respect to h^* values with WL algorithms on IPC23LT domains. – indicates that the memory limit was reached while embedding the entire dataset with the corresponding WL algorithm. Lower (↓) values are better.

Synthesising Novelty Heuristics



$$h^{\mathcal{F}}_{\text{pn}}(s) = \min_{\sigma \subseteq \mathcal{F}(s), \text{s.t.} \nexists t \in C, \sigma \subseteq \mathcal{F}(t)} |\sigma|$$

Lipovetzky and Geffner 2017 for $\mathcal{F}$(s) = id(s)

$$h^{\mathcal{F}}_{\text{qb}}(s) = \begin{cases} -|\mathcal{F}_{\text{new}}(s)|, & \text{if } |\mathcal{F}_{\text{new}}(s)| > 0, \\ +|\mathcal{F}_{\text{old}}(s)|, & \text{otherwise,} \end{cases}$$

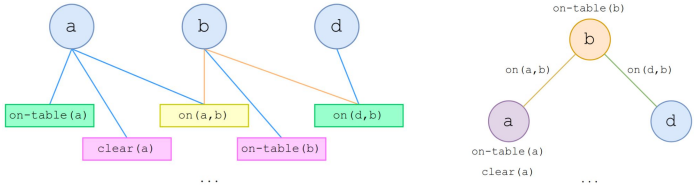
$$\mathcal{F}_{\alpha}(s) = \{p \in \mathcal{F}(s) \mid h(s) \triangle \min_{t \in C, p \in \mathcal{F}(t)} h(t)\}$$

Katz et al. 2017 for $\mathcal{F}$(s) = id(s)

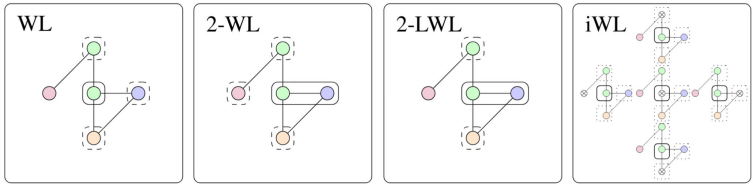
	at	wl	at;wl	at‖wl
qb	508	462	527	543
pn	612	556	599	**644**

WLPlan

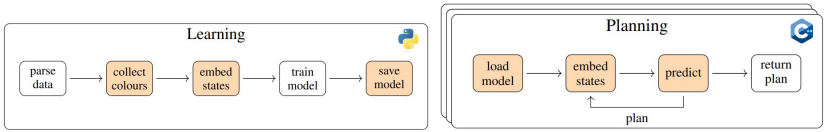
1. Graph representations of planning tasks

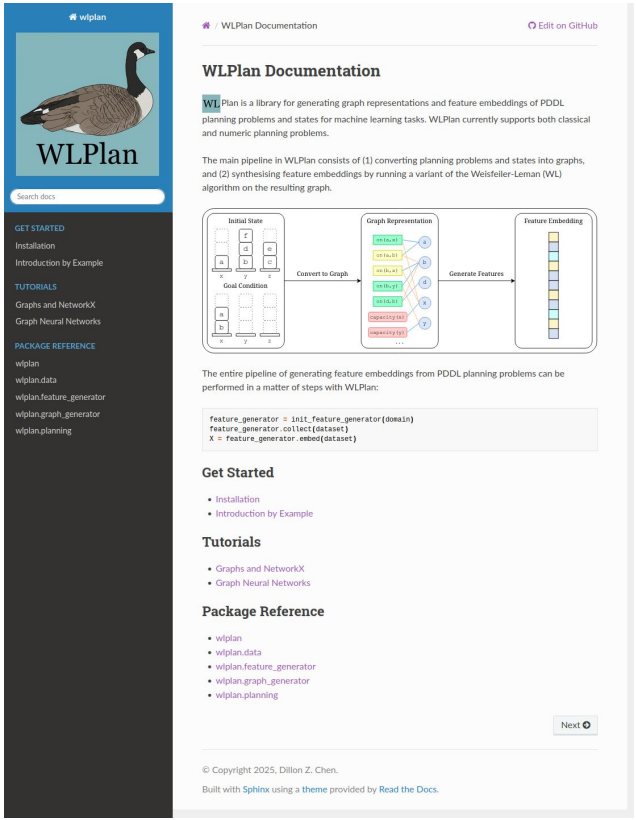


2. Embeddings of planning tasks and graphs



3. Serialisation of models





https://github.com/dillonzchen/wlplan

Implementing a L4P Pipeline from Scratch

1. Parsing training data
2. Manipulating training data
3. Building a model
4. Training a model
5. Evaluating a model

https://github.com/l4p-tutorial/heuristic-learning