16–782: Planning and Decision Making in Robotics (Fall 2025)

Homework 3: Domain Independent Symbolic Planner

Professor: Maxim Likhachev TAs: Alvin Zou, Barath Satheeshkumar

Due: November 11, 2025, 11:59pm

Henry Kou (AndrewID: hkou)

1 Introduction

In this assignment, we implement a **domain-independent symbolic planner** operating over the STRIPS representation. The environment is provided as an `Env` object, containing:

- Initial conditions
- Goal conditions
- Actions (with preconditions and effects)
- Symbols

The planner must compute a sequence of `GroundedAction`s that transforms the initial state into the goal state. We evaluate performance on three provided environments:

1. Blocks World
2. Blocks + Triangles
3. Fire Extinguisher

We compare uninformed search against heuristic-informed search.

2 Results

We evaluate all three environments with Dijkstra (mode = 0) and three A* heuristics: (1) goal-miss count, (2) delete-list penalty, and (3) combined.

2.1 Blocks World

Table 1: Blocks World: performance by mode (speedup vs. mode 0).

Mode	Heuristic	Plan Length	Expanded	Time	Speedup (Expanded)	Speedup (Time)
0	Dijkstra (no h)	3	11	17 ms	–	–
1	Goal-miss count	3	5	11 ms	2.20×	1.55×
2	Empty-Delete-List	3	14	19 ms	0.79×	0.89×
3	Combined (1)+(2)	3	5	**4 ms**	**2.20**×	**4.25**×

Plan (all modes produce length 3):

```
1. MoveToTable(A, B)
2. Move(C, Table, A)
3. Move(B, Table, C)
```

2.2 Blocks + Triangles

Table 2: Blocks + Triangles: performance by mode (speedup vs. mode 0).

Mode	Heuristic	Plan Length	Expanded	Time	Speedup (Expanded)	Speedup (Time)
0	Dijkstra (no h)	6	870	10 s	–	–
1	Goal-miss count	6	74	536 ms	11.76×	18.66×
2	Empty-Delete-List	6	385	3 s	2.26×	3.33×
3	Combined (1)+(2)	6	88	682 ms	9.89×	14.66×

Plan (length 6):

```
1. MoveToTable(T0, B0)
2. MoveToTable(B0, B1)
3. MoveToTable(T1, B3)
4. Move(B1, B4, B3)
5. Move(B0, Table, B1)
6. Move(T1, Table, B0)
```

2.3 Fire Extinguisher Domain

Plan (length 21):

Table 3: Fire Extinguisher: performance by mode (speedup vs. mode 0).

Mode	Heuristic	Plan Length	Expanded	Time	Speedup (Expanded)	Speedup (Time)
0	Dijkstra (no h)	21	364	1 s	–	–
1	Goal-miss count	21	340	1 s	1.07×	1.00×
2	Empty-Delete-List	21	397	1 s	0.92×	1.00×
3	Combined (1)+(2)	21	386	1 s	0.94×	1.00×

```
1.  MoveToLoc(A, B)
2.  LandOnRob(B)
3.  MoveTogether(B, W)
4.  FillWater(Q)
5.  MoveTogether(W, F)
6.  TakeOffFromRob(F)
7.  PourOnce(F)
8.  LandOnRob(F)
9.  MoveTogether(F, W)
10. Charge(Q)
11. FillWater(Q)
12. MoveTogether(W, F)
13. TakeOffFromRob(F)
14. PourTwice(F)
15. LandOnRob(F)
16. MoveTogether(F, W)
17. FillWater(Q)
18. Charge(Q)
19. MoveTogether(W, F)
20. TakeOffFromRob(F)
21. PourThrice(F)
```

3 Heuristic vs. Uninformed Search: Discussion

Across all three domains, heuristic search consistently reduces the number of expanded states compared to uninformed Dijkstra search, often by a dramatic margin. The magnitude of the improvement, however, depends strongly on the structure of the domain.

When heuristics help. In the two block-stacking domains (Blocks World and Blocks+Triangles), the goal configuration imposes a strict relational structure among objects (`On(A,B)`, `Clear(x)`, etc.). Uninformed Dijkstra search explores large portions of the search space before encountering states that make progress toward these relations. In contrast, the goal-miss heuristic (mode 1) directly measures how many goal facts remain unsatisfied, providing a strong gradient toward useful states. As a result:

- In **Blocks World**, expansions drop from 11 to 5 (2.2× improvement), and runtime improves from 17 ms to 4 ms.
- In **Blocks+Triangles**, the improvement is even more dramatic: expansions decrease from 870 to 74 (11.76×), and runtime improves from 10 s to under 0.6 s (18.66×).

This dramatic reduction occurs because the heuristic prunes states that are incompatible with the final goal state, focusing search on action sequences that manipulate the correct subset of objects in the correct order.

When heuristics offer limited benefit. In some domains, the search space is dominated by long, unavoidable action sequences, leaving little room for heuristic pruning. The Fire Extinguisher domain is an example: regardless of the search strategy, the agent must repeatedly perform the same mandatory loop of operations (move, land, fill, charge, take off, pour) to extinguish the fire. Because nearly all intermediate states lie on this single narrow path, even a goal-directed heuristic cannot distinguish "promising" from "unpromising" states. Consequently, heuristic search yields only a minor reduction in expansions ($364 \rightarrow 340$) and no runtime improvement, illustrating that heuristics provide limited benefit when domain structure tightly constrains the valid plan trajectory.

Takeaway: Heuristics provide the biggest benefit when they capture structure that the search would otherwise rediscover by brute force.

4 Planner Design

4.1 State, Action, and Environment Representation

Facts / State. A world state is a set of grounded facts:

$$S \subseteq \mathcal{G} \equiv \{\texttt{GroundedCondition}(\texttt{predicate}, \texttt{arg\_values}, \texttt{truth})\}.$$

In code, a state is a `std::unordered_set<GroundedCondition,...>` held in the `Node::conditions` field.

Action Schemas. Each action schema `Action` stores (i) a name, (ii) formal parameters `args`, and (iii) sets of preconditions and effects as `Condition` objects (which may be negated). Actions are *grounded* by substituting formal parameters with concrete symbols (Section 4.3). The planner never hard-codes domain knowledge; all logic is schema-driven.

Environment. `Env` stores: the symbol universe, the initial and goal grounded facts, and the action schemas. The parser (`create_env`) constructs `Env` from the domain file using regex-based extraction of symbols, conditions, and action definitions.

4.2 Search Graph and Node Structure

We perform forward search over the implicit graph whose nodes are world states S and whose edges are applicable grounded actions a:

$$S' = \gamma(S, a) \quad \text{(apply STRIPS add/delete effects)}.$$

Each search node is:

$$\texttt{Node} = \langle S,\ g,\ h,\ f = g{+}h,\ \texttt{parent},\ \texttt{action} \rangle.$$

Here, `action` is the grounded action that produced this node from `parent`. Plans are reconstructed by backtracking parents (Algorithm 2).

4.3 Grounding Strategy

For each action schema with arity k, we precompute all ordered k-tuples of distinct symbols. Concretely:

1. Generate all k-combinations of the symbol set.
2. For each combination, generate all $k!$ permutations.

This yields a reusable cache `arity` $\mapsto$ {tuples} (`generateGroundingMap`). For a tuple $\vec{x}$, we build a *grounded* action by substituting formals with $\vec{x}$ in both preconditions and effects (`updateActionWithSymbols`, which uses `update_preconditions_args` and `update_effects_args`).

Applicability Test. `ArePrecondSatisfied` converts each (possibly negated) grounded precondition into a `GroundedCondition` and checks set membership in the current state's fact set.

Transition Function. `applyEffect` realizes the STRIPS update:

$$S' \;=\; (S \setminus \text{Delete}) \cup \text{Add}.$$

4.4 Search Algorithm

We run best-first graph search (A*) with a min-heap OPEN ordered by $f{=}g{+}h$. The start node packs the parsed initial facts and $g{=}0$. The goal test checks that all grounded goal facts are present in the current state's fact set (conjunctive goal).

Algorithm 1 A* for Domain-Independent STRIPS Planning

Input: `Env` with Σ (symbols), S_0 (initial facts), G (goal facts), action schemas $\mathcal{A}$
OPEN $\leftarrow$ min-heap by f; CLOSED $\leftarrow \emptyset$; cache $\mathcal{T} \leftarrow$ `generateGroundingMap`$(\mathcal{A}, \Sigma)$
push $\langle S_0, g{=}0, h{=}0, f{=}0, \bot, \bot\rangle$ to OPEN
while OPEN not empty **do**
    $n \leftarrow$ pop-min(OPEN)
    if $G \subseteq n.S$ **then return** `backTrack`(n)
    end if
    for schema $A \in \mathcal{A}$, tuple $\vec{x} \in \mathcal{T}[\text{arity}(A)]$ **do**
        $a \leftarrow$ `updateActionWithSymbols`$(A, \vec{x})$
        if `ArePrecondSatisfied`$(a, n.S)$ **then**
            $S' \leftarrow$ `applyEffect`$(a, n.S)$
            if $S' \notin$ CLOSED **then**
                $g' \leftarrow n.g + 1$; $h' \leftarrow \hat{h}(S', G, a)$; $f' \leftarrow g' + h'$
                if $S' \in$ OPEN **then** `UpdateOpenList` to better path
                elsepush new node
                end if
            end if

            insert n into CLOSED

            return `failure`

4.5 Heuristics

We support multiple heuristic modes via `calcHeuristic`:

$$\begin{aligned}
\text{Mode 0 (Dijkstra):} \quad & h(S) = 0. \\
\text{Mode 1 (Goal-miss count):} \quad & h(S) = |\{g \in G \mid g \notin S\}|. \\
\text{Mode 2 (Delete-penalty):} \quad & h(S) = \sum_{e \in \text{effects}(a)} \mathbf{1}[\neg e] \cdot c_{\text{del}}, \text{ with } c_{\text{del}}{=}2. \\
\text{Mode 3 (Combined):} \quad & h(S) = \text{Mode1}(S) + \text{Mode2}(S, a).
\end{aligned}$$

Mode 1 is an admissible relaxation akin to ignoring interactions between goals. *Mode 2* approximates the empty-delete-list heuristic flavor by discouraging actions that delete facts (a proxy for regression difficulty). The mode is selected by a single `constexpr int mode` switch in the planner loop.

4.6 Backtracking and Output

Algorithm 2 Plan Reconstruction (`backTrack`)

Input: goal node n
$A \leftarrow$ empty list
for $t \leftarrow n$; $t \neq \perp$; $t \leftarrow t$.`parent` **do**
  if t.`parent` $\neq \perp$ **then**
    prepend `GroundedAction`(t.`action.name`, t.`action.args`) to A
  end if
end for
return A =0

5 Compilation and Running

Build (CMake).

```
mkdir build
cd build
cmake ..
cmake --build . --config Release
```

Or with g++:

```
g++ ../src/planner.cpp -o planner -std=c++11
```

Run.

```
./planner [envName]
# examples:
./planner Blocks.txt
./planner BlocksTriangle.txt
./planner FireExtinguisher.txt
```

6 Selecting the Heuristic Mode

The planner supports four heuristic modes, corresponding to the `calcHeuristic` function:

Mode	Description
0	Dijkstra / Uninformed Search ($h = 0$)
1	Goal-Miss Count
2	Empty-Delete-List Penalty
3	Combined (Goal-Miss + Delete Penalty)

How to Change the Heuristic

To change which heuristic the planner uses, modify the following line inside `planner.cpp`:

```
constexpr int mode = 0;   // choose heuristic mode here
```

Set the value to one of {0, 1, 2, 3} depending on which heuristic you want to evaluate. Then rebuild.