Prof. Alexander Bochman

Moshe Gotam

Artificial Intelligence

Holon Institute of Technology, Computer Science

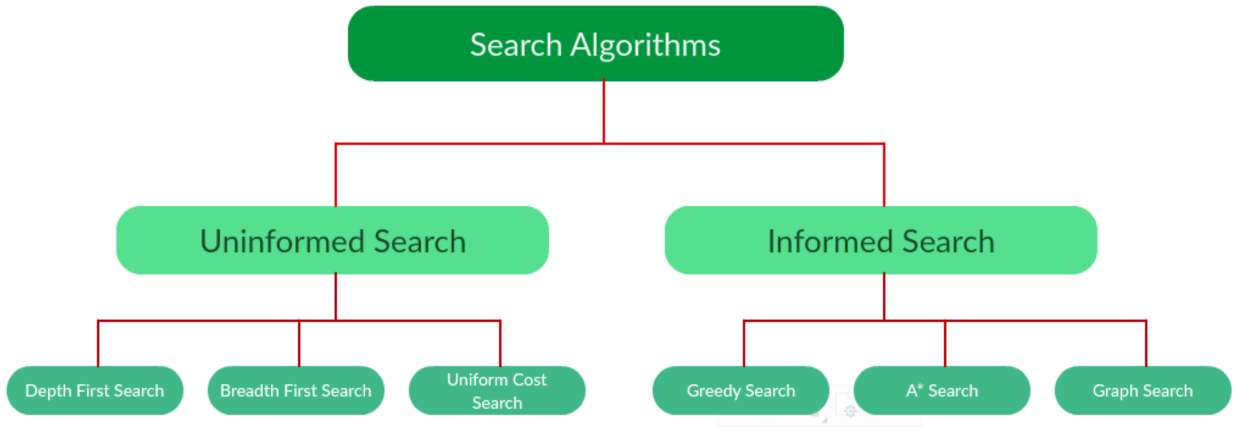
Table Of Contents

Part 1: Algorithm Background

1.1 Search Algorithms in AI

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

There are far too many powerful search algorithms out there to fit in a single article. Instead, this article will discuss six of the fundamental search algorithms, divided into two categories, Informed Search and Uniformed Search.



1.2 Search-Based Problem-Solving Agent

Goal-based agent, decide what to do by finding sequences of actions that lead to desirable states. The process of looking for such a sequence of actions is called **search**. A **problem** can be defined components – initial state, actions, transition model, goal test and path cost.

Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal.

A **state-space problem** consists of

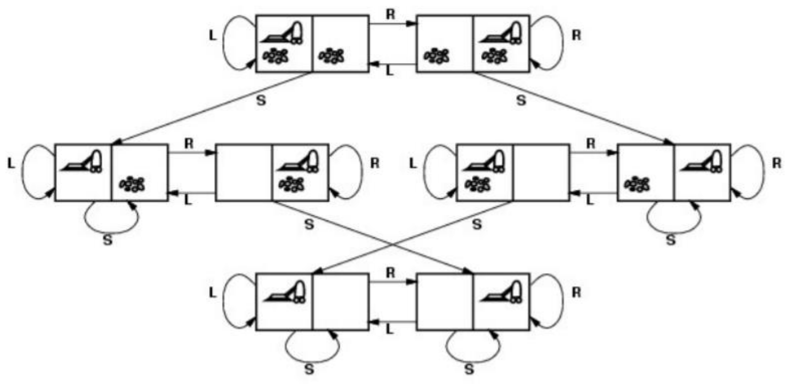
- a set of states
- a distinguished set of states called the start states
- a set of actions available to the agent in each state
- an action function that, given a state and an action, returns a new state
- a set of goal states, often specified as a Boolean function, goal(s), that is true when s is a goal state
- a criterion that specifies the quality of an acceptable solution. For example, any sequence of actions that gets the agent to the goal state may be acceptable, or there may be costs associated with actions and the agent may be required to find a sequence that has minimal total cost. This is called an optimal solution. Alternatively, it may be satisfied with any solution that is within 10% of optimal.

<u>Example:</u> *Vacuum-Cleaner*

Initial state - Any state (8 states)

Actions - Left/Right or Suck

Transition model - Complete state space

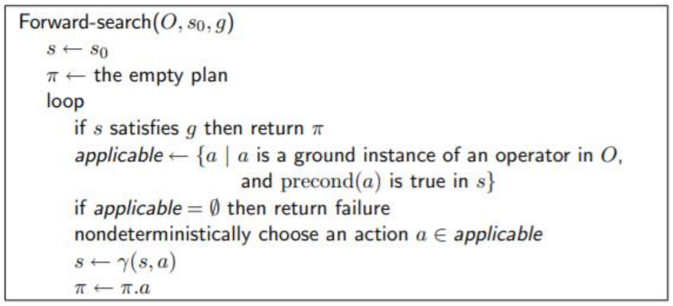


Goal test – whether both square clean

Path cost – each step costs 1

1.2.1 Forward search

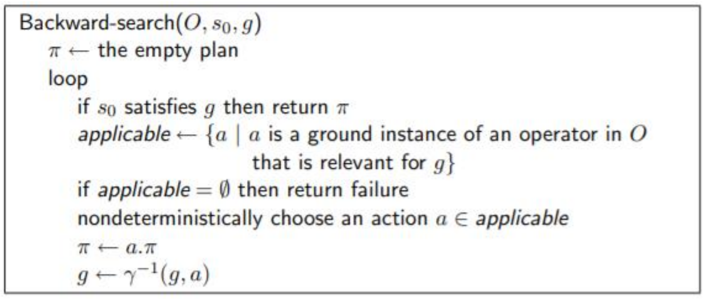
Considered one of the simplest planning algorithms. The algorithm is node deterministic, it takes the input statement of the planning problem as a P, if P is solvable then Forward-Search returns a solution plan, otherwise it returns failure.



```
Forward-search($O, s_0, g$)
    $s \leftarrow s_0$
    $\pi \leftarrow$ the empty plan
    loop
        if $s$ satisfies $g$ then return $\pi$
        applicable $\leftarrow \{a \mid a$ is a ground instance of an operator in $O$,
                        and $\text{precond}(a)$ is true in $s\}$
        if applicable $= \emptyset$ then return failure
        nondeterministically choose an action $a \in$ applicable
        $s \leftarrow \gamma(s, a)$
        $\pi \leftarrow \pi.a$
```

1.2.2 Backward search

Planning can also be done using backward search, the idea is to start at the goal, and apply actions to reach an initial state. This method is sometimes called "back propagation."



```
Backward-search($O, s_0, g$)
    $\pi \leftarrow$ the empty plan
    loop
        if $s_0$ satisfies $g$ then return $\pi$
        applicable $\leftarrow \{a \mid a$ is a ground instance of an operator in $O$
                        that is relevant for $g\}$
        if applicable $= \emptyset$ then return failure
        nondeterministically choose an action $a \in$ applicable
        $\pi \leftarrow a.\pi$
        $g \leftarrow \gamma^{-1}(g, a)$
```

The main advantage is that it allows us to consider only relevant actions, because it's starts from the goal the actions that he can make are one that one of the conjuncts of the goal.

1.2.3 Graph Search

Trees are directed graphs without cycles and with nodes have more than parent. To solve a problem, first define the underlying search space and then apply a search algorithm to that search space. Many problem-solving tasks can be transformed into the problem of finding a path in a graph. Searching in graphs provides an appropriate level of abstraction within which to study simple problem solving independent of a particular domain.

1.3 Planning

Planning can be defined as finding a sequence of actions that leads to a goal state starting from any of the initial states. **Solution** (obtained sequence of actions) is optimal if it **minimizes** sum of action costs.

Let us consider what can happen when an ordinary problem-solving agent using standard search algorithm – depth-first, A*, and so on – comes up against large, real-world problems. The most obvious difficulty is to minimize the number of irrelevant actions to 0, the search algorithm will have to examine all outcome states to find one that satisfies the goal.

The next difficulty is finding a good **heuristic function**. heuristic (from Greek εὑρίσκω "I find, discover") is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution.

1.3.1 Planner Language

Planning Domain Definition Language (PDDL) is a family of languages which allow us to define a planning problem. As planning has evolved, so too has the language used to describe it and as such there are now many versions of PDDL available with different levels of expressivity.

In artificial intelligence, action description language (ADL) is an automated planning and scheduling system in particular for robots. It is considered an advancement of STRIPS. Contrary to STRIPS, the principle of the open world applies with ADL: everything not occurring in the conditions is unknown (Instead of being assumed false). In addition, whereas in STRIPS only positive literals and conjunctions are permitted, ADL allows negative literals and disjunctions as well.

In the ADL we have to files – **Domain file**, where we write the “Black Box”, and the **Problem file**, where we set initial state and the goal for the agent.

To make it possible for a planning algorithm to take advantage of the logical structure of the problem, we have a representation of 3 functions:

1.3.1.1 Representation of states - predicate

Planner decompose the world into logical conditions and represent the state as literal, different from STRIPS, in ADL we can have positive and negative literal in state.

For Example - (have ?p - person ?m - material) – represent the state if the person has a certain material. The closed-world assumption is used here, meaning that any conditions that are not mentioned in a state are assumed false.

1.3.1.2 Representation of Goals

A goal is partially specified state. The planner will strive to get to get to that goal using the predicate we wrote earlier. As I mentioned earlier we write the goal in our Problem file.

1.3.1.3 Representation of Actions

The "executioner" of the agent. An action defines a transformation the state of the world. This transformation is typically an action which could be performed in the execution of the plan, such as picking up an object, constructing something or some other change.

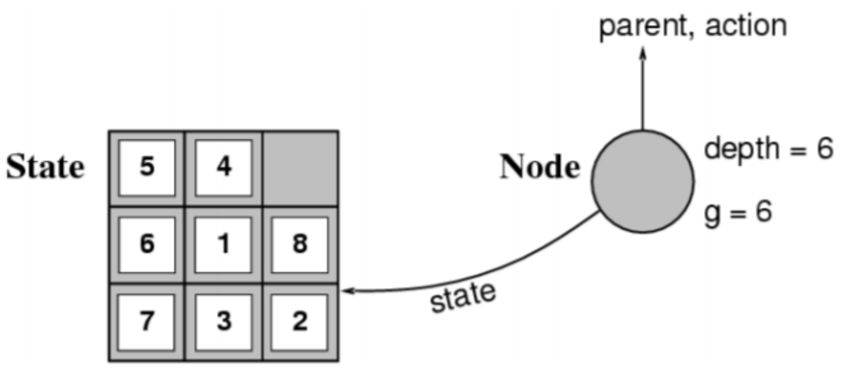
Actions consist 3 parts:

- The action name and parameter list
- The preconditions – what must be true or false before the action can be executed
- The effect – describe what states the action change

1.3.2 States versus Nodes

A **State** is a representation of a physical configuration, and a **Node** is a data structure constituting part of a search tree that include state, parent node, action, path cost and depth.

Example:



1.4 Planner Limitations

To implement ADL, I will use Visual Studio Code (VSCode) with PDDL extension. The program uses a web application - Solver.Planning.Domains - The solver component of planning.domains is an automated planner and validator in the cloud. You can invoke the software either by sending links to the PDDL files or sending raw PDDL content in JSON format directly to retrieve or validate a plan.

The planner has limit of number of types defined, after 20 types the planner wasn't able to read the domain file, also the planner has 10 seconds limit to plan a solution for the problem file.

Part 2: Introducing the Problem

2.1 Defining the problem

The problem that the agent will try to solve is to build houses under all the restriction. He will also need to know to solve problem if certain type of house or rooms need to build. To get the material, he will have to buy them from 4 different shops, which each provide other materiel.

2.1.1 Type of shops

We have 4 different shops in the planner, in each you can buy different material:

	Martial you can buy															
Shop Name	Br	Fl	Wi	Bal	Cab	Fe	Cl	Be	Co	La	To	Bat	TV	Car	Pl	La
Workshop	V	V	V	V	V	V										
Ikea							V	V	V	V	V	V				
Walmart													V	V		
Nursery															V	V

2.1.2 House Fundamental

	Martial you can buy															
Shop Name	Br	Fl	Wi	Bal	Cab	Fe	Cl	Be	Co	La	To	Bat	TV	Car	Pl	La
Walls	V															
Floor	V	V														
Window			V													
Ceiling	V															

2.1.3 Type of Houses

There are 3 types of houses –

- "Regular" house
- Garden house, which has a garden that contain lawn & plants
- Penthouse, which must have all rooms type (bath, living and bed) and a balcony

2.2 The Domain

Here we define the objects, predicates and actions

2.2.1 Initial Requirements

The planner will use ADL, ":adl" is a super requirement which adds the following requirements:

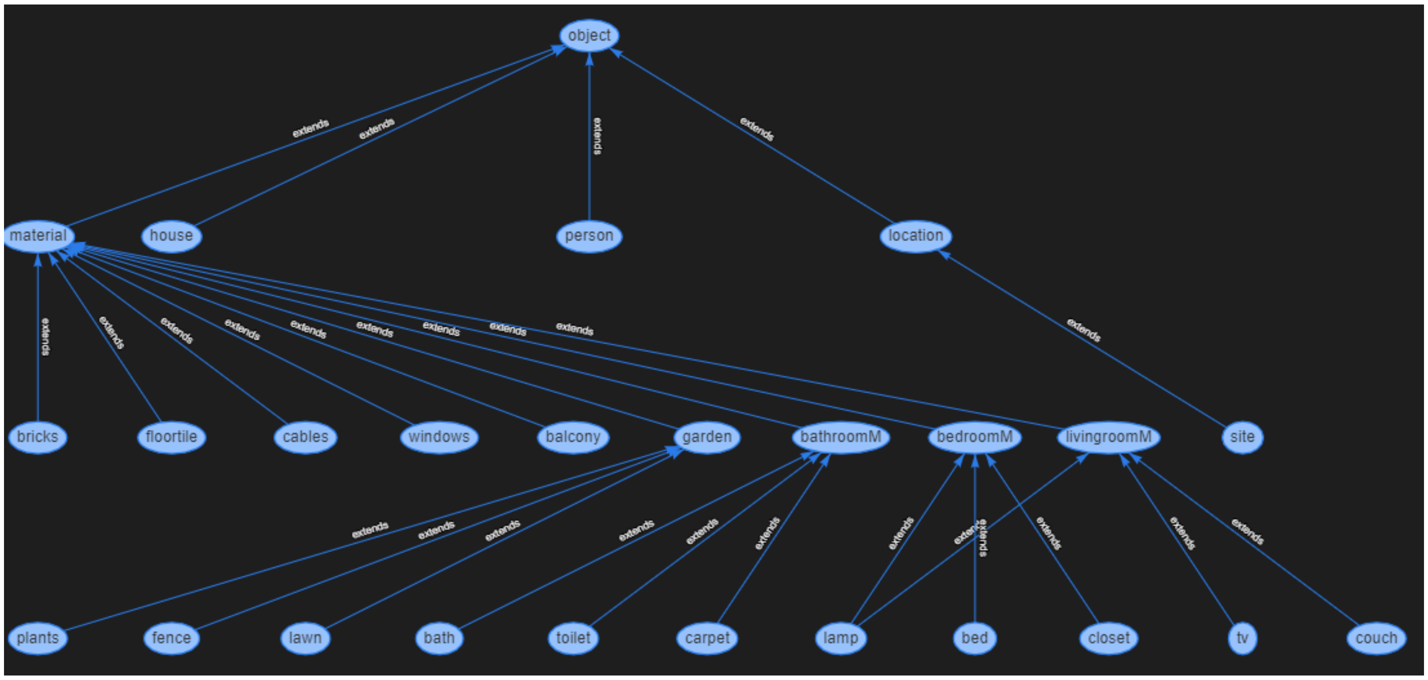
- STRIPS - Allows the usage of basic add and delete effects as specified in STRIPS. e.g.
- Typing - Allows the usage of typing for objects. Typing is similar to classes and sub-classes in Object-Oriented Programming
- Disjunctive Preconditions - Allows the usage of **or** in goals and preconditions
- Negative Preconditions - Allows the usage of **not** in goals and preconditions (which is missing in STRIPS)

This requirement also adds equality, quantified-preconditions and condition-effects which we will not use.

2.2.2 Objects

The objects in the plan can be split to 4 main objects – person, location, house and material.

- material house person - **object**
- bricks floortile cables windows balcony garden bathroomM bedroomM livingroomM - **material**
- plants fence lawn - **garden**
- bath toilet carpet - **bathroomM**
- lamp bed closet - **bedroomM**
- tv couch lamp - **livingroomM**
- **site**



2.2.3 Predicates

Predicates apply to a specific type of object, or to all objects. Predicates are either true or false at any point in a plan and when not declared are assumed to be false (except when the Open World Assumption is included as a requirement).

The "meaning" of a predicate, in the sense of for what combinations of arguments it can be true and its relationship to other predicates, is determined by the effects that actions in the domain can have on the predicate, and by what instances of the predicate are listed as true in the initial state of the problem definition.

Main Purpose	**Predicate (Parameters)**
	walls-build (?h – house)
	windows-fitted (?h - house)
	floor-built (?h - house)
Build House	ceiling-built (?h - house)
	have-fundamental (?h - house)
	have-balcony (?h – house)
	house-build (?h - house)

Material	have (?p - person ?m - material)
Garden	have-plants (?h - house)
	have-fence (?h - house)
	have-lawn (?h - house)
	have-garden (?h - house)
Site	foundations-set (?s - site)
	build-building (?s - site)
	finish-building (?s - site)
	cable-building (?s - site)
Location	at-workshop (?p - person)
	at-wallmart (?p - person)
	at-site (?p - person)
	at-ikea (?p - person)
	at-nursery (?p - person)
Rooms	have-living-room (?h - house)
	have-bath-room (?h - house)
	have-bed-room (?h - house)
	have-rooms (?h - house)
House Type	is-garden-house (?h - house)
	is-penthouse (?h - house)
	is-regular-house (?h - house)
Location	at-workshop (?p - person)
	at-wallmart (?p - person)
	at-site (?p - person)
	at-ikea (?p - person)
	at-nursery (?p - person)

2.2.4 Actions

An action defines a transformation the state of the world. This transformation is typically an action which could be performed in the execution of the plan, such as picking up an object, constructing something or some other change.

2.2.4.1 Go to locations

Each location has different actions –

- Go-to-workshop
- Go-to-IKEA
- Go-to-Wallmart
- Go-to-Nursery

* I didn't had object for locations because there were to many objects for the planner, so I decided to minimize the locations types to predicates

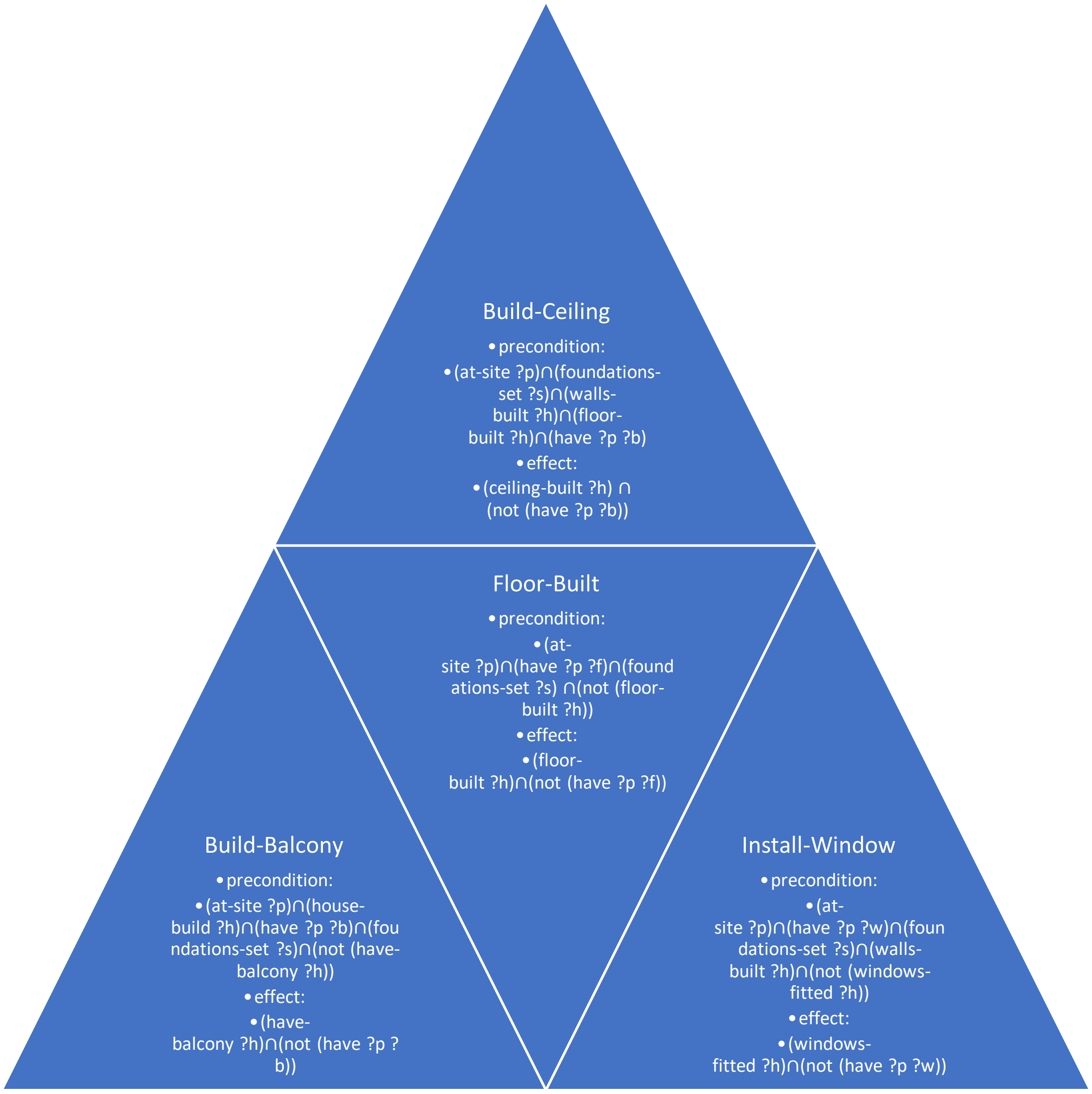
2.2.4.2 Buy Material

In each shop you can buy different material (2.2.1), these actions are critical for the planner, without the material the fundamentals in the houses cannot be build. One of the goals of the planner is to minimize the amount of times we buy the material. The actions are:

- Buy-material-workshop
- Buy-material -IKEA
- Buy-material -Wallmart
- Buy-material -Nursery

2.2.4.3 Build House Fundamental

To show the best way the actions to build a house is with a pyramid.


Build-Ceiling
• precondition:
• (at-site ?p)∩(foundations-set ?s)∩(walls-built ?h)∩(floor-built ?h)∩(have ?p ?b)
• effect:
• (ceiling-built ?h) ∩ (not (have ?p ?b))
Floor-Built
• precondition:
• (at-site ?p)∩(have ?p ?f)∩(foundations-set ?s) ∩(not (floor-built ?h))
• effect:
• (floor-built ?h)∩(not (have ?p ?f))
Build-Balcony
• precondition:
• (at-site ?p)∩(house-build ?h)∩(have ?p ?b)∩(foundations-set ?s)∩(not (have-balcony ?h))
• effect:
• (have-balcony ?h)∩(not (have ?p ?b))
Install-Window
• precondition:
• (at-site ?p)∩(have ?p ?w)∩(foundations-set ?s)∩(walls-built ?h)∩(not (windows-fitted ?h))
• effect:
• (windows-fitted ?h)∩(not (have ?p ?w))

To check the house, have the foundations we have the action **Check-House-Fundamental**:

Parameters - ?s -site ?h - house ?p - person

Preconditions:

(at-site ?p) ∩ (foundations-set ?s) ∩ (walls-built ?h) ∩ (windows-fitted ?h) ∩ (floor-built ?h) ∩ (ceiling-built ?h)

- ✓ The person needs to be in the site to check the house
- ✓ The site needs to have foundations set

- ✓ The house has to have – walls, windows, floor and ceiling

Effect - (have-fundamental ?h)

2.2.4.4 Rooms

There are 3 types of different rooms – Bedroom, living room and bath room.

- BUILD-BED-ROOM

Parameters - ?s - site ?h - house ?p - person ?la - lamp ?be - bed ?cl – closet

Precondition:

(at-site ?p) ∩ (foundations-set ?s) ∩ (have-fundamental ?h) ∩ (have ?p ?la) ∩ (have ?p ?be) ∩ (have ?p ?cl)

- ✓ The person needs to be in the site to build the bed room
- ✓ The site needs to have foundations set
- ✓ The house has to have – walls, windows, floor and ceiling (fundamental)
- ✓ The person has to have the room material – lamp, bed and closet

Effect:

(have-bed-room ?h) ∩(not (have ?p ?la)) ∩ (not (have ?p ?be)) ∩ (not (have ?p ?cl))

- BUILD-LIVING-ROOM

Parameters - ?s - site ?h - house ?p - person ?la - lamp ?tv - tv ?co - couch

Precondition:

(at-site ?p) ∩ (foundations-set ?s) ∩ (have-fundamental ?h) ∩ (have ?p ?la) ∩ (have ?p ?tv) ∩ (have ?p ?co)

- ✓ The person needs to be in the site to build the living room
- ✓ The site needs to have foundations set
- ✓ The house has to have – walls, windows, floor and ceiling (fundamental)
- ✓ The person has to have the room material – lamp, tv and couch

Effect :

(have-living-room ?h) ∩ (not (have ?p ?la)) ∩ (not (have ?p ?tv)) ∩ (not (have ?p ?co))

- BUILD-BATH-ROOM

Parameters - ?s - site ?h - house ?p - person ?ba - bath ?to - toilet ?ca - carpet

Precondition:

(at-site ?p) ∩ (foundations-set ?s) ∩ (have-fundamental ?h) ∩ (have ?p ?ba) ∩ (have ?p ?to) ∩ (have ?p ?ca)

- ✓ The person needs to be in the site to build the bath room
- ✓ The site needs to have foundations set
- ✓ The house has to have – walls, windows, floor and ceiling (fundamental)
- ✓ The person has to have the room material – bath, toilet and carpet

Effect :

(have-bath-room ?h) ∩ (not (have ?p ?ba)) ∩ (not (have ?p ?to)) ∩ (not (have ?p ?ca))

2.2.4.5 Rooms Type

- BUILD-ROOMS-TYPE-A

Parameters - ?s - site ?h - house

Precondition:

(have-balcony ?h) ∩ (have-bath-room ?h) ∩ (have-bed-room ?h)

- ✓ The house has to have a balcony, bath room and bed room

Effect :

(have-rooms ?h)

- BUILD-ROOMS-TYPE-B

Parameters - ?s - site ?h - house
Precondition:
(have- living-room ?h) ∩ (have-bath-room ?h)

- ✓ The house has to have a living room and bath room

Effect :
(have-rooms ?h)

- BUILD-ROOMS-TYPE-C

Parameters - ?s - site ?h - house
Precondition:
(have- bed-room ?h) ∩ (have-bath-room ?h)

- ✓ The house has to have a bed room and bath room

Effect :
(have-rooms ?h)

2.2.4.6 Garden

- PLANT-PLANTS

Parameters - ?s - site ?h - house ?p - person ?pl – plants
Precondition:
(have ?p ?pl) ∩ (at-site ?p) ∩ (foundations-set ?s) ∩ (have-lawn ?h) ∩ (not (have-plants ?h))

- ✓ The person needs to be in the site to plant the plants
- ✓ The site needs to have foundations set
- ✓ The house has to have a lawn
- ✓ The person has to have the plants
- ✗ The house doesn't have plants

Effect:
(have-plants ?h) ∩ (not (have ?p ?pl))

- BUILD-FENCE

Parameters - ?s - site ?h - house ?p - person ?fe - fence
Precondition:
(have ?p ?fe) ∩ (at-site ?p) ∩ (foundations-set ?s) ∩ (not (have-fence ?h))

- ✓ The person needs to be in the site to build the fence
- ✓ The site needs to have foundations set
- ✓ The person has to have the fence
- ✗ The house doesn't have a fence

Effect:
(have-fence ?h) ∩ (not (have ?p ?fe))

- BUILD-LAWN

Parameters - ?s - site ?h - house ?p - person ?l - lawn
Precondition:
(have ?p ?l) ∩ (at-site ?p) ∩ (foundations-set ?s) ∩ (not (have-lawn ?h))

- ✓ The person needs to be in the site to build lawn to the house
- ✓ The site needs to have foundations set
- ✓ The person has to have the lawn

- ✗ The house doesn't have a lawn

Effect:

(have-fence ?h) ∩ (not (have ?p ?l))

- BUILD- GARDEN

Parameters - ?h - house ?pl - plants ?l - lawn ?f - fence ?p - person

Precondition:

(at-site ?p) ∩ (have-fence ?h) ∩ (have-lawn ?h) ∩ (have-plants ?h) ∩ (not (have-garden ?h))

- ✓ The person needs to be in the site to build the garden
- ✓ The house has to have – fence, lawn and plants
- ✗ The house doesn't have a garden

Effect:

(have-garden ?h)

2.2.4.7 Penthouse

- BUILD- PENTHOUSE

Parameters - ?s - site ?h - house ?p - person

Precondition:

(at-site ?p) ∩ (house-build ?h) ∩ (have-balcony ?h) ∩ (have-bath-room ?h) ∩ (have-bed-room ?h) ∩ (have-living-room ?h) ∩ (not (is-garden-house ?h)) ∩ (not (is-regular-house ?h))

- ✓ The person needs to be in the site to check if the house is a penthouse
- ✓ The house has to have – house foundations, balcony, bath room, bed room and living room
- ✗ The house isn't a garden house or regular house already

Effect:

(is-penthouse ?h) ∩ (finish-building ?s)

2.2.4.8 Site

- SET-FOUNDATIONS

Parameters - ?s - site ?p - person

Precondition:

(at-site ?p)

- ✓ The person needs to be in the site in order to set the foundations

Effect:

(foundations-set ?s)

- CABLE-BUILDING

Parameters - ?s - site ?p - person ?c - cables

Precondition:

(at-site ?p) ∩ (finish-building ?s) ∩ (have ?p ?c)

- ✓ The person needs to be in the site to put cables around the building
- ✓ All of the house in the building needs to be build
- ✓ The person has to have cables

Effect:

(cable-building ?s) ∩ (not (have ?p ?c))

- BUILD-BUILDING

Parameters - ?s - site ?h1 - house ?h2 - house ?h3 - house ?h4 – house ?p - person

Precondition:

(at-site ?p) ∩ (is-garden-house ?h1) ∩ (is-penthouse ?h2) ∩ (is-regular-house ?h3)

- ✓ The person needs to be in the site to check the building
- ✓ The house has to have three houses with 3 different types

Effect:
(have-building ?s)

2.3 The Problem

A problem forms the other half of a planning problem. In the domain we express the global “worldly” aspects of a problem, such as what actions we can perform and what types of objects exist in the world we’re planning in.

The problem then solidifies this expression by define exactly what objects exist, and what is true about them and then finally what the end goal is. What state we want the world to be in once the plan is finished.

2.3.1 Objects

The objects block allows us to declare a set of objects which exist within our problem. each object name must be unique, and should be typed. If not typed then they will typically take on the properties of the base type object.

2.2.2 Init

The initial state (init) defines specifically what predicates are true at the start of the problem. This is not a logical expression because it is simply a list of predicates which are true. Unless the planner or domain specify otherwise all problems have the “closed world” assumption applied meaning anything not specified as true is considered false. Therefore, we only need to list things which are true.

2.3.3 Goals

The goal is a logical expression of predicates which must be satisfied in order for a plan to be considered a solution. In essence it is what we want the world to look like at the end. Note that as a logical expression, if this expression excludes some predicate, then the value of that predicate is not considered important. This means that a goal should not only consist of the predicates that should be true, but also the predicate which should be false.

Part 3: Planner Implementation

This part will cover the implementation of the Domain and Problems using PDDL (Planning Domain Definition Language) which was inspired by STRIPS and ADL, also a brief explanation of the source code involving the **Planning.Domains** API (introduced in 1.2.6) using VSCode as I mention earlier.

Planning.Domains is the definitive home of Artificial Intelligence Planning. The site features a wide variety of tools and resources to help you with AI Planning. Included in its toolset is a Planning as a Service (PaaS) API. An online editor for writing and testing domains and problem files.

To show the problem and possible solution I will show problem file and the output from **Planning.Domains**.

3.1 Problems

3.1.1 One Person, 2 House (Regular & Penthouse)

Problem File

(:**objects**

moshe - person s1 - site h1 h2 - house br1 - bricks ba - balcony bath1 - bath to - toilet ca - carpet la - lamp tv1 - tv co - couch be - bed cl - closet w - windows c - cables fl - floortile pl - plants l - lawn fe - fence)

(:**init**

(at-site moshe))

(:**goal** (and

(is-regular-house h2) (is-penthouse h1)))

Solution

Nodes generated during search: 404
Nodes expanded during search: 42
Plan found with cost: 41
Planner found 1 plan(s) in 0.826secs.

3.1.2 One Person, 4 House (Regular, Penthouse and 2 Garden)

Problem File

(:**objects**

moshe - person s1 - site h1 h2 h3 h4 - house br1 - bricks ba - balcony bath1 - bath to - toilet ca - carpet la - lamp tv1 - tv co - couch be - bed cl - closet w - windows c - cables fl - floortile pl - plants l - lawn fe - fence)

(:**init**

(at-site moshe))

(:**goal** (and

(is-regular-house h2) (is-penthouse h1) (is-garden-house h3) (is-garden-house h4)))

Solution

Nodes generated during search: 20883
Nodes expanded during search: 1641
Plan found with cost: 125
Planner found 1 plan(s) in 6.089secs.

3.1.3 Two Person, 4 House (Regular, Penthouse and 2 Garden)

Problem File

(:**objects**

moshe israel - person s1 - site h1 h2 h3 h4 - house br1 - bricks ba - balcony bath1 - bath to - toilet ca - carpet la - lamp tv1 - tv co - couch be - bed cl - closet w - windows c - cables fl - floortile pl - plants l - lawn fe - fence)

(:**init**

(at-site moshe) (at-site israel))

(:**goal** (and

(is-regular-house h2) (is-penthouse h1) (is-garden-house h3) (is-garden-house h4)))

Solution

Nodes generated during search: 43734
Nodes expanded during search: 1898
Plan found with cost: 114
Planner found 1 plan(s) in 3.686secs.

3.1.4 Two Person, 5 House (Regular, 2 Penthouse and 2 Garden)

Problem File

(:**objects**

moshe israel - person s1 - site h1 h2 h3 h4 h5 - house br1 - bricks ba - balcony bath1 - bath to - toilet ca - carpet la - lamp tv1 - tv co - couch be - bed cl - closet w - windows c - cables fl - floortile pl - plants l - lawn fe - fence)

(:**init**

(at-site moshe) (at-site israel))

(:**goal** (and

(is-regular-house h2) (is-penthouse h1) (is-garden-house h3) (is-garden-house h4) (is-penthouse h5)))

Solution

Nodes generated during search: 90635
Nodes expanded during search: 2972
Plan found with cost: 138
Planner found 1 plan(s) in 6.265secs.

3.1.5 Two Person, 7 House (2 Regular, 2 Penthouse and 2 Garden)

Problem File

(:**objects**

moshe israel - person s1 - site h1 h2 h3 h4 h5 h6 - house br1 - bricks ba - balcony bath1 - bath to - toilet ca - carpet la - lamp tv1 - tv co - couch be - bed cl - closet w - windows c - cables fl - floortile pl - plants l - lawn fe - fence)

(:**init**

(at-site moshe) (at-site israel))

(:**goal** (and

(is-regular-house h2)(is-penthouse h1)(is-garden-house h3)(is-garden-house h4)(is-penthouse h5)(is-regular-house h6)))

Solution

Planner found 0 plan(s) in 10.722secs.

3.2 Conclusion

	Nodes generated during search	Nodes expanded during search	Plan found with cost	Time
One Person, 2 House	404	42	41	0.826
One Person, 4 House	20883	1641	125	6.089
Two Person, 4 House	43734	1898	114	3.686
Two Person, 5 House	90635	2972	138	6.265
Two Person, 7 House				10.722

Time

We can see the difference how much time does the planner takes to plan with one person to two people, almost half the time for the same type and number of houses.

We reached the limit for the planner when we tried to plan 7 houses with 2 people.

Plan

We have always 1 or 0 plans, that because of the heuristic of the plan. The solution is linear, which means there is always one plan to solve the problem - There is only one way build a house based on the planner.

Part 4 Bibliography

1. PDDL Planning Wiki - https://planning.wiki/
2. Artificial Intelligence: A Modern Approach (3rd Edition), Chapter 3-11 - link
3. Artificial Intelligence: Foundations of Computational Agents, 2nd Edition - link