Automated planning for strategy generation in dynamic domains

A minor thesis submitted in partial fulfilment for the
requirements for the degree of
Master of Information Technology

Ujjwal Batra
School of Computer Science and Information Technology
RMIT University
Melbourne, Victoria, Australia

November 2, 2019

Declaration

This thesis contains work that has not been submitted previously, in whole or in part, for any other academic award and is solely my original research, except where acknowledged.

This work has been carried out since February 2019, under the supervision of Supervisor(s): Sebastian Sardina and John Thangarajah.

Ujjwal Batra
School of Computer Science and Information Technology
RMIT University
November 2, 2019

Acknowledgements

I would like to take this opportunity to convey my special thanks to my supervisor Dr Sebastian Sardina for his incredible support during the past year. His support is beyond ordinary supervision.

Thanks to Dr John Thangarajah for his useful comments on my work, and for great discussion and feedback.

Abstract

Much of the Multi-Agent systems tend to rely on a programming based approach, where a programmer uses an Agent-Oriented Language to encode the strategies of the domain. Agent-Oriented languages provide higher-level abstractions than Object-Oriented Programming and handle the dynamic nature of complex multi-agent domains. However, Agent-Oriented Programming requires contemplating strategies for all the possible scenarios in the domain; which is sometimes not possible in the case of dynamic multi-agent domains due to a large number of changing details. This work will explore the use of automated planning for complex and dynamic domains, which involves generating high-level strategies that can be executed by an execution framework.

Contents

Chapter 1

Introduction

Automated planning has been of profound interest in the field of artificial intelligence. The notion of achieving a goal is considered important, and automated planning is the sub-area of artificial intelligence that is concerned with producing a procedural course of actions or a *plan* to reach a goal (Ghallab et al., 2004). Automated planning has achieved some success in practical applications such as the mars rover (Estlin et al., 2003) and the sheet-metal bending operations (Gupta et al., 1998).

The application of autonomous systems are still mostly based on programming based approaches (Belecheanu et al., 2006). The programming based approach uses predefined procedures to achieve a goal. The Programming of predefined procedures to achieve a goal requires encoding a procedure for every possible situation.

However, it is not always possible to predict all situations, especially applications involving multiple autonomous agents. Due to the complex and dynamic nature of the system, the agent might enter a state where it does not have a procedure to execute. As recently argued by Nau (2007), it is desirable for an agent to have the ability to dynamically generate a sequence of actions to work towards achieving its goal.

To address the complexities that arise due to the dynamic nature of the environment, we propose the use of automated planning to generate high-level strategies that can be executed by an execution framework. For this thesis, we will use Planning Domain Definition Language (PDDL) (Fox and Long, 2003) to encode the information of the domain Agents in the City which is a part of the International Multi-Agent Contest[1]. The encodings will then be used for the generation of a high-level plan.

This leads to the research questions that will be the focus of this work.

1.1 Research Questions

Research Question 1: How can a complex domain be modeled in PDDL to deal with the complexities of a dynamic domain?

Research Question 2: What features of PDDL can be added to the encodings to reach a high quality plan?

1.2 Contributions

This work has made the following contributions to the field of automated planning of dynamic domains:

- Encoding complex processes of the domain using scalable techniques.
- Adding multiple features of PDDL to the encodings to reach a more detailed plan.
- Comparison of plans obtained by executing PDDL encodings on different planners to check for available features.

[1] `https://multiagentcontest.org/`

Finally, an experimental evaluation and analysis of these encodings are presented.

1.3 Organization

The rest of the thesis is organised as follows:

- Chapter 2 takes a detailed look at relevant literature in the field of automated planning.
- Chapter 3 describes the details and complexities of the Agents in the City domain, and the evaluates the solution provided by the winning team.
- Chapter 4 includes details of our implementation, and an experimental evaluation of the implementation.
- Chapter 5 concludes this work.
- Chapter 6 discusses future work.

Chapter 2

Automated Planning

Automated planning is a field of artificial intelligence that is responsible for generating a sequence of actions to achieve a goal (Ghallab et al., 2004). Figure2.1 shows a planning system that produces a sequence of actions or *plan* using a model-based approach. Here, the *planner* produces a plan based on the following input:

- An *Operator* or an action describes information about the preconditions under which it is applicable and its effect on the world.
- The *initial state* is a set of first-order predicates which are initially true.
- The *goal* is a set of predicates that need to be achieved.
- *Information* is the domain knowledge that tailors a planner for a particular domain such as the mars rover (Estlin et al., 2003) or sheet-metal bending operations (Gupta et al., 1998).

Domain-independent systems are not tailored for a particular domain and do not require domain-specific *information* as an input (Nau et al., 2003). Since domain-independent planning systems are not tailored for a specific domain, they need domain information as input to search for a sequence of actions to achieve a goal.

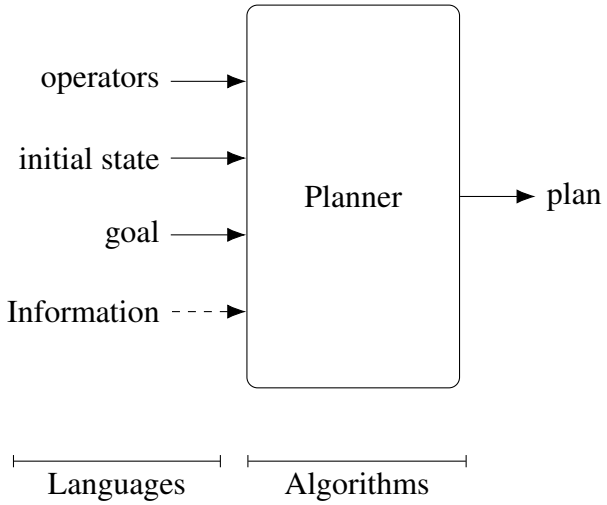

Figure 2.1: High level architecture of a planning system

The planner uses a conceptual model of a *state-transition system* (also called a discrete event system). Where a *state* is represented by a set of propositions. A state-transition system is a 3-tuple $\Sigma = (S, A, \gamma)$, where

$S = \{s_0, s_1, s_2, \dots\}$ is a finite set of states
$A = \{a_1, a_2, \dots\}$ is a set of operators
$\gamma : S \times A \to 2^S$ is a state-transition function.

If $s' \in \gamma(s, a)$, where $a \in A$ is an action, then the model contains a state transition from s to s'. Given the state-transition system Σ, the purpose of the planner is to form the conceptual model from the input, and find which actions to apply to achieve the goal state when starting from a given state. A *planning language* is used to encode the initial state, goal, and suitable actions that are possible. The encoding is then passed to the planner.

There are multiple planning languages and planners available. This work focusses on a domain-independent planning system because it can be applied to different domains with less effort, as no procedural programming or specific heuristic is required. In this chapter, we will discuss some of the most relevant languages and planners used for domain-independent planning.

2.1 Languages for Planning

A planning language allows specification of the domain by encoding suitable actions. Attempts to formalize a language for automated planning started with STRIPS (Stanford Research Institute Problem Solver), which was developed for robot research (Fikes and Nilsson, 1971). STRIPS was used as a base for the creation of more planning languages like PDDL (McDermott et al., 1998). In this section, we will describe the most common planning languages.

2.1.1 STRIPS

STRIPS is a very well known planning language, it was considered a standard in the field of automated planning. STRIPS represents the world model using restricted formulas from first-order predicate calculus. For example, to describe a world model in which a truck `truck1` is at storage `STORAGE0`, and `truck1` is carrying an item `ITEM5`, would include the following atoms:

```
AGENT-AT-LOCATION(truck1, STORAGE0)
AGENT-CARRYING-ITEM(truck1, ITEM5)
```

A problem space in STRIPS is defined using three key entities: initial state (s_0), a set of operators or actions, and a set of propositions that is needed to be achieved or *goal state*. Suppose we have a truck `truck1` in storage `STORAGE0`, and item `ITEM5` is in resource node `node1` and `truck1` has to deliver `ITEM5` to `storage1`. The initial state and goal state might be declared as follows:

$$s_0 : \left\{ \begin{array}{l} \texttt{AGENT-AT-LOCATION(truck1, STORAGE0)} \\ \texttt{ITEM-IN-NODE(ITEM5, node1)} \end{array} \right\}$$

$$s_g : \texttt{ITEM-IN-STORAGE(ITEM5, storage1)}$$

Every action includes a set of literals for specifying preconditions under which the action is applicable on, and its effect on the world. The effect of an action is mentioned using an *add list* and a *delete list*. The delete list includes the predicates that will no longer be true after the application of that action, and in the add list one specifies the predicates that will become true in the new state. For example, consider a navigational action `goto(facility1, facility2)` to model the agent from location `facility1` to `facility2`. Such an action might be modeled in STRIPS as follows:

```
goto(agent, facility1, facility2)
    Precondition:
        AGENT-AT-LOCATION(agent, facility1)
    delete list
        AGENT-AT-LOCATION(agent, facility1)
    add list
        AGENT-AT-LOCATION(agent, facility2)
```

STRIPS comes with a very intuitive syntax and a sophisticated way of modelling a planning problem. The disadvantages of using STRIPS is that only actions with very simple effects can be represented which can lead to unsound inferences in some cases (Lifschitz, 1987). This limitation is overcome by planning languages which came into existence after STRIPS.

2.1.2 Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) was an attempt to standardise planning languages. STRIPS, amongst other languages, inspires PDDL (McDermott et al., 1998).

PDDL evolved over the years, and various features were added to PDDL. The most recognised versions of PDDL are 1.0 (used in IPC 1998), 2.1 (used in IPC 2002) and 3.0 (used in IPC 2006). In this section, we will discuss the key features added to PDDL within different versions.

2.1.2.1 PDDL1.0 and PDDL1.2:

The initial versions of PDDL introduced the idea of defining the planning problem in two major parts: *domain description* and *problem description*.

Domain description A domain description specifies the domain and its dynamics, that is, how the world changes as the agent performs actions. Actions are at the core of the domain model and encode the causal laws of the domain. An action is described by its preconditions, stating when the action can be legally executed, and its effects, stating what are the changes in the world arising from its execution. For example, the navigational action `goto(facility1, facility2)` to model the agent moving from location facility1 to location facility2, can be modeled in PDDL as follows:

```
(:predicates
    (AGENT-AT-FACILITY ?a - agent ?f - facility)
)
(:action goto
    :parameters(?a - agent,
        ?f1 - facility, ?f2 - facility)
    :precondition(and
        (AGENT-AT-FACILITY ?a ?f1)
        (not (= ?f1 ?f2))
    )
    :effect(and
        (AGENT-AT-FACILITY ?a ?f2)
        (not (AGENT-AT-FACILITY ?a ?f1))
    )
)
```

In the above encoding, it can be seen that the action schema of PDDL is inspired from STRIPS. Like STRIPS, PDDL actions have an action name, parameters,

a precondition and an effect. The `goto` action can only be applied when the precondition of the action holds true. After the application of the action, the effects of the action are added to the new state. In this example, the new state will have the fact that the agent `?a` is at `?f2`, and `?a` is not at `?f1`.

Similarly, multiple actions can be applied to make the system go through a series of changes in state to reach the goal.

Problem description Problem description specifies the *initial state*, *objects* and *goal* that a planner should consider for finding a plan. Components of the problem description are described below:

- **Initial state:** Initial state in PDDL is described in the *(:init)* section of the problem description. Initial state is defined as a list of predicates that are initially true. All the other predicates are false by default. For example, we might use the following declaration for the situation where initially all trucks (`truck1`, `truck2`, `truck3`, `truck4`) are at `storage1`, and `item1` is available in node1:

```
(:init
    (AGENT-AT-FACILITY truck1 storage1)
    (AGENT-AT-FACILITY truck2 storage1)
    (AGENT-AT-FACILITY truck3 storage1)
    (AGENT-AT-FACILITY truck4 storage1)
    (ITEM-IN-NODE item1 node1)
    (ITEM-IN-NODE item2 node2)
)
```

  Here, `truck1, truck2, truck3` and `truck4` are the *objects* of type truck; `storage1`, `node1` and `node2` are types of facility; and, `item1` and `item2` are of type item.

- **Goal:** The goal description contains the predicate that must be true to achieve the goal. For example, if we want to transport `item2` to `storage2` and the `truck1` should be at storage2 then we might define the goal as:

```
(:goal (and
    (STORAGE-CONTAINS-ITEM storage2 item2)
    (AGENT-AT-FACILITY truck2 storage1))
)
```

The encoding of the domain and problem description is together used to build the model of the world by the planning algorithm to formulate a plan.

2.1.2.2 PDDL2.1: Beyond Classical Planning

PDDL version 2.1 allowed the representation and solution of many more real-world problems by introducing features such as *numeric expressions, plan metrics* and *durative actions* which allowed the planner to optimise the plan based on domain requirements.

Numeric expressions: Numeric expressions are declared using domain functions and support basic arithmetic operators. They are useful in applications where the possible roles need a numeric value such as the quantity a truck can carry or the remaining capacity of a truck. Suppose in the store action we want the agent to give only 1 unit of item to the storage, then the action might be described as:

```
; in domain description
(functions(and
    (AGENT-CARRYING-ITEM ?a - agent ?i - item)
    (AGENT-CAPACITY ?a - agent)
    (ITEM-IN-STORAGE ?i - item ?s - storage))
)
```

```
(:action store
:parameters(?a - agent, ?s - storage, ?i - item)
:precondition(and
    (>= (AGENT-CARRYING-ITEM ?a ?i) 1)
    (AGENT-AT-FACILITY ?a ?s)
)
:effect(and
    (decrease(AGENT-CARRYING-ITEM truck1 item1) 1)
    (increase(AGENT-CAPACITY truck1) 1)
    (increase(ITEM-IN-STORAGE item1 storage1) 1))
)

; in problem description
(:init
    AGENT-AT-FACILITY(truck1, storage1)
    ITEM-IN-NODE(item1, node1)
    (= (AGENT-CARRYING-ITEM truck1 item1) 0)
    (= (AGENT-CAPACITY truck1) 100)
    (= (ITEM-IN-STORAGE item1 storage1) 0)
)
```

In this scenario the precondition specifies that the agent should be carrying more than 1 unit of the item. When the action finishes the agents releases 1 unit of the item, the capacity of agent increases by 1, the quantity of item carried by the agent decreases by 1 and the quantity of item in storage increases by 1.

Plan Metrics: Until version 2.1, the plan quality was judged by the length of the plan. Version 2.1 allows us to add plan metrics to the problem description which enables the planner to optimise (minimization/maximization) the plan based on the requirements of the domain. For example, the planner might be required to

minimize the distance travelled. The metric for this scenario can be declared in the following way:

```
; in domain description
(:functions(and
    (total-cost)
    (DISTANCE ?f1 - facility ?f2 - facility))
)
(:action goto
    :parameters(?a - agent,
        ?f1 - facility, ?f2 - facility)
    :precondition(and
        (AGENT-AT-FACILITY ?a ?f1)
        (not (= ?f1 ?f2))
    )
    :effect(and
        (AGENT-AT-FACILITY ?a ?f2)
        (not (AGENT-AT-FACILITY ?a ?f1))
        (increase (total-cost) (DISTANCE ?f1 ?f2))
    )
)

; in problem description
(:objects
    truck1 - truck
    storage1 storage2 - storage
)

(:init
    AGENT-AT-FACILITY(truck1 storage1)
    ITEM-IN-NODE(item1 node1)
    ((DISTANCE storage1 storage2) 150)
```

```
        (= (total-cost) 0)
    )

    (:metric
        (minimise (total-cost))
    )
```

In this case, the optimisation works on total-cost, where the total-cost is the distance travelled by an agent. The initial values of the functions are described in `:init`, and the planner increases the total-cost by the distance every time the `goto` action is used. The optimisation criteria is mentioned in `:metric` part of the problem description, which is minimization of the total-cost for the above example.

Durative actions: The idea of durative actions is based on temporal planning (Smith and Weld, 1999; Bacchus and Kabanza, 2000; Do and Kambhampati, 2001) where the action can go on for a certain duration of time. Durative actions are more expressive than previous actions, because in real world applications performing an action will consume some time and it will have various effects at different intervals of time.

A durative action allows the modelling of conditional effects where the condition explicitly mentions whether the predicate must hold at the start of the interval, at the end of the interval or over the interval. The effects that happen at a particular interval are known as *discrete effects* and the effects that happen over a period of interval are called *continuous effects*.

For example, the `goto` action can be converted to a durative action:

```
(:durative-action goto
    :parameters(?a - agent,
        ?f1 - facility, ?f2 - facility)
    :duration(= ?duration (DISTANCE ?f1 ?f2))
```

```
    :condition(and
        (at start (AGENT-AT-FACILITY ?a ?f1))
        (at start (not (= ?f1 ?f2)))
    )
    :effect(and
        (at start (not
            (AGENT-AT-FACILITY $?a$ $?f1)$))
        (at end
            (AGENT-AT-FACILITY $?a$ $?f2))
        (at end (increase (total-cost)
                (DISTANCE $?f1$ $?f2$)))
    )
)
```

In the above goto `action`, the action takes time equal to the distance between two facilities. The preconditions are replaced by the condition which now has a time interval. In durative actions predicates in effects and conditions have extra information about the time interval. The predicate `(not (AGENT-AT-FACILITY ?a ?f1))` becomes true as soon as the agent action is invoked, to reflect the fact that the agent has left $?f1$, and when the action is ending the predicate `(AGENT-AT-FACILITY ?a ?f2)` becomes true to reflect the new location of the agent. Also, the total-cost of the action is increased at the end.

2.1.2.3 PDDL3.0: Adding Constraints

PDDL 3.0 was the official language of the 5^{th} IPC in 2006(Gerevini and Long, 2006). It increases the expressive power of PDDL and improves the plan quality by introducing the following features:

Strong constraints: Strong constraints are the conditions that must be satisfied by any valid plan. For example, for best performance of a drone, the user can have a constraint that it cannot carry more than 100 units of any item otherwise it might break.

Soft constraints and preferences Soft constraints are the conditions that are desirable outcomes that the user would prefer to see satisfied rather than not satisfied. It is not essential for a planner to satisfy all the soft constraints, it is acceptable to not satisfy a soft constraint in case of a conflict with another constraint or a goal. For example, it might be ideal to use all the available trucks to achieve concurrency and better plan quality. Such a condition can be used as a soft constraint to guide the planner to generate a more desirable plan.

2.2 Planning algorithms

There are different types of planning algorithms used in domain-independent planning systems. This work focusses on a specific type of domain-independent planning system called *classical planning*.

Most research and applications of automated planning have focused on classical planning, which is a type of domain-independent planning. Classical planning imposes the following restrictive assumptions on the state-transition system $\Sigma = (S, A, \gamma)$:

1. **Finite states:** In system Σ, there are a finite number of states.
2. **Fully observable:** In system Σ, one has complete knowledge about the state of the system.
3. **Deterministic actions:** All actions in the system Σ are deterministic, such that the application of the action will always lead to the same effect on the system.

4. **Explicit goal(s):** The planner is given a specific goal S_g to achieve.

5. **Sequential plans:** The output of the planning problem is a linearly ordered sequence of actions.

6. **Static:** Only the agent is considered to be moving and rest of the environment is assumed to be static.

Solving a classical planning problem amounts to solving a reachability problem in a graph (Ghallab et al., 2004). In classical planning, a heuristic based planner builds a graph using the planning language input, and then finds a heuristic for that domain. After finding a good heuristic function, the planner searches through a graph of states, also called *state-space*.

The problem of searching through state-space might look trivial, but it is a PSPACE-complete problem. The state-space can be in the order of 2^n, where n is the number of predicates of the domain. Ghallab et al. (2004) shows that even for simple problems, the state space can become significantly larger than the number of particles in the universe.

2.3 Types of classical planning algorithms

There are different types of planning algorithms available, and each one of them provides a different set of advantages and disadvantages. In this section, we will discuss the algorithms that are relevant to this thesis.

Commands to run these planners can be found in appendix A

2.3.1 Heuristic-based Planners

Heuristic Based Planners are one of the most used planners for classical planning. They use variants of the A* algorithm with the advance heuristics to guide the

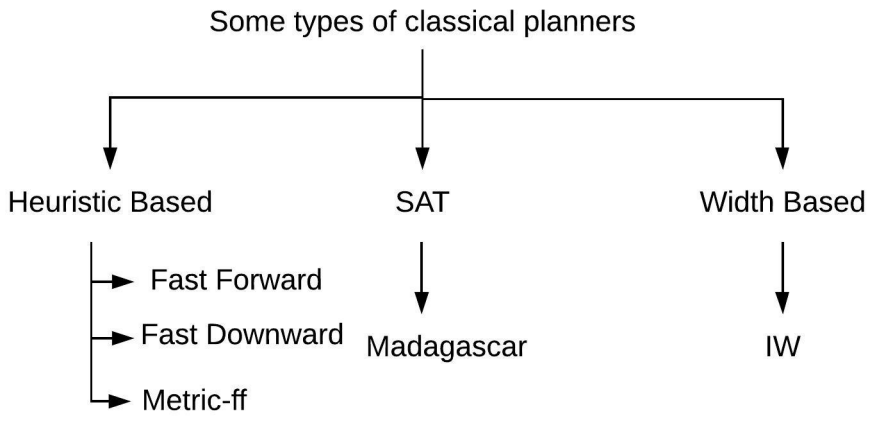

Figure 2.2: Most commonly used classical planners

search. One of the problems for classical planning was *"How to obtain a good heuristic?"*.

HSP (Bonet et al., 1997) is considered a breakthrough in extracting a helpful heuristic by relaxing the problem P into a simpler problem P'. HSP first obtains the relaxed planning problem, P', by ignoring the *delete list*. In other words, P' is like P except that delete lists are assumed empty. As a consequence, actions may add new predicates but not remove existing ones. HSP estimates the heuristic function $h(s)$ to solve the planning problem *P*, by estimating it to the optimal heuristic $h'(s)$ obtained for the relaxed problem.

Here, we will discuss how *Fast Forward*, *Fast Downward* and *Metric-ff* extended HSP to calculate a heuristic and extract a good plan.

Fast Forward Planner: The Fast Forward (FF) planner relies on the forward search in state space to generate a plan (Hoffmann, 2001). While FF is based on HSP, it uses a different method to obtain the heuristic.

FF forms a layered planning graph, which is a directed graph containing two types of nodes: *fact node* and *action node*. The layers of the graph alternate between

action and *fact layers*, where one fact and action layer together forms a *time step*. In the first time step, step 0, the fact layer represents to the initial state, and the action layer contains all the actions applicable to the initial state. The layers in subsequent step i contains all the facts that can be made true in step i, and all the actions that are applicable to the given facts.

After reaching the goal at the top graph layer, if step i has a goal planner it adds it as a goal to be achieved to step $i-1$, this procedure is stopped after reaching the first layer, and $\langle O_1, O_2, \ldots, O_{m-1} \rangle$ where O_i is action taken at step i.

FF planner lacks a few features like support for metrics and the ability to produce parallel plans. FF planner improves runtime performance and provides a shorter plan length over ancestors due to the above method of finding the relaxation.

Fast Downward Planner: Fast downward (FD) is a classical planning system based on heuristic search which supports all features of PDDL2.2. FD has shown to be notably successful; it won the 4th International Planning Competition (Edelkamp and Hoffmann, 2004).

Like FF, FD is also based on forward search in state space. However, unlike most of the classical planners, FD does not use propositional representation of a planning problem directly. FD planner took motivation from *"How would humans solve a planning problem?"*, and the most common approach of dividing problem into sub problems is used in FD. The planner uses hierarchical decomposition of planning tasks for the calculation of the heuristic function, called *casual graph heuristic*.

FD planner generalises the problem to several levels of abstraction, forming an abstraction hierarchy. The topmost level is the most abstract level, where all the preconditions are ignored. Successive levels have more details than the previous layer until the final layer of the hierarchy equals to the planning task (Helmert, 2006).

For example if a `truck1` is standing at the `storage1`, it needs to gather an

item from `node1` and transport the item to `storage2`. Then the planner might generate the following plan:

```
GOTO truck1 storage1 node1
GATHER truck1 item1
GOTO truck1 storage1 storage2
STORE item1 storage2
```

Metric-FF: Though the planning language, PDDL, supported numeric fluents before Metric-FF, none of the planners fully supported numeric fluents in preconditions, and effects of the actions. Metric-FF planner extends FF planner to handle quantifiers, disjunctions and numerical fluents in preconditions, effects and goals (Hoffmann, 2003).

Metric-FF supports the following operators:

- **Arithmetic operators:** $+, -, *$ and $/$.
- **Comparison operators:** $=, >, <, <=$ and $>=$.
- **Assignment operators:** $:=, +=, -=, *=$ and $/=$.

Metric-FF is highly useful in generating plans with more detail by adding numerical variables to the domain and problem description. Suppose a scenario where we have to transport 5 units of `item1` from `node1` to `storage2`. The planner might generate the following plan:

```
GOTO truck1 storage1 node1
GATHER truck1 item1
GATHER truck1 item1
GATHER truck1 item1
GATHER truck1 item1
```

```
GATHER truck1 item1
GOTO truck1 storage1 storage2
STORE item1 storage2
STORE item1 storage2
STORE item1 storage2
STORE item1 storage2
STORE item1 storage2
```

In the above example, action gather and store, will be repeated to accommodate the requirement of storing 5 units of the item. Due to the level of detail Metric-FF can handle, it has been proven to be one of the best numeric planners in IPC-3 (Gerevini et al., 2003).

LAMA LAMA is a classical planning system based on Fast downward planning system. In addition to FD, it uses the FF heuristic to improve performance by calculating heuristics derived from *landmarks* (Richter et al., 2008). Landmarks are the propositions that need to be true for all plans for a given planning problem.

LAMA uses landmarks to direct its search towards the state where most amount of landmarks have already been achieved. After LAMA has found an initial solution with a greedy best-first search, it tries to find alternative plans that are more optimal in terms of action cost. This approach has been shown to be very efficient compared to other planners (Richter et al., 2010).

2.3.2 SAT based

SAT planners are based on the idea of planning as a satisfiability problem (Cook, 1971); which was first introduced by Kautz et al. (1992) in their SATPLAN system. SAT planner produces a plan where two actions can be performed simultaneously if they can be ordered in such a way that the older action does not disable

the later action, also called $\exists - step$ plan.

SAT planner translates the classical planning problem into a formula of propositional variables. Then the planner reduces the planning problem to a series of satisfiability tests. Recent SAT based planners like Madagascar (Rintanen et al., 2006) rival the performance of planners based on other paradigms.

2.3.3 Width/novelty based

Width based planners uses a forward state breadth-first search algorithm with a modification to make it faster. A new concept known as *state novelty* is used to prune a state in the state space. The state novelty is the size of minimal set of atoms that has been seen for the first time. The higher the novelty measure, the less novel that state is (Geffner and Lipovetzky, 2012).

For this thesis, we have used Iterative Width (IW) planner (Geffner and Lipovetzky, 2012). It has been shown that the IW planner works better when the goal is a single atom. Novelty based planners has shown good results in the General Video Game (GVG) Competition 2016 (Perez-Liebana et al., 2016).

Chapter 3

Agents in the City domain

The Agents in the City domain is one of the several domains in the Multi-Agent International Contest[1]. It has two teams of 28 agents competing against each other to earn as much virtual currency as possible. The virtual currency *massium* can be earned by completing jobs, and the domain provides certain requirements to complete a job (Ahlbrecht et al., 2018).

Completion of a job requires the assembly of *items*. There are two types of items, the *base items* that can be gathered from a facility, and the *advanced items* can be assembled in a facility.

The advanced items are assembled by agents. There are four types of agents in the domain: drone, motorcycle, car, and truck. Each agent has different abilities, and can perform different roles. For example, a drone can travel from one location to another even if there is no road, while other types of agents require a road to travel between two locations. Each agent is charaterised by following attributes:

- **Battery:** How long an agent can last without charging.
- **Carrying capacity:** How much volume an agent can carry.

[1] `https://multiagentcontest.org/`

- **Speed:** How fast an agent can move.
- **Vision:** How far an agent can perceive.
- **Skill:** How fast an agent can complete tasks.

Apart from agents, the domain also has facilities such as *shops*, *charging stations*, *workshops*, *research nodes*, *storage* and *dumps*. These facilities are positioned randomly over the map. Each facility is recognised by a unique name and location. These facilities serves the following purpose:

- **Shop:** Agents can trade the assembled items in a shop.
- **Charging station:** A charging station can be visited by the agents to charge their battery.
- **Workshop:** A Workshop is a type of facility where agents can assemble an item.
- **Storage:** Storages are the facilities which can store items up to a specific volume.
- **Dump:** An agent can destroy an item at dump (to free capacity).

Even though there are enough agents and facilities available, completing the jobs could still get difficult. For example, a job requirement of completing `job1` is given in the following way:

```
job(job1, storage1, 339, [required(item10, 1),
    required(item5, 1), required(item8, 2)])
```

In the above scenario, `job1` requires one unit of `item10`, one unit of `item5`, and two units of `item8`. For the completion of the job `job1`, all the mentioned items are to be stored in the storage `storage1` to earn a reward of 339.

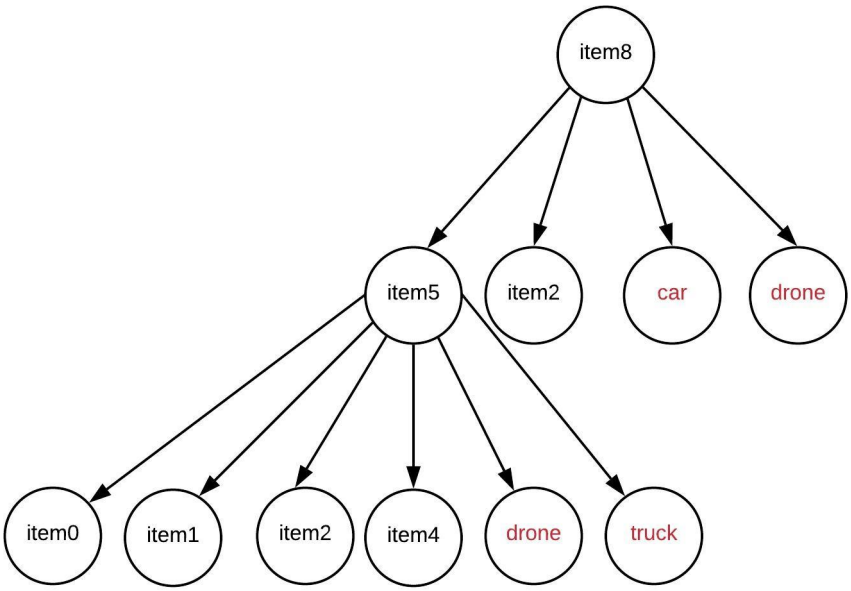

Figure 3.1: Assembly requirements of item 8

Figure 3.1 shows the requirements of assembling `item8`. It can be seen from the figure that, the assembling of `item8` require two roles (a car and a drone) and two items (`item5` and `item2`) to be present at the same workshop. Items that are present at the leaf nodes are the base items, but `item5` itself has to be assembled before the assembling of `item8` takes place. In addition to the required agents, any number of agents can participate in the assembly process. The domain do not impose any restriction on the order of items supplied or limit the number of agents participating in the assembly process.

For the assembling of `item8`, any car and drone can be present, and more agents can participate by supplying the required items. For example, in addition to the required agents (let us say `drone1` and `car2`), `truck1` can supply both of the required items.

Assembling an advance item is considered difficult to achieve because it requires collaboration, cooperation and contemplation from different agents in the domain.

Completing a job can be a very cumbersome task, as it might need multiple advanced items. To accomplish tasks, the domain provides the following list of *actions* that an agent can perform:

- **goto:** Moves an agent towards a destination.
- **give:** Gives an item to another agent at the same location.
- **store:** Stores an item in a storage.
- **assemble:** Called by the main agent assembling an item.
- **assist_assemble:** Called by all agents, apart from the main agent, to participate in the assembly.
- **dump:** Used to get rid of an item at the dump facility.
- **recharge:** Recharges their battery by 1 unit.

It is on the execution engine/program to use these actions effectively to achieve the goal.

3.1 Existing solutions

In this section, we will discuss the solutions submitted by the winning team *BusyBeaver* for the Agents in the City 2017 contest.

BusyBeaver's implementation focuses on selecting the jobs that are most profitable, and attempts to assemble items in advance to get a head start on the job. The encoding of the domain is done in AgentSpeak (Bordini and Hübner, 2005) and Python. AgentSpeak is an Agent-Oriented Programming Language, and it relies on the reactive nature of agents to complete the jobs.

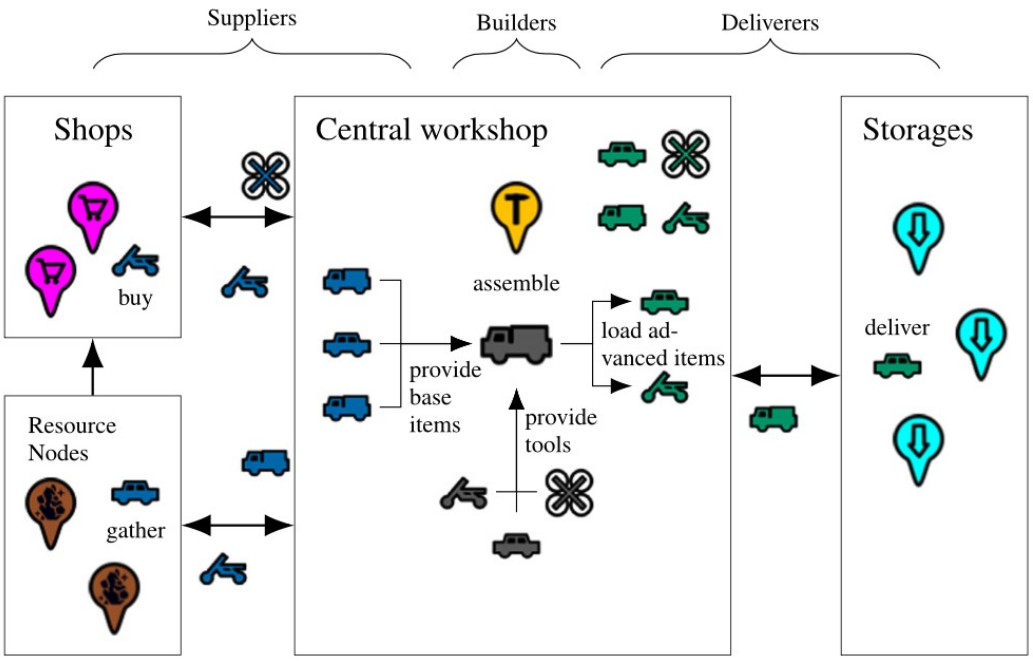

Figure 3.2: Architecture of BusyBeaver's strategy

The figure 3.2 shows the strategy used by the team Busybeaver to optimise the assembly process. They divided the agents into three major categories: *suppliers* to gather the base items, *builders* to assemble advanced items and *deliverers* to deliver the items to the storage.

The interaction in the above strategy is handeled by a leader which does high-level planning such as deciding which items need to be gathered. But basic actions such as charging are left upon the agents to decide by themselves.

BusyBeaver won the Agents in the City 2017 contest (Ahlbrecht et al., 2018), and the key reason of their win reactive nature of the agents, and the organised collaboration amongst the agents. The reactive nature of the agents enabled the system to start working towards the completion of jobs as soon as they are available.

The table 3.1 shows the distribution of agents by role across group used by BusyBeaver. The distribution of the agents is programmed into the system, and that causes several bottlenecks in the system. In the situation where a lot of items are

	Suppliers	Builders	Deliverers
Drones	1	2	1
Motorcycles	4	1	3
Cars	4	1	3
Trucks	4	1	3

Table 3.1: Distribution of agents by role across group

being assembled in workshops, the suppliers will be idle because there is no current requirement of the items while the builders and deliverers will be completely accupied. This programmed strategy fails to adapt in situation of uneven work loads.

The assembling of advanced items happen in any of the four workshops in the domain. It can also be seen from the table 3.1 that there is only one car, motorcycle and truck available for assembling an item. In the situation where two assemblies need a drone to be present at the workshop, the assembling will have to wait for the drone to be free from another workshop.

Chapter 4

Automated planning for Agents in the City

In this chapter, we will describe how can we model the Agents in the City domain for automated planning, and the results and analysis of the model on various planning algorithms.

As mentioned above, the aim of the Agents in the City is to earn maximum amount of virtual currency, and virtual currency can be earned by completing jobs in the Agents in the City domain. For completing a job, the agents needs to assemble complex items and transport it to a storage. The task of assembling as item is highly complex, and can be accomplished in different ways.

We investigate how automated planning can be used for completing a job. To do so, we used iterative development technique to develop various domain models for encoding the domain information. The models cover domain actions, dummy actions, action cost, and numeric fluents. While using domain actions with dummy actions turn out to be feasible, it seems that planners have a hard time optimising on action cost. The experiments also show that given enough time using numeric fluents is also possible.

The chapter is organised as follows. Section 4.1 describes the encoding of static

features of the domain and the basic processes of the domain. Section 4.2 describes the ways to make the basic model scalable using different features of PDDL. Section 4.3 describes the encoding of action cost to achieve a more desirable plan. Section 4.4 describes the encoding of the numeric details of the domain such as the agents carrying capacity, to the automated planning model. Finally, section 4.5 describes the result and analysis of the execution of PDDL encodings on various planners.

4.1 Basic Model

The basic aspects of planning, for the domain Agents in the City involve encoding the static and dynamic domain information into PDDL. The domain has static constructs, which do not change during the execution, such as the four types of agents (truck, car, motorcycle and drone).

All the provided object types, such as agents, items, storage, workshop and resource nodes, can be declared in the following way:

```
(:types
   agent facility item
   truck motorcycle drone car - agent
   storage workshop node - facility
)
```

In the above definition, there are three types of objects, namely, `agent`, `facility` and `item`. In the second line there are four types of agents namely `truck`, `motorcycle`, `car` and `drone`. Finally, there are three types of facilities in the domain namely `storage`, `workshop` and `node`.

Assembling: The domain requires us to complete jobs to obtain reward. The most important part of completing a job is the assembling of items. For example, consider the assembly of `item5`, which requires requires roles (car and a drone), and certain parts (`item1`, `item2` and `item4`). This scenario can be described as:

```
item(item5, 5, roles([car, drone]),
                parts([item1, item2, item4]))
```

A direct approach to model the assemble action would require the following predicates:

```
(:predicates
    (assemble-require-role ?i - item ?a - agent)
    (assemble-require-item ?i - item ?iR - item)
)
```

In the aforementioned example, the first predicate `assemble-require-role` is used to specify the roles required for assembling an item For example,

```
(assemble-require-role item5 drone1)
(assemble-require-role item5 car1)
```

The above predicate means that the assembly of item5 requires `drone1` and `car1` to be present for assembly.

```
(assemble-require-item item5 item1)
(assemble-require-item item5 item2)
(assemble-require-item item5 item4)
```

The above predicates signify that the assembly of `item5` requires `item1`, `item2` and `item4`.

Unfortunately, the above approach does not work, because it requires the specification of predicates which are not flexible with the domain requirements. The

predicate `(assemble-require-role item5 drone1)` is specifying `drone1` for the assembling process. However, there could be another drone available at a more convenient location.

The action of assembling an item, will require all the needed roles and items to be available at the workshop. The assembly of different items require different number of items and roles. The naive approach above fails to achieve flexible assembly.

The process of assembling is considered difficult to model because any number of agents can participate, in addition to the required roles. Because of the difficulties, we decided to model the assembly requirements by encoding it directly into the action schema.

Here is an example of how the assembling of `item5` is modeled:

```
(:action assemble_i5_car
    :parameters ( ?c - car ?d - drone
                  ?a1 ?a2 ?a3 - agent
                  ?w - workshop)
    :precondition (and
        (agent-at-facility ?d ?w)
        (agent-at-facility ?c ?w)
        (agent-at-facility ?a1 ?w)
        (agent-at-facility ?a2 ?w)
        (agent-at-facility ?a3 ?w)

        (agent-carrying-item ?a1 item1)
        (agent-carrying-item ?a2 item2)
        (agent-carrying-item ?a3 item4)
    )
    :effect (and
        (not (agent-carrying-item ?a1 item1))
```

```
            (not (agent-carrying-item ?a2 item2))
            (not (agent-carrying-item ?a3 item4))
            (agent-carrying-item ?c item5)
        )
    )
```

In the above action, the first five preconditions capture the requirements that the agents are at the workshop where `item5` is being assembled. The last three preconditions capture the requirements that `item1`, `item2` and `item4` are being made available by the respective agents.

The first three lines of effect of the action is capturing the information that the required items are being consumed. The atom `(agent-carrying-item ?c item5)` represents the fact that the assembled item `item5` is being released to the car.

The action does not impose any condition on `?a1`, `?a2` and `?a3`, that is why it is not necessary that `?a1`, `?a2` and `?a3` are three different agents. They all could be a single agent providing all items. This encoding only puts a condition that a car (`?c`) and a drone (`?d`) has to be present.

The minimum number of agents participating in the assembly is two, because either of the car and drone participating can be `?a1`, `?a2` and `?a3`. The maximum number of agents participating in the assembly would be five where `?c`, `?d`, `?a1`, `?a2` and `?a3` are all different agents. This encoding successfully satisfies the domain requirements.

In addition to the assemble action, we completed the encoding of the following actions for basic modelling:

Gather: The `gather` action is used to mine an item from a resource node, and it can be encoded in the following way:

```
(:action gather
    :parameters (?a - agent
        ?i - item ?n - resourceNode)
    :precondition (and
        (agent-at-facility ?a ?n)
        (item-in-resourceNode ?i ?n)

    )
    :effect (and
        (agent-carrying-item ?a ?i)
    )
)
```

Here, the first precondition captures the information that the agent `?a` has reached the resource node, and that the second precondition captures that the required item `?i` is available in the node.

The effect captures the information that the agent `?a` has now acquired the item `?i`.

Goto: The `goto` action is explained in section 2.1.2.1.

Store: The `store` action is used to store an item in a storage. We used the following encoding for the `store` action:

```
(:action store
:parameters (?a - agent
    ?s - storage ?i - item)
:precondition (and
    (agent-carrying-item ?a ?i)
    (agent-at-facility ?a ?s)
```

```
        )
        :effect (and
            (not (agent-carrying-item ?a ?i))
            (item-in-storage ?i ?s)
        )
    )
```

In the above encoding, the preconditions capture the information that the agent `?a` is carrying the item `?i`, and `?a` is at the storage `?s`. When the action is applied, it has an effect in which `?i` is taken from `?a` and added to `?s`.

Give: The `give` action is important in the domain because it is used for the transfer of an item from one agent to another. It can be encoded in the following manner:

```
    (:action give
        :parameters (?a1 ?a2 - agent
            ?i - item ?f - facility)
        :precondition (and
            (agent-carrying-item ?a1 ?i)
            (not (agent-carrying-item ?a2 ?i))

            (agent-at-facility ?a1 ?f)
            (agent-at-facility ?a2 ?f)
        )
        :effect (and
            (not (agent-carrying-item ?a1 ?i))
            (agent-carrying-item ?a2 ?i)
        )
    )
```

Here, the first two preconditions capture the fact that the agent `?a1` is carrying the item `?i`, while `?a2` does not have `?i`. Last two preconditions of the action captures the fact that both agents (`?a1` and `?a2`) are at the same facility.

The effect of the action is that the agent `?a2` has the item `?i` and the agent `?a2` does not have the item `?i` anymore.

Scalability of the Basic Model: While the PDDL specification is parameterised, the actual planners instantiate all variables with all possible objects in the problem definition. This process is called *grounding*. For example, in `assemble_i5_car` action the parameter `?c` will be replaced by each car available, `?d` will be replaced by each drone available, and so on.

This means that in a scenario where there are four cars, drones, motorcycles, trucks and workshops; the total number of grounded `assemble_i5_car` actions are:

$$\begin{aligned}\text{number of actions after grounding} = {} & (\text{number of cars}) \\ & \times (\text{number of drones}) \\ & \times (\text{number of agents})^3 \\ & \times (\text{number of workshops}) \\ & = \mathbf{262{,}144}\end{aligned}$$

As we can see, the calculation of all the combinations by replacing parameters of an action with objects leads to *combinatorial explosion*. Since there are 6 items to be assembled in the domain, it is impractical for the domain to use basic modelling because the above encoding is not scalable.

4.2 Scalable domain model

As we saw before, the scalability of the domain model is directly affected by combinatorial explosion on grounding. Since the number of combinations of an action depends on the number of parameters of actions. We will reduce the number of grounded actions by introducing *dummy actions*.

The idea is to replace complex actions with many parameters like the `assemble_i5_car` above with a sequence of actions with relatively fewer parameters. Importantly the execution of all those actions gives a result equivalent to the original action.

For example, the action `assemble_i5_car` can be broken into following modelling and assembly process:

1. The first step involves selecting and committing to a workshop where the assembly will happen. This is done with the following action:

   ```
   (:action prep_assemble_item5_finalise_workshop
   :parameters (?w - workshop))
   ```

2. The second step involves getting all the required items to the workshop. This is done with the following actions:

   ```
   (:action prep_assemble_item5_arrange_item1
   :parameters (?a - agent ?w - workshop))
   ```

   ```
   (:action prep_assemble_item5_arrange_item2
   :parameters (?a - agent ?w - workshop))
   ```

   ```
   (:action prep_assemble_item5_arrange_item4
   :parameters (?a - agent ?w - workshop))
   ```

3. The third step is to make sure that the required roles are present at the workshop. This can be done using the following action:

   ```
   (:action prep_assemble_item5_arrange_roles
   :parameters (?c - car ?d - drone ?w - workshop))
   ```

4. The fourth step of the process is to assemble the item. This can be done in with the following action:

```
(:action assemble_i5_car
:parameters (?c - car))
```

5. The fifth step involves consuming the required items for assembly, and releasing the agents that are providing the item. This can be achieve using the following actions:

```
(:action consume_item1_assemble_i5
:parameters (?a - agent))

(:action consume_item2_assemble_i5
:parameters (?a - agent))

(:action consume_item4_assemble_i5
:parameters (?a - agent))
```

6. The sixth step involves releasing all the resources such as the agents and the workshop. This can be done in the following way:

```
(:action post_assemble_i5_freeup_everything
:parameters (?c - car ?d - drone ?w - workshop))
```

7. The seventh step involves releasing the assembled item to the main agent. this can be done in the following way:

```
(:action release_assembled_item5
:parameters (?a - agent))
```

For the complete description of each action see Appendix B.

Observe that the number of parameters in the above encoding is up-to two, except for `:action prep_assemble_item5_arrange_roles` where the number of parameters is three.

For the scenario considered in the section 4.1, new total number of actions after grounding can be calculated in the following way:

number of actions after grounding = (number of workshops)
+ (number of agents $\times$ number of workshops) $\times 3$
+ (number of cars $\times$ number of drones $\times$ number of workshops) $\times 2$
+ (number of cars)
+ (number of agents) $\times 4$
$= 4 + (16 \times 4) \times 3 + (4 \times 4 \times 4) \times 2 + 4 + 16 \times 4$
$=$ **392**

So, by splitting the action `assemble_i5_car` into dummy actions, the number of actions after grounding can be reduced from 262,144 to 392. Which completely removes the scalability issue for the agents in the city domain as we will see in experiments.

Since dummy actions are introduced, it becomes necessary to use auxiliary predicates to take care of the flow in an assembly process. For example, for the action `prep_assemble_item5_arrange_item1` used above for arranging `item1` for the assembly of `item5`. The action is encoded in the following way:

```
(:action prep_assemble_item5_arrange_item1
    :parameters (?a - agent ?w - workshop)
    :precondition (and
        (or (agent-committed ?a item5)
            (not (agent-busy ?a)))
        (workshop-allocated ?w item5)
        (agent-at-facility ?a ?w)
        (agent-carrying-item ?a item1)
    )
    :effect (and
        (item-arranged-for-assembly item1 item5)
        (agent-providing-item-for-assembly
            ?a item1 item5)
        (agent-busy ?a)
        (agent-committed ?a item5)
```

```
        )
    )
```

Earlier for the assembly process, there was only one action for the assembly on an item, so the agent did not have to be declared busy or committed. Now, the plan flows from one action to another for assembling a single item. To take care of the situation where an agent should not be able to leave during the assembly; the predicates `agent-committed` and `agent-busy` are used.

Using existential quantifier: Further parameter reduction can be sometimes achieved by using the existential quantifier. The action `prep_assemble_item5_arrange_item1` can be encoded in the following way:

```
    (:action prep_assemble_item5_arrange_item1
        :parameters (?a - agent)
        :precondition (and
            (or (agent-committed ?a item5)
                (not (agent-busy ?a)))
            (exists (?w - workshop)
                (and (workshop-allocated ?w item5)
                    (agent-at-facility ?a ?w)))
            (agent-carrying-item ?a item1)
        )
        :effect (and
            (item-arranged-for-assembly item1 item5)
            (agent-providing-item-for-assembly
                ?a item1 item5)
            (agent-busy ?a)
            (agent-committed ?a item5)
        )
    )
```

Here, the keyword `exists` is used as an existential quantifier to completely remove the parameter `?w - workshop`, because it is used only in the precondition, and not in the effect of the action.

4.3 Capturing action cost

The Above actions have no cost, this means that the agents can perform actions even when it is not necessary. For example, an agent might continue giving items to other agents. To make sure that the agents do not use any unnecessary action, we added a cost to every action in the domain definition.

For example, the action `goto` will have the following definition:

```
(:functions
    (total-cost)
    (distance ?from - facility ?to - facility)
)

(:action goto
:parameters (?a - agent ?loc1 ?loc2 - facility)
:precondition (and
    (agent-at-facility ?a ?loc1)
    (not (= ?loc1 ?loc2))
    (not (agent-busy ?a))
)
:effect (and
(agent-at-facility ?a ?loc2)
        (not (agent-at-facility ?a ?loc1))
        (increase (total-cost)
            (distance ?loc1 ?loc2))
```

```
        )
    )
```

In the above action, the function `(total-cost)` is added to keep a track of how much is the cost of using `goto`. Every time the action `goto` is performed, the cost is increased by the distance between the two locations.

The fact that we need to minimize the `(total-cost)` of the plan can be encoded in the problem description in the following way:

```
(:metric
    minimize (total-cost))
)
```

Similarly, an action cost is added to all the actions. When the planner receives the input from PDDL encoding, it tries to keep the action cost to minimum, therefore any unnecessary action will not be performed.

4.4 Capturing agent capacity

In the domain Agents in the City, every action has a certain carrying capacity. It is important for an agent to carry items within their carrying capacity. Numeric fluents can be used as a function in PDDL to track the carrying capacity of an agent in the following way:

```
(:functions
    (agent-carrying-item ?a - agent ?i - item)
    (agent-capacity ?a - agent)
)
```

In the above example, the fluent `agent-carrying-item` tracks the amount of each item carried by an agent; and, `agent-capacity` keeps a track of the remaining capacity of the agent.

When the agent gets an item (through either `gather` or `give` action), the remaining capacity of the agent decreases while the amount of item carried by that agent increases.

The following fluents can be added to the effect of the `gather` action to achieve the aforementioned functionality:

```
(:action gather
    :parameters (?a - agent
        ?i - item ?n - resourceNode)
    :precondition (and
        (item-in-resourceNode ?i ?n)
        (agent-at-facility ?a ?n)
        (not (agent-busy ?a))
        (>= (agent-capacity ?a) 1)
    )
    :effect (and
        (increase (agent-carrying-item ?a ?i) 1)
        (decrease (agent-capacity ?a) 1)
    )
)
```

In the above action, the atom `(>= (agent-capacity ?a) 1)` in the precondition makes sure that the action is only executed when agent has the capacity to carry more. The first atom in the effect increases the amount of item `?i` the agent `?a` is carrying by 1, and the second line decreases the remaining capacity of the agent `?a` by one.

Encoding Version	IW	FD	Metric-FF	Madagascar
V1	8.29s	5.14s	2.44s	24.86s
V2.1	218.97s	56.19s	195.18s	42.42s
V2.2	209.99s	80.279s	135.91s	MEM
V3	74.79s*	TO	459s*	41.85s*
V4	NR	NA	6430s	NA

NA = Not Applicable
NR = Plan not found
TO = Timeout
MEM = Runs out of memory
* = Execution successful, but total-cost ignored by the planner

Table 4.1: Execution time of all versions on different planners

4.5 Results and analysis

For obtaining the results, each domain model version was executed on different planners. Table 4.1 summarises the execution details of all PDDL encodings on IW[1], Fast Downward (FD)[2], Metric-FF[3] and Madagascar[4] planners. Table 4.2 describes the pros and cons of using each version.

In the tables 4.1 and 4.2, the encoding versions are:

- V1: The PDDL encodings of the basic model discuss in section 4.1.
- V2.1: The encodings with dummy actions, explained in section 4.2.
- V2.2: The PDDL encodings with the use of existential quantifier, explained in section 4.2.
- V3: The version V3 refers to the PDDL encodings with use of the action cost, also described in section 4.3

[1] http://lapkt.org/index.php?title=Documentation#IW_Plus
[2] http://www.fast-downward.org
[3] https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html
[4] https://research.ics.aalto.fi/software/sat/madagascar/

Encoding Version	Scalable	DuA/TA	Numeric Fluent support	Parallel actions
V1	✗	0	✗	✓
V2.1	✓	0.80	✗	✓
V2.2	✓	0.80	✗	✗
V3	✓	0.80	✗	✓
V4	✓	0.80	✓	✗
DuA/TA = ratio of Dummy Actions to Total number of actions				

Table 4.2: Pros and cons of each version of the domain model

- V4: The version V4 refers to the PDDL encodings that includes use of numeric fluents to capture the carrying capacity of the agents, also described in section 4.4

The following observations can be made from the tables 4.1 and 4.2:

Scalability: The scalability of the domain model is affected by DuA/TA ratio, where DuA is the number of dummy actions used in the domain description, and TA is the total number of actions in the domain description. High DuA/TA ratio means that the execution platform will have to extract details from the dummy actions, as the dummy actions are not supported by the domain.

In case of V1, just domain actions are used for domain description, that is the reason behind zero DuA/TA ratio. Zero DuA/TA ratio also means that the plan can directly be executed by the execution platform as only domain actions are used for modelling. The model V1 is not a scalable approach, because of combinatorial explosion no more than 2 items could be assembled, and that is why the execution time is the least as not all details were added to the domain model. When dummy actions are introduced in further version (V2, V3 and V4), the DuA/TA ratio increases to .80 but the domain model is now scalable, and all the items in the domain were added to the model.

Performance of Madagascar planner: The Madagascar planner has the best performance in the domain in terms of execution time. Madagascar planner generated a plan where multiple actions could be performed concurrently. It is very helpful feature for this domain because there are multiple agents in the domain, so performing multiple actions on different agents at the same time will help us accomplish tasks much faster.

Capturing required quantity: Out of all the planners we have used, only Metric-FF supports using mathematical operators to deal with numeric fluents in PDDL. Even though the execution time of V4 on Metric-FF in significantly more than any other execution, the plan generated is also a lot more detailed. For example, the following plan will be generated when `truck1` has to acquire 5 units of `item1`:

```
GOTO truck1 storage1 node1
GATHER truck1 item1
GATHER truck1 item1
GATHER truck1 item1
GATHER truck1 item1
GATHER truck1 item1
```

The above plan was generated on Metric-FF, any other planner used for this work would have mentioned `GATHER truck1 item1` only once. The only disadvantage of using V4 is that it is time consuming. But it could be helpful in applications such as mining, as the plan required in mining is detailed and there is enough time to generate the plan. Once a detailed plan is generated, it can be directly implemented. Where as if numeric fluents are not used, then the plan might require extra effort to be made ready for execution.

Optimal action cost: V3 supported optimal action cost, although no planner used could actually optimise on the `total-cost` fluent. The plan retrieved for V3 was not optimal in terms of the action cost.

Use of existential quantifier: The difference between the versions 2.1 and 2.2 is the use of existential quantifiers for reducing the number of parameters. Every planner interprets the existential quantifier in different manner. For example, IW planner converts the existential planner to a fluent, and that is why the performance of V2.1 and V2.2 was similar on IW planner.

By observing the performance of V2.2 on different planners, it cannot be concluded that reducing paramters using the existential quantifier is helpful.

Chapter 5

Conclusion

In this work, we have encoded different processes of the Agents in the City domain to PDDL, and different features of PDDL have been incorporated to achive a detailed plan. The encodings were then examined with different planners to analyse the plans and execution time of the encodings.

We presented a way of modelling a complex domain with a large number of details in PDDL. Our experimental study has shown that, introducing dummy actions to reduce the number of parameters of actions significantly reduces the number of grounded actions, hence improving the scalability of the domain.

Another significant contribution of this thesis is to encode numeric fluents to obtain a detailed plan. This enables the use of automated planning in complex applications where a detailed plan is a necessity.

Chapter 6

Future Work

The techniques used for modelling the Agents in the City domain seems to be promising for large scale dynamic problems, but this research is still in its infancy and further research is required to fully understand it. For example, it is not clear why using existential quantifiers to reduce the number of parameters reduced the number of grounded actions. Even though reducing the number of parameters has been an efficient technique to reduce the number of grounded actions. As it was briefly mentioned, it might be due to the fact that some planners translate existential quantifiers to parameters in an action.

Finally, the use of action cost in the encoding did not provide an optimal plan on any planner used for this work. it would be interesting to see why is that happening, and if any other planner can support optimising the plan on action cost.

Appendix A

Commands to run planners

The following commands were used for the planners:

- Metric-ff:

```
ff -o domain.pddl -f problem.pddl
```

- IW:

```
iw_plus-ffparser/rp_iw \
    --domain domain.pddl \
    --problem problem.pddl
```

- Fast Downward:

```
./fast-downward.py domain.pddl task.pddl \
    --evaluator "hff=ff()" \
    --evaluator "hcea=cea()" \
    --search "lazy_greedy([hff, hcea], \
     preferred=[hff, hcea])"
```

- LAMA:

```
./fast-downward.py \
    --alias seq-sat-lama-2011 \
    domain.pddl \
    task.pddl
```

- Madagascar:

```
./Mp domain.pddl problem.pddl
```

Appendix B

PDDL encodings for dummy action

```
(:action prep_assemble_item5_finalise_workshop
    :parameters (?w - workshop)
    :precondition (and
        (not (workshop-busy ?w))
        (not (assembly-lock item5))
    )
    :effect (and
        (workshop-busy ?w)
        (workshop-allocated ?w item5)
        (assembly-lock item5)
    )
)

(:action prep_assemble_item5_arrange_item1
    :parameters (?a - agent ?w - workshop)
    :precondition (and
        (or (agent-commited ?a item5)
            (not (agent-busy ?a)))
        (not (item-arranged-for-assembly item1 item5))
```

```
        (workshop-allocated ?w item5)
        (assembly-lock item5)
        (agent-carrying-item ?a item1)
        (agent-at-facility ?a ?w)
    )
    :effect (and
        (item-arranged-for-assembly item1 item5)
        (agent-providing-item-for-assembly
                                ?a item1 item5)
        (agent-busy ?a)
        (agent-commited ?a item5)
    )
)

(:action prep_assemble_item5_arrange_item2
    :parameters (?a - agent ?w - workshop)
    :precondition (and
        (or (agent-commited ?a item5)
            (not (agent-busy ?a)))
        (not (item-arranged-for-assembly item2 item5))

        (workshop-allocated ?w item5)
        (assembly-lock item5)
        (agent-carrying-item ?a item2)
    )
    :effect (and
        (item-arranged-for-assembly item2 item5)
        (agent-providing-item-for-assembly
            ?a item2 item5)
        (agent-busy ?a)
        (agent-commited ?a item5)
```

```
    )
)

(:action prep_assemble_item5_arrange_item4
    :parameters (?a - agent ?w - workshop)
    :precondition (and
        (or (agent-commited ?a item5)
            (not (agent-busy ?a)))
        (not (item-arranged-for-assembly item4 item5))

        (workshop-allocated ?w item5)
        (assembly-lock item5)
        (agent-carrying-item ?a item4)
    )
    :effect (and
        (item-arranged-for-assembly item4 item5)
        (agent-providing-item-for-assembly
            ?a item4 item5)
        (agent-busy ?a)
        (agent-commited ?a item5)
    )
)

(:action prep_assemble_item5_arrange_roles
    :parameters (?c - car ?d - drone ?w - workshop)
    :precondition (and
        (or (agent-commited ?c item5)
            (not (agent-busy ?c)))
        (or (agent-commited ?d item5)
            (not (agent-busy ?d)))
        (assembly-lock item5)
```

```
        (not (required-roles-arranged-for-assembly
                                    item5))
        (workshop-allocated ?w item5)

        (agent-at-facility ?c ?w)
        (agent-at-facility ?d ?w)
    )
    :effect (and
        (required-roles-arranged-for-assembly item5)
        (assembly-required-agent ?c item5)
        (assembly-required-agent ?d item5)
        (agent-commited ?c item5)
        (agent-commited ?d item5)
        (agent-busy ?c)
        (agent-busy ?d)
    )
)

(:action assemble_i5_resources_aquired
    :parameters ()
    :precondition (and
        (assembly-lock item5)
        (not (assembly-resources-acquired item5))

        (item-arranged-for-assembly item1 item5)
        (item-arranged-for-assembly item2 item5)
        (item-arranged-for-assembly item4 item5)

        (required-roles-arranged-for-assembly item5)
    )
```

```
    :effect (and
        (assembly-resources-acquired item5)
    )
)

(:action assemble_i5_car
    :parameters (?c - car)
    :precondition (and
        (assembly-lock item5)
        (assembly-resources-acquired item5)
        (agent-commited ?c item5)
        (not (item-assembled item5))
        (assembly-required-agent ?c item5)
    )
    :effect (and
        (item-assembled item5)
        (assemble-main-guy ?c item5)
    )
)

(:action consume_item1_assemble_i5
    :parameters (?a - agent)
    :precondition (and
        (assembly-lock item5)
        (item-assembled item5)
        (agent-carrying-item ?a item1)
        (agent-providing-item-for-assembly
                                ?a item1 item5)
    )
    :effect (and
        (not (agent-busy ?a))
```

```
        (not (agent-carrying-item ?a item1))
        (not (agent-commited ?a item5))
        (assembly-item-consumed item1 item5)
        (not (item-arranged-for-assembly item1 item5))
        (not (agent-providing-item-for-assembly
                                ?a item1 item5))
    )
)

(:action consume_item2_assemble_i5
    :parameters (?a - agent)
    :precondition (and
        (assembly-lock item5)
        (item-assembled item5)
        (agent-carrying-item ?a item2)
        (agent-providing-item-for-assembly
                                ?a item2 item5)
    )
    :effect (and
        (not (agent-busy ?a))
        (not (agent-carrying-item ?a item2))
        (not (agent-commited ?a item5))
        (assembly-item-consumed item2 item5)
        (not (item-arranged-for-assembly item2 item5))
        (not (agent-providing-item-for-assembly
                                ?a item2 item5))
    )
)

(:action consume_item4_assemble_i5
    :parameters (?a - agent)
```

```
    :precondition (and
        (assembly-lock item5)
        (item-assembled item5)
        (agent-carrying-item ?a item4)
        (agent-providing-item-for-assembly
                                ?a item4 item5)
    )
    :effect (and
        (not (agent-busy ?a))
        (not (agent-carrying-item ?a item4))
        (not (agent-commited ?a item5))
        (assembly-item-consumed item4 item5)
        (not (item-arranged-for-assembly item4 item5))
        (not (agent-providing-item-for-assembly
                                ?a item4 item5))
    )
)

(:action post_assemble_i5_freeup_everything
    :parameters (?c - car ?d - drone ?w - workshop)
    :precondition (and
        (assembly-lock item5)
        (assembly-item-consumed item1 item5)
        (assembly-item-consumed item2 item5)
        (assembly-item-consumed item4 item5)
        (assembly-required-agent ?c item5)
        (assembly-required-agent ?d item5)
    )
    :effect (and
        (not (item-assembled item5))
        (not (agent-commited ?c item5))
```

```
        (not (agent-commited ?d item5))
        (not (agent-busy ?c))
        (not (agent-busy ?d))
        (not (required-roles-arranged-for-assembly
                                            item5))
        (not (workshop-allocated ?w item5))
        (not (workshop-busy ?w))

        (not (assembly-item-consumed item1 item5))
        (not (assembly-item-consumed item2 item5))
        (not (assembly-item-consumed item4 item5))

        (not (assembly-required-agent ?c item5))
        (not (assembly-required-agent ?d item5))

        (not (assembly-resources-acquired item5))

        (assembly-procedure-complete item5)
    )
)
```

Bibliography

T. Ahlbrecht, J. Dix, and N. Fiekas. Multi-agent programming contest 2017. *Annals of Mathematics and Artificial Intelligence*, 84(1-2):1–16, 2018.

F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial intelligence*, 116(1-2):123–191, 2000.

R. A. Belecheanu, S. Munroe, M. Luck, T. Payne, T. Miller, P. McBurney, and M. Pěchouček. Commercial applications of agents: Lessons, experiences and challenges. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1549–1555. ACM, 2006.

B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *AAAI/IAAI*, pages 714–719, 1997.

R. H. Bordini and J. F. Hübner. Bdi agent programming in agentspeak using jason. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 143–164. Springer, 2005.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

M. B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into csp. *Artificial Intelligence*, 132(2):151–182, 2001.

S. Edelkamp and J. Hoffmann. Classical part, 4th international planning competition, 2004.

T. Estlin, R. Castano, R. Anderson, D. Gaines, F. Fisher, and M. Judd. Learning and planning for mars rover science. *Issues in Designing Physical Agents for Dynamic Real-Time Environments: World modelling, planning, learning, and communicating*, page 9, 2003.

R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

M. Fox and D. Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

H. Geffner and N. Lipovetzky. Width and serialization of classical planning problems. 2012.

A. Gerevini and D. Long. Preferences and soft constraints in pddl3. In *ICAPS workshop on planning with preferences and soft constraints*, pages 46–53, 2006.

A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in lpg. *Journal of Artificial Intelligence Research*, 20: 239–290, 2003.

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

S. K. Gupta, D. A. Bourne, K. Kim, and S. Krishnan. Automated process planning for sheet metal bending operations. *Journal of Manufacturing Systems*, 17(5): 338–360, 1998.

M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

J. Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.

J. Hoffmann. The metric-ff planning system: Translating“ignoring delete lists”to numeric state variables. *Journal of artificial intelligence research*, 20:291–341, 2003.

H. A. Kautz, B. Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer, 1992.

V. Lifschitz. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, 1987.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl-the planning domain definition language, 1998.

D. S. Nau. Current trends in automated planning. *AI magazine*, 28(4):43–43, 2007.

D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. Shop2: An htn planning system. *Journal of artificial intelligence research*, 20: 379–404, 2003.

D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *AAAI*, volume 8, pages 975–982, 2008.

S. Richter, J. T. Thayer, and W. Ruml. The joy of forgetting: Faster anytime search via restarting. In *Twentieth International Conference on Automated Planning and Scheduling*, 2010.

J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

D. E. Smith and D. S. Weld. Temporal planning with mutual exclusion reasoning. In *IJCAI*, volume 99, pages 326–337. Citeseer, 1999.