Planning Automation on Farmbot: The Classical Planning Approach

Master of Computer Science Research Project (100 points)

By:

Author: Jiayuan Chang

Student Number: 1137973

Email: jiaychang@student.unimelb.edu.au

Supervised by:

Supervisor: Nir Lipovetsky

Department: School of Computing and Information Systems

Email: nir.lipovetzky@unimelb.edu.au

School of Computing and Information Systems

University of Melbourne

Victoria, Australia

31 - October - 2021

Abstract

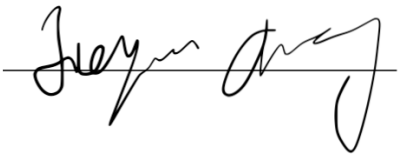
Autonomous system has an increasingly important role in small-scale urban agriculture. It has the potential to enable people without professional agricultural skills to utilize the under-exploited urban space on top of, or between the urban buildings to increase food production. It is important for global food security given the background of increasing global population and decreasing total arable land in the future. The aim of this research is to propose the classical planning approach to automate an existing low-cost robotic platform: Farmbot, which is designed for small-scale backyard farming and evaluate the feasibility of this approach. Specifically, we first propose the appropriate planning domain and problems on Farmbot using the planning domain definition language (PDDL) to allow the common tasks on the Farmbot to be achieved by automatically generated plans. Then, we explore the meaningful way to apply plan metric in order to optimize those plans. Later, we propose an agent planning program, which enables the automatic formulation of a sequence of planning problems and continuous plan generation over the planning domain on Farmbot. A ROS project is developed to incorporate these planning components with a controlling interface to enable plan dispatching and execution. The results and evaluation from multiple experiments, including a benchmarking and a simulation show the feasibility of such an approach and also validate our PDDL modelling, our agent planning program, the approach we apply plan metric to optimize the plans and the entire system in the ROS project. This project introduces a new research problem and increases the uptake of planning technology in real-world problems, most importantly it opens up many options and opportunities for future research topics.

Keywords: Classical Planning, PDDL, Farmbot, Agent Planning Program, ROS, Simulator, ROSplan

Declaration

I certify that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- Clearance for this research from the University's Ethics Committee was not required.
- The thesis is 29836 words in length, excluding text in images, captions, tables, bibliographies and appendices.

Signed:  Date: 31/10/2021

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Nir Lipovetsky, who provided support, feedback and encouragement that enable me to undertake this project. His suggestion and sharing of knowledge allow me to quickly gain relevant knowledge and explore a new area that is full of opportunities for future applications and cross-disciplinary collaboration.

Contents

List of Figures

List of Tables

I Introduction

AI planning is the study that aims to automate the reasoning about plans, more specifically about formulating the plan to achieve a given goal in a given situation [1]. As its name implies, it is one of the important research areas in Artificial Intelligence [1]. The early research mainly focused on specific algorithms that are tied to particular problems and therefore, do not generalize [2]. Later, with the evolution of AI planning technology, model-based planning techniques such as classical planning were developed so that an algorithm for solving planning problems can generalize over a set of problems expressible via the same planning model. Model-based planning often involves two parts: a formalization technique that describe the initial situation, the actions available to change it, and the goal condition in the form of a general representation, known as model, and a planning system, known as planner/solver that takes this general representation as input and compute the solution, or plan composed of those actions that will accomplish the goal when executed from the initial situation. [1].

The Planning Domain Definition Language (PDDL) is one such formalization technique. It is a description language based on the STRIPS formalization [3] which represents the knowledge in propositional logic. It allows users to model problems, which can then be solved by a problem-independent planner that takes in the formalization described using PDDL, converts the model into a search space or logical reasoning problem and applies some problem solving techniques such as heuristic search or Boolean satisfiability to solve it effectively. The planner does not need to know about the formal description of the problem and hence, can be applied to any problem expressible using the PDDL language [1]. Although PDDL is not the only available modelling language, with its role in the International Planning Competition [4], PDDL becomes the de-facto standard for representing classical planning problems [2] and it is now the most supported language in terms of available planning systems [1]. As it continues to be developed by worldwide communities, more features were added to PDDL, which enable it to model more expressive domains such as numeric proposition, temporal plannings [5] and continuous processes [6]. Due to its usefulness and popularity, many researchers apply PDDL to solve real world problems [7]–[11].

In the area of agriculture, automation has increasingly become the interest of many researchers. With the development of digital agriculture, we see many existing works apply AI technologies in attempt to automate the agricultural process [12]–[16]. However, the vast majority of these works we reviewed addressed their respective problem using non-generalized techniques, often developed with partnership from industry and adopt different approaches for automation. These approaches are not international standard and often close-source which are only available to users who purchase the product from these

relevant researches.

In this paper, we present our PDDL approach to automate agricultural process through the Farmbot platform. Farmbot is a comparatively low-cost robotic device designed for small-scale urban/backyard family farming. It is a generic, computer numerically controlled (CNC), 3-axis Cartesian robot. Most importantly, it is open-source, which means its design and other works related to it are widely available to the worldwide communities. Farmbot in its current form, does not have any automation feature. It requires human to program it by composing its available actions (i.e. moving to a certain position, turning on water pump, etc.) into a sequence of actions that solve certain tasks (watering all plants, etc.). Users can do so by utilizing a web-based drag-and-drop programming interface or writing scripts that use the API which call and execute every single action. The nature of this ”sequence of actions” resemble the solution of classical planning or the plan generated by a PDDL planner. Hypothetically, given an appropriate modelling of the domain related to the Farmbot using PDDL language, those manually composed sequences of actions can be generated automatically by planners. Without generating the plan automatically, users need to manually program a different sequence of actions for every task and for every different garden layout. Using the PDDL, a sequence of actions which otherwise requires complex change manually now only requires changes to several predicates that describe the plant location in the PDDL file, then a different sequence can be generated. Therefore, it has great advantage to use AI planning to automate the Farmbot and address the current limitation of requiring manual programming of the task-solving sequences.

The aim of this research is to present an appropriate domain modelling using the PDDL language and, enable the use of automated planning to solve common tasks on Farmbot (i.e. placing seeds to all available positions; checking soil moisture for all plants; watering all plants, etc.). However, such plans will only be text output by the planner program if we cannot instruct Farmbot to execute the plan, therefore, we also aim to integrate the planning (the PDDL and plan generation) with the actuation (the plan dispatching and controlling of Farmbot) with the ROSplan framework [17] and design the appropriate action dispatch interface between ROSplan and Farmbot’s API. We immediately recognize a challenge: since the ROSplan accepts only one PDDL domain file, it is necessary to model only one single domain that incorporates all the possible predicates and actions. Each domain action should also be effectively modelled to correspond to an action that can be executed by the Farmbot, so that the plan dispatching, which sends out instructions to the Farmbot to execute every action in a sequence and the feedback mechanism, which reports the success of current action and require the next action to be dispatched (or report failure and halt the system) can be designed with minimal complexity. We will discuss the details of PDDL and ROSplan later in the

literature review. PDDL has many extensive features beyond the strict classical planning model to support multiple levels of expressiveness and utility. These extensive features often result in a more complicated problem that requires additional problem solving effort. Action cost plan metric is one of the extensive features we are interested, which enables us to find a more optimal plan but require additional modelling effort and more sophisticated and expensive planners to solve. We will devise two PDDL domains, one pure STRIPS domain and one with the additional action cost plan metric. A benchmarking experiment will be conducted between these domains and their corresponding state-of-the-art planners. We aim to find a trade-off between the advantage of the extensive domain and the complexity of solving such a domain. In other words, investigating whether the additional PDDL feature brings us enough advantage in modelling Farmbot and compensate the extra modelling and problem solving effort. Furthermore, we aim to propose an agent planning program (APP) [18] over our proposed PDDL domains, which allows us to synthesize a transition system so that our Farmbot does not halt after a single plan is executed, but instead, carries out consecutive plans even in an infinite loop if necessary. For example, with only the PDDL and the planner, only one plan can be generated and executed at a time. With APP, the agent can synthesize multiple planning problems and execute the plans continuously and be able to make decisions on the next planning problems. Therefore, many tasks can be completed one after another based on the defined transition rules (i.e. placing seeds to all available positions, then watering all of them, then removing the weeds around all the seeded positions, etc.). However, introducing the APP will introduce another challenge to our domain modelling: we need to take into consideration the resulting world state of a plan, meaning that the outcome of the domain actions in a plan should satisfy the condition in the goal state that allows the existence of the precondition for any possible next plan. We will explain this in detail later. Finally we will evaluate the automated system using a simulator we develop for this research. Overall, we aim to show the feasibility of automating the operation of Farmbot through the use of classical planning and PDDL, also to be the first work that proposes the relevant problems to the best of our knowledge. With the success of this research, in the future, we might be able to extend the system to a more sophisticated, fully automated farming solution that can be applied in many areas of agriculture.

This thesis is organized as follows. We will first conduct a literature review that covers important existing works, these include the background and the relevant technologies, the works that contribute meaningful knowledge to our work, the gaps in the current knowledge that motivate our research, and the problems and challenges we will address. We then provide a short section on the problem statement. After that, We describe problems on Farmbot and propose the classical domain modelling in PDDL based on classical planning model and PDDL 1.2 language standard. Later we extend it to include the action cost

plan metric introduced in PDDL 2.1. After that, we propose the integration architecture between PDDL and Farmbot through Robot Operating System (ROS) [19] and ROSplan [17]. A comparison between these domains will be done through a benchmarking of their corresponding planners while scaling up the problem size and analyzing the relevant performance metrics. Later, we propose the Agent Planning Program for our system, our implementation to integrate it, as well as the adaptions made to the original one proposed in [18]. Last but not least, for evaluating the whole system, we create a simulator in Unity Engine [20]. The result will be used to validate the correctness of the system as well as to select the best domain and the corresponding planner alongside the findings from the benchmarking experiment. Finally, we formally conclude our research, present the outcomes, contributions, significance and the limitations, and discuss any possible future direction to the work.

II Literature Review

As worldwide population growth and urbanization continue in the following decades, it is estimated that by 2050, the population around the world will increase by around 3 billions [21]. This requires a 70%-110% increase in agricultural production to meet the global food demand [22]. However, due to the increasing land required for industrialization, urbanization and people's occupation, the total arable land available by then will be only half of the size as today's [22]. It is important to increase the efficiency of agricultural production and create more farming space from places that are not traditionally considered for agriculture. Small-scale urban agriculture emerges as a solution. It grows food in the urban area on the unexploited space between or on top of the urban buildings and therefore, utilize those unconventional space to increase food production. This form of agriculture is commonly in practice in the developing world as a strategy to increase food security [23]–[25], but is not often considered as a mean to increase the total food production due to several disadvantages. Compared to conventional large-scale agriculture, small-scale farm on top or near the urban building are not easy to mechanize, requiring more labour input and reducing economic viability [26]. Also, the success of urban small-scale agriculture in contributing to the global food production requires more people to be mobilized as farmers in their second job, meaning that if such a farming practice is massively adopted, the majority of the operators will be non-professional farmers who might not have the luxury of time and necessary knowledge to precisely control the growing and harvesting of the plants in their farms. It potentially degrades the economic performance of small-scale urban farming and causes food waste [27].

With the development of precision agriculture, automated agricultural robots become a strong solution to overcome these disadvantages [28]. They are complex integration of digital technologies capable of performing precise control over the agricultural process and minimizing human labour input or intervention. However, the existing automated solutions are mostly based on specific platforms with specialized hardware actuators, such as weeding [13], pesticide spraying [29] and harvesting [14]. Therefore, they do not generalize for all the necessary tasks one needs to perform on a farm. To fully automate a small-scale farm, the individual, family or business owner needs to adopt and maintain all different task-specific solutions at the same time. It will not just increase the cost significantly, but also introduce unnecessary complexity for running and automating in a small-scale farming scenario where such a system is intended to be deployed. Some recent studies looked into the design of the generic robot which can potentially be adapted for different tasks and lower the hardware expense. Such a platform typically features a universal tool mount that can easily switch the physical manipulator and allows great flexibility [28]. Amongst them, a 3-axis computer numerically controlled (CNC) Cartesian gantry robot called Farmbot [30] (shown in

Figure 1) receives most of our attention and becomes the chosen platform for us to conduct this research due to several reasons. It is simply designed and manufactured platform, which gives its relatively low cost. With its universal tool mount installed at the lower end of the z-axis (vertical gantry), it can flexibly switch tools and perform different tasks. With these simple and generic designs, the automation solutions based on the Farmbot platform can potentially be low cost. In addition, it is an open-source hardware, which allows Farmbot and the research around it to be more accessible worldwide. Even though other similar platforms exist [31, 32], Farmbot is the only commercially available, ready-to-use platform that is both open-source and supported with its developer documentation and programming interface, therefore it enables us to focus on automating the platform rather than having to develop our own hardware. Most importantly, contrasting the many study on the hardware design for the generic platform that potentially reduces the hardware expense on small-scale agriculture, the study to automate such a platform remains a gap in the current knowledge.

Fig. 1: Photo showing the simple design of the 3-axis Cartesian robot Farmbot

Currently, there is no specific study on the automation for Farmbot. While other automation solutions exist on other modern agricultural robotic platforms, many of these solutions rely on the programming-based approach [28]. That is, the software that enables them to perform tasks is manually composed. The robots execute individual actions that are manually arranged in a sequence one by one. This approach is often characterized by the lack of flexibility, meaning a small change in the working environment

will require a modification in the existing composition of the actions. Farmbot in its default form is no exception. Users need to program it via the web app by manually arranging its available actions into a sequence and set the timer or event for such a sequence to be triggered (Figure 2), or do so by writing a script which call each of the actions in a sequence through the official API. Although the programming-based approach is a proven way in many existing automation solutions [13, 14, 29], repeating the same approach to automate Farmbot contributes no novelty to the current knowledge. A more elegant approach is to explore methods that can automatically generate such a sequence to complete a task [1]. In artificial intelligence, classical planning [33] is such an approach. According to the definition [1], a problem is an instance of classical planning if it can be formally modelled as $S = \langle S, s_0, S_G, A, f \rangle$. It is made up of a finite set of states S, an initial state s_0, a set of goal states S_G, and a set of actions $a \subseteq A(s)$, where each action a applicable from s, maps one state s into another state s' denoted by transition functions $s' = f(a, s)$. Meaning that transition, or action $a \subseteq A$ enables the transition from state $s \subseteq S$ to state $s' \subseteq S$. The solution is a plan, which is a sequence of actions $a_0, ..., a_n$ that transforms a given initial state s_0 into a goal state $s_G \in S_G$ and leaving a generated state trajectory $s_0, s1, ..., s_G$. Transitions between states are assumed to happen instantaneously and deterministically (transition function $s' = f(a, s)$ with action a will 100% change the world state from s to s', contrasting to an action with probabilistic outcome). The world is fully observable. The cost of each action is assumed to be uniform, which is equal to 1. Provided that if there is an appropriate modelling to any problem, a plan, which is a sequence of actions will be the solution to the problem. In the case of the Farmbot, if we model the problem according to the classical planning model and map each of the available actions on Farmbot to an action defined in the model, a solver to classical planning model will be able to automatically produce the solution, which is a sequence of actions that takes Farmbot from a certain initial state to a goal state and hence, complete a task.

The motivation behind innovating the classical planning model, or in general, the model-based approach is to enable separation of the problem description and problem solving. It allows any problem, applicable to such a model to be described in a standard form that contains the information of the state space, the initial state, the actions available to change states and the goal state. Therefore, a solver, which focuses on problem solving, can take in this standard form and produce a plan composed of the sequence of actions that will accomplish the goal when executed from the initial state [1]. The solver can be independent of the actual problem itself as long as it is able to solve the problem based on this standard form and therefore, enabling people who apply classical planning to real world problems to focus on constructing the model which formalizes the problem and use an existing problem-independent solver to produce the solution. This motivation gives rise to the STRIPS language [3], a language for describing the problem instance

Fig. 2: Example of programming farmbot by manually arranging the available actions into a sequence that complete certain task, using the official web app interface

of the classical planning model. STRIPS further refines the definition of the classical planning model by formalizing how the states and actions should be represented. In STRIPS formalization, the states are represented in propositional logic. A state, including the initial state and the goal state, is represented by a conjunction set of a number of facts (terms 'atom', 'proposition', 'fluent', 'literal' are interchangeable). Facts are Boolean-valued, it is true if it is present in the conjunction set and false otherwise. An action enabling the transition from state to state is done by adding new facts that are true in the new state, and deleting facts that are no longer valid, and the action is only applicable from that state if the precondition (also a set of facts) of the action is present in the conjunction set representing the current world state. A problem in STRIPS can be formalized as a tuple $P = \langle F, O, I, G \rangle$ where F is a set of all facts, $I \subseteq F$ is the initial state, $G \subseteq F$ is the goal state, O is the set of all transitions, or actions. Each action $o \in O$ is represented by an Add list Add, a Delete list Del and a Precondition list Pre where $Add, Del, Pre \subseteq F$. The problems described in STRIPS are usually converted to graph search problems, where states are vertices, the applicable actions between states are directed edges connecting the 'from' states and the 'to'

states. The initial state is the starting vertex of the graph search algorithm, and the algorithm terminates when the vertex of the goal state is found, and the path the algorithm traverses from the initial vertex to the goal vertex is returned as the plan (as illustrated in Figure 3). This powerful description language allows any classical planning problem expressible using STRIPS to be solved by common graph traversal algorithms such as breadth first search (BFS) [34] and depth first search (DFS) [35]. However, different from conventional graph traversing problems, the planning problems usually have a very large state space. The complexity of blind search algorithms like BFS and DFS will scale up very quickly and might run out of memory (for BFS) or be trapped in a cycle (for DFS) [33]. With efficiency and safety in mind, heuristic search algorithms [36] like A star search (A*) [37] and enforced hill climbing (EHC) [33] become more common as the solver for STRIPS problems. The use of heuristic search reveals another powerful aspect of STRIPS, which enables delete relaxation and efficient inference of well-informed heuristics such as h_{Add}, h_{Max} and h_{FF} [33].

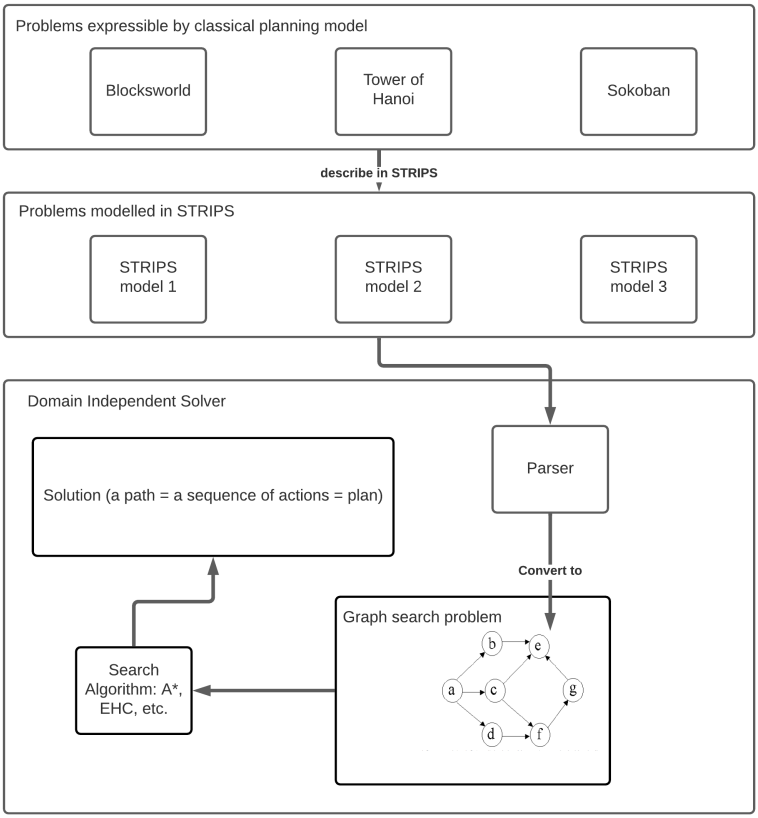


Fig. 3: Illustration of how STRIPS allow domain independent solver to automatically generate plan for the problem modelled in STRIPS language

The implementation of STRIPS gives rise to the planning domain definition language (PDDL) [1].

PDDL is a super-set of the STRIPS language, and slightly more generalized. The oldest PDDL standard that is still in use: PDDL1.2 [38] is functionally identical to STRIPS. It phases out the STRIPS due to its more flexible syntax that reduces hardships and tedious works in modelling classical planning problems. A PDDL model typically separates the representation into two files: a planning domain and a planning problem. The domain file defines the actions and the predicates, whose parameters are instantiated on objects defined in the problem file to form the set of propositions and set of ground actions [2]. The problem file defines the initial state, the goal state and the objects that are used as the parameters for the predicates and PDDL actions defined in the domain file [2]. Allowing propositions to be instantiated from fewer predicates with parameter objects is a great step forward in reducing repetitive works in modelling. Separation of the definition of domain and problem allows multiple planning problems with different initial states, goal states and objects but with the same proposition and action templates to be instantiated on the same planning domain, further reducing tedium in modelling effort. The solver program of the PDDL is called the PDDL planner. A PDDL planner usually consists of a parsing and knowledge compilation component that convert the given PDDL models (domain and problem files) into a specific data structure representing the search problem, and the problem solving component which will apply one or more solving algorithms that take in the search problem and output the plan (for planners with multiple solving algorithms, the planner will usually apply the default one first and switch to others if it fails, or start with a specific one if users instruct).

For each PDDL standard [5, 6, 38], there exists a range of different planners that can solve the problem modelled by the PDDL of that specific standard [39]–[43]. However, the choice of planner depends on the PDDL standard and the language features used for modelling the actual problem. Different features are supported by different planners and some planners only support part of the syntax, features and assumptions set by a certain PDDL standard. Generally, planners for higher PDDL standards are backward compatible with earlier PDDL standards. However, using a planner designed for higher PDDL standards with more features on a search problem modelled in lower PDDL standard is often unnecessary and wasting computational resources. This is because planners for higher PDDL standards usually have more complicated knowledge compilation that results a much difficult search problem. That is to say, when using the PDDL to model problems, it is better to always consider features available for lower PDDL standards and move on to higher standards only when necessary. There always exists a trade-off between the utility of different PDDL standards and the computational cost required by the corresponding planners. Also, for planners that are all designed for the same PDDL standard, the performance difference exists. The report of each international planning competition (IPC) [4, 44] is considered an important literature

that indicates the best performing planners given certain PDDL standards. Combining with the existing literature specific to each planner, including the the baseline and the state-of-the-art [39]–[43], researchers who adopt PDDL to model their problems of interest are generally able to choose the top performing planners for computing plans. However, the benchmarking results in the existing literature are generated on some typical PDDL models and scenarios such as blocks-world [4]. These models are designed for the purpose of competition and the performance results of planners on these models cannot be generalized to all other domains, especially on new domains proposed by various researchers. Therefore, a common practice observed from existing literature [8]–[11] that proposes a PDDL model on a new problem is to select a few planners reported to have good performance and benchmark them on their proposed PDDL models.

PDDL 1.2 [38], which is the earliest standard from 1998 that is still widely supported, can model only the pure STRIPS problems. While almost all existing planners can solve the simplest STRIPS model, the Fast-Forward planning system called the FF planner [39] is the most popular one designed to efficiently solve this PDDL standard. It is considered a standard planner to compare with as a baseline in research proposing new planners [40, 45] and still remains a robust, excellent up-to-date planner for suitable problems, particularly where EHC, h_{FF} and delete relaxation provide enough guidance. FF planner parses the PDDL and represents the search problem exactly in pure STRIPS, a conceptual search graph is then established where vertices are states represented by the conjunction of Boolean-valued propositions, connected by edges representing valid actions that enable transitions between states. The planner then apply enforced hill climbing (EHC) guided by h_{FF} heuristic function [46] to find a solution path through such a search graph that maps to a plan leading from the initial state to the goal state. When EHC fails to find a plan, a complete greedy best first search (GBFS) [33] is triggered, guided by the same h_{FF} heuristic. As the knowledge compilation of FF planner is comparatively efficient and remains consistent across all its derivatives and extensions, the complexity of the FF planner depends mainly on the heuristic search algorithm. More specifically, the part that computes the heuristic value at a given state. The heuristic function is essential to the heuristic search algorithms used in the FF planner, and acts as a 'guide' that leads the algorithm to explore the state that is closer to the goal state first [47]. A good heuristic function should achieve a balance between computational complexity and informedness. For example, a heuristic function that always returns a constant value will be fast to compute, but it informs nothing as the algorithm will not be able to tell the difference between any two states that it might explore next. On the contrary, a heuristic function that always return perfect information such as exactly how far the state is from the goal state will be very difficult to compute as it is equivalent to solving the entire search

problem from any state. The h_{FF} heuristic utilizes a strategy called delete relaxation [33, 46] that exploits the STRIPS representation of the problem. It constructs a relaxed version of the original problem by ignoring the effect of the Delete list Del of all the actions in the original STRIPS problem. The resulting relaxed STRIPS problem will be much easier to solve, therefore h_{FF} explicitly solves this relaxed problem and return the cost of the solution plan of this relaxed problem as the heuristic. With this strategy, h_{FF} achieves a excellent informedness while still being efficient enough to compute. However, it still results in a high computational time complexity: on average more than 70% of the time spent by the FF planner on any PDDL model is used to compute h_{FF} and hence, it becomes the major performance bottleneck of the FF planner [39].

Recently, a new planner called best first width search (BFWS) [40], which is also designed for PDDL 1.2 introduces the new concept of 'novelty'. While using the same knowledge compilation which parses the PDDL files and establishes the search problem as the FF planner, it applies a different class of search algorithm also named BFWS. BFWS, unlike heuristic search, can be seen as a blind search algorithm. However, different from other traditional blind search algorithms like BFS and DFS which explore the vertices based on a FIFO (first in, first out) and FILO (first in, last out) queue respectively, BFWS explores new vertices based on a priority queue who rank the states in increasing order of the novelty (state/vertex with smaller novelty get explored first). As in STRIPS, states are represented by the conjunction of propositions, 'novelty' of a state is the size of the smallest subset of that conjunction of propositions that have not been visited by the search algorithm before. Take the example from the popular planning domain Blocks-world, if an initial state consists of the propositions: on(B, Table), on(A, B), on(C, A), on(D, Table), clear(C), clear(D), which means that block B is on the table, A is stacked on top of B, C is on A, and nothing on C, D is on the table alone and nothing on D, then the next possible state with the set of propositions: on(B, Table), on(A, B), clear(A), on(D, Table), on(C, D), clear(C), which is reachable from the initial state by the action that removes C from A and place on top of D has a novelty of 1 as the size of the smallest subset that is 'novel' (never seen in any state previously visited by the algorithm, can be either on(C, D) or clear(A)) is 1. Novelty can be larger, for example 2, if every subset that contains only a single proposition is seen, and any subset that has size 2 such as the conjunction of both on(B, Table), on(A, B) is never seen, or in other words, both propositions never appear together in previously visited states, and so on for higher novelty. The novelty of a state is independent of the goal state. It rather captures how 'novel' a state is with respect to the past. Smaller novelty means that the state is more 'novel' as it can bring a unique feature to the search, and avoid the states that have an larger overlapping conjunction set of propositions with previously visited states because any previously visited

state cannot be the goal, otherwise the algorithm already returns with a plan. Similar to the concept of goal-counting heuristic, BFWS, which prefers to explore the state with smaller novelty first, will tend to explore the parts of the search graph that are away from the non-goal history and hence, closer to the goal. The advantage of BFWS is that the novelty is extremely fast to compute, but at the cost of being less informed than h_{FF}. In theory, BFWS should compute the plan faster in time than FF planner but will tend to explore more states and return a less optimal plan due to the less informedness of novelty compared to h_{FF}. The advantage of BFWS in time complexity is supported by evidences from original literature [40]. However, there is no direct comparison showing the disadvantage of having a less informed novelty to guide the search algorithm. It becomes a small interest in this project for us to investigate it when we compare these two planners later in our experiment.

Although many existing works put a tremendous amount of effort into improving the planners for pure STRIPS PDDL 1.2, its application value is still compromised due to the lack of many features required for solving real world problems. Therefore, PDDL2.1 [5] was proposed to extend the syntax set in PDDL1.2 and allowed more expressive features beyond the pure STRIPS model. One of the most important features in PDDL2.1 is the plan metric such as the total plan cost. In PDDL1.2, action cost is assumed uniform as 1 and ignored by planners. They succeed if they find a valid plan and disregard whether it is the optimal path in the search problem. But in many real world problems, it is often desirable to take into consideration of the total plan cost. plan metric in PDDL2.1 enables modelling of the action cost of individual action in the STRIPS and hence, provides a parameter for planners to find the plans that optimize certain metrics such as cost. Even though different from the traditional STRIPS classical planning model, this feature still remains in the realm of classical planning as it can be achieved by assigning weight to the edge in the search problem without fundamentally changing the knowledge compilation. However, this introduces new challenges to planner design: solving optimal plan will require the planner to explore and check all possible plans in the entire search space, and the search space in planning problems is typically very large [48], making the exhaustive search approach for getting the optimal plan unrealistic. Although the class of optimizing planners exist [49], the results from the past competition show that optimizing planners tend to take a long time, sometimes up to hours on a small size problem [4, 44]. In the case of Farmbot and other real life problems, where time constraint on solving problems exists, optimizing planners should not be considered due to the time complexity required for finding an optimal plan. Another class of planners for solving the action cost plan metric takes a more realistic 'satisficing' strategy [42, 43]. These planners are often referred to as the 'anytime' planner. The principle of satisficing planner is to accept a user-defined time limit, then the planner repeat the search many times, each repeated search is bound by the cost of

the plan found in the previous search. The planner iteratively improves the quality of the plan until the time limit is reached. It does not guarantee an optimal plan. However, it often finds a relatively good plan, not far from the optimal one if not the same in terms of the total cost, in an acceptable amount of time.

The LAMA planner [42] is one of the most outstanding anytime planners for solving problems with the action cost plan metric. It was the winner of 2008 IPC [50] and still follows closely the winning planners in 2018 IPC [51]. The LAMA planner builds on the Fast Downward planning system [52] and parses the PDDL and represents the search problem in finite-domain, which is a more compact representation compared to STRIPS called SAS+. The biggest difference of SAS+ is that it uses multi-valued variables as opposed to Boolean-valued STRIPS for compact storage of state node information. The LAMA planner also use heuristic search which employs multiple heuristics including a variant of the h_{FF} heuristic and the landmark heuristic [52]. The search algorithm is a variant of A* called weighted A* [33]. Unlike EHC or BFWS, A* considers not just the heuristic but also the cost of the already-traversed section in the search graph. Weighted A*, compared to regular A*, has an additional weight parameter on the heuristic. The LAMA planner iteratively decreases the weight for each repeated search, gradually putting more emphasis on the cost. Therefore the planner continues to search for plans of better quality until the time limit is reached [42]. In 2018 IPC [51], the Dual-BFWS-LAPKT planner [43] emerged to surpass the LAMA planner in the competition. The only difference between them is that the Dual-BFWS-LAPKT uses the BFWS algorithm [40] to complete the first search and find a plan, then switch to Weighted A* and use the cost of the first plan to bound the following search. BFWS and the 'novelty' alone will completely ignore the action cost, but, since it calculates the first plan very fast, it gains Dual-BFWS-LAPKT a faster start compared to the LAMA. However, it is not safe to say that Dual-BFWS-LAPKT is better than LAMA in all domains especially new PDDL models proposed by various researchers. BFWS tends to find the first plan with worse quality than Weighted A*, leaving more search effort to the following search. It still requires comparison on the specific PDDL model to find the most suitable planner.

Although plan metric provides parameters to optimize the plan quality, it is not always necessary. The existing pure classical planners [39, 40] often employ powerful and well-informed heuristic functions or novelty that often guide the search towards an optimal path. In some real world problems, the PDDL standard 1.2 and its specific planners are sufficient enough. Sometimes it is because the domain configuration results in search problems that are simple enough for pure classical planner to find the optimal plan [53], other times there is no such way to define the plan metric or the resulting optimized plan is not much better to compensate the extra time cost and modelling efforts. Using plan metric when

unnecessary will result in increased complexity in modelling and solving the problems, while having no improvement to the plan quality. From the existing works which proposed new PDDL models, we see some of them apply plan metric for optimizing certain aspects of the plan [54, 55] and some of them not [10, 56] even though it can be applied. It all depends on the resulting search problem from the specific PDDL modelling. On Farmbot, it seems like the only place to apply plan metric is the overall distance of the route moved by the Farmbot to accomplish certain tasks. It is worth investigating where to apply plan metric, also finding out whether it is necessary to apply it to model the domain and problems on Farmbot.

These reviewed literature on PDDL above allows us to identify the usefulness of PDDL and give us a strong motivation to apply PDDL to automate the Farmbot platform: besides serving as a well-known standard language for modelling problems and being adopted for solving many real world problems [7]–[11, 54]–[56], it is supported and developed by worldwide AI planning communities over years. Not only these existing supported features across different PDDL standards provide a flexible tool for modelling the real world problems, but also the various domain independent planners [39]–[42, 45, 57] enable us to focus on proposing the PDDL model itself, and choose the appropriate planner for solving the problem. Most importantly, we identify some properties of the nature of tasks on the Farmbot platform that fits perfectly for applying classical planning model and PDDL. Firstly, the tasks on Farmbot are accomplished by sequences of individual actions that are programmed manually, automating the composition of these sequences is a major way to automate the Farmbot, classical planning allows such sequences to be generated as solution plan given that appropriate modelling of the problems is provided. Secondly, the Farmbot can be seemed as a fully observable world with a single agent and deterministic transitions. These are assumptions of the classical planning model, which means that classical planning approach is adequate for the problem. Some literature argues that robotic platforms are usually subjected to a dynamic working environment [28]. Hence, it is often considered as an ideal scenario for applying non-deterministic models in probabilistic planning [58] where transitions have non-deterministic effects, instead of the deterministic one assumed in classical planning model. Despite that, one piece of literature [59] on probabilistic planning competition reveals that a planner based on deterministic planning techniques and re-planning mechanism called FF-Replan outperforms all other probabilistic planners over problems with no critical failure. This further gives us the confidence to apply classical planning to the Farmbot. Even though uncertainty handling is not part of the interest in this paper, future research focusing on uncertainty handling can implement it by introducing the re-plan mechanism similar to FF-Replan [59], based on the classical planning approach adopted in this research, instead of having to adopt a different approach based

on probabilistic planning. Furthermore, our observation on the Farmbot indicates that unforeseen changes to the environment or goals are rare enough, usually that means some mechanical failures which always require human intervention. These reasons make classical planning a suitable approach.

However, we still identify several gaps in the existing literature. Firstly, to the best of our knowledge, no existing research has studied the applications of PDDL in agricultural robots to automate the agricultural process, let alone the Farmbot. This is because the STRIPS model, also the PDDL uses propositional variables to represent the problems. It might not be suitable for many real world problems as their environments should be best represented in continuous values. Using propositional variables will make the modelling process tedious as each unique continuous value will require a unique proposition. Consider a simple planning problem representing a thermometer with temperature ranging from 0 to 40 degrees. Continuous value representation will just need one numeric parameter to indicate the temperature whereas propositional representation will need to define 40 unique propositions, each representing a temperature degree, as well as some accompanying rules saying each proposition should exclusively appear in the world state. Despite that, many existing studies show applications of PDDL across many different domains where traditionally continuous value representation is considered more appropriate [8]–[11, 54]. The lesson learnt from these existing works is that propositional variables are adequate for modelling problems which involve continuous aspects as long as the numeric changes of these continuous aspects are not the interest of the modelling. In robotics, where the physical states are often considered continuous, some studies looked into applying PDDL to model the robotic domains [60, 61]. The PDDL domain modelling in these studies usually maps each physical state of the robot in the environment to a STRIPS state represented by a set of different propositions, these propositions describe all the objects, including the robot in the environment, as well as their situations and relationships. Each state transition maps to an available action that can be taken by the robot to manipulate these objects, their situations and relationships in the environment. The modelling of Farmbot problem in PDDL should be guided by the same approach and model each state transition to an action available on Farmbot according to the official programming API (The web app mentioned above is just a graphical programming interface that calls the API). Furthermore, the existing applications of PDDL show that each different problem requires unique modelling that is dependent on the nature of the specific problem. It is very likely that the approaches these existing works used to model their problems will not be useful for us, we will have to devise the PDDL modelling without taking reference from how other works modelled their problems.

Secondly, besides the lack of direct reference on the construction of the PDDL model on Farmbot, most

existing works related to robots only present the formal PDDL model and rarely combine the planning with execution that allows the robotic platform to execute the plan. The approach to automating the composition of the sequence of actions on Farmbot should not only allow the sequence of actions to be generated automatically through AI planning, but also enable automatic deployment and execution of the generated plan. Recently, a study benefits us by proposing ROSplan [17]. ROSplan is a framework based on the robot action execution interface provided by the robot operation system (ROS) [19], and integrates the AI planning with plan execution. This framework allows us to embed PDDL models and planners into controlling software and provides a mechanism that parses the plan output from the planner into actions in a queue waiting to be dispatched to the robot. ROSplan provides a useful tool to combine planning and execution, but it will require users to implement the actual action interface between the ROS side and the robot side. On Farmbot, this action interface will need to take in each action from the queue, call the API on Farmbot to execute the action and monitor whether the single execution succeed or not, then provide the feedback back to the top level dispatch cycle to request the next action to be dispatched from the queue or abort. The Farmbot API is designed for simple scripting that calls individual action, but not for integration with other software. Therefore, it does not provide a good interface structure to enable such ”dispatch and feedback” loop, particularly an easy way to query the feedback on execution in an anytime blocking manner during the loop. Designing an action interface that connects the ROSplan with Farmbot could be one of the implementation challenges. In addition, the case study introduced alongside the ROSplan [17] shows that the PDDL model embedded with ROSplan should be engineered to not just capture and express the problem, but also work with the ROSplan framework. ROSplan has an important component called knowledge base, which stores all the propositions, predicates and actions defined in the PDDL domain as well as the information of current state and goal state from the PDDL problem [17]. Once the program starts, the knowledge base will be instantiated by the PDDL domain file and the PDDL problem file. Due to this implementation, every running instance of the ROSplan accepts only one PDDL domain, meaning that all the possible problems on Farmbot should be condensed into one PDDL domain instead of many. Therefore, it is necessary to incorporate all possible propositions and actions in that one single PDDL domain modelling. The downside is that every time a planner computes a plan, there will be many propositions and actions that are not needed for that specific plan but have to be included in this one PDDL domain, which increases the complexity of the search problem (more possible state nodes and edges/transitions between states). This brings challenges to the PDDL modelling as it will require appropriate design of that one PDDL domain file so that it covers all necessary aspects of the problem while being as concise as possible in terms of the predicates and actions. Furthermore, the knowledge base, mainly the information of current state, will be updated as the plan dispatches (added or removed

of a series of propositions from the Add and Del effect of actions, therefore move from the initial state to the goal state). However, ROSplan provides additional services (ROS service) [19] to add or remove proposition so that the state change can happen not only after the action defined in the PDDL domain has been dispatched. Compared to a PDDL-only system where the world state can only be altered by the Add and Del effect of the actions, and the same initial state and goal state will always result in the same plan and hence, same changes to the propositions in the world state. These additional services provided by ROSplan allows us to attach different Add and Del effect to the same action (for example, a PDDL action on Farmbot could be to check soil moisture, the Farmbot can use the services to add different propositions to the world state according to the sensor reading, in addition to the Add and Del effect of such an action), giving extra flexibility to the PDDL modelling.

Thirdly, an important observation on Farmbot is that many functionalities are not achievable by one single plan. For example, a complex task can only be achievable through a series of plans, it will be first planting several seeds in the garden bed, then switch to monitoring the moisture level of the soil around them and watering them if necessary, this monitoring and watering can potentially be in a loop that runs continuously until the plants are ready to be harvested. This requires continuous planning and execution of many different plans, each has a different goal state. More importantly, the agent should be able to decide the next goal, switching to a different planning problem to allow the continuous generation of consecutive plans. Currently in PDDL, each domain and problem only determine one planning problem and hence, aim to find a single plan that achieves one goal at a time. Therefore, it has limitations when tasks cannot be expressed in terms of a single goal [62]. Recently, a study proposed the concept of Agent Planning Program (APP) [18], which is a continuous planning model for solving multiple planning problems and generating multiple plans that are required for certain real-world planning applications. The conceptual model of APP is technically a transition system based on finite state machine/automaton (FSA) [63]. An APP over a planning domain is defined by a tuple $A = \langle F, V, v_0, s_0, D \rangle$, the F is a set of facts, or propositions of the planning domain, V is a set of program states, v_0 is the initial program state, s_0 is the initial world state, and D is a set of transitions between program states. Each program state maps to a node in the APP (FSA) model. Each transition maps to an edge that represents a single planning problem whose plan takes the agent from one program state to the other. The autonomous agent starts from the initial program state, then traverses through a sequence of transitions according to the definition of the APP. Each transition has its own transition goal and hence, the agent accomplishes a series of goals one after the other. Each transition $d \in D$ can be formally defined as a tuple $d = \langle v, \gamma, \psi, \phi, v' \rangle$. v and v' correspond to the initial and the goal state of that planning problem. γ is the $guard$, which is essentially

the precondition, represented by a set of propositions that must hold in the current world state: the initial world state of the corresponding transition so that the transition is valid. ϕ is $achievement\ goal$, which is a set of propositions that must hold in the resulting world state: the goal state of that transition. ψ, is called $maintenance\ goal$, which is essentially the invariant propositions of the transition that much be true before, after and during the state trajectory of the transition. Figure 4 depicts an example APP for an academic's daily life routine taken directly from [64], it only shows the achievement goal of the transitions.

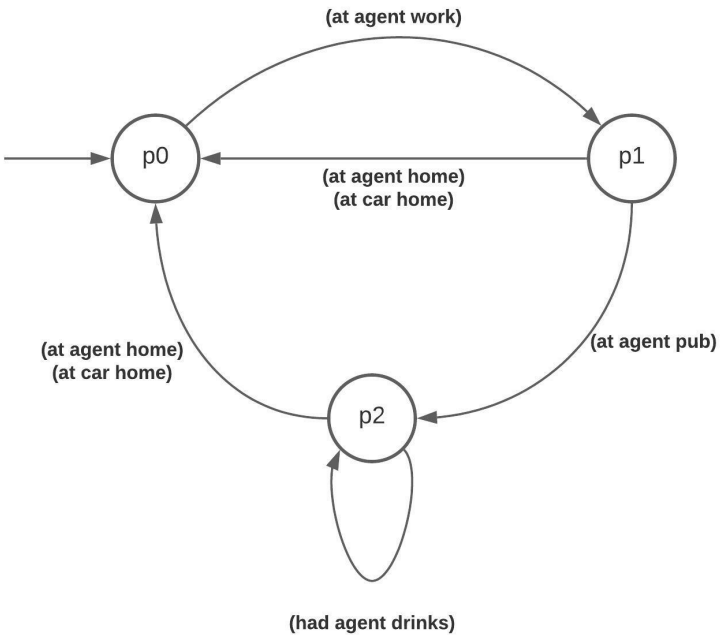


Fig. 4: An example APP describes the daily routine of an academic with only the achievement goals

However, the existing work of the APP is essentially a higher-level representation of problems with complex goals that cannot be achieved by just one plan while still remaining in the realm of classical planning [62]. This might not be directly applicable to the problems on Farmbot as what we need is a goal reasoning mechanism [65] that allows the agent to dynamically determine the next goal based on conditions in the current world state after the previous plan is finished and hence, formulate a series of planning problems. In the original literature [18], the solution of an APP instance is called a $realization$, which is a policy function $\sigma(v, s, g)$ that maps a program state (node in the APP) v, a world state in the planning problem s and a transition with achievement goal g (which is essentially the goal of that planning problem corresponding to that transition) to a plan π that will achieve g from s. This policy function

is pre-computed through an offline plan realization algorithm [62], meaning that the agent has a global view in which all the goal states are known and plans are pre-computed too. Therefore, the dynamic goal reasoning is not achievable through the offline plan realization. However, this is not saying that the APP is not useful, we observe an important property that is never pointed out by any previous literature, that is the APP can be used for goal reasoning if there is a way to adapt the plan realization to an online algorithm. That means, upon the transition to the next program state, instead of looking up the pre-computed policy function for the next plan, the agent will dynamically determine the next transition goal, and compute a valid plan via an 'online' approach. The main challenge to apply APP and enable the Farmbot to achieve complex goals that require synthesizing multiple planning problems is that we need to devise such an adaption to the original work. The notion of online realization was proposed in [62], but it is unlikely to be applied to this project without adaption. Furthermore, having to compute the plan online as described above means that the only goal-determining factor will be the $guard$. It is possible that some APP will have two transitions with identical conditions in the $guard$. This might require us to extend the existing tie-breaking strategy for agent to reason the next goal. Other challenges we identify include having to design the PDDL modelling in favor of the application of APP. More precisely, to allow consecutive planning problems to be produced in the APP, careful design of the actions and predicates in the PDDL domain modelling will be required so that the state trajectory of the plan in the previous transition should result in a world state that satisfies the precondition, or the guard γ of the next transition. Also, implementing a working program that combines its theoretical algorithms and actual controlling interface based on ROS can be challenging as we observe no literature for reference that applies APP in a real-world problem beyond theoretical study. But, the actual implementation of embedding an APP model on top of the PDDL model could benefit from the ROSplan [17], which provides those ROS services mentioned above that allow the APP to query the knowledge base for the guard γ, maintenance goal ψ and achievement goal ϕ and therefore, use it for deciding the next goal and determining the current state.

For evaluation, the existing literature focusing on proposing new PDDL models often conducts benchmarking experiments by selecting several planners and applying them on the proposed domains [10, 11, 54]. Some important metrics collected include the time taken to find a plan by different planners, as well as the quality of the solution if the action cost plan metric is enabled. To thoroughly test the capability of planners on new domains, some papers [10, 55] will scale up the complexity of the planning problem by increasing the number of objects, therefore increasing the number of states (node) and the number of available actions (edge) in the search problem to test the scalability of planners. In works that are interested in finding the appropriate PDDL modelling [54, 55], authors propose several PDDL domains

across different PDDL standards, and compare them through various experiments including comparing the performance of the planners on each domain for those different PDDL standards. The most appropriate domain is selected not just based on the performance metrics of their suitable planners, but also the expressiveness and utility of the domain and the relative effort to model it. When real-world scenarios are involved in the PDDL modelling [8, 54, 60], simulation is often seen as an experimental method. However, the exact data to collect from the simulation will depend on the actual experiment design. Generally, the experiment, particularly data collection and analysis is often not the most important part in papers that propose new domain modelling [8]–[10, 54, 56]. Often it is used to certify the claim that their PDDL approach is a valid one instead of using it for quantitative analysis. However, they are enough to inspire the design of our experiment and evaluation method that allow us to report the most appropriate PDDL domain modelling and the suitable planner for it. Furthermore, no existing experiment can be found on evaluating the application of the APP (because there is no paper that applies APP to real-world problem), which is another gap in the current knowledge that we will need to address in this project.

III Problem Statement

The aim of this research is to propose and evaluate a working solution to automate tasks on the Farmbot platform based on the PDDL and classical planning approach. The PDDL is a useful and powerful language for modelling problems expressible in classical planning model, the solution to such a model is a plan, which resembles the manually programmed sequence of actions on the Farmbot platform for carrying out the tasks. With the appropriate PDDL model, we should be able to automatically generate these sequences and automate the works on Farmbot. However, this has not been done by any previous research. This gap in the current knowledge motivates us to open up a new research area across the realms of AI planning and digital agriculture, and propose any useful methodology for the future study. We divide our main goal into 3 research questions, along with hypotheses and aims of each of them:

RQ 1: Can we successfully model the appropriate domain on Farmbot using PDDL to enable the automatic generation of task-accomplishing sequences of actions by domain-independent planners?

PDDL is based on STRIPS formalization, which uses propositional variables. Hypothetically, any real world planning problem can be modelled in Boolean-valued propositions, but there is great cost in modelling effort for those problems involving continuous values. Farmbot potentially involves continuous values, which is the Cartesian coordinate of the objects in the garden bed. But, another hypothesis is that as long as the changes of values are not essential for planning, these numeric objects can be treated as propositional objects. We will address this by proposing the actual domain modelling. However, this research question will not be fully answered until a series of evaluations are carried out. First, we will formulate a set of planning problem scenarios and run the planner to validate if the expected plans are produced. Then, we can further evaluate the correctness when we conduct the benchmarking experiment. Finally, we deploy plans by integrating with the controlling framework. i.e. ROS and evaluate through simulation and answer whether the plans are correct when they are used to instruct the Farmbot.

RQ 2: Does the action cost plan metric allow us to improve the plan quality significantly? Does the improvement compensate the extra modelling effort and planner complexity?

In PDDL 2.1, the classical planning model was extended to allow the plan metric, which can be used for the planner to optimize the plan. However, this will require extra modelling effort to extend the pure classical domain and also need to apply a different class of planners which are more computationally expensive. Hypothetically, whether the plan metric can be effectively applied to the planning domain, and whether the plan metric is actually useful depends on the actual problems. After proposing the pure classical planning domain, we will extend it to a domain that uses the plan metric. We will find out how

it should be appropriately applied in order to have meaningful semantics. We will compare these two domains by conducting a benchmarking experiment using two state-of-the-art planners for each domain while scaling up the planning problems. We will also deploy the plans to simulation so we can evaluate how they perform when actually controlling the Farmbot. The combined evidence will be used to draw conclusions that address this question and report the most appropriate domain out of these two (pure STRIPS or with plan metric). Also the most appropriate state-of-the-art planner for the selected domain will be reported (if applicable).

RQ 3: Can the agent planning program (APP) be adapted for goal reasoning to synthesize multiple planning problems over the proposed domain on Farmbot?

An important observation on automating the Farmbot is that the automation should allow Farmbot to achieve a series of goals by continuously executing many plans instead of just halting after one goal is achieved. The original PDDL has does not support such a feature as described in the literature review. Relative to AI planning technology in general, APP has the potential to be adapted for goal reasoning. We aim to implement and adapt the original APP to accomplish this aim, and integrate it with the ROS-based controlling software deployed over the simulation to evaluate its correctness.

IV The PDDL Model of Farmbot

In this section, we describe the tasks on Farmbot that can be expressed by planning problems. We will demonstrate how we model the domain in PDDL by providing some fragments of PDDL file and the explanations on the semantics and decisions. We first model the planning domain in pure STRIPS formalization that is available in PDDL 1.2. Then, we extend it to PDDL 2.1 and introduce action cost to the planning domain. Lastly, we describe some scenarios and formulate the corresponding planning problems over our domain, we will utilize an online planning environment to solve these problems. The purpose is to examine and evaluate whether the domain modelling allows the generation of expected plans. The result will be used as part of the mixed evidence to support the answer to our RQ 1.

*A. **Planning Problems on Farmbot***

Farmbot, specifically the model Genesis XL [66] (Shown in Fig.1), is situated on a garden bed of 1.8m x 6m. Farmbot is a 3-axis Cartesian robot, meaning that the rails and the gantries span across the entire garden. However, the exact position of the Farmbot is represented by the position of its tool tip, or actuator on the lower-end of the z-axis gantry. Besides the main body of the Farmbot, a separate tool bay is installed on one of the sides of the garden bed, holding a seeder, a weeder, a watering nozzle, a soil moisture sensor and a seed container.

The tasks on Farmbot are accomplished by executing sequences of discrete actions. These sequences are manually programmed by arranging the available actions provided by the Farmbot API. Despite that many unique sequences can be composed by rearranging these actions in different manner, the essential tasks on Farmbot can be categorized into 5 classes as shown in Table I. All different sequences can be seen as the variations of these 5 categories due to different garden layouts with different location and number of plant. The manual programming approach will need to adjust these sequences according to garden layout each time even though they are performing the same essential task.

Therefore, the planning problems on Farmbot is to automatically obtain the correct plans that map to those sequence of actions in Table I by providing proper representations of the initial state and the goal state of the planning problems, the objects involved in the planning problems, the planning domain and the operators (actions). In PDDL, the initial state and the goal states are represented by a set of propositions, the initial state is the initial conditions of the world state, the goal state is the conditions that need to be satisfied. Each action is an operator that changes the world state by adding or removing some propositions from the set. The plan generated by a domain independent planner will be the sequence of actions that

TABLE I: Essential tasks on Farmbot

Task	General pattern of the sequence of actions to complete the task
Place seeds on all desired locations	Start (move to seeder's location) (pick up seeder) (move to seed tray) (pick up seed) (move to first location) (place seed) .. [repeat depends on garden layout] .. (move to last location) (place seed) End
Check moisture of soil around each plant	Start (move to moisture sensor's location) (pick up moisture sensor) (move to first plant) (check moisture) .. [repeat depends on garden layout] .. (move to last plant) (check moisture) End
Water all plants	Start (move to watering nozzle's location) (pick up watering nozzle) (move to first plant) (turn on water pump) (turn off water pump) .. [repeat depends on garden layout] .. (move to last plant) (turn on water pump) (turn off water pump) End
Detect weed around each plant	Start (move to first plant) (detect weed) .. [repeat depends on garden layout] .. (move to last plant) (detect weed) End (notes: no need of picking up tool because the camera is always attached to the gantry)
Remove weed around each plant	Start (move to weeder's location) (pick up weeder) (move to first plant) (move to weed 1 of the first plant) (remove weed 1 of the first plant) (move to weed 2 of the first plant) (remove weed 2 of the first plant) ..(move to weed x of the first plant) (remove weed x of the first plant) .. [repeat depends on garden layout] .. (move to last plant) (move to weed 1 of the last plant) (remove weed 1 of the last plant) (move to weed 2 of the last plant) (remove weed 2 of the last plant) ..(move to weed x of the last plant) (remove weed x of the last plant) End

change the initial world state to the world state that meets the goal condition.

To represent these tasks in the PDDL, we need to consider how all these aspects in the environment can be expressed in propositional variables. All physical objects, including these tools and the Farmbot as well as the plants grown in the garden bed, the seeds in the seed container, etc. can be represented as the PDDL objects in the planning problems. Their situations and relationships which describe the world state (including the initial state and goal state), such as if Farmbot is currently equipping with any tool or the seed container is containing some seed objects, can be defined using predicates that are instantiated on these objects. Representing the location and position will require some reasoning. On Farmbot, all positions/locations are represented in a Cartesian grid with 3 dimensions (x, y, z). Intuitively, this should be modelled using some continuous variables instead of propositional variables. However, continuous values are needed only if we are interested in the changes of these values. On Farmbot, we only need position and location to indicate some unique landmarks, so that the positional relationship such as if certain objects like plants or tools are at certain locations can be modelled using predicates. Therefore,

modelling positions or locations as atomic propositions will satisfy the need. Also, as we are aiming to combine planning with actuating using ROSplan, the translation of proposition to actual continuous coordinate can be done after the planning, specifically when the plan is dispatched to the actual controlling software that instructs the Farmbot to move to the exact Cartesian coordinate.

Consider a scenario where 5 plants (A, B, C, D, E) are presented at locations Loc1, Loc2, Loc3, Loc4, Loc5. If we expect to obtain a plan that fulfilled the task of watering some of the plants, then a goal state should describe which plants require watering, and the initial state should describe which plants are already watered, the current position of the Farmbot gantry, the positions of those plants, as well as the equipped tool on the tool mount. For instance, plant A, C, and D are required to be watered, the initial state describes the Farmbot is at location Locf, and is equipped with weeder tool. The described scenario can be written as the following fragment of example PDDL problem. (Notes: it is not syntactically complete, refer to it as PDDL pseudo code)

```
(define (problem example) (:domain farmbot)
(:objects ...)
(:init (plant-at A Loc1)
       (plant-at B Loc2)
       (plant-at C Loc3)
       (plant-at D Loc4)
       (plant-at E Loc5)
       (farmbot-at Locf)
       (carry-tool weeder)
        ...
       (tool-at LocX wateringNozzle)
       (tool-rack LocW weeder)
)

(:goal (and (watered A)
            (watered C)
            (watered D)
        )
)
```

The plan we will obtain from a domain independent planner will contain the following actions:

```
(move Locf LocW)
(put-down-tool weeder LocW)
(move LocW LocX)
(pick-up-tool wateringNozzle LocX)
(move LocX Loc1)
(water A Loc1)
(move Loc1 Loc3)
(water C Loc3)
(move Loc3 Loc4)
(water D Loc4)
```

The above example provided a direction on how the planning problems can be modelled in PDDL. We also observe several implications for the domain modelling:

1) As we are considering the integration with ROSplan [17], the domain will be a single large domain file.
2) According to the description of planning problems above. The domain should be able to work with different planning problems, essentially the problems whose plans fulfill the 5 essential tasks in Table I. These planning problems all have different initial and goal conditions
3) The plan generated should be meaningful for the Farmbot to execute, which means that each domain operator (action) defined in the PDDL domain should map to a real action available on the Farmbot platform.
4) From 1 and 2, the proposition in the PDDL model will need to describe all the conditions observable in the garden bed. These include the positions and situation of the plants, tools, seeds and the Farmbot gantry, etc. Therefore the predicates defined in the domain file should be the descriptive facts or adjectives to the objects (plants, tools and locations, etc.) supplied by the problems.
5) To allow the plans to be generated from the given initial state to the goal state, the actions defined in the domain should allow the existence of these plans, which enable the transition of states that results in the state sequence from the initial state to the goal state.

B. The PDDL Domain of Farmbot

We model the environment of the garden bed as discrete locations represented by individual objects. Other physical objects, including the plants, seeds, tools, tool holders, seed container as well as Farmbot itself are also represented by individual objects. Therefore, the domain will define the scope of those objects by employing typing:

```
(:types
    position
    plant
    seed
    weed
    tool
    toolholder
    seedcontainer
)
```

The predicates in the PDDL domain will describe their position, situation and relationship. Therefore, the domain contains the following predicates:

```
(:predicates
    (tool-mount-free)
    (farmbot-at ?x - position)
    (carry-tool ?t - tool)
    (carry-camera)
    (tool-at ?x - position ?t - tool)
    (tool-rack-at ?tr - toolholder ?t - tool ?x - position)
    (no-plant-at ?x - position)
    (plant-at ?x - position ?p - plant)
    (checked-weed-exist ?x - position)
    (weed-at ?x - position ?w - weed ?p - plant)
    (weed-removed ?x - position)
    (no-weed-at ?x - position)
    (carry-seed ?s - seed)
    (seeder-free)
    (container-at ?x - position ?c - seedcontainer)
    (container-has ?c - seedcontainer ?s - seed)
    (match-seed-type-n-plant-type ?s - seed ?p - plant)
    (checked-moisture ?x - position ?p - plant)
    (need-water ?x - position ?p - plant)
    (not-need-water ?x - position ?p - plant)
    (watered ?x - position ?p - plant)
)
```

These predicates are sufficient to capture several important aspects of the world state. The (farmbot-at) describes the position of the Farmbot's gantry (or tool tip/actuator). This predicate allows defining the initial position of the Farmbot, which is an compulsory definition in the initial state of any planning

problems. Also, we can therefore define the most important state transition operator (move), which is relevant to any tasks performed by the Farmbot:

```
(:action move
    :parameters (?x ?y - position)
    :precondition (and
        (farmbot-at ?x)
    )
    :effect (and
        (not (farmbot-at ?x))
        (farmbot-at ?y)
    )
)
```

This action is interpreted as moving Farmbot's position from x to y, the precondition is that Farmbot has to be at x, after the transition takes place, the Del effect remove (farmbot-at ?x) and the Add effect add (farmbot-at y), which means that farmbot is no longer at x, and now at y. The x and y here are parameters that will be replaced by any two position objects. Furthermore, this action is important as it allows state transition of the Farmbot's physical position because before farmbot performs any action at any given position, it needs to be physically present at there. Therefore, all the following actions will have (farmbot-at) in their precondition, indicating the physical presence of Farmbot's actuator. These actions include the following 2 that utilize the predicate (carry-tool):

```
(:action pick_up_tool
    :parameters (?x - position ?t - tool)
    :precondition (and
        (tool-at ?x ?t)
        (tool-mount-free)
        (farmbot-at ?x)
    )
    :effect (and
        (not (tool-mount-free))
        (not (tool-at ?x ?t))
        (carry-tool ?t)
    )
)

(:action put_down_tool
    :parameters (?x - position ?t - tool ?tr - toolholder)
```

```
    :precondition (and
        (carry-tool ?t)
        (tool-rack-at ?tr ?t ?x)
        (farmbot-at ?x)
    )
    :effect (and
        (not (carry-tool ?t))
        (tool-mount-free)
        (tool-at ?x ?t)
    )
)
```

The (pick_up_tool) action exists because it will allow a state transition for Farmbot to equip itself with specific tools. Before any relevant actions can be performed, such as watering plants or removing weeds, the Farmbot needs to equip with the relevant tool. Another observation is that Farmbot can only carry one tool at a time, therefore the predicate (tool-mount-free) is in the precondition of this action, indicating the Farmbot must not carry any other tool when picking up a new tool. Also, it is possible that Farmbot is equipped with the wrong tool at the initial state, therefore the state transition operator (put_down_tool) is necessary for the agent to re-gain the (tool-mount-free) proposition in its world state. Allowing switching to a different tool.

The predicate (carry-seed) is in the Add effect of action (pick_up_seed), meaning that the Farmbot is currently carrying a seed. It is a precondition in the action (place_seed), indicating the Farmbot must carry a seed before it can place it. Also, the Farmbot needs to carry the seeder tool before it can pick up any seed, hence requiring (carry-tool seeder) in the precondition of the (pick_up_seed) action. Furthermore, it cannot carry any other seed if it already picks up one, as the seeder on Farmbot is a vacuum suction tube with a maximum capacity of one seed and hence, the (seeder-free) predicate is introduced. Therefore, the following 2 actions is defined:

```
(:action pick_up_seed
    :parameters (?x - position ?s - seed ?c - seedcontainer)
    :precondition (and
        (container-has ?c ?s)
        (seeder-free)
        (container-at ?x ?c)
        (farmbot-at ?x)
        (carry-tool seeder)
```

```
    )
    :effect (and
        (not (seeder-free))
        (carry-seed ?s)
        (not (container-has ?c ?s))
    )
)

(:action place_seed
    :parameters (?x - position ?s - seed ?p - plant)
    :precondition (and
        (no-plant-at ?x)
        (carry-seed ?s)
        (farmbot-at ?x)
        (match-seed-type-n-plant-type ?s ?p)
        (carry-tool seeder)
    )
    :effect (and
        (not (carry-seed ?s))
        (not (no-plant-at ?x))
        (seeder-free)
        (plant-at ?x ?p)
    )
)
```

In real life scenario, the seed container only contains a limited amount of seeds, in pure STRIPS representation, we cannot define numeric value so we will need to have objects that represent individual seeds. The predicate (container-has) is the proposition that indicates whether a specific seed is present in the seed container and it has to be presented in the initial state, after it is picked up, this (container-has) predicate for the corresponding seed will be deleted, indicating the seed is consumed. When placing the seed, it is important to know which plant object is added into the world state with the predicate (plant-at). Therefore the (match-seed-type-n-plant-type) predicate is introduced as a necessary guard ('n' is 'and' in naming).

Furthermore, the rest of the predicates are used in the following actions, the semantics are self-explanatory:

```
(:action check_need_water
```

```
    :parameters (?x - position ?p - plant)
    :precondition (and
        (plant-at ?x ?p)
        (farmbot-at ?x)
        (carry-tool soilsensor)
    )
    :effect (and
        (checked-moisture ?x ?p)
    )
)

(:action water_plant
    :parameters (?x - position ?p - plant)
    :precondition (and
        (need-water ?x ?p)
        (farmbot-at ?x)
        (plant-at ?x ?p)
        (carry-tool wateringnozzle)
    )
    :effect (and
        (not (need-water ?x ?p))
        (watered ?x ?p)
    )
)

(:action detect_weed
    :parameters (?x - position ?p - plant)
    :precondition (and
        (carry-camera)
        (farmbot-at ?x)
        (tool-mount-free)
        (plant-at ?x ?p)
    )
    :effect (and
        (checked-weed-exist ?x)
    )
)
```

```
(:action remove_weed
    :parameters (?x - position ?w - weed ?p - plant)
    :precondition (and
        (farmbot-at ?x)
        (carry-tool weeder)
        (weed-at ?x ?w ?p)
    )
    :effect (and
        (weed-removed ?x)
        (not (weed-at ?x ?w ?p))

    )
)
```

These actions are sufficient to compose all the sequences of actions that perform the essential tasks on Farmbot platform. Last but not least, a fully installed Farmbot garden bed will always have the tool bay and all the 4 default tools (seeder, watering nozzle, weeder and soil moisture sensor) available, therefore the objects representing them and their positions will remain constant in the domain file:

```
(:constants
    seeder wateringnozzle weeder soilsensor - tool
    seedtray - seedcontainer
    seederrack wateringnozzlerack weederrack soilsensorrack - toolholder
    seederPos wateringnozzlePos weederPos soilsensorPos posSeedTray - position
)
```

Furthermore, these following propositions are always true in the initial state of any planning problems on Farmbot, indicating the initial arrangement of these tools in the world state. Also, a weed detecting camera is always attached to the Farmbot's gantry, hence a predicate (carry-camera) should always be present in the initial state:

```
(tool-rack-at seederrack seeder seederPos)
(tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
(tool-rack-at weederrack weeder weederPos)
(tool-rack-at soilsensorrack soilsensor soilsensorPos)
(tool-at seederPos seeder)
(tool-at wateringnozzlePos wateringnozzle)
(tool-at weederPos weeder)
(tool-at soilsensorPos soilsensor)
(container-at posSeedTray seedtray)
```

```
(carry-camera)
```

C. *Classical Planning Model*

The first model is a classical planning model in PDDL that employs PDDL 1.2 language standard [38] based on pure STRIPS formalization. Therefore, there are only two requirements in the domain file.

```
(:requirements
    :strips
    :typing
)
```

A full PDDL domain file is provided in the Appendix A. This domain is the same as described in section IV-B above. The domain only relied on propositional variables to describe all the physical objects and their situations. The pure propositional variables are sufficient to capture the world states as well as the actions that enable state transitions.

Though relatively simple to model, there are some limitations: we have to define individual seeds in the seed container as individual objects to prevent the planner from finding a plan that picks up seeds that do not exist. The consequence is that it introduces tedious work when defining the initial condition where the seed container contains many seeds, and also for each seed object, we have to define the accompanying predicates (container-has ?c ?s) and (match-seed-type-n-plant-type ?s ?p). The former is used in the pick_up_seed action as a precondition which only allows the seed to be picked up if it exists in the seed container. The latter is used in action place_seed to specify which plant object is introduced after that specific seed is planted.

D. *Classical Planning Model With Action Cost Plan Metric*

The second model involves the plan metric introduced in PDDL 2.1 [5], it enables the definition of action cost for each operator in the PDDL domain, and the plan metrics for the plan to satisfy in the planning problems. PDDL 1.2 or pure STRIPS model assumes actions to have uniform cost. It is often not true in many real world problems. Therefore, action cost is a very important feature that allows planners to treat actions differently and prioritize certain actions to improve the plan quality while reaching the goal. It is useful when applying to problems where finding a valid plan is not the only requirement but

also the plan with certain metrical properties such as a smaller total cost is preferred.

On Farmbot, completing a task with a sequence of actions that has a smaller total cost is also preferred. In the context of Farmbot, action cost can be the effort required to execute that action. On the physical level, such an effort can be the energy (electricity to drive the motor) or time expense. All the actions on Farmbot will require these two efforts, but not all of them can be optimized using the plan metric. The essential tasks in Table I above generally involve a similar pattern: the farmbot start from a certain position, move the gantry and tool mount to pick up a specific tool if necessary, and repeat the sub-sequence of moving the gantry to each location, performing certain actions and so on. Consider the same scenario we described in section IV-A above, where the initial state has many unwatered plants and the goal state requires all of them to be watered. The plan will involve many actions that are essential for adding the goal propositions into the world state and enabling the goal conditions to be met, including the action that water each plant. These actions must be included in a valid plan and they have no alternative with a smaller action cost. Therefore, the only possible place to optimize the plan in terms of cost is the route that the Farmbot visit each plant and water them. More precisely, minimizing the unnecessary travel distance between all the positions that farmbot will move to in order to complete the task and hence, minimize the extra time and energy spent on a non-optimal traveling route.

The domain with action cost will be extended based on the classical domain in section IV-D above. To include action cost, we first introduce numeric fluent to the domain requirement:

```
(:requirements
    ...
    :fluents
)
```

Then we need to declare the additional section called functions in the PDDL domain. Functions are specific predicates in PDDL domain for the numeric fluent. Because the only meaningful plan metric to optimize is the total travel distance, or the route for Farmbot to complete a certain task as we mentioned above, and the action 'move' is the only action that allows Farmbot to move and visit each of the positions where it have to perform certain other actions to achieve the goal state. Therefore, the action 'move' is the only meaningful action to apply the action cost for planners to opt for a better route, and the cost is determined by the distance of the two positions between which the action move is taken. So, we need two functions, one to keep track of the total action cost, and the other is used to assign the individual cost that is the distance between the 2 position parameters of the action 'move'.

```
(:functions
    (move-distance ?x - position ?y - position) - number
    (total-cost) - number
)
```

Also, we need to modify the action 'move' as follows:

```
(:action move
    :parameters (?x ?y - position)
    :precondition (and
        (farmbot-at ?x)
    )
    :effect (and
        (not (farmbot-at ?x))
        (farmbot-at ?y)
        (increase (total-cost) (move-distance ?x ?y))
    )
)
```

A full PDDL domain is included in the Appendix B. Compared to the classical domain, the only change is the new effect (increase (total-cost) (move-distance ?x ?y)). This is saying when each time the Farmbot moves from position x to position y, the total cost of the entire plan increases by the moving distance between x and y.

In the PDDL problem file, the initial values of these functions should be instantiated in the initial state:

```
(define (problem example) (:domain farmbot)
(:objects Loc1 Loc2 Loc3 Loc4 Loc5 - position)
(:init
        ...
        (= (total-cost) 0)
        (= (move-distance Loc1 Loc2) 23) #the value for each (move-distance x y) is just exam
        (= (move-distance Loc2 Loc2) 21)
        (= (move-distance Loc1 Loc3) 32)
        ...
        (= (move-distance Loc4 Loc5) 22)
        (= (move-distance Loc5 Loc4) 22)
)
```

```
(:goal (and
            ...
        )
)
```

The semantics is to assign the initial total-cost to 0, and assign a cost value between each two position objects that reflects the distance. Then we need to specify the plan metric to optimize by including the following definition in the problem file:

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

Therefore, the planner will keep in mind the value of the total-cost and find a valid plan with the better combined cost of all the 'move' actions in the plan, or the total traveling distance.

The limitations to model the action cost mainly come from modelling the initial state of the planning problems. We have to provide the instantiation of the (move-distance x y) between every two positions in the garden bed. It is equivalent to manually defining the weight of every edge in a fully connected network and it introduces tedious works. However, this way of modelling the action cost is still reasonable as the same instantiations of all the (move-distance x y) apply to all different planning problems as long as the position objects do not change, meaning that if we define it once, we can use it in every problem files. On Farmbot, the positions of plants usually form a logical grid, meaning that it is very likely the position objects will always be the same for each unique garden bed size. Also, the PDDL instantiation for all the (move-distance x y) can be generated by a script which takes in a garden layout defined in texts. We will describe the script later in section V when discussing the experiment where we use a script to scale up the problems and conduct benchmarking on planners for the domain.

*E. **Evaluation***

This evaluation is a functional testing and validation focusing on finding out whether the PDDL modelling of Farmbot we provided above allows the planner to generate plans as we expected. We will formulate 5 scenarios and the corresponding planning problems, and will run planner on each of them and manually examine the details of the plans to see if the correct plans are found. We will only conduct the evaluation on the first PDDL domain: the pure classical domain because the second domain with action cost plan metric is extended from the first one and has mostly the same semantics and modelling. The

main purpose is to find out whether the decisions made on modeling those predicates and the actions in the provided domains are appropriate and can result in the expected plans from the given planning problems. Therefore, if the first domain passes the evaluation then so does the second domain.

Table II describes the scenarios, and the PDDL problem files of the corresponding planning problems are attached in Appendix C.

TABLE II: Scenarios for evaluation

Scenario number	Description
Scenario 1	The goal is for Farmbot to grow 4 plants at 4 different positions. The initial world state indicates that these 4 positions are empty, there are 4 seeds in the container, each for one plant respectively. The Farmbot is initially at a position called 'home' and the tool mount is equipped with no tool. The initial state also describes the available tools and their positions.
Scenario 2	The goal requires Farmbot to water 4 plants at 4 different positions. The initial world state describes which plant at which position, and the available tools and their positions. The Farmbot is initially at a different position called 'home', and it is equipped with the weeder, which is the wrong tool for the current task.
Scenario 3	The goal requires Farmbot to check the soil moisture of 4 plants at 4 different positions. The initial world state describes which plant at which position, and the available tools and their positions. The Farmbot is initially at position 'home' and is equipped with the wrong tool. This time, the goal also requires the Farmbot to return to 'home' after the task is completed.
Scenario 4	The goal requires Farmbot to detect the weed around 4 plants at 4 different positions and return to the initial position. The initial world state describes which plant at which position, and the available tools and their holders' positions. The Farmbot is initially at position 'home', and equipped with unnecessary tool 'weeder'. It needs to get rid of it before its camera can function properly.
Scenario 5	The goal requires Farmbot to remove 3 weeds at 3 positions around 2 plants. The initial state describes the 2 plants and 3 weeds, where 2 weeds are around one plant and 1 weed is around the other. The initial state also describes the Farmbot's initial position 'home' and the available tools and their positions. The Farmbot starts with no equipped tool.

This functional testing is conducted using an online PDDL editing environment [67][1]. This is the most suitable and easy-to-use platform for testing classical planning domains and problems in PDDL. It has a planning service provided by an integrated online planner. The research and infrastructure for

[1]http://editor.planning.domains/

classical planning model are the most mature, therefore we avoid the complication of setting up our own environment.

The results show that all the problems over the proposed PDDL domain allow the domain independent planner to generate the expected plans. The modelling decisions on the domain propositions, predicates and actions are adequate and appropriate. The continuous element, such as the coordinate can be represented in individual propositional variables. The results and observations are displayed in Table III

TABLE III: Results and observation for evaluation

Scenario number	Outcome	Observation
Scenario 1	Plan is found	The plan is correct, the Farmbot start by moving to the position of the seeder, picking up the seeder and then repeat the actions of moving to the seed tray, picking up one seed, then moving to each position to place the seed one by one into each of the 4 positions.
Scenario 2	Plan is found	The plan is correct and is aware of the presence of the wrong tool, the Farmbot will first move to the position of the tool holder of the weeder, put down the weeder then move to and pick up the watering nozzle, then it moves to one of the plants, water it, then move to the next one and so on, until all 4 plants are watered.
Scenario 3	Plan is found	The correct plan is found, similar to the previous scenario, the Farmbot will first put down the wrong tool, and pick up the right one, which is the soil sensor. Then, it will visit all 4 plants one by one and check the soil moisture of them. At the end, it moves back to the position 'home' as it is in the goal state.
Scenario 4	Plan is found	The correct plan is found, and the process is very similar to the previous scenario, the Farmbot will put down wrong tool first, but the difference is that detecting weed does not require picking up a tool as the weed detecting camera is built into the tool mount. The Farmbot will directly move to each of the plants and detect weeds around them. At the end, it moves back to the initial position 'home'
Scenario 5	Plan is found	The correct plan is found, the Farmbot will first move to and pick up the weeder. Then, it moves to the first plant, removes the two weeds around it, then it moves to the next plant and remove the remaining weed.

This evaluation provides sufficient evidence to critically examine our modelling decisions. It shows that our PDDL domains are valid and allow a domain independent planner to generate plans as expected, which

helps us partially answer our RQ1. However, this evaluation has limitations and is not conclusive as only a few representative scenarios are selected to test the validity of the PDDL model, we did not cover all the possible planning problems. In addition, the plan metric in the second domain is not evaluated. Both domains will be further evaluated later in section V and section VIII using their corresponding planners in our own experiment environment, and we will combine all the evidence, including this evaluation as part of the mixed evidence at the end to support conclusive answers to our research questions.

V Benchmarking Experiment

In this section, we conduct the experiment to compare the performance of different planners on the two PDDL domains we present above. These domains contain different features that are supported by different PDDL standards and hence, require different planners. The first domain, which is the classical domain, requires classical planners such as BFWS and FF. The second domain with action cost plan metric requires anytime satisfying planners such as LAMA and Dual-BFWS-LAPKT. The purpose of this experiment is to gather results based on the performance metrics, such as time taken to find a plan and the plan quality of the corresponding planners that are applied to these domains, and analyze these results to gather evidences that help us answer whether the use of action cost plan metric allows us to significantly improve the plan quality compared to the pure classical domain where plan metric is ignored. Also, whether the improvement in total plan cost compensates the extra time expense in finding such an improved plan. The results and evidences we gathered from this experiment will be used to answer mainly the RQ 2, though the correctness of running the entire experiment can be used as an evidence of the correctness of the PDDL modelling, which contributes to answering the RQ 1.

A. *Method and Experiment Settings*

The experiment will be conducted in an automated way using a script written in the Python programming language [68] version 3.6.9. The script takes in a PDDL domain and a problem, then it will iteratively scale up the problem size by inserting new objects and goals, call the planner on this domain and the scaled-up problem to find a plan, and parse the output by the planner to collect useful metrics for analysis.

The planning problems we used for this experiment has the initial state that describes a number of position objects with plants at each of them that require watering. The goal condition is for the Farmbot to water all of them. For the pure classical domain in section IV-C, this problem is modelled as follows:

```
(define (problem experiment-problem) (:domain farmbot)
(:objects
    ...
    position1 position2 position3 position4 position5 ... - position
    plant1 plant2 plant3 plant4 plant5 ... - plant
)
(:init

        ...
        (plant-at position1 plant1)
```

```
        (plant-at position2 plant2)
        (plant-at position3 plant3)
        (plant-at position4 plant4)
        (plant-at position5 plant5)
        ....
        (need-water position1 plant1)
        (need-water position2 plant2)
        (need-water position3 plant3)
        (need-water position4 plant4)
        (need-water position5 plant5)
)

(:goal (and
        (watered position1 plant1)
        (watered position2 plant2)
        (watered position3 plant3)
        (watered position4 plant4)
        (watered position5 plant5)
        ...
    )
)
```

The problem size is defined as the number of position objects in this problem, a problem size of one means that there is only one plant object at a position object with the goal of watering this plant. Increasing the problem size by one will mean to insert an additional position object, plant object, and introduce additional accompanying propositions such as the (plant-at) and (need-water) in the initial state as well as the (watered) proposition in the goal state. This will proportionally increase the instances of the predicates and actions as defined in the domain, thus the number of states (nodes) and actions (edges) in the search problem. Therefore, scaling up the problem. By running the planners on a problem with increasing problem size and gathering performance metrics. We can tell how well the planner scale on such a domain and therefore, provide us evidences on selecting the appropriate domain and planner.

The reason why this planning problem is selected for the experiment is because this problem allows meaningful optimization of the plan cost proposed in the second PDDL domain. Consider an alternative problem such as placing seeds and growing plants at each position, where the Farmbot has to repeat the movement back-and-forth between the seed container to pick up one seed at a time and the desired position to place such a seed. There is no way that the visiting order of these positions can be optimized

because as soon as Farmbot places a seed at one position, it needs to move to a fixed position where the seed container is to pick up the next seed. This is required in the state trajectory to reach the goal in this case and hence, the visiting order does not matter and the total travelling distance is always the same. To support this, we have done a preliminary experiment that runs LAMA planner and Dual-BFWS-LAPKT planner to solve problems that require placing seeds to grow plants on all desired positions as problem size scales up from 1 to 100. These two anytime planners are given enough timeout limit (90 seconds) to optimize the plan. However, almost all the plans are not optimized significantly in term of the plan cost. In particular, the Dual-BFWS-LAPKT could not find an optimized plan most of the time, and LAMA only optimized some redundant steps (such as moving to another position and then moving to the desired next position) because the overall traveling route to visit each position cannot be optimized. Figure 5 and Figure 6 provide the evidence to support our choice.

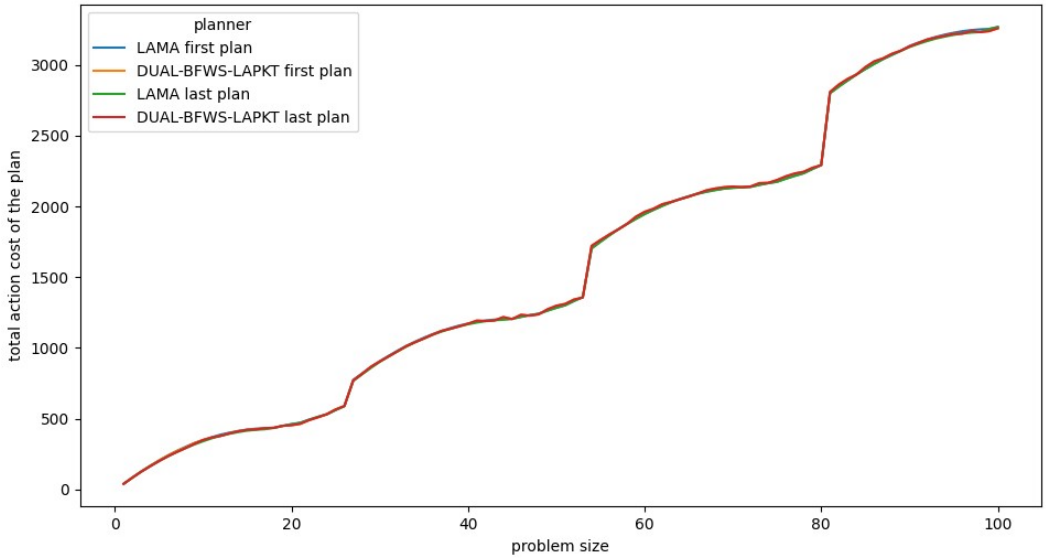


Fig. 5: Comparing the plan cost of first plan and last plan found by LAMA and Dual-BFWS-LAPKT during the given timeout limit, we can see that they almost overlap and the difference before and after optimization is insignificant. For the Dual-BFWS-LAPKT, the first plan is the last plan most of the time because it could not find another optimized plan

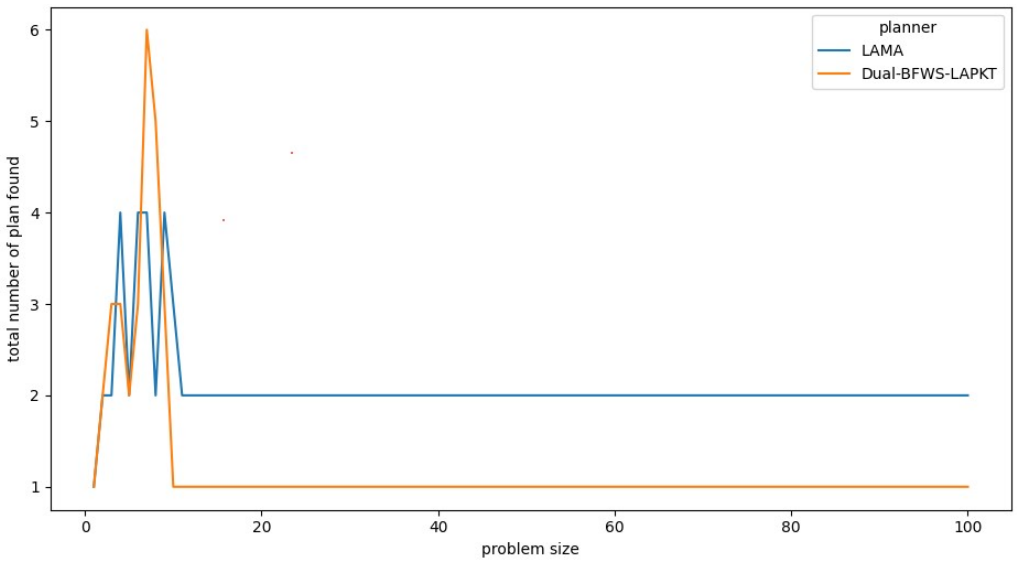


Fig. 6: Comparing total number of plan found between LAMA and Dual-BFWS-LAPKT in 90 second as the problem scales up, Dual-BFWS-LAPKT could not optimize the plan at all for most of the time as it only found 1 plan, where LAMA only optimize slightly

Other possible planning problems, like the remaining ones in Table I are similar to the chosen problem because the aforementioned visiting order can be optimized. Therefore, allowing meaningful optimization of the plan metric. The evidence and reasoning we had so far are sufficient to support the choice.

The script will repeat the process of scaling-up the planning problem, computing the plan, and parsing the output for analysis, starting with problem size from 1 to 100. The garden bed where we installed our Farmbot is 1.8m x 6m. It is estimated to accommodate around 50 plants, giving it 50 positions objects on a planning problem at maximum. However, scaling the problem up to 100 will give us clearer and more conclusive evidences when comparing how the different planners scale in terms of the performance.

Furthermore, for the second PDDL domain, which is the classical domain with the plan metric, scaling up the problem requires additional steps. In addition to scaling up the planning problem by inserting new objects and propositions, we also need to insert instantiations of the (move-distance) between all the position objects. This is done by a function in the script that takes in a number of position objects, which is the current problem size, and automatically generates the layout of the garden that contains the given number of position objects. The generated layout arranges the position objects in a grid-based manner and is represented by a matrix, the function then computes the Manhattan distance between every two objects and inserts the instantiation of the (move-distance) into the initial state. In a preliminary experiment, we

implement two different approaches for generating the garden layout. The first one appends new position objects in the horizontal row, and starts a new row once the current row contains 9 objects. The second one involves the concept of blocks, each block has a size of 9 x 3 and contains up to 27 objects. It will fill up the current block with new objects in a top-to-down, left-to-right manner, then move to fill up the next block once the current block is full. Consider letting this function to generate a garden layout that contains 50 position objects. Figure 7 illustrates the garden layout generated by these two approaches respectively.

```
[
...
[pos46, pos47, pos48, pos49, pos50, ...],
[pos37, pos38, pos39, pos40, pos41, pos42, pos43, pos44, pos45],
[pos28, pos29, pos30, pos31, pos32, pos33, pos34, pos35, pos36],
[pos19, pos20, pos21, pos22, pos23, pos24, pos25, pos26, pos27],
[pos10, pos11, pos12, pos13, pos14, pos15, pos16, pos17, pos18],
[pos1 , pos2 , pos3 , pos4 , pos5 , pos6 , pos7 , pos8 , pos9 ],
]

[
...
[pos28, pos31, pos34, pos37, pos40, pos43, pos46, pos49, ...],
[pos29, pos32, pos35, pos38, pos41, pos44, pos47, pos50, ...],
[pos30, pos33, pos36, pos39, pos42, pos45, pos48, ...],
[pos1 , pos4 , pos7 , pos10, pos13, pos16, pos19, pos22, pos25],
[pos2 , pos5 , pos8 , pos11, pos14, pos17, pos20, pos23, pos26],
[pos3 , pos6 , pos9 , pos12, pos15, pos18, pos21, pos24, pos27],
]
```

Fig. 7: Illustration of the difference between the two garden layouts with 50 position objects resulted from two different layout generation approaches, the top shows the first one and the bottom shows the second one.

These numbers, or parameters that determine the row size or the block size are decided arbitrarily, the point is to result in different patterns of the layout. The second approach results in a more staggered layout. The motivation behind testing these two approaches is due to a conference paper on the effect of domain configuration on planner performance [53]. In the paper, the ordering of predicate declarations were tested for the impact on planners. For our preliminary experiment, the hypothesis is that if we build

up the garden layout in a non-staggered way, and the distance between newly introduced position objects increases linearly, we might create a problem that is much easier for the planner to optimize. And in reality, we might not always have this 'ideal' layout. However, the result of this preliminary experiment shows that the difference between these two layout patterns is not significant. Figure 8 shows the total cost of the last found plan by the LAMA planner in 90 seconds timeout limit as problem size increases. We will explain things such as what is the last found plan, timeout limit or planner in detail later, this figure here is to help justify the choice of layout generation approach. Even though numeric difference exists due to the garden layouts differ significantly hence will result in different plan cost, and there are some sudden rises of plan cost as problem size increases because the newly inserted position is placed in a new row or new block, we can still see that the two line do not divert from each other, meaning that they scale up similarly. It is suggesting that the choice between these two garden layout generation approaches will not significantly affect the conclusion we can draw from the results. However, it is important to be consistent. Therefore, we will stick to approach two for all experiments. Also because a more staggered layout might reflect the real world application.

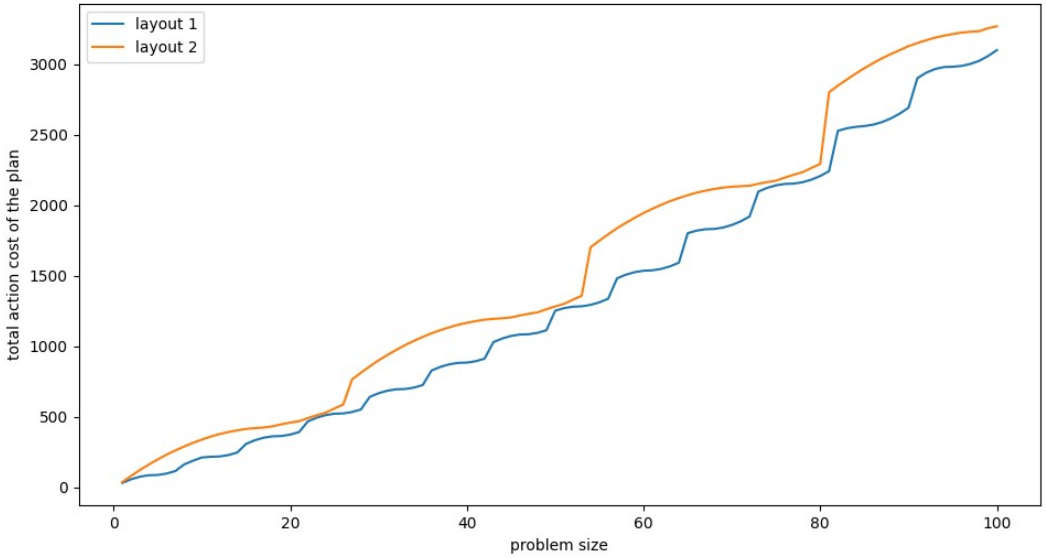


Fig. 8: Plot showing how two garden layout generation approaches affect the planner performance

We will apply different planners for each of the three domains. The planner FF and BFWS are the state-of-the-art planners for the first pure classical STRIPS domain, they will return the output as soon as a plan is found, disregarding the quality. Guided by the h_{FF} heuristic and novelty respectively, the plan they first discovered will generally be the easiest one to find in the search problem. For the second classical domain with action cost plan metric applied, the LAMA planner and the Dual-BFWS-LAPKT

planner are the state-of-the-art. They are anytime planners that require users to provide a timeout limit. These two planners will iteratively improve the quality of the plan based on the plan metric, or total plan cost in our case until the timeout. For both anytime planners, the time limit we allow is 90 seconds. After some preliminary experiments, we found that 90 seconds will allow the results to show some conclusive evidences that differentiate the planners as well as the domains. Enabling us to critically analyze the results and extract the meaningful conclusion. We will discuss it in detail in the later sections.

All of these planners will produce an output file once they are called to solve a problem. The output file indicates whether plans can be found. If the answer is positive, this output file will contain the performance metrics we are interested besides the plan itself. We can use these metrics to evaluate the effectiveness of planners for each domain. For FF and BFWS on classical STRIPS domain, we are most interested in the time taken for finding the plan, as well as the total cost of the plan. However, FF and BFWS on pure classical domain do not allow declaration of action cost. The planners will report the plan length, which is the number of actions in the plan as the plan cost because the cost is assumed to be uniform as 1. We can work around this by parsing the detailed plan and calculating the action cost. For the LAMA and Dual-BFWS-LAPKT, which are the anytime planners, they are slightly different. In addition to the same performance metrics as the FF and BFWS, we are also interested in how many plans they find for each of the problem sizes in the given time limit. Then, if there are more than one plan found, which means that the plan is optimized, we are most interested in the difference of time taken and the cost between the first plan found and the last plan found before the timeout. These can allow us to analyze how much extra time the planners take to optimize the plan and how well the plan is optimized. We will show these performance metrics that we can directly parse from the output of planners in section V-B, then, we will also discuss the information we can extrapolate from these results through further analysis in the discussion section V-C. Furthermore, each of the same configuration (domain, planner, problem size etc.) will be run 10 times and the mean average will be taken as the result to minimize the impact of the performance fluctuation of the computing hardware. Hypothetically, we will only see variations in the time expense for the planners from the results of the multiple repetitions of the same configuration, this is because the same algorithm on the same search problem will always follow the same search path and produce the same plan. There is no stochasticity in the planners we use. Also, the variations in time taken across different repetitions of the same configuration are expected to be small, because the fundamental search problem remains the same, the search effort required for the algorithm will be the same, the variations can only be caused by the fluctuation in hardware performance such as CPU.

This experiment will be run on a desktop computer with Intel 10900k CPU, 32 gigabytes of memory running Ubuntu 20.04 LTS [69]. Figure 9 shows the process diagram of this experiment. Figure 10 provides a simple diagram showing the components and the relationships of the script. To summarize the experiment: for the two domains we proposed in section IV, we will run their respective planners to solve the problems, and scale up the problem size from 1 to 100 to see how each planner performs when the complexity of the search problem increases. Each setting will be run 10 times and averaged out for the results. The results will be used to help determine the appropriate planning domain between them for the planning problems on Farmbot as well as the optimal planner for this domain.

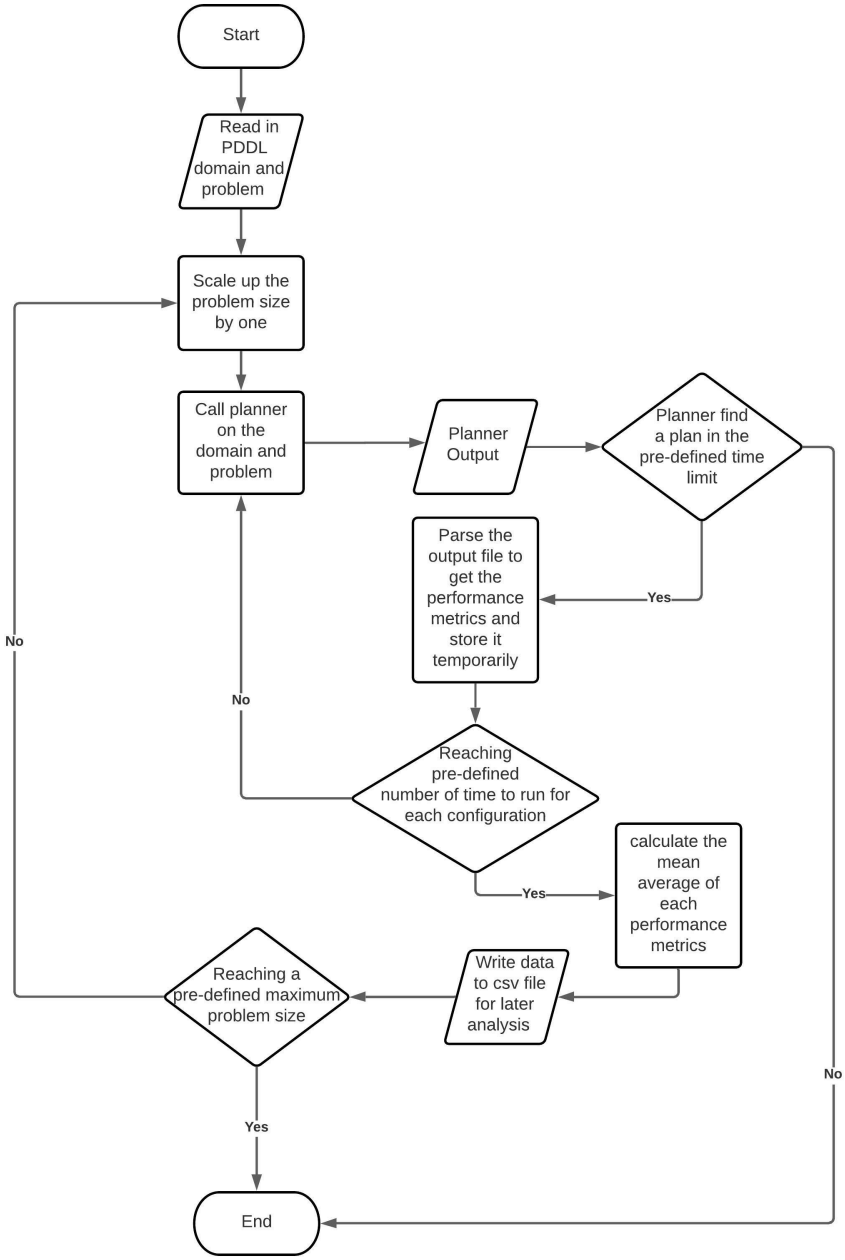


Fig. 9: Process diagram of the scripts used for the experiment

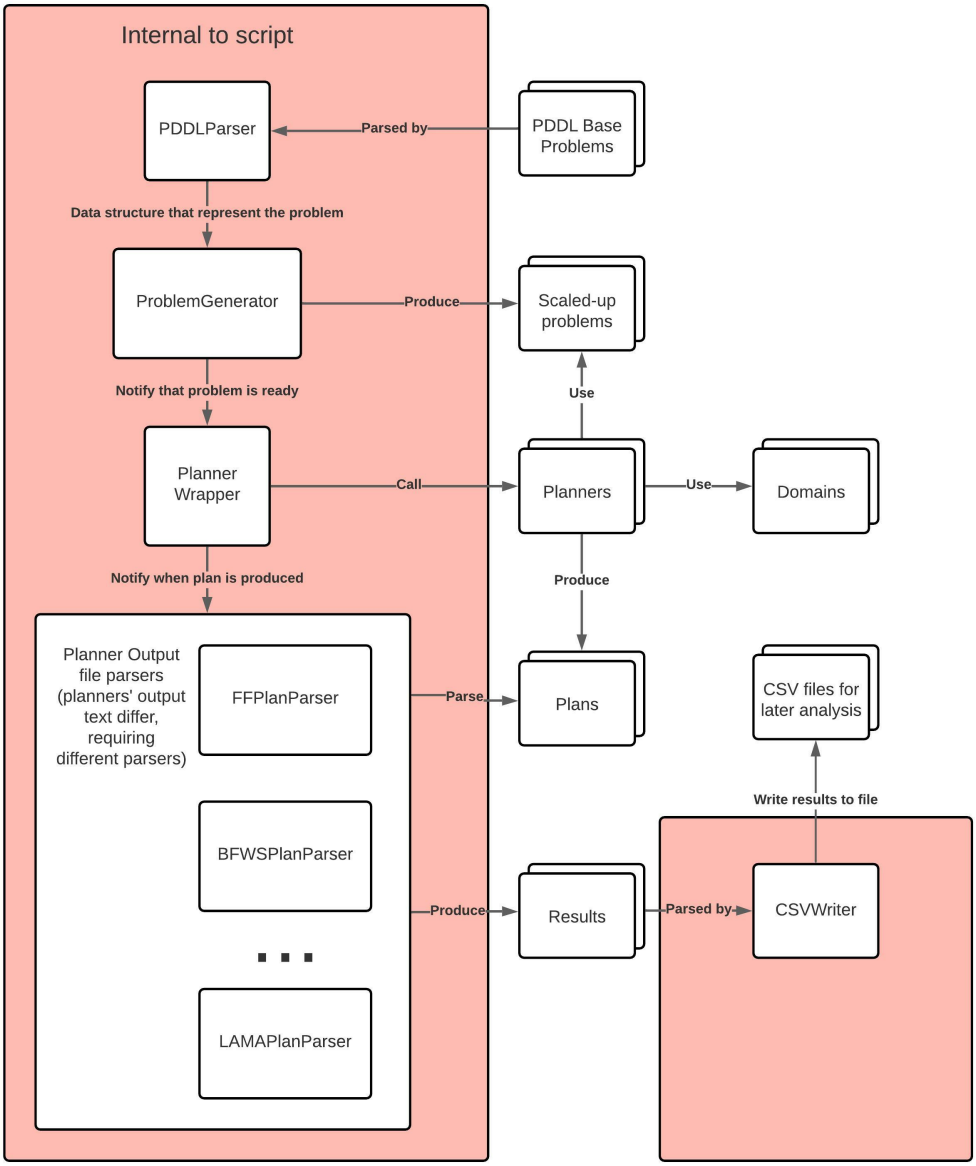


Fig. 10: Diagram showing the components of the script and their interactions with internal components or external files and programs

B. *Results and Interpretation*

We gather the performance metrics on how planners solve the problems with increasing problem size on their respective domain as described above. The results are stored in csv files that contain the individual performance metrics for all problem sizes. These csv files are large and each contains thousands of entries. Instead of displaying the raw data in tables, we use the python programming language version 3.6.9 [68] and the library matplotlib [70] to produce plots that are easier to view and interpret.

*1) **Classical planning model***

The planners BFWS and FF are applied to the pure classical domain. Figure 11 shows the time taken to find a plan as the problem size increases. **The solid plot is the mean average value whereas the shaded area around the mean represents the standard deviation of the 10 repetitions** (This visual representation is the same for all the following plots, if not specified otherwise). The time performance of the FF planner degrades much faster as the problem size becomes larger, starting at around 40 position objects. This is very likely due to the computational complexity of the h_{FF} heuristic used by the EHC algorithm in the FF planer starts to meet performance bottleneck. On the other hand, the novelty used by the BFWS is much easier to compute, resulting in a smaller increase in terms of the time expense. It is certain that BFWS takes less time than the FF for the same problem size. At the problem size of 100, the mean value of time taken by the FF planner is 1.68 compared to 0.33 of the BFWS, which is around 5 times smaller. The standard deviations are very narrow, indicating all the results of the 10 repetitions agree with the pattern shown in the mean values and hence, does not affect our conclusion.

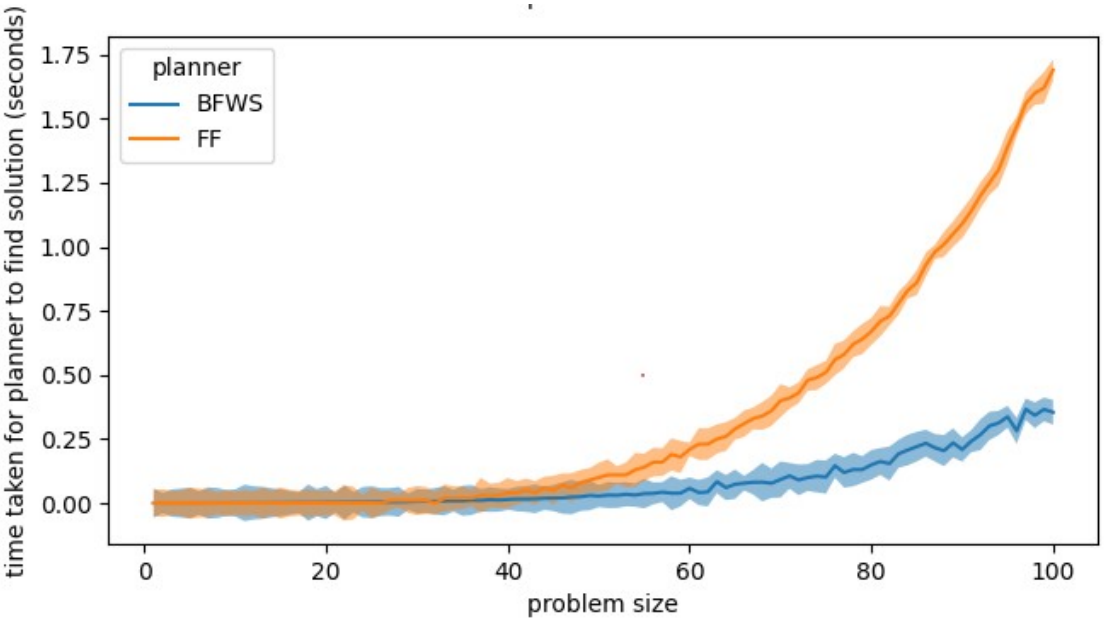


Fig. 11: Plot showing the time taken to find a plan on the given problem size between BFWS planner and FF planner. The shaded area represents the standard deviation

However, in real life scenarios, having a problem that contains 100 position objects or even bigger is very unlikely on Farmbot. Furthermore, even though FF scales up much faster, the magnitude of the time expense is still considered very small, which is 1.68 seconds taken to find the plan at problem size 100. Therefore, we consider that the difference in the time expense is not significant enough in the context to allow us to exclude the FF planner. BFWS is the clear winner to compute the plan faster, but it comes

at the cost of using a less informed novelty compared to the h_{FF} heuristic, which means the plan found by BFWS is likely to be less optimal. In addition to time taken to compute the plan, the plan quality is considered a more important performance metric as it can directly impact the plan efficiency, potentially costing more time to execute a less optimal plan. Figure 12 shows the length of plan found by these two planners. The shaded area does not appear as all 10 repetitions produced the same results. This is because the same algorithm will most likely to produce the same search path on the same search problem configuration, plans of all 10 repetitions of the same configuration are always the same. We can clearly see that most of the time, the FF finds a shorter plan to complete the same task. However, when the problem size is small, specifically below 20. The plots between BFWS and FF overlap. Another observation is that the plan length of the BFWS does not divert from that of the FF, visually it just maintains a consistent range of difference above the FF, and the difference is very narrow.

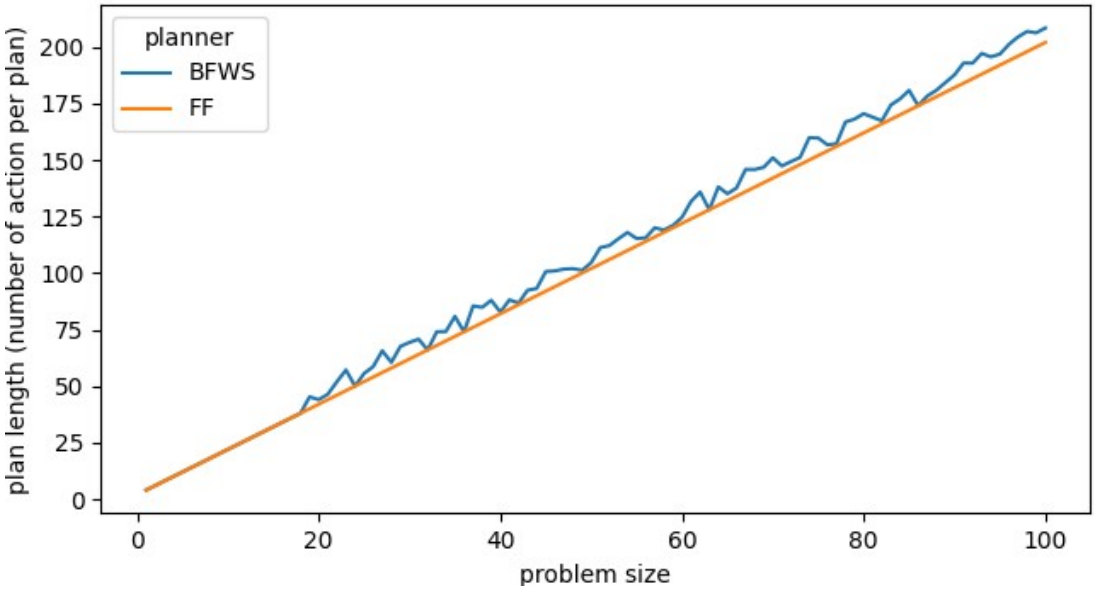


Fig. 12: Plot showing the length of the plan on the given problem sizes between BFWS planner and FF planner

However, this is not to say that the plan quality of the plan found by the BFWS using the less informed novelty is not worse than the more computationally expensive FF with the h_{FF} heuristic. The plan length is only the number of domain actions in the plan, and does not provide sufficient evidence on the quality of the plan and hence, we cannot determine which planner finds a better plan. The limitation of the pure classical domain is that it assumes all actions to have uniform cost, which is not always true. Both the BFWS and FF planner do not accept the plan metric syntax and do not allow the definition of non-uniform action cost. Therefore, they do not report directly from the planner outputs. However, in the script, we can still generate the garden layout and store the information of the action cost in a look-up table, later

when we obtain the detailed plan from the planner output, we can parse the plan and then calculate the cost by adding the cost of individual (move) actions in the plan. This produces the results in Figure 13, which shows the different plan costs of the plans found by these two planners when the problem scales up. Because the plans of the same configuration are the same, therefore the shaded area does not appear.

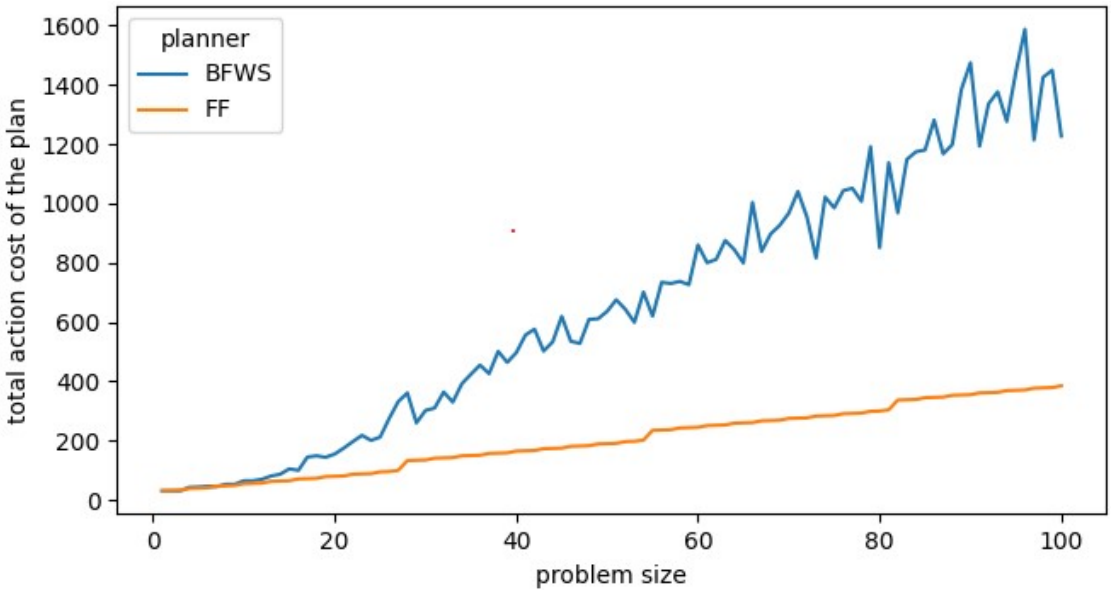


Fig. 13: Plot showing the plan cost of the plan found by BFWS planner and FF planner on the given problem sizes

The significant difference between the planner now emerges. From Figure 13, we can clearly see that the plan quality in terms of the total action cost is generally much worse in the plans found by the BFWS. At the start when the problem size is small, the numeric difference is also small. As the problem size increases, the BFWS scales up much faster and diverts from that of FF. Meaning that the novelty used by the BFWS could be even less informed to guide the search algorithm when the search problem becomes more complicated. In addition, the plan quality of BFWS seems to fluctuate a lot as the problem size increases, this is probably due to the less informed novelty tends to have more random or non-optimal node expansion behaviour during the search, and occasionally increase the plan cost by some random value. The FF, on the other hand, increases relatively consistently. Although there are three points where sudden rise occurs. That is due to the garden layout generation approach we described above fills up the whole block and inserts the next position object in the new block relatively far away from the existing blocks.

Overall, BFWS compute the plan faster over our domain and the time taken scales up much slower, though in terms of the absolute numeric difference, the FF planner is not considered unacceptably slow,

around 1.7 seconds to find a plan for problem size 100 is still very fast. When comparing the plan quality, the FF shows a great advantage of the BFWS and the plan found by the FF is considered much more optimal. Generally, the FF is the more favorable planner over the first pure classical PDDL domain. However, the evidences we have so far do not allow us to conclusively opt for this domain and the FF planner, we still need to compare the second domain with the action cost plan metric with the two anytime planners and place the results in a greater context later when we compare across the domains.

2) ***Classical planning model with action cost***

In this domain, the LAMA and Dual-BFWS-LAPKT are planners designed to optimize the plan cost. They are anytime planners that find as many plans as possible and iteratively improve the plan quality within the given time limit. Figure 14 shows the total number of plans they find in the 90 seconds. Generally both planners found multiple plans within the 90 seconds time limit and hence iteratively improve the plan quality many times, there are a few exceptions for the LAMA planner where is has downward spikes in the plot. All the 10 repetitions for the same configuration produce the same result, therefore it is probably due to the specific search problems that LAMA failed to improve the plans. We also notice that the total number of plans has an upward trend as problem size increases, due to smaller size problems are generally easier to find more optimal plans over the first few tries. It seems this upward trend gradually stabilizes as the problem size further increases, possibly due to more difficult search effort is required to further optimize the plan when the search problems become more complicated.

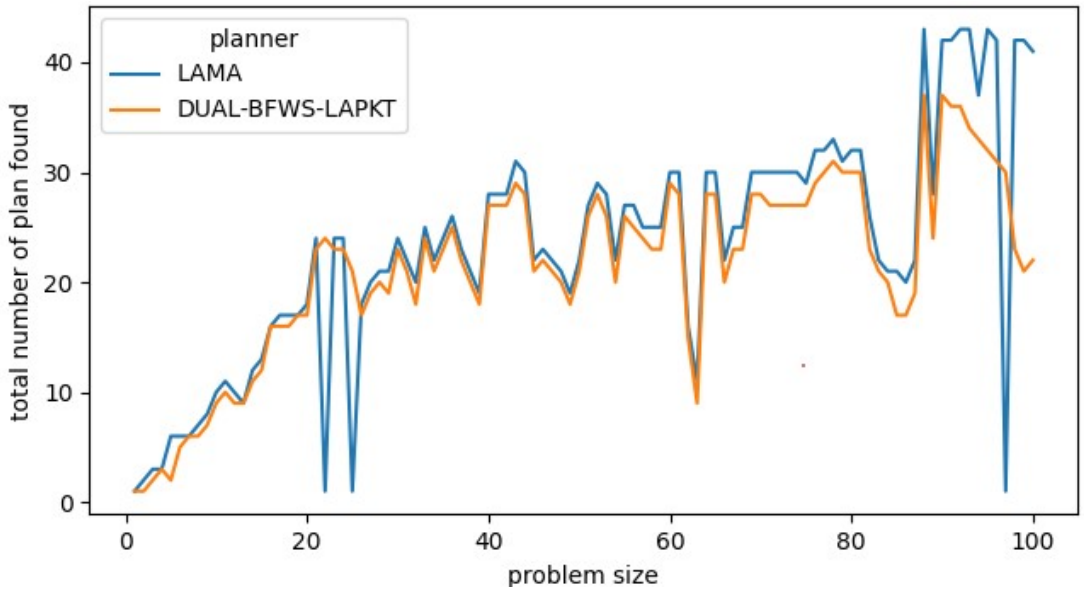


Fig. 14: Comparing number of plan found by LAMA and Dual-BFWS-LAPKT in the given timeout limit as problem size increases

The total number of plans found does not indicate how well the two planners optimize the plan. To investigate this, we can compare the difference of time taken and plan quality between the first plan and the last plan found by these two planners for each problem size, so that we gain insights on generally how much extra time is spent and how well the planners improve the plan. Figure 15 shows the difference of the total plan cost between the first plan and the last plan found by both planners in 90 seconds time limit, and Figure 16 shows the difference of the time taken between finding the first and the last plan.

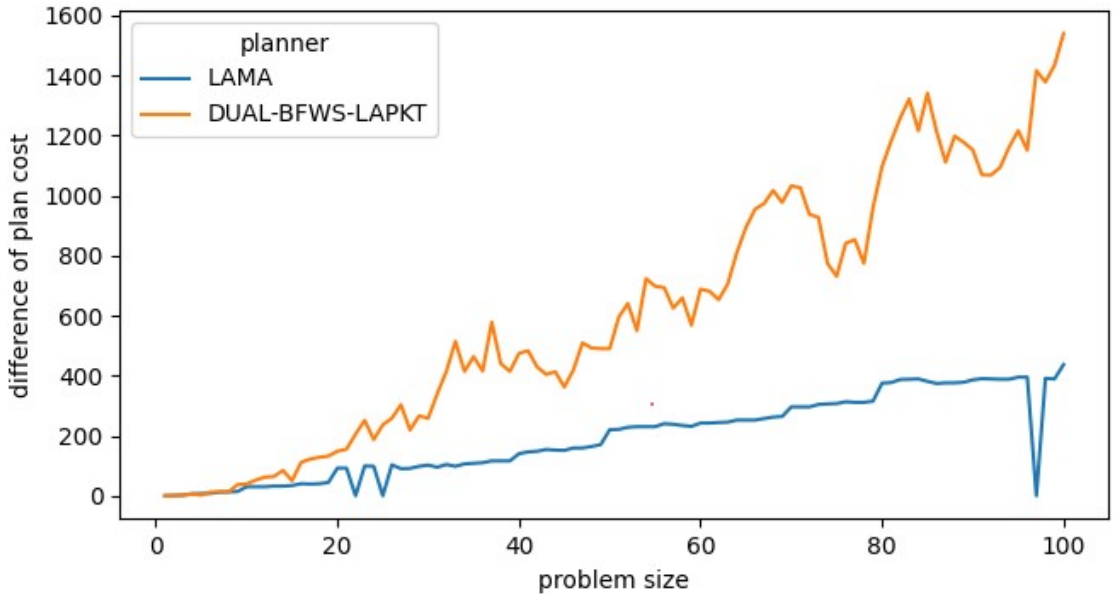


Fig. 15: Difference of plan cost between the first plan and the last plan found by LAMA and Dual-BFWS-LAPKT

There are a few observations from these differences. Both planners can effectively optimize the plans in the 90 seconds time limit, and the magnitude of the plan cost they optimize increases with the problem size. Figure 16 also shows that the extra time taken for the optimization does not scale up with the problem size, after the initial significant increase at around problem size 20, it gradually stabilizes at the same magnitude over the remaining problem sizes. In addition, The standard deviations are very insignificant, consider that Figure 15 shows no shaded area because all the results of the 10 repetitions are the same, these slight variations in time taken can only be caused by fluctuations in the computing hardware and have no meaningful implications.

From the plot of the mean value, we can observe that the extra time taken to optimize the plan between two planners follows a similar pattern in the trend, we cannot conclusively say that one takes more time than the other, but the numeric value of the optimized plan cost scales up differently. From Figure 15, we can see that Dual-BFWS-LAPKT significantly optimize more costs that LAMA. However, this is not

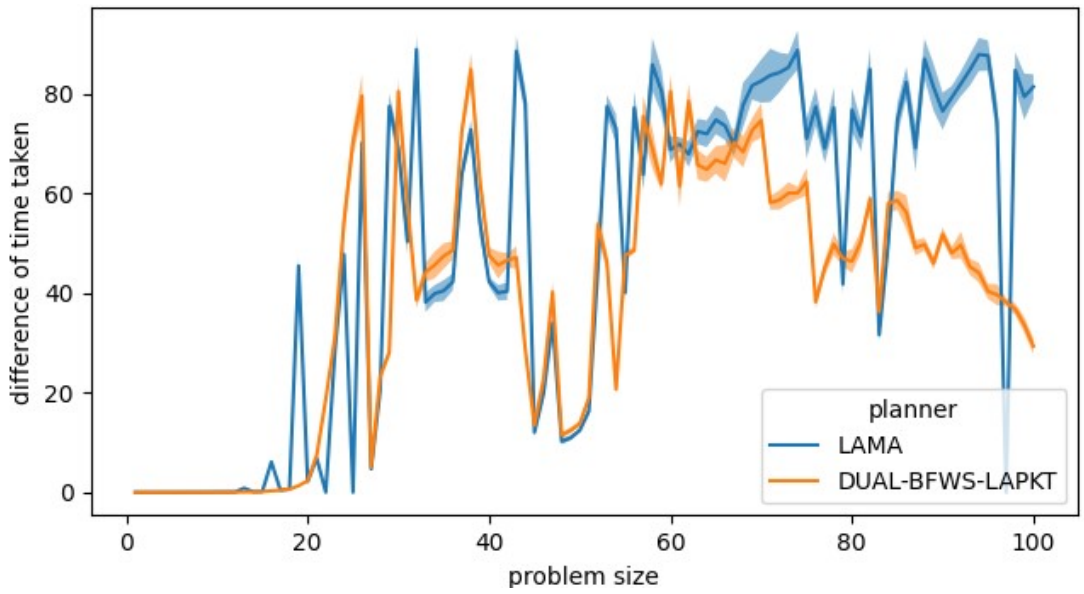


Fig. 16: Difference of time taken between the first plan and the last plan found by LAMA and Dual-BFWS-LAPKT. Shaded area represents the standard deviation

suggesting that the Dual-BFWS-LAPKT is more capable of optimizing the plan and finding a more optimal plan before the timeout limit. The Dual-BFWS-LAPKT uses the BFWS to find the first plan and then utilize that cost of that plan to bound the WA* search that is used in the following iterative improvement. Unlike LAMA, which also uses WA* for the first search, the plan quality of the first plan found by Dual-BFWS-LAPKT can be much worse than the LAMA, leaving it more room for optimization. Figure 17 shows the total plan cost of the first plan found by these two planners with increasing problem size. We can see that the first plan found by LAMA using the WA* is much better than the Dual-BFWS-LAPKT using the BFWS.

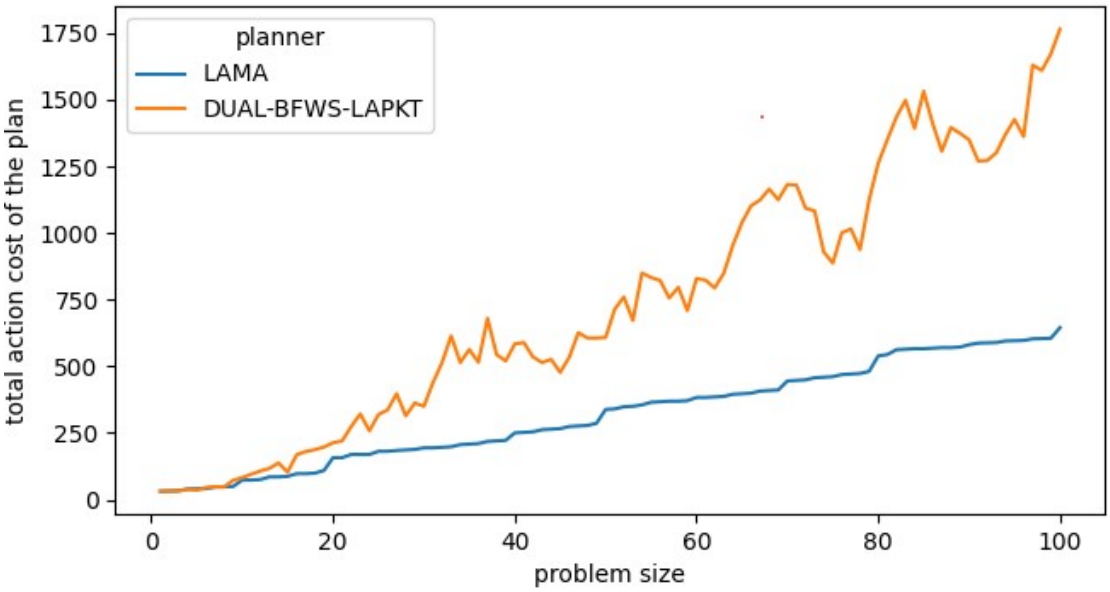


Fig. 17: Comparing plan cost of the first plan found between LAMA and Dual-BFWS-LAPKT

Also, considering the numeric scale in Figure 17 and Figure 15, the improvement of the plan quality by both planners is very significant. Even though the Dual-BFWS-LAPKT optimized much more plan cost within similar time range, it is still not considered as an advantage because the first plan found by the Dual-BFWS-LAPKT is much worse. To compare which planner can result in a more optimal plan, we need to compare the plan quality of the last found plan. Figure 18 shows the plan cost of the last plan found by these two planners.

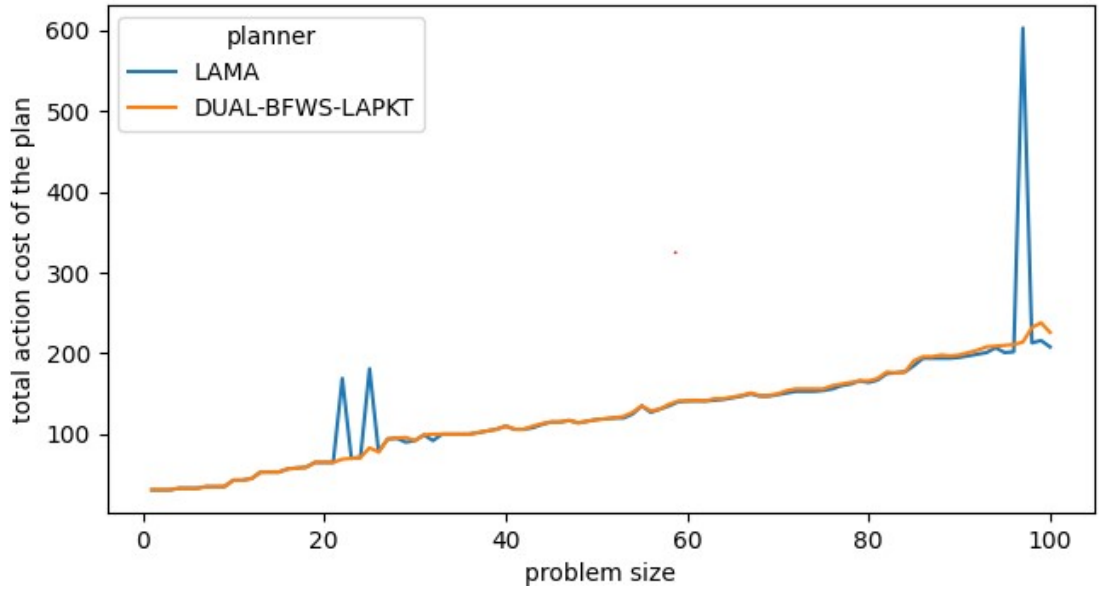


Fig. 18: Comparing plan cost of the last plan found between LAMA and Dual-BFWS-LAPKT

Generally, the numeric difference between the plan cost of the last found plans, which are the most optimized plans before the timeout limit by these two planners is not significant. Even though the LAMA occasional has only a few much worse last found plans in terms of plan cost (more precisely, the three spikes in Figure 18) likely due to the specific search problem configuration as all 10 repetitions produce the same results (no shaded area in the plot), overall trends between these two planners are very close and nearly overlap most of the time. The only thing that can differentiate them is the time taken to find these optimized plans. Figure 19 shows the time taken to find the last plan.

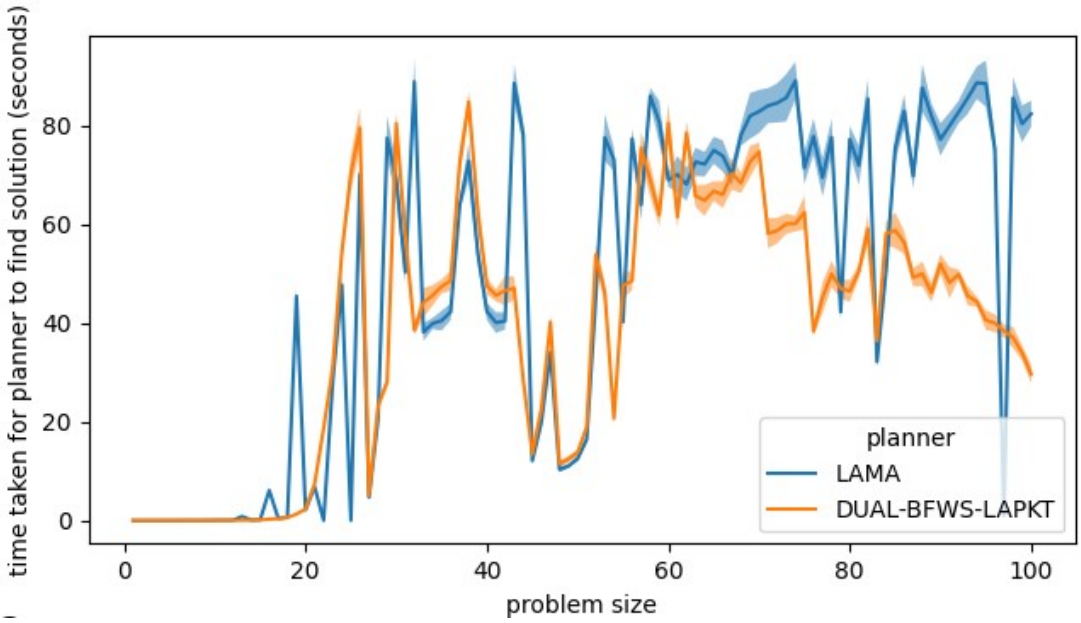


Fig. 19: Comparing time taken to find the last plan between LAMA and Dual-BFWS-LAPKT

In Figure 19, there are significant fluctuations because the search problem configuration is different at each problem size. We also observe that generally both planners scale up similarly, there is a drastic rise from problem size 20 and then stabilize and fluctuate. The standard deviations are very insignificant and therefore patterns indicated by the mean values can be generalized to all 10 results. However, the Dual-BFWS-LAPKT takes a shorter time and shows decreasing trend as the problem size goes beyond 60. The reason is unknown and with the evidences we have so far, we consider these two planners very similar to each other in terms of performance. Furthermore, the two anytime planners will always take up the 90 seconds before they terminate, the time expense we plot in those figures is less significant for the anytime planners as the 90 seconds are always needed. However, it indicates the appropriate timeout limit we should set for each planner.

In addition, Figure 19 looks very similar to Figure 16. This is because Figure 16 is just the difference

of time taken between the first found plan and the last found plan, essentially it is Figure 19 subtracted by the time taken to find the first plan. Figure 20 shows the time taken between these two planners to find the first plan. We can see that the time taken for the first plan by both planners is very tiny and hence, results in the similarity between Figure 19 and 16. Also, it shows that the BFWS, which is the first search algorithm used to find the first plan by Dual-BFWS-LAPKT is much faster than WA*, but considering Figure 17, it is at the cost of finding a much worse first plan in terms of the plan quality.

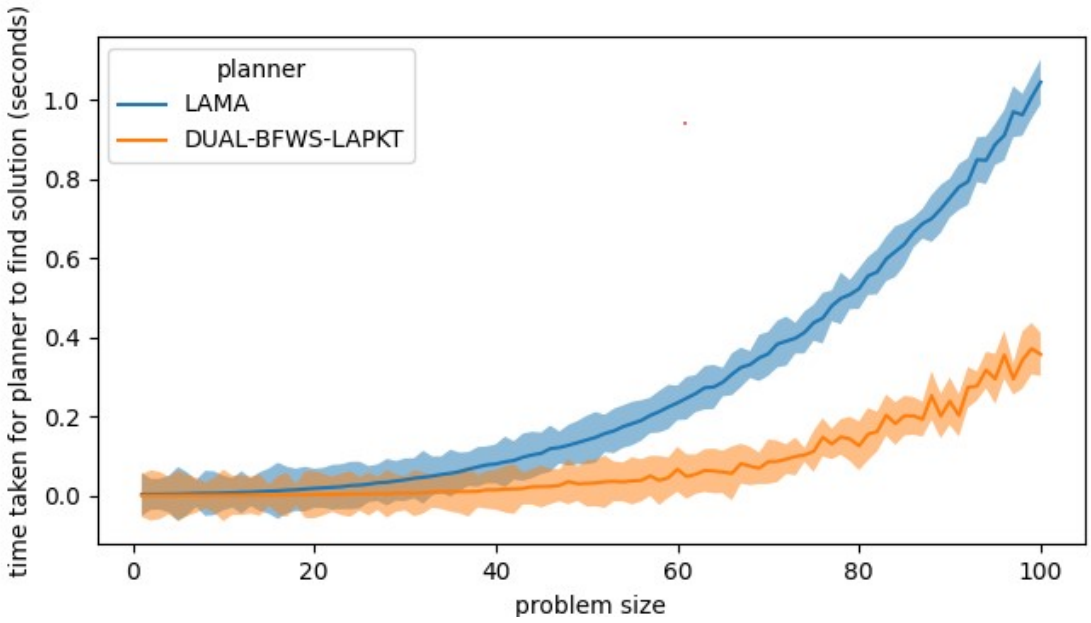


Fig. 20: Comparing time taken to find the first plan between LAMA and Dual-BFWS-LAPKT. The standard deviations represented by the shade area are also very narrow

Even though both planners are very similar to each other in performance, the LAMA has spikes in costs, and generally takes longer to find the last optimized plan when the problem size increases beyond 60 for unknown reasons. These factors make Dual-BFWS-LAPKT more preferable. However, we have no conclusive evidence to explain that and hence, we cannot confidently suggest a winner of the two with the evidence we have so far. Furthermore, our aim is to show whether the action cost plan metric is preferable, so we also need further analysis that compares with the pure STRIPS classical domain in order to draw the conclusion that answers our question.

C. Analysis and Discussion

To answer this question of whether the action cost plan metric is necessary for improving the plan quality, comparing planner performance within the same domain is not enough, instead we have to take

it a step further to compare between the first and the second domains. In addition, we will **only plot the mean average values and ignore the standard deviations** from now on. This is because from the previously displayed results, the variations across the repetitions of the same configuration only apply to the time expense due to fluctuation in hardware performance, and these variations are very insignificant. They do not affect the trends and patterns indicated by the mean values. Since we are interested in analyzing and comparing the trends, patterns and scale of the results rather than the exact numeric value, showing the mean values only provides sufficient evidence for answering our question. Another reason is that the standard deviations provide no meaningful implications, removing them from the plots allows us to have a clearer observation on other useful information such as the patterns.

Figure 21 shows the plan quality of plans found by all four planners as problem size increases, BFWS and FF using the pure classical domain, Dual-BFWS-LAPKT and LAMA on the domain with the plan metric. For the Dual-BFWS-LAPKT and LAMA, the last found plan is used.

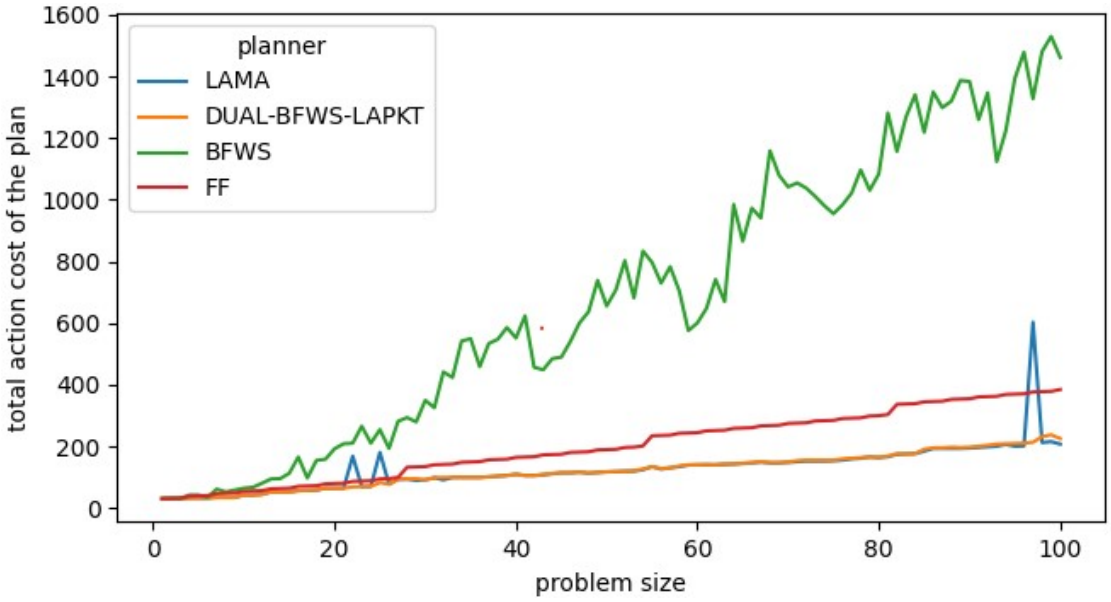


Fig. 21: Total plan cost of the plan found by all four planners. For the Dual-BFWS-LAPKT and LAMA, the last found plan is used.

Evidently, the plan quality found by the two anytime planners over the second PDDL domain that uses the plan metric is better. Even though LAMA has a few exceptions where it performs even worse than the FF on the classical domain, but generally the optimized plan quality is much better. Furthermore, on the pure classical domain, FF is significantly better than the BFWS, and the result is closer to the two anytime planners than to the BFWS.

To make the results more visually distinctive for comparison, instead of plotting the raw data, we apply the log scale to the data in Figure 21. Figure 22 shows the plan quality of plans found by all four planners as problem size increases, where log_e is applied to the values on y-axis. For the Dual-BFWS-LAPKT and LAMA, the last found plan is used again.

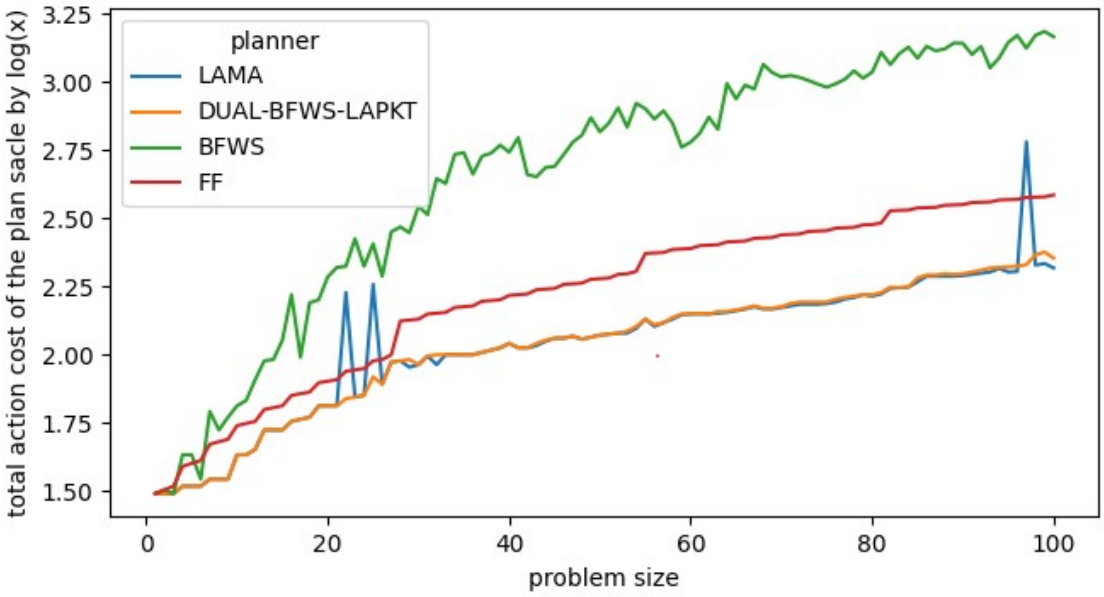


Fig. 22: Total plan cost of the plan found by all four planners, log base e scale is used for the plan cost. For the Dual-BFWS-LAPKT and LAMA, the last found plan is used.

We can see that LAMA and Dual-BFWS-LAPKT generally find the best plan most over all the problem sizes and the difference between these two anytime planner is not significant. Therefore, we can say that the action cost plan metric has a meaningful impact on improving the plan quality.

Figure 21 and Figure 22 also show that the FF planner finds much better plans compared to BFWS on the same pure classical domain, and the plan quality is even closer to that of the two anytime planners. This is because the h_{FF} heuristic is a very informed heuristic that efficiently estimates the cost of the future actions leading to the goal. Essentially the h_{FF} heuristic is the cost of the relaxed plan (explicit solution to the delete-relaxed plan, as explained in the literature review). Even though it does not directly consider the action cost, it is still sensitive to the cost and allows the search algorithm to choose which is the next search node to consider. When the problem size is relatively small, specifically smaller than around 20, the FF planner on classical domain seems to be very close to the anytime planners even without considering the action cost and employing the same optimizing strategy. Only when the problem size starts to get large, the action cost plan metric then starts to enable anytime planners to find a significantly better

plan. This shows the increasing impact of optimizing the action cost with the plan metric and anytime planners.

However, if we consider the time taken for the anytime planners to optimize the plan and achieve the plan quality in their last found plan, the FF seems to achieve a result that is not worst compared to the BFWS, and requires only a fraction of the time. Figure 23 shows the time taken to find plans found by all four planners as problem size increases. For the Dual-BFWS-LAPKT and LAMA, the last found plan is used.

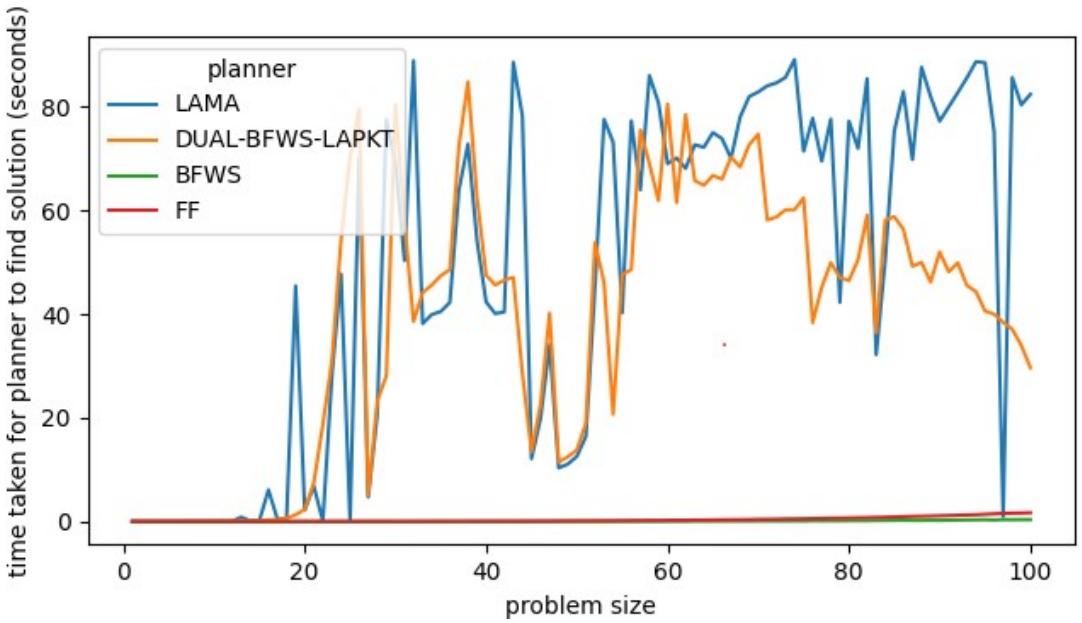


Fig. 23: Time taken of the plan found by these four planners. For the Dual-BFWS-LAPKT and LAMA, the last found plan is used.

The anytime planners over the domain with the plan metric generally spend much more time on optimizing the plan. Figure 23 shows that extra time required by the two anytime planners could really take up the whole 90 seconds for their last found plans whereas the two classical planners only take a few seconds. Besides, the actual time is less important because the two anytime planners will always spend the entire timeout limit before it terminates. Even though the action cost plan metric is evidently useful in improving the plan quality, the FF planner on the classical domain without the plan metric might still be desirable in some situations. Particularly when the problem size is not that large and slightly less optimal plan is acceptable, as well as when time constraint is applied and the two anytime planners are not allowed to have sufficient timeout limit.

Furthermore, even though Figure 23 shows that the two anytime planners could really take up the

whole 90 seconds timeout limit, the actual time they need can be much shorter than that. Figure 24 plots the cumulative difference of the plan cost between the plans found at any time point during the 90 seconds timeout limit and their respective last found plans (the horizon x-axis indicates when during that 90 seconds the plans are found and the vertical y-axis shows the cumulative difference of those plans that are all found at the same time relative to their respective last found plans of the same problem size), this indicates how many more the combined total plan cost of those plans found at the same time compared to their respective last optimized/found plan, all the problem sizes are combined. Essentially, it suggests that within which time in that 90 seconds that the plans are optimized the most, and starting from when in that 90 seconds, the plan quality is not improved much and hence, the timeout limit more than that is unnecessary.

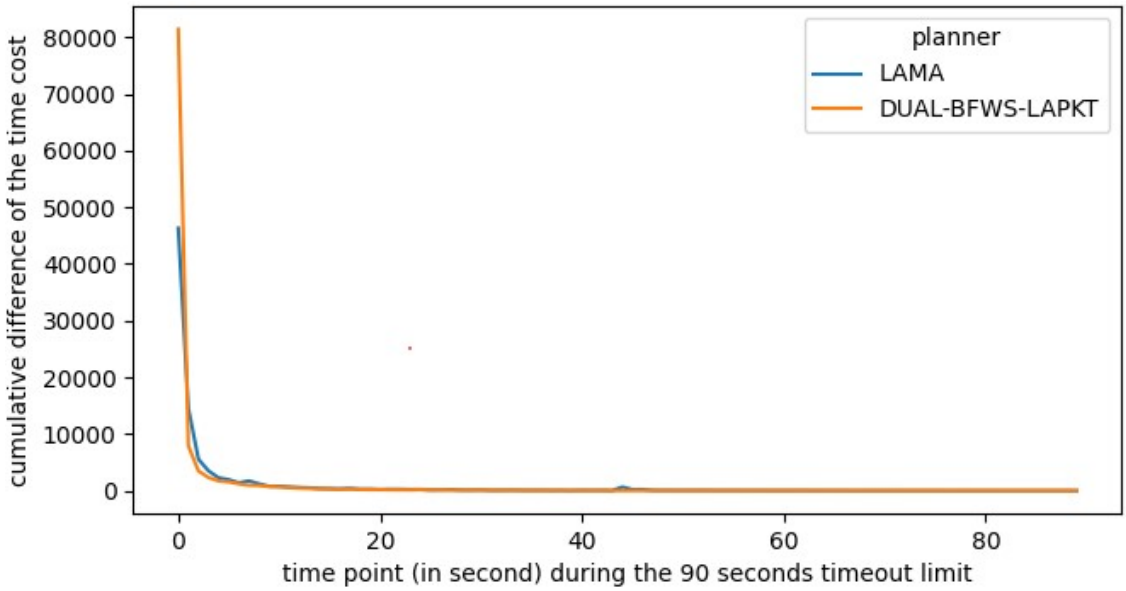


Fig. 24: The cumulative difference plot showing how many more the combined total plan cost of those plans found at the same time compared to their respective last found plan, for both LAMA and Dual-BFWS-LAPKT planners

From Figure 24, it seems that most of the plans get optimized the most in the first 10 seconds, and they appear to have no optimization after 20 seconds. It shows that having a timeout limit of 90 seconds is unnecessary and might cancel the advantage by the two anytime planners using the plan metric compared to the FF on the pure classical domain, especially when the problem size is small.

To summarize, with all the evidence we gathered so far, we can certainly say that the plan metric used on the second PDDL domain allows us to apply anytime planners and significantly improve the plan

quality. However, consider that the FF over the pure classical domain has very close plan quality when the problem size is small, and the significant extra time require to optimize the plan quality by the two anytime planners, we cannot decisively say the second domain with plan metric will always be the better choice. We need to resort to further experiment later in section VIII and conduct a simulation that will take into account not only the time taken for the planners to compute or optimize the plans, but also the time for executing plans to find out if a slightly less optimal plan is acceptable when it can significantly reduce the time expense of the planners. All the findings and evidences by then will be combined to make a final conclusion. Furthermore, this benchmarking experiment also contributes several other meaningful points to answering our research questions:

1) The choice of whether to use the plan metric might depend on the specific problem. Therefore to compare planners thoroughly, we need to narrow the planning problem down to some specific scenarios which contain distinctively different problem sizes.
2) Even though the time taken is less important for the anytime planners because it will always take up the whole timeout period before it terminates, the results of time taken still allow us to determine the appropriate time limit for our next experiment based on simulation in section VIII. Figure 25 scales the horizontal value of Figure 24 by log_e to show when exactly within the 30 seconds, the plans are optimized most. We see that for the Dual-BFWS-LAPKT, setting the timeout limit more than 30 seconds is completely unnecessary as no further optimized plan can be found after that. For LAMA, 50 seconds is the time when no further optimization can happen. From the original Figure 24, it seems only a very tiny amount of plan cost is optimized after 20 seconds for both planners. When the problem size is small (smaller than around 16), Figure 23 shows that the last optimized plan can be found in 5 seconds. Therefore 90 seconds might be too long and sometimes unrealistic for application. In the simulation we can adjust this timeout limit to a more appropriate number.
3) The results give us insights on how planners scale up, and how different planners perform in many different problem sizes and scenarios. The analysis we made in this section will be used as part of the mixed evidence to support answers to our research questions.
4) Through this benchmarking, all the planners found valid plans on both domains for all the problem sizes without giving any error. This is an indirect but strong evidence showing that our PDDL modelling is correct. Which helps us positively answer our RQ 1. However, we will answer it with more evidence at the end when the simulation experiment is successful.

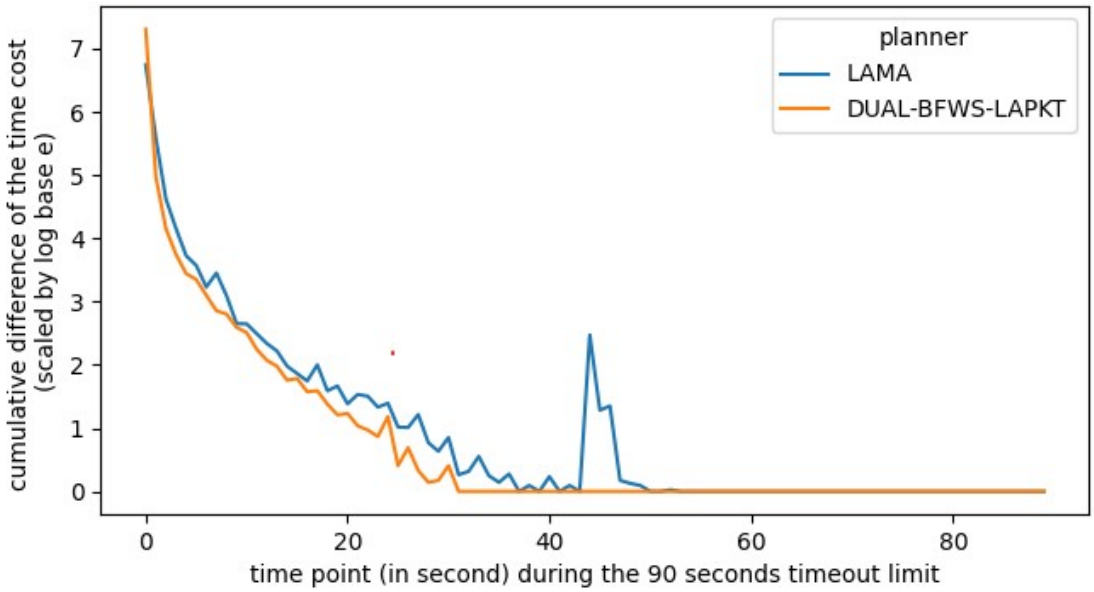


Fig. 25: The log-scaled cumulative difference plot showing how many more the combined total plan cost of those plans found at the same time compared to their respective last found plan, for both LAMA and Dual-BFWS-LAPKT planners

Furthermore, one observation worth mentioning is that the original algorithm of BFWS can easily be modified to prefer shorter plans: we can combine the novelty with an accumulated cost in the evaluation function that guides the search so that the search algorithm is cost-sensitive, and when it selects the next search node to explore, it will not just consider the novelty but also the overall quality of the plan. However, testing this is beyond the scope of this research, and is a potential direction for the future.

VI Integration of Planning and Controlling

In this section, we present the implementation of a ROS project that integrates the planners and controlling components using the ROS [19] and ROSplan framework [17]. The purpose is to allow the Farmbot to execute the plan generated by planners on the PDDL models we proposed above. The plan outputs of planners are in pure text. This ROS project is able to parse the plans from the planners and dispatch it to the Farmbot and instruct the Farmbot to execute the plan. It is important because we can examine how well the domain modelling and plan generations actually work with a real world platform. Also, it provides an infrastructure that allows us to conduct simulation later.

A. *Integration Architecture*

Figure 26 illustrates the architecture of this ROS project, also showing how each component interacts.

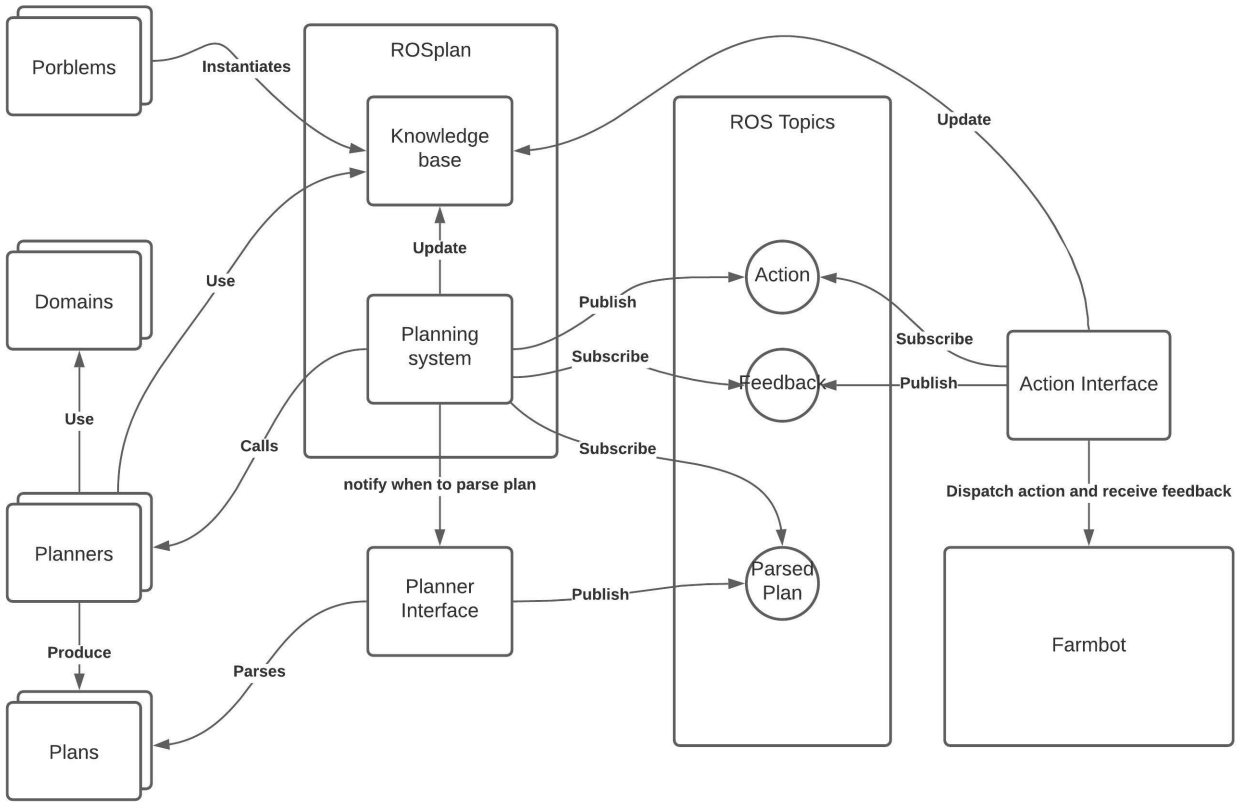


Fig. 26: High-level diagram showing the integration of planning and controlling

The ROSplan provides two essential components, the knowledge base and the planning system, both are individual ROS nodes. The knowledge base stores the current planning problem, with its current world state (initial state) and the goal, it can be instantiated by a PDDL problem file upon the start of

the program. The planning problem stored in the knowledge base can also be updated through a ROS service. The planning system manages all the components and the workflow of the ROSplan, it will let the knowledge base generate the PDDL problem file based on the current planning problem stored in the knowledge base, call the planner on the supplied domain file and the generated problem file, then call the user-supplied planner interface to parse the plan and publish that to a ROS topic, finally it will dispatch the parsed plan to the user-supplied action interface that interacts with the actual Farmbot, and update the current world state stored knowledge base.

We need to implement two other essential components, the planner interface and the action interface. The former is responsible for parsing the plan outputs by the user-supplied planner and the latter is responsible for taking the dispatched actions from the planning system and instructing the Farmbot. Users need to provide their own implementation because the text format of plans produced by each different planner requires a different parsing algorithm and each different robotic platform requires a different communication format. They are essentially an 'adapter layer' of the different user-chosen planners and robots. However they should comply with several requirements, particularly on how they take in the input (i.e. raw plan and action message) and provide the output (i.e. parsed plan and feedback message), these are mainly for correct interfacing with other ROSplan components.

The planner interface is very trivial to implement, the official ROSplan project [71] provides an example template, the only aspect we need to replace is the parsing algorithm, more precisely how to parse the plain-text planner output and extract the individual actions from the plan, then publish them in a list of action objects to the 'parsed plan' topic. The action interface, on the other hand, requires some more clarification, we will explain them in the following section VI-B.

B. Action Interface

The action interface connects the ROSplan and the Farmbot. when the ROSplan planning system starts dispatching the plan, it will publish the first action in the plan to a ROS topic 'action_dispatch', then it will wait for a Boolean true or false message from the ROS topic 'action_feedback' which indicates whether that single action has been carried out successfully ('action_dispatch' and 'action_feedback' correspond to the Action and Feedback ROS topics in Figure 26). If it receives true, then it will publish the second action and wait for the feedback, and so on. Otherwise, it will abort the entire program. The user-implemented action interface is responsible for two things: (1) taking the action messages from the 'action_dispatch'

topic, translate it into direct instructions and send it to the relevant robotic platform. (2) Query whether the action is successfully executed according to the different feedback mechanism employed by different platform, and publish that feedback of success as a true or false message to the 'action_feedback' topic.

The Farmbot programming API is based on the MQTT protocol [72], it receives the instruction by listening to any instruction message published to MQTT topic 'from_clients', and publish any feedback message to the MQTT topic 'from_device'. The format of these messages is a raw JSON string that contains a message id indicating the purpose of the message (instruction or feedback) and a message body with relevant parameters. For instruction message, the message body contains the string name of the action, and the parameters for the action. For the feedback message, it contains the status id and debug messages. The action interface should subscribe to the ROS action topic for any dispatched action, translate the action to the instruction message and send to the 'from_clients' MQTT topic, then it keeps listening to the 'from_device' MQTT topic, if the message arrived is the relevant feedback message to the last sent instruction, then it parses the message and sends it to the ROS feedback topic.

In the PDDL modelling, we model the positions on the garden bed as atomic propositions. However, when actually instructing the Farmbot, we need to provide the (x, y, z) coordinate as the parameters to the action 'move' instead of the proposition. This is done by the action interface as well, which translates each proposition representing the position object to the corresponding (x, y, z) coordinate. We store the 'proposition to coordinate' pair in a lookup table and populate it with all the positions when the program starts.

The implementation of action interface can be expressed as the following pseudo-code in Algorithm 1:

Algorithm 1: Action Interface

```
current_action_id := null;
lookup_table := {position1: (162, 190, 220), position2: (...), ...};

ROS.Subscribe("action_dispatch", actionHandler);
/* subscribe to ROS topic "action_dispatch", when message comes, handle it with
   function actionHandler                                                      */

MQTT.Subscribe("from_device", feedbackHandler);
/* subscribe to MQTT topic "from_device", when message comes, handle it with
   function feedbackHandler                                                    */

Function actionHandler(actionMsg)
    currentAction := actionMsg;
    instructionMessage := actionToInstruction(currentAction);
    MQTT.Publish("from_clients", instructionMessage);
    current_action_id := instructionMessage.actionId;
end

Function feedbackHandler(farmbotFeedback)
    feedback := translate(farmbotFeedback);
    if feedback.actionId == current_action_id then
        ROS.Publish("action_feedback", feedback.success);
        /* success is true or false Boolean value                              */
    end
end

Function actionToInstruction(action)
    ... /* This function takes in the action message from the ROS action topic, and
       convert it to the instruction message in JSON string format, including
       convert the proposition of positions into coordinates that are
       understandable by the Farmbot. Requires lookup_table                    */
    return instructionMessage
end

Function statusToFeedback(farmbotFeedback)
    ... /* This function takes feedback status message from the Farmbot that is
       published to the MQTT 'from_device' topic, and parses the message for
       information such as whether the action succeeded, the action id etc.    */
    return feedback
end
```

Figure 27 illustrates how these different handlers and topics interact.

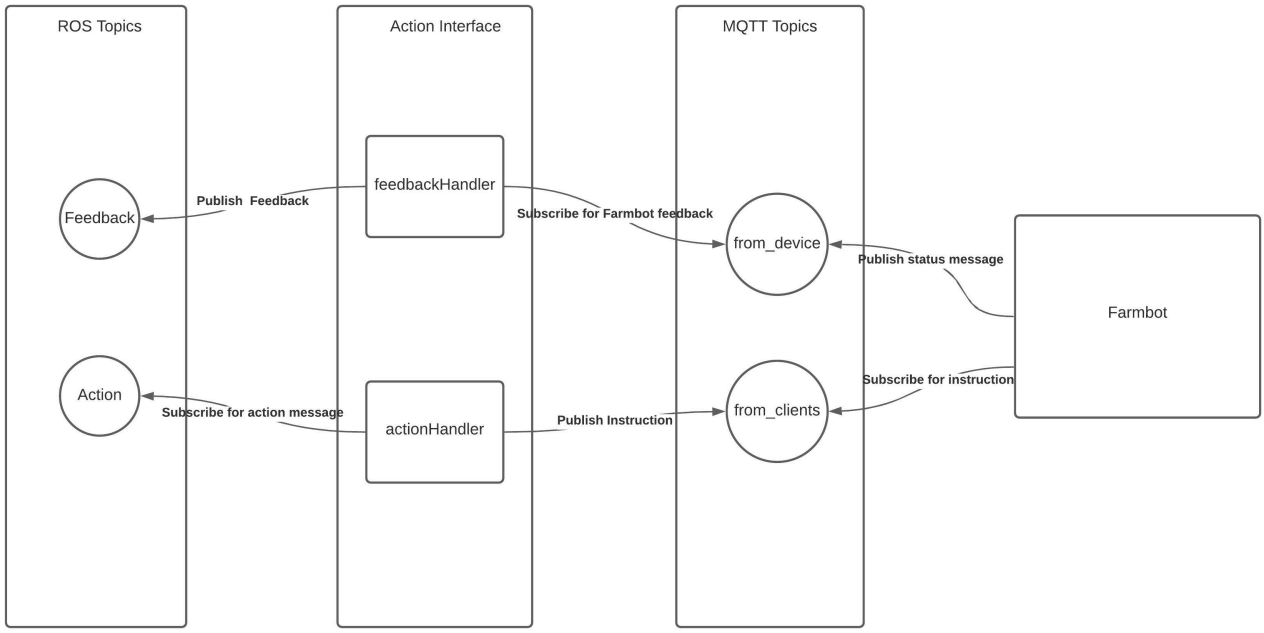


Fig. 27: Illustration of the action interface in details

VII Agent Planning Program over The Farmbot PDDL Domain

In pure PDDL planning, certain aspects cannot be automated, especially when the tasks can only be achieved through a sequence of plans where the autonomous agent should reason about the consecutive planning goal one after another. In this section, we utilize the APP to achieve the goal reasoning and continuous planning over the Farmbot domains. Besides, our application requires us to extend the original APP, particularly on interactive goal reasoning and online plan realization where autonomous agent interacts with the environment (mainly through sensing) and receive information that will affect the world state for the next planning problem. The necessary adaptions will be proposed. Furthermore, we will describe how we implement and integrate our APP with the ROS-based controlling program.

A. *The Agent Planning Program*

Agent planning programs are transition system that allows the autonomous agent to traverse through a series of planning problems over a shared domain. Consider the tasks described in Table I. When the agricultural process is fully automated, the Farmbot should automatically go through the following process: it starts from an initial world state with an empty garden bed. Then, it places seeds at some desired position to grow plants, by doing so, it arrives at a state where it should then look after those plants. Therefore, it will alternate the tasks of checking the moisture of soil for each plant, watering them if necessary, detecting the weed around each plant and removing them if any.

The APP that enables the described transitions will need to have the appropriate program states so that the corresponding planning problems can take the agent through each of them. Therefore the agent planning program over the Farmbot domain will involve 4 program states:

1) The initial program state p_0 or v_0, corresponding to the program state where Farmbot starts with an empty garden bed.
2) A state p_1, where Farmbot can arrive from p_0 by placing the seeds on all the desired positions, indicating that it arrives at a state where it should then look after those plants.
3) A state p_2, where Farmbot can arrive from p_1 by checking the soil moisture of all the plants. Then, Farmbot can return to p_1 by watering all the plants
4) A state p_3, where Farmbot can arrive from p_1 by detecting the weeds around all the plants. Then, it can return to p_1 by removing all the plants.

The transitions between these states will involve the following planning problems:

1) The transition d_0 from p_0 to p_1, corresponding to the planning problem of starting from an empty garden bed, then placing seeds on all the desired positions. The guard γ should have the (no-plant-at) propositions that describe these desired positions as empty, and the (container-has) propositions that describe available seeds in the seed container. The achievement goal ϕ should have the (plant-at) propositions which indicate that the desired positions have plants on them. The maintenance goal ψ does not seem necessary for this transition and hence, $\psi = \emptyset$.
2) The transition d_1 from p_1 to p_2, corresponding to the planning problem of checking the soil moisture level of all the plants. The $\gamma = \emptyset$ here as no precondition is required for this transition. However, the maintenance goal ψ should have the (plant-at) propositions that indicate the existence of the plant. The achievement goal ϕ should have the (checked-moisture) propositions indicating that all existing plants have been checked.
3) The transition d_2 from p_2 back to p_1, corresponding to the planning problem of watering all the plants and return to the normal 'looking after' state. The maintenance goal ψ should have the (plant-at) propositions that indicate the existence of the plant, same as above. The achievement goal ϕ should have the (watered) propositions indicating that all existing plants have been watered. The guard γ can be empty as there is no other transition which the agent can choose.
4) The transition d_3 from p_1 to p_3, corresponding to the planning problem of detecting the weed around all the plants. The $\gamma = \emptyset$ here as no precondition is required for this transition. The maintenance goal ψ should have the (plant-at) propositions that indicate the existence of the plant. The achievement goal ϕ should have the (checked-weed-exist) propositions indicating that all existing positions with plants have been checked.
5) The transition d_4 from p_3 back to p_1, corresponding to the planning problem of removing all the weeds and returning to the normal 'looking after' state. The guard should have the (weed-at) propositions indicating the presence of weed and the maintenance goal ψ is not necessary and hence, $\psi = \emptyset$. The achievement goal ϕ should have the (weed-removed) propositions for all positions indicating that all weeds are removed.

This transition system can then be visualized as the following APP-like structure, in Figure 28.

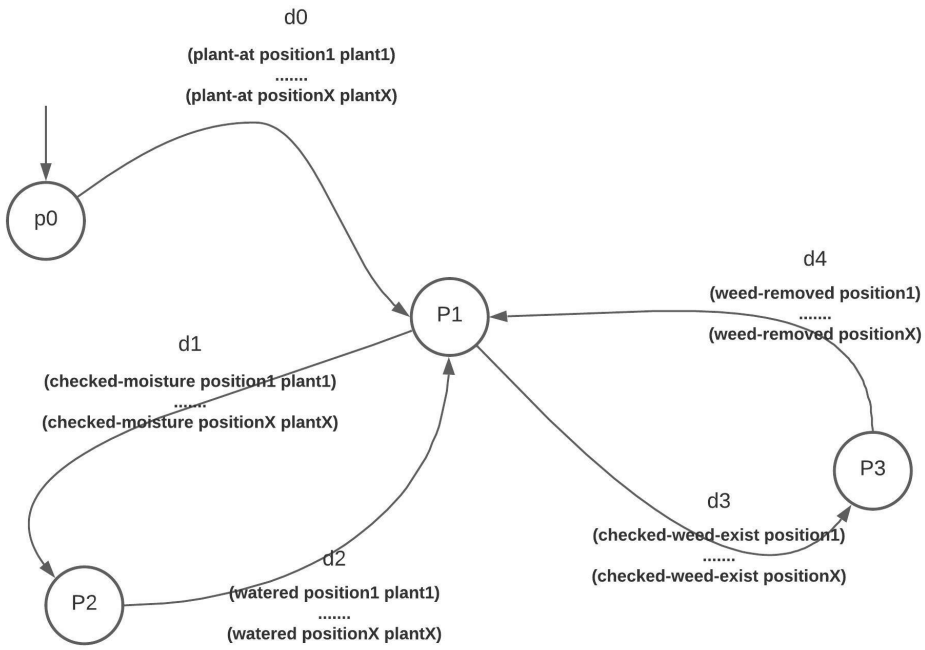


Fig. 28: How the APP over the Farmbot domain looks like, only the achievement goal is shown

However, what we have here is essentially a 'task reasoning' model instead of a full APP definition. In contrast to the original APP, here the exact content in the guard, maintenance goal and achievement goal for each transition cannot be determined unless the garden layout, more precisely, the exact objects (positions, plants, seeds, etc.) are known. The original APP requires the exact definition of each transition in terms of the five-tuple $\langle v, \gamma, \psi, \phi, v' \rangle$, meaning that the description of 'watering all plants', 'placing seed on all positions' are not sufficient to define the APP in terms of the original definition because the propositions on exactly what plants and what positions, etc. are not provided. Instead, the transition system in Figure 28 can be instantiated on different garden layouts. What it provides is a definition on what is the next high-level planning goal for the agent to take. Therefore, our APP is essentially a model for the transition of the 'task' states on Farmbot, and its detailed definitions of the transitions are template can be instantiated on different numbers of position objects and plant objects (different garden layout).

This adaption allows us to define the APP for our domain as a formal method for goal reasoning. Despite different garden layouts, the structure shown in Figure 28 indicates that the high-level transition goal will always remain the same. We can define our APP over the Farmbot domain as a tuple $A = \langle F, V, v_0, s_0, D \rangle$ where F is all the propositions in the classical domain we proposed above. V is the set of all program

state $\langle p_0, p_1, p_2, p_3 \rangle$, v_0 = p_0. s_0, the initial world state, corresponds to a set of propositions that describe an empty garden bed with many position objects, the Farmbot and the initial position it is at, all the tools rested on the tool bay and all the seeds in the seed container. And, $D = \angle d_0, d_1, d_2, d_3, d_4 \rangle$, the set of all the transitions described above. Though the exact definition of each transition, particularly the exact propositions in the guard, the maintenance goal and the achievement goal depend on the garden layout. These exact garden layouts will be explained when we describe the scenarios for the simulation in section VIII.

B. Changes to The Original PDDL Modelling and Implications

Furthermore, The original PDDL domains have to be modified slightly. Considering the transition d_1 and d_2, where the same actions will be applied to all the plants in the garden bed. In reality, not all the plants require watering every time after the Farmbot checks their soil moisture with the soil sensor. It depends on the sensor readings. ROSplan provides us the ROS services to modify the current world state stored in the knowledge base as mentioned in the literature review. Therefore, with the ROSplan, during the transition d_1, each time a plant is checked with the soil sensor, the action interface can actively update the world state by inserting the predicate (need-water) or (not-need-water) according to the sensor reading. When later the transition d_2 happens, the plan towards the same transition goal will only water the plants that require watering. However, to reach the transition goal of watering all the plants, there should be the (watered) propositions for all plants in the goal state, even for those that do not require watering. Therefore we introduce another action as follows:

```
(:action skip_watering_plant
    :parameters (?x - position ?p - plant)
    :precondition (and
        (not-need-water ?x ?p)
        (plant-at ?x ?p)
    )
    :effect (and
        (not (not-need-water ?x ?p))
        (watered ?x ?p)
    )
)
```

The semantics of this action should be interpreted as: for those plants that do not require watering according to the soil sensor reading, the current world state is updated with the predicate (not-need-water)

for each of them when Farmbot checks the soil moisture. Then later when planning towards the transition goal of watering all the plants, the planner can generate the plan that does not water the unnecessary plants, but the goal condition (watered) in the goal state can still be reached even though some of the plants will not have the (need-water) proposition. A slight modification is also required in the action interface, when the action dispatcher sees this action, it will just ignore it and do not send any instruction to the Farmbot. Therefore, no action will be taken on those plants that do not require watering but the transition goal of watering all plants is still reachable. This is a typical case where we utilize the ROS and ROSplan to accomplish something that is not feasible on a PDDL-only system.

This is also the case for the transition d_3 and d_4, which require the action interface to actively update the world state indicating whether there is any weed detected around each plant. Therefore, different plans are required for removing the weeds and transition towards the same goal. This mean a similar action to the one above should be introduced in both PDDL domains:

```
(:action skip_removing_weed
    :parameters (?x - position)
    :precondition (and
        (farmbot-at ?x)
        (no-weed-at ?x)
    )
    :effect (and
        (weed-removed ?x)
        (not (no-weed-at ?x))
    )
)
```

There are several implications to these modifications:

1) The state trajectory during the APP transition is now affected by the sensor reading, meaning that the agent is now 'interactive' and the actions have fully observable non-deterministic effects that depend on the environment or the simulator, and the plan realization for the same transition will have different resulting world states.
2) Therefore the same program state of the APP, particularly for the program state p_2 and p_3, will now have a dynamic corresponding world state each time the agent moves to them due to the conditional action effects to the state trajectory. For example in p_2, different plants will have the (need-water) propositions in the corresponding world state each time the agent arrives at this program state by checking the soil moisture of each plant.

3) This further implies that some transitions, specifically d_2 and d_4, will probably start from a different world state each time, even though the program states are always the same.
4) For them to reach the same transition goal from a different initial world state of that specific planning problem, a different plan is required for that specific transition each time.
5) This means that the detailed plan can only be determined when the autonomous agent is at the start program state for that transition, because every time the initial world state at that start program state is different. Pre-computed plan realization of the original APP proposed in [18] will not work. We have to have some approach that allows us to alternate the planning for the next transition goal and the transitioning (execute that plan and move towards the transition goal). And, the planning should happen just before the transitioning at each program state when the corresponding world state is known.

The original APP proposed in [18] did not specify whether each program state should correspond to a unique world state. However, from the plan realization approach in [18], plans for all transitions are pre-computed, assuming a fixed unique world state for each program state. This is not suitable for our application because if that is the case, for every unique garden layout and number of position objects, there will be 2^n unique program states replacing each of our program states p_2 and p_3, where n is the number of position objects (same as the number of plants). This is because, for all the n plants in the garden bed, each of them can have a (need-water) or (not-need-water) propositions in the resulting world state in program state p_2, similar for p_3. If we adopt the original APP, the effort in constructing such an APP will be unimaginable. Instead, we relax the original APP to allow one program state to correspond to multiple different actual world states, except for the initial program state. We consider this as another adaption to the original work on APP, and adding value to it for real-world applications. However, as mentioned before, this will require a different plan realization approach to the original pre-computed approach. It will be explained in detail in the following section where we discuss the implementation.

C. Implementation and Integrating with The ROS Project

The actual implementation of the agent planning program will require us to extend the ROS project from section VI and implement a controlling program based on the ROSplan planning system so that the Farmbot can generate multiple planning problems, execute plans that allow it to transition between the program states, and accomplish multiple goals. When the project starts, the ROSplan will first instantiate the knowledge base with the initial world state of the APP defined in a problem file. Then, the controlling

program will take in an APP definition and construct the transition system. After this, it starts from the initial program state, selects the next applicable transition, finds a plan towards that transition goal, and finally executes the plan and transitions to the next program state. It continues the transition from one program state to the other program state and so on, according to the APP definition. There are several things we will clarify in the following subsections:

1) How the APP model, like the one in Figure 28 is represented in order for the controlling program to take in the APP definition and parse it to construct the transition system?
2) What is the initial world state and how can we utilize the ROSplan to set up this initial world state in our ROS program?
3) How the transition process, or the plan realization is implemented in the controlling program and How we utilize the ROS services provided by the ROSplan to implement it?
4) During plan realization, how the next applicable transition is selected (goal reasoning)?

1) ***Representing the agent planning program***

For the program to understand the APP over the Farmbot domain, we utilize the JSON format to represent the APP definition. The information in the JSON file includes all its program states and the transitions between them. The following code fragment shows part of the full JSON representation of the APP over the Farmbot domain in Figure 28:

```
{
    "States" : [
         "p0",
         "p1",
         "p2",
         "p3"
    ],
    "Transitions" : [
        {
            "name" : "d1",
            "StartState" : "p0",
            "EndState" : "p1",
            "Guards" : [
                {
                    "predicate" : "no-plant-at",
                    "parameters" : {
                        "x" : "position1"
```

```
            }
        },
        ...
    ],
    "MaintenanceGoals" : [],
    "AchievementGoals" : [
        ...
    ]
},
{
    "name" : "d2",
    "StartState" : "p1",
    "EndState" : "p2",
    "Guards" : [],
    "MaintenanceGoals" : [
        {
            "predicate" : "plant-at",
            "parameters" : {
                "x" : "position1",
                "p" : "plant1"
            }
        },
        ...
    ],
    "AchievementGoals" : [
        ...
    ]
},
...
{
    "name" : "d4",
    "StartState" : "p3",
    "EndState" : "p1",
    "Guards" : [
        ...
    ],
    "MaintenanceGoals" : [],
    "AchievementGoals" : [
```

```
                ...
            ]
        }
]
}
```

This definition contains a set of program states, and a set of transitions each is represented as a JSON object containing the full definition of the 5 tuple $\langle v, \gamma, \psi, \phi, v' \rangle$. Each of the guard, maintenance goal and achievement goal is a set of propositions, and each of the propositions is represented as a JSON object containing the predicate name and the parameters that instantiate the predicate to form the full proposition.

The controlling program will parse this whole JSON object and construct a transition system with a logical structure resembling the structure shown in Figure 28. Then the agent transition will base on such a logical transition system.

2) ***Initial world state***

Because our controlling program is based on the ROSplan framework, when the ROSplan starts, it will always take in a problem file that is the initial planning problem used to instantiate the knowledge base. We utilize this to instantiate the knowledge base with the initial world state of the agent planning program. The initial world state, as mentioned before, describes an empty garden bed with all the empty position objects, the available seeds in the container, the initial Farmbot position, and all the available tools. The following fragment of PDDL problem file describes the initial world state

```
(define (problem initial-world-state) (:domain classical-domain-farmbot)
(:objects
    home position1 position2 position3 ... - position
    seed1 seed2 seed3 ... - seed
    plant1 plant2 plant3 ... - plant
)
(:init
    (farmbot-at home)
    (carry-camera)
    (tool-mount-free)
    (seeder-free)

    (tool-rack-at seederrack seeder seederPos)
    (tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
```

```
    (tool-rack-at weederrack weeder weederPos)
    (tool-rack-at soilsensorrack soilsensor soilsensorPos)

    (tool-at seederPos seeder)
    (tool-at wateringnozzlePos wateringnozzle)
    (tool-at weederPos weeder)
    (tool-at soilsensorPos soilsensor)

    (container-at posSeedTray seedtray)

    (container-has seedtray seed1)
    (container-has seedtray seed2)
    (container-has seedtray seed3)
    ....

    (match-seed-type-n-plant-type seed1 plant1)
    (match-seed-type-n-plant-type seed2 plant2)
    (match-seed-type-n-plant-type seed3 plant3)
    ....

    (no-plant-at position1)
    (no-plant-at position2)
    (no-plant-at position3)
    ....
)

(:goal (and
))
)
```

The missing parts in this problem file depend on the exact garden layout (number of position objects), the existing part only shows 3 position objects that will require the Farmbot to place seeds on them and grow plants. It can be scaled up according to different garden layouts.

For the second domain where plan metric is applied, we also need to define the additional metric (move-distance) between every pair of position objects as well as instantiate the initial plan cost (total-cost) to 0. Declaration for the plan metric in the (:metric) section should also be included for enabling the corresponding planners to improve the plan quality in terms of the defined plan metric.

3) ***Transitions and online plan realization***

After the program starts, the knowledge base is instantiated with the initial world state and the transition system is established from the JSON format representation of the APP. The program will set current program state to the initial program state in the transition system. Starting from the initial program state, the program will then transition through a series of program states and execute many plans. Considering the modification to the PDDL domain mentioned above, the implication to the APP is that the same program state will have different world states, requiring different plans towards the next transition goal. Plan for the next transition can only be computed after the previous transition is completed and the agent is at a known starting world state. A solution for an APP is called plan realization, the original literature on APP [18] employs an aforementioned offline approach that pre-computes the plan realization as a policy function that maps each unique tuple of program state, world state and achievement goal (transition goal) to a pre-computed plan. For our application, the plan realization needs to be a sequence of alternating transition from new program state and world state and planning for that transition goal from that specific world state. That is, the autonomous agent alternates the plan computation and the execution of that plan. At every program state, the agent is likely to be at a new world state that is never encountered before, therefore instead of asking the policy function for the pre-computed plan for the current program state, corresponding world state and the decided transition goal, the agent actively determines the next transition goal through goal reasoning, and then computes the plan. This approach is referred to as online plan realization [62].

The definition of online realization in the context of our application extends that in the [62], and is defined as follows:

Definition 1. The online realization of the agent planning program $A = \langle F, V, v_0, s_0, D \rangle$ over the Farmbot domain is a sequence of $(s^0, v^0)d^1\pi^1(s^1, v^1)d^2\pi^2...(s^{n-1}, v^{n-1})d^n\pi^n(s^n, v^n)$ where:

- $n \geq 0$
- $s^0 = s_0$ is the initial world state, $v^0 = v0$ is the initial program state
- $d^i = v^{i-1} \overset{\gamma^i, \psi^i, \phi^i}{\rightarrow} v^i = \sigma(D, v^{i-1}, s^{i-1})$ for each $0 \leq i \leq n$, is the selected transition through the goal reasoning function σ that takes the agent from v^{i-1} to v^i. The goal reasoning function σ takes in the start program state v^{i-1}, filtered out the non-applicable transitions from D (i.e. the transitions whose v in $\langle v, \gamma, \psi, \phi, v' \rangle \neq v^{i-1}$ and $\gamma \cup \psi \not\subseteq s^{i-1}$), and select the next transition from the remaining applicable transitions based on tie-breaking strategy we will discuss later.
- π^i is the solution plan of the planning problem whose initial state is s^{i-1} and the goal state is ϕ^i of

the selected transition d^i, and is computed online. The application of π^i results in the world state s^i such that $\phi^i \subseteq s^i$ while ψ^i of the selection transition d^i holds throughout the state trajectory of π^i.

The following pseudo-code in Algorithm 2 describes the implementation in the ROS-based controlling system:

Algorithm 2: Transition system

```
ROS.knowledge_base.reset_current_state("initial_world_state.pddl");
/* Instantiate knowledge base with initial world state s0                    */

app := parse_and_construct_app_from_JSON_file("APP on Farmbot.json");
current_program_state := app.initial_program_state;

while program is running do
    selected_next_transition := σ();
    /* Goal reasoning                                                        */

    ROS.knowledge_base.update_current_goal(selected_next_transition.achievement_goal);

    ROS.planning_system.generate_planning_problem("problem.pddl");
    /* Generate the planning problem from the knowledge base into pddl problem
       file "problem.pddl"                                                   */
    ROS.planning_system.call_planner("domain.pddl", "problem.pddl", "plan.txt");
    /* Run the planner which takes the supplied planning domain, generated
       problem and write the planner output to a plan file                   */
    ROS.planning_system.parse_plan("plan.txt");
    /* This is where user-supplied planner interface comes into play         */

    while not ROS.planning_system.plan_dispatch_completed do
        ROS.planning_system.dispatch_one_action();
        ROS.wait_for_action_to_finish();
        ROS.knowledge_base.check_if_condition_hold(selected_next_transition.maintenance_goal);
        /* Making sure that ψ holds throughout the state trajectory of the plan */
    end

    ROS.knowledge_base.check_if_condition_hold(selected_next_transition.achievement_goal);
    /* Just validating whether the achievement goal or transition goal is met */
    if None of the above steps went wrong then
        current_program_state := app.get_state(selected_next_transition.end_program_state);
        /* Move to the next program state                                    */
    else
        throwError()
    end
end
```

This implementation can be summarized as: starting from the initial program state, the agent selects one applicable transition based on the goal reasoning, updates the ROSplan knowledge base with that transition goal via the provided ROS services, finds the plan towards that transition goal from the current world state, dispatches the plan for it to be executed by the Farmbot and makes sure the maintenance goal of the current transition holds throughout the state trajectory of the plan, and finally validates that the transition goal is achieved and moves to the next program state.

*4) **Goal reasoning and tie-breaking***

In the online plan realization and transition we described above, the most important part is the 'selection' of the next transition goal, which gives this entire process the 'online' property. This selection is goal reasoning, and it is about formulating and selecting the goal for the next planning [65]. Since the transition goal is essentially the achievement goal of the selected transition, therefore the goal reasoning in the context of our application should be about selecting the one applicable transition. Determining whether the transition is applicable is mainly based on two criteria: (1) whether the start state of the transition matches the current program state, and (2) whether the guard and maintenance goal hold in the current world state. This is also another reason why the goal reasoning needs to be carried out in an online approach before the planning because the corresponding world state for each program state is dynamic and depends on the state trajectory caused by the previous plan.

However, for our application, an important observation indicates that having more than one applicable transitions from a given program state is possible. For example in p_1 we will have a non-deterministic transition, it can go through both d_1 and d_3 as they are equally applicable according to the above two criteria. Also, another important observation is that we want the autonomous agent to alternate between equally applicable transitions so that all transitions are ensured to be visited, instead of just randomly selecting one and resulting in uneven distribution of the number of executions between these equally applicable transitions. The original APP will not enable us to break ties meaningfully, therefore we extend the definition in the following way:

Definition 2. The extended agent planning program A is defined as a five-tuple $A = \langle F, V, v_0, s_0, D, \sigma \rangle$ where:

- F, V, v_0, s_0 remains the same as the original definition.
- σ is the goal reasoning function.

- D is a set of transition $D = \langle d_0, d_1, d_2, d_3, ... \rangle$ where each transition d is defined as a six-tuple $d = \langle v, \gamma, \psi, \phi, v', h \rangle$. $v, \gamma, \psi, \phi, v'$ are the same as the original definition, h is a newly introduced element called $history$. h is initialized as 0 and records the number of times this transition has been selected during the online plan realization of that APP.

When we have more than one equally applicable transitions according to the two criteria above, the selected applicable transition is the one with the least h among them. Therefore, we extend the APP in such a way so that we can break ties meaningfully and the goal reasoning should always result in the transition goal from only one applicable transition. The aforementioned goal reasoning function σ can be implemented as the following pseudo-code in Algorithm 3:

Algorithm 3: Goal Reasoning

```
Function GoalReasoning
    selected_transition := null;
    for d ∈ D do /* For each transition d in the set of all transitions D */

        start_state ← d.start_state;
        conditions ← d.guard ∪ d.maintenance_goal;

        if ROS.knowledge_base.condition_hold(conditions) && start_state == current_program_state then
            if selected_transition ≠ null then
                selected_transition := whichever_has_smaller_h(d, selected_transition);
            end
        end
    end

    assert selected_transition ≠ null;
    selected_transition.h++;
    /* Increment history by 1 */
    return selected_transition;
end
```

VIII Simulation

In this section, we present the results of a simulation experiment where we deploy the whole ROS project we implemented above, which is the controlling program including the planning, the plan executing and the APP to a simulator. The simulator is essentially a virtual Farmbot that will receive and execute the dispatched action from the ROS-based project and provide the action feedback just like the real Farmbot.

There are 3 main purposes for the simulation:

1) To further evaluate the correctness of our PDDL domain modelling proposed in section IV and answer the RQ1 with combined evidences across previous evaluations and experiments.
2) To gather and analyze more data on the performance difference between the first classical domain and the second classical domain with plan metric, and combine with results from the benchmarking experiment to find conclusive answers to the RQ2.
3) Last but not least, to evaluate our APP, and show that our implementation allows the sequence of transitions of APP, especially the correctness of the goal reasoning, the tie-breaking and online plan realization, which enable the autonomous agent to continue planning after the previous transition is traversed. It helps us answer the RQ3 and evaluate whether our adaption to the original APP is successful.

Furthermore, this simulation also aims to show that the overall system is working as expected. The simulation is needed due to the extended lockdown, as the original intention is to evaluate the system on the actual Farmbot.

*A. **Simulation Experiment Scenarios and Settings***

From the results and analysis of the benchmarking experiment in section V, we know that when the problem size is relatively small, particularly when below 20, the pure classical domain with FF planner has to potential to become a better choice as the plan quality is not significantly worse while having the advantage of not needing the extra time to optimize the plan metric. The domain with the plan metric shows the significant advantage as the problem size becomes larger. Furthermore, the artificial layouts are matrix-based and very simple, also the Manhattan distance used for the plan metric is discrete-valued, which might result in relatively easy search problem configurations and hence, affects the performance of planners. In the simulation, we can narrow down the scenarios to some specific garden layouts that are more natural compared to the matrix-based ones. The results showing different advantages of domains

and planners in different problem sizes also suggest that in order to thoroughly compare the two domains, the simulation needs to be done on both a small size problem and a large size problem, more precisely, a small size garden with a relatively smaller number of objects, and a large size garden with more objects.

Therefore, we conduct the simulation on two garden bed layouts. The smaller one contains 15 position objects, whereas the larger one contains 50 position objects. 15 is chosen for the smaller garden because we are interested in seeing the results when the problem size is below 20, but we want neither to be too small nor at the exact maximum. Therefore, 15 is considered a number in the middle. 50 is chosen for the larger garden bed as it is sufficient for a large problem size, also any number beyond will not be meaningful as the real garden is not large enough to accommodate that. Figure 29 and Figure 30 show these two garden layouts.

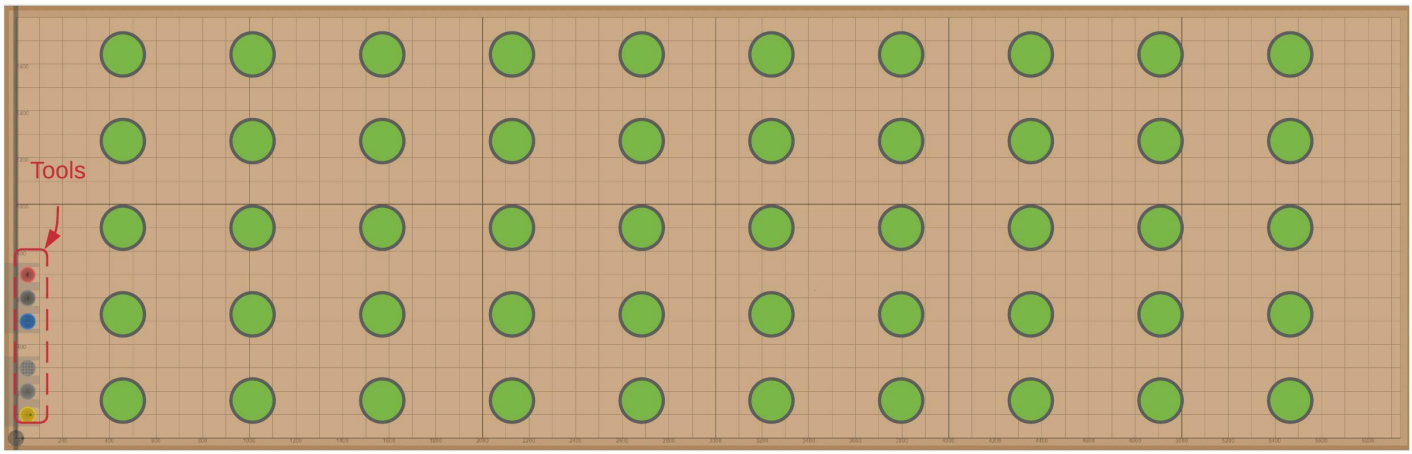


Fig. 29: A garden layout for the simulation, drawn on the real 1.8m x 6m garden, viewing from above, containing 50 position objects. The area highlighted is the tool bays, containing the necessary tools such as weeder, seeder etc. The coordinates are shown in mm, the bottom-left is (0, 0), the scale is annotated at every 200 mm

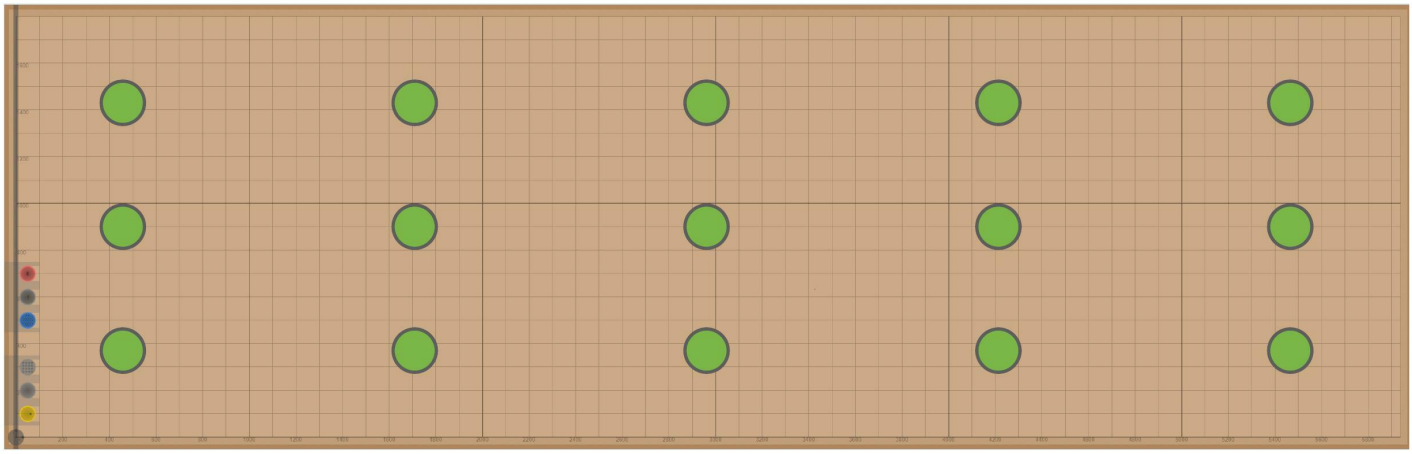


Fig. 30: A garden layout containing 15 position objects for the simulation

The position objects are arranged in horizontal manner, as illustrated in Figure 31.

pos11	pos12	pos13	pos14	pos15
pos6	pos7	pos8	pos9	pos10
pos1	pos2	pos3	pos4	pos5

Fig. 31: Illustration of arrangement and ordering of the position objects, correspond to the garden layout in Figure 30.The larger garden layout is arrange in the same way, just containing 50 objects

However, different from the benchmarking experiment, the exact position of each object now is its own (x, y) coordinates as in Figure 29 and Figure 30, instead of the index in the matrix. Meaning that the action cost plan metric for the domain action (move) in the second PDDL domain, which is defined by the travel distance for Farmbot to move its tool tip from one position to the other is calculated as the Euclidean distance between these two positions. Denoted by the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Furthermore, the positions of each tool (the highlighted part in Figure 29) and the distance from each other position is now taken into consideration, instead of being assigned as 1 in the benchmarking experiment (We did so because we thought the Farmbot would always need to pick up the necessary tool for the task in order to reach the goal state, therefore the distance to tools cannot be optimized anyway). Therefore, we have garden layouts and the resulting problem configurations that are closer to the real garden environment.

Both domains proposed in section IV, also the 4 planners, 2 for each domain will be run in the simulation. They are combined with the ROS-based project along with agent planning program which is for continuous planning and goal reasoning. All these components will be integrated into a single ROS project that instructs the virtual Farmbot in the simulator to perform tasks. The details of the implementation will be explained in the next section. Considering the two different garden layouts, there will totally be 8 different simulation configurations.

Each configuration will be run for 21 transitions. In the APP definition, the autonomous agent first grows plants at all the positions, then rotate between checking the moisture then watering all plants (or making sure plants are watered) and detecting weeds around all plants and removing them. 21 transitions will allow the program to grow plants and rotate the aforementioned 4 transitions 5 times. We decide on this number because we also conducted some preliminary experiments beforehand, and 21 transitions will be sufficient for us to observe an adequate amount of continuous transitions and evaluate the concatenated sequence of planning problems with different initial states and goals.

In addition, the same simulation configuration will be repeated 5 times. Theoretically, the same planning problem with the same domain configuration will always result in the same plan by the same planner, therefore 1 time will be sufficient. However, we choose to run 5 times because we need to consider that our adaption to the APP will result in certain randomness as some transitions will require a different planning problem each time, leading to different state trajectory and resulting world state due to the conditional action effects caused by the possible sensor's update to the current world state. In this case, there is no need to take the average of any data from the same configuration as the planning problems for each simulation configuration could be different due to the randomness, we will simply show the results of all 5 repetitions. Running each configuration more than 5 times will further minimize the influence of the randomness on our conclusion. However, after some preliminary experiments, we consider running each configuration more times will not be meaningful and critical to the result analysis.

The data we will collect from the simulation is the total time taken from the start to the end for each simulation configuration. This total time includes the time taken for the ROS program to start, the time taken for the autonomous agent to continuously go through 21 transitions during which it reason the next goal at each program state, the time taken to calculate the plan during online realization and the time taken to dispatch and execute the plan for each transition. In addition, the single data of the total time can be broken down into several different time duration and recorded separately. They are:

- The total time taken for the planners to calculate the plan for the online plan realization of all 21 transitions during the simulation. This will tell us whether the time taken by the planners alone is significant. For the two anytime planners, the time taken will always be the timeout limit, which is much longer than the two non-anytime planners. This will allow us to find out in the context of deploying the system, whether spending extra time with the two anytime planners on the domain using plan metric allows us to find a better plan which might reduce the total time taken.
- The time taken for dispatching and executing the actions. The execution of plans will take time in the simulator, and a less optimized plan will take more time to execute. This allows us to quantitatively compare the effect of plans with different quality. Combine with the total time taken by the planners, the results will give us a suggestion about the trade-off, that is whether the time saved on the execution of the optimized plan can compensate the extra time taken by the anytime planners. This part can be further broken down into two different time duration:
    1) The time spent on dispatching and executing the (move) actions. In the domain modelling, the action cost is essentially equivalent to the routing distance for the farmbot to complete all the tasks. The cost is incurred only when the Farmbot is moving from one position to the other, so the action cost will only occur when the simulator is executing the action (move). Therefore, this time is correlated to and can be used to represent the total action cost. It will combine the time taken for all the (move) actions in the 21 plans executed for the 21 transitions. Taking this out alone will allow us to analyze the plan quality between different planners and the impact of that on the whole execution time. This is the part where the plan quality can be optimized, a plan with worse quality will mean that more time is spent on the action (move) by taking a less optimal route.
    2) The remaining dispatching and executing time, which is the part that cannot be optimized. Taking it out and inspecting it alone will allow us to identify and isolate the factors that are not influenced by the plan quality, and allow us to have a more accurate conclusion regarding whether the action cost is necessary.
- The remaining time of the total time taken for each simulation. This includes some program overheads as well as the overheads of the APP. Inspecting this part alone will allow us to show that whether our adaption and implementation of the APP and the rest of the system add additional time complexity.

Since the time taken by the planners will be taken into consideration, setting the timeout limit carefully for the two anytime planners is crucial. According to the result analysis of the benchmarking experiment, generally for the problem size of 15, the time taken for the last found plan is generally less than 5 seconds.

It is meaningless to let the planner continue to run after that because it generally cannot find any more plan afterward. For the problem size of 50, we decide that 30 seconds for the Dual-BFWS-LAPKT and 50 seconds for LAMA according to Figure 25. Therefore, we will allow 5 seconds timeout limit for the two anytime planners over the second PDDL domain on simulation configuration involving the smaller garden layout (15 position objects) and 30 seconds for Dual-BFWS-LAPKT and 50 seconds for LAMA on the larger layout (50 position objects).

The direct observation of the simulation is also important, we will discuss any significant observation later in the discussion. The experiment will be run on the same desktop computer as the one used for the benchmarking experiment above, the additional specification will includes a GPU of ASUS GeForce RTX 3080 TUF as the simulator is built on a game engine.

B. Infrastructure for Simulation and Implementation

1) Implementation of the simulator and integration with the ROS project

The simulator is implemented in the Unity Engine [20] as a 3D project. Figure 32 is a screenshot of the Unity scene showing the virtual Farmbot in the simulator. The 3D CAD model was provided as a free resource by the official Farmbot provider [73], whereas all additional components, including the scripting, moving animation etc. in the simulator have been implemented from scratch and will be made publicly available.

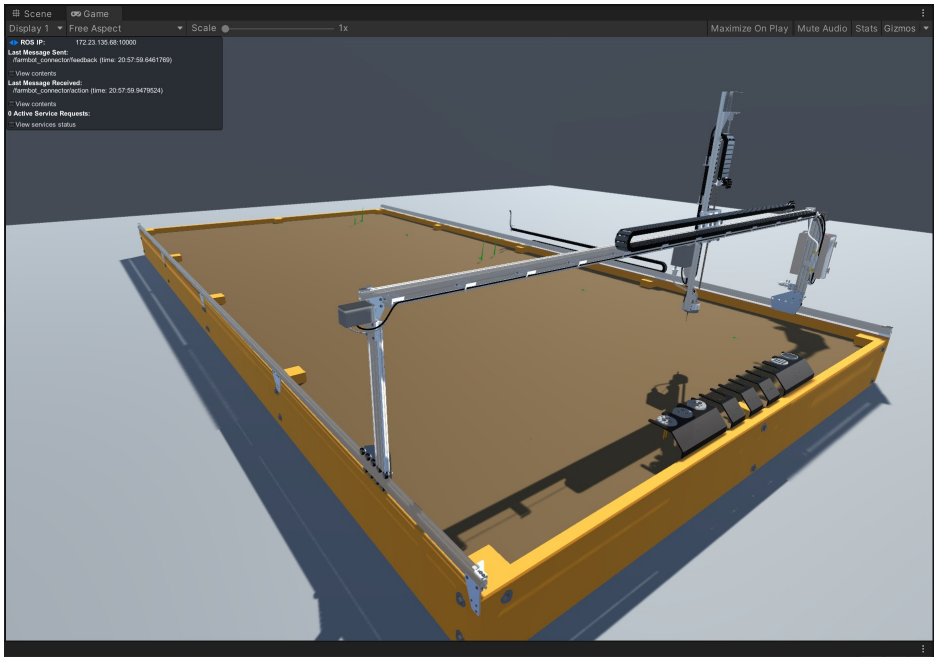


Fig. 32: The Farmbot simulator, screenshot taken during a simulation run, part of the garden already has plants and weeds

Back in Figure 26, we also showed how the planning components based on ROS are combined with the Farmbot. For the simulation, the real Farmbot is replaced by the simulator. That means the simulation infrastructure is consisted of 2 components: the Unity project, which is the simulator itself and the aforementioned ROS project that combines agent planning program, planning, and plan dispatching. These 2 components are bridged using a Unity package [74] provided by the official Unity developer, which allows the Unity to communicate with ROS using the same publish and subscribe message protocol like ROS. The low-level message conversion, serializing and de-serializing will be handled by the package and the user treat the whole system as within the ROS message system. The simulator functions like the real Farmbot, it will receive and execute dispatched actions and provide feedback. However, different from the real Farmbot, we can directly utilize the same message protocol as ROS and minimize the complication of relaying message from ROS to MQTT, and vice versa. The original action interface will also need to be slightly modified, the general implementation is still the same, except that it will just take the dispatched action from the planning system, then instruct the simulator instead of the real Farmbot. Figure 33 illustrates the architecture of the entire simulation infrastructure, extending from Figure 26 and Figure 27.

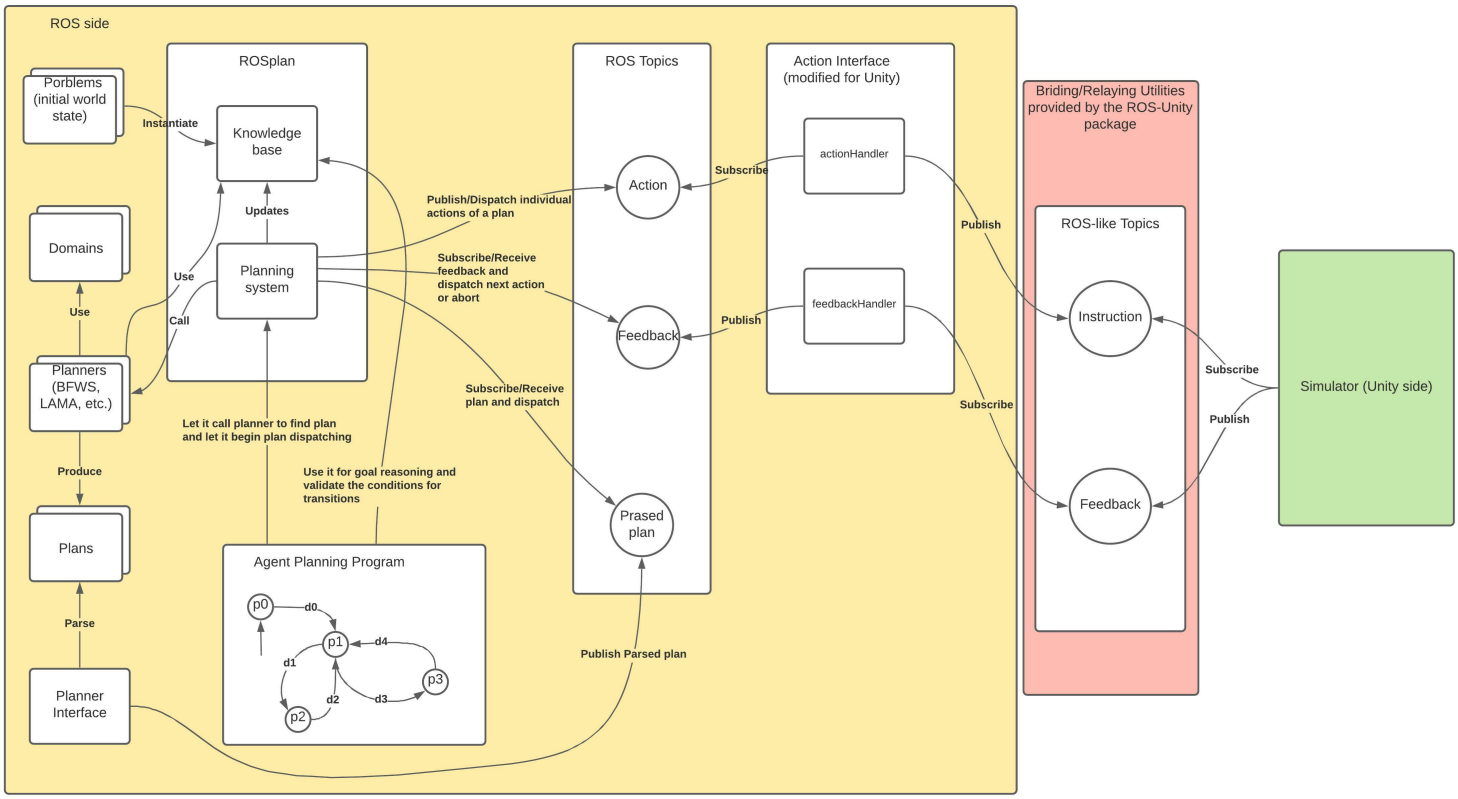


Fig. 33: High-level architecture of the entire simulation infrastructure, including the aforementioned ROS project that incorporates the agent planning program and the modified action interface for the simulator

*2) **The process of simulation program***

The whole process of the simulation can be summarized as follows. Once the ROS project starts, the agent planning program will start from the initial program state, then start the transition by reasoning about the next transition goal, creating the next planning problem and finding the plan, then it dispatch the plan for the simulator to execute, complete that transition and move to the next program state. It goes on for 21 transitions before that simulation end while the relevant data is collected. The same thing will be done on all 8 configurations over 2 domains, 4 planners (2 for each domain) and 2 garden layouts. Each configuration will be repeated 5 times. Figure 34 is a process diagram that illustrates the whole process.

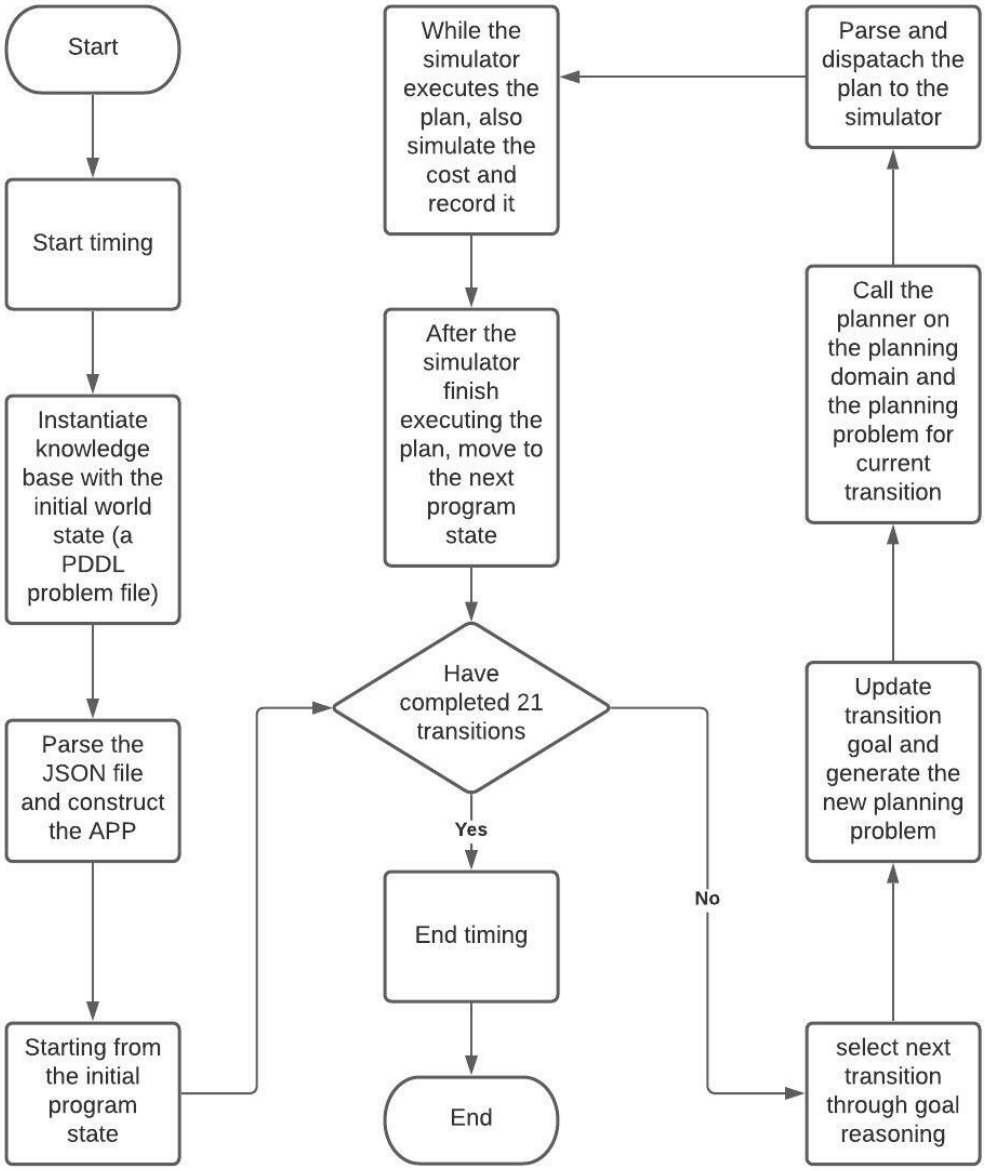


Fig. 34: Process diagram of the simulation

3) ***How the moisture level is simulated***

As mentioned in section VII-B, the modelling and implementation decision allows the dynamic state trajectory of certain transitions and the resulting world state of those transitions to be dependent on the sensor readings. Specifically the plan that check the moisture of all the plants will result in a dynamic initial world state for the next transition where some plants require watering and others do not. Therefore, we need a mechanism in the simulator that updates the soil moisture level of each plant, and provides the controlling program a way to determine the moisture and update the ROSplan knowledge base accordingly when dispatching the action (check_need_water).

Therefore, we create a Unity script dedicated for this, and assign it to every existing plant objects in the simulator. It will keep a variable that indicates the moisture level of the soil around this plant. This variable will be instantiated to some certain initial value, and will be reduced by certain amount per unit time, which simulates the losing of soil moisture over time. Each time the (check_need_water) is dispatched to the simulator, it will compare this variable to a pre-defined threshold. If this comparison indicates a less-than relationship, the current world state in the knowledge base will be updated with the proposition (need-water) for that corresponding plant, and otherwise if the variable is higher than the threshold. When the (water_plant) action is dispatch, this variable of the corresponding plant object will be increased by certain amount.

There are several parameters for simulating the moisture level:

- Initial moisture level when the script is assigned.
- The amount to reduce the moisture level per unit time.
- The time unit to reduce the moisture level by certain amount.
- The threshold by which whether the plant requires watering is determined
- The amount of moisture added each time the (water_plant) is executed on the corresponding plant object

After some preliminary experiment, these five parameters are set to 20, 0.02, 1 second, 50 and 10 respectively. Since the major interest of this simulation is to evaluate the whole system, therefore these values are selected based on our decision to make sure that we can cover as many scenarios as possible. For example, if the moisture is losing extremely slow, then for the entire simulation, all the plants would probably never require watering and we always see the exact same planning problem for the same transition and hence, does not enable us to see how the online plan realization work from the same program state in response to the different initial world state.

4) ***How the weed is simulated***

Similar to the soil moisture, the simulator will need to simulate a mechanism where weeds can occur and impact the state trajectory for certain transitions, which causes the dynamic corresponding world state to some program state. More precisely the transitions that require the agent to detect weeds and remove them. Since the existence of weeds are represented as proposition (weed-at) in the planning problem, and is added to the world state stored in the knowledge base when there are detected weeds around every single plant during the execution of action (detect_weed) for that corresponding plant. The exact

simulation mechanism should randomly generate weeds and allow the controlling program to know which weeds are around which plant, then it can update the knowledge base accordingly.

Therefore, the Unity script created for this simulates weeds in the following process described in Algorithm 4.

Algorithm 4: Simulate Weeds

parameter : spawn_rate, loop_rate, r

while *True* **do**

  for $plant \in ExistingPlantsinGarden$ **do** /* For each existing plant in the garden */

    $ran \leftarrow Random(0, 1)$;

    /* generate a random number between 0 and 1 */

    if $ran > spawn\_rate$ **then** /* spawn_rate is the probability of whether to spawn a weed */

      $spawn\_position \leftarrow RandomlyChoosePosition(r, plant)$;

      /* r is the radius of a circle around the plant where the new weed should be spawned within, it can be done with the Unity API: Random.insideUnitCircle * r + plant.position */

      $newly\_spawn\_weed \leftarrow InstantiateWeed(spawn\_position)$;

      /* spawn a weed prefab at position $spawn\_position$ */

      $plant.weeds \leftarrow plant.weeds \cup newly\_spawn\_weed$;

      /* each plant object in the simulator maintain a list that contains the weeds belonging to this plant, when the action (detect_weed) is executed on the current plant, it will be used to update the knowledge base with the propositions indicating the existence of these weeds, their corresponding objects and positions in the knowledge base and the simulator will also be created. When the (remove_weed) is executed, the weed will be removed from the list and all its corresponding propositions in the knowledge base and the game objects in the simulator will be removed */

    end

  end

  Sleep for some time before the next iteration so the loop rate of the while loop matches $loop\_rate$;

  /* having the weeds being spawned too frequent is not realistic */

end

The parameters for simulating weeds are:

- The loop rate of the repeating the whole weed-generating process. It will help control the frequency of seeing newly generated weeds.
- The spawn rate of weeds, which is a probability to decide whether a weed should be spawn around each plant at each iteration of the while loop. It will also help control the frequency of new weeds, but the different is that this will result in intended uneven distribution of weeds for each simulation. Some plants will have more weeds than other. It allows the next transition to start from different initial world states after the once plan is executed to detect all the weeds due to the random weed generation and hence, cover more possible world states for us to evaluate how well the agent achieve the sequence of continuous transitions.
- The horizontal radius to consider when a new weed is spawn around each plant. This is for controlling how far away the newly spawn weed is from the plant visually, and it help simulating how the real Farmbot will detect the existence of threat of a weed to a plant, which is by checking the soil surface within certain radius of that plant using the camera and computer vision and reporting feedback that can be use to update the knowledge base.

After some preliminary experiment, these parameters are set to 3 loops per minute (0.05 hz), 0.1, and 10 cm (in Unity world scale, where the 1.8m x 6m garden bed and Farmbot is also re-sized to the Unity world scale). These parameters are decided so that the spawning of weeds is visually realistic. More importantly, the simulation of weeds based on these selected values simulates the random initial world state that allows us to observe how the online plan realization can handle dynamic world states at the exact program state and enable the autonomous agent to continuously traverse through transitions in the APP.

C. Results and Discussion

The simulation shows the sequence of transitions and continuous planning for the new goals. The Farmbot completed all the 21 transitions and perform the tasks as expected. Once one transition was traversed, it continued to the next one by correctly selecting the next goal and successfully computing the plan which enables the next transition to the desired program state. The resulting plan realization can be produced from the concatenation of a sequence of planning problems, plans, program states and world states as mentioned in section VII-C3. Furthermore, all the planning problems in the benchmarking experiment above only shows one type of tasks which involves similar initial states. In the simulation,

the agent encounters different planning problems with different initial states. All the planning problems are successfully solved, indicating that the planning domains we proposed are appropriate.

The data we collect from the simulation is attached as Table IV in Appendix D, it contains the total time taken for the each simulation trial. Furthermore, the total time taken is constituted of multiple different time duration, this includes the time taken for planners to find plans, the total dispatching and execution time for each plan, and the time taken for other overheads such as the APP and the goal reasoning, they are also recorded separately in Table IV. The total dispatching and execution time is further separated and recorded into two columns: (1) the total time spent on dispatching and executing all the (move) actions that enable the Farmbot to travel between position objects and complete the task and (2) the rest of the dispatching and execution time spent by other actions. The former is directly correlated to the plan cost as mentioned before. For now on, for convenience, we will refer to this time in (1) directly as 'cost'.

To interpret the data, we produce clustered and stacked bar plots in the following way: we stack the bar of different colors showing the numeric quantity of the each time duration that constitute the single total time taken for each simulation. We plot the data of each repetition of the same simulation configuration into such a stacked bar and cluster all 5 repetitions of the same simulation configuration together. Figure 35 and Figure 36 are the resulting plots, the horizontal axis represents the different planners on their respective PDDL domain and has no numeric scale whereas the vertical axis is the total time taken for each simulation configuration measured in seconds. In addition, the data for each of the two different garden layouts are separated, therefore Figure 35 shows the results in the small garden with 15 position objects and Figure 36 shows the results in the large garden with 50 position objects.

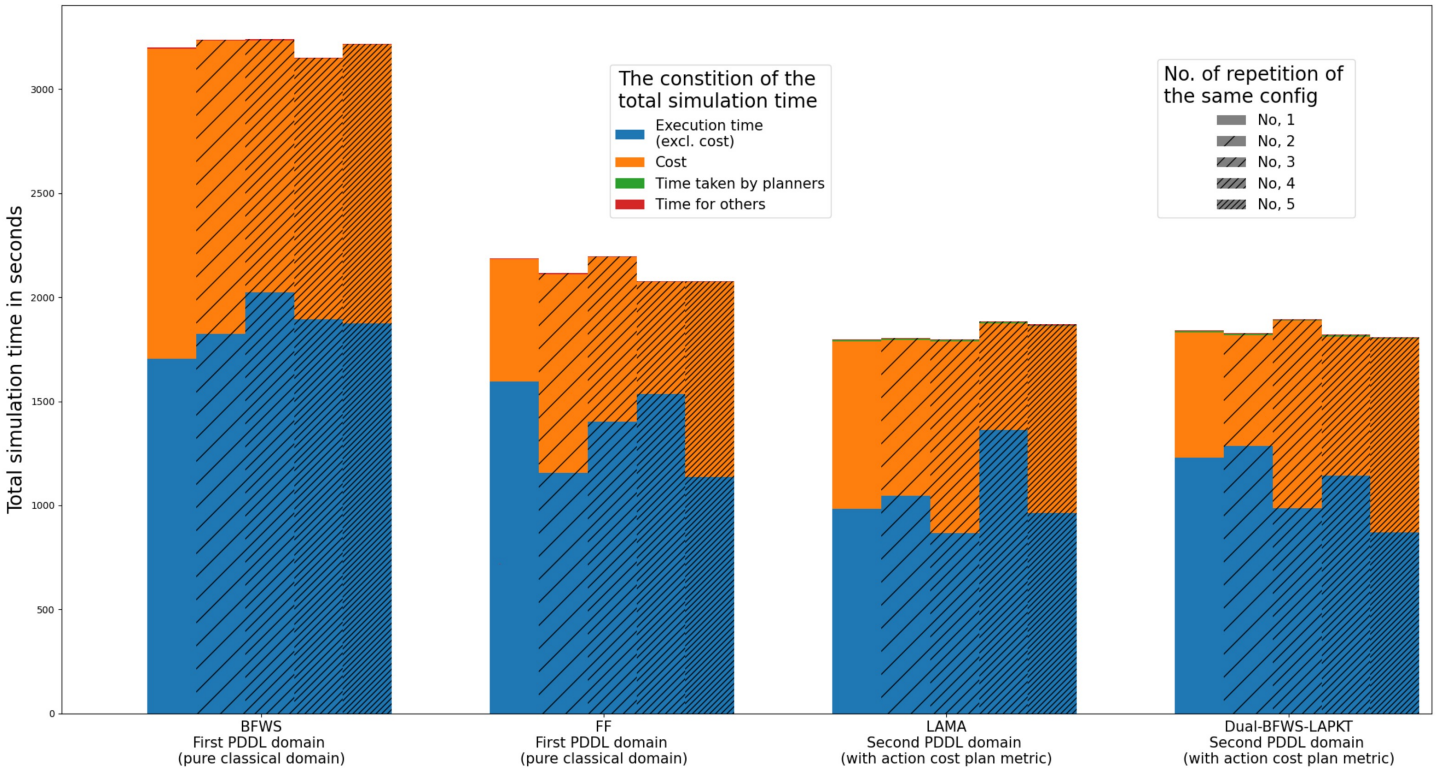


Fig. 35: Clustered and stacked bar plots showing the results of the total time taken (s) of the simulation on small size garden layout with 15 position objects. The total time taken is separated into 4 different time duration stacked together. The 'Cost' is the aforementioned combined time taken for dispatching and executing the (move) action, whereas the Execution time (excl. cost) is just the dispatching and execution time for the remaining actions

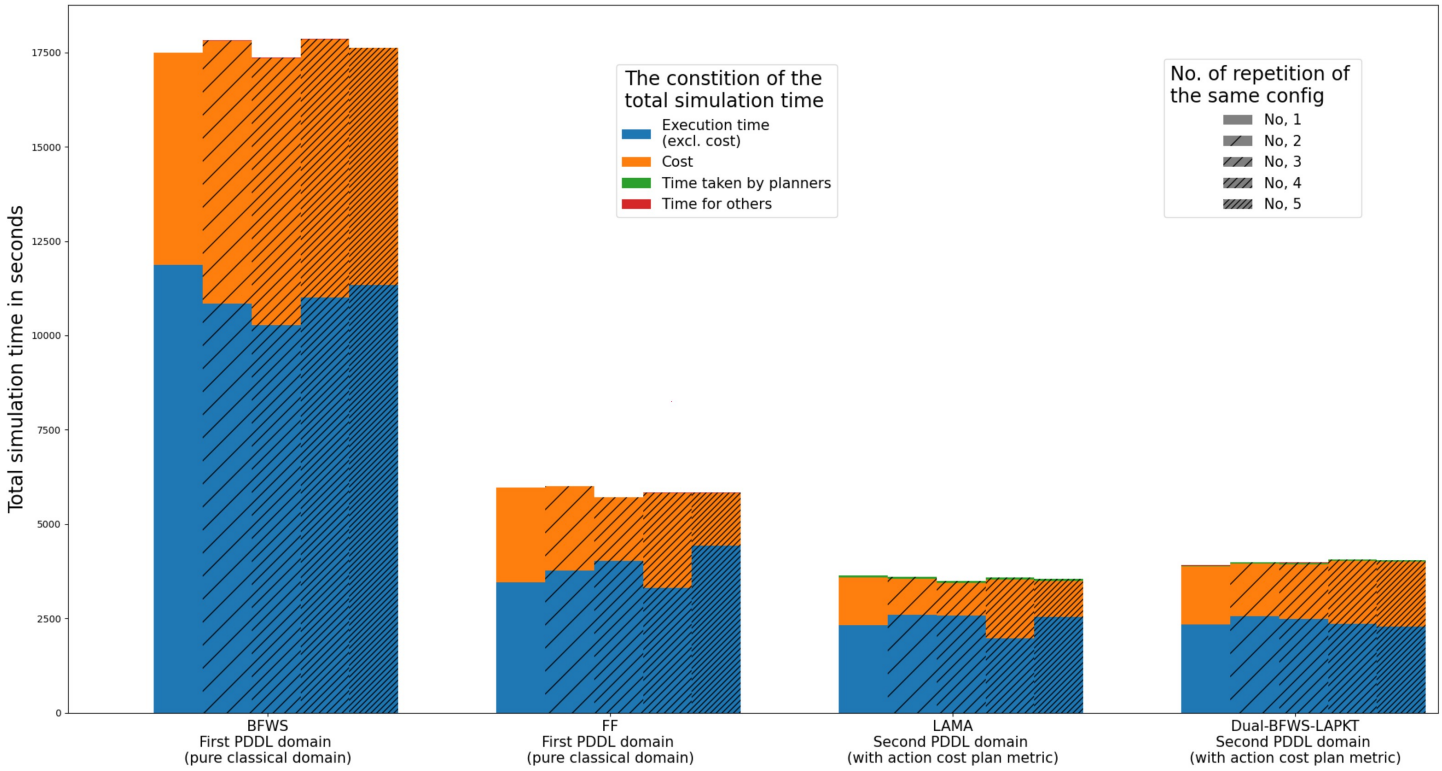


Fig. 36: Clustered and stacked bar plots showing the results of the total time taken (s) of the simulation on small size garden layout with 50 position objects.

The first observation from these two plots is that the time taken by the planners (the green bar in Figure 35 and Figure 36 and the data of 'Time taken by planner' in Table D) is just a very insignificant portion compared to the total time cost. For the two planners on the pure classical domain, the planner time cost is invisible on the chart where the total time for the entire simulation can easily surpass hours. For the two anytime planners, because the time taken is considered as the entirety of the timeout limit instead of the actual time when the last plan is found within the timeout limit, the time is significantly larger than the non-anytime planners. However, compare to the total time, it is still considered a negligible portion and is merely visible.

In contrast to the time taken by the planners, the absolute majority of the time was spent on dispatching and executing the plan (orange and blue bar in Figure 35 and Figure 36 and the data of 'Cost' and 'Time for plan dispatching and execution (excl. cost)' in Table D). By comparing the difference of total time between them, we can observe that the quality of the plan is extremely important and is the main determining factor to the efficiency of the farmbot at performing tasks because it affects the time taken for dispatching and executing the plan. Longer time cost on executing actions means more energy, such as electricity is

also spent. The LAMA and Dual-BFWS-LAPKT are clearly spending less time on it because the plans they found are more optimal. The results are generally compatible to that of the benchmarking experiment above, that is the LAMA and Dual-BFWS-LAPKT have similar results, FF is slightly worse by around 2 to 3 hundred second when the garden layout size, or essentially the problem size and complexity is small, and the become even more worse compared to the two anytime planners when the problem size become larger, where its spent around 2500 seconds more time on the simulation completing essentially the same tasks. The BFWS, on the other hand, is evidently much worse than others. It has almost double the time cost compared to the two anytime planners even when the garden is small, and almost 5 times more cost on the large garden layout.

The evidence we extract from the results strongly indicates the usefulness of action cost plan metric and the corresponding planners at producing more optimal plans and reduce the time and energy cost on executing such plans. From direct observation during the process of the simulation, we found that with BFWS, the Farmbot completely ignored the order it visits the plant objects in the garden bed [2]. Usually, after it finished the task at one plant, instead of moving to the closest one next to it and shorten the overall traveling distance, it would sometimes move all the way across the garden to the furthest one on the other side.

With the simulation, we can visualize the negative impact of the less optimal plans more effectively, through considering the dispatching and execution time cost also observing the behaviour of Farmbot during the simulation. In the benchmarking experiment, we could not make a decisive conclusion on whether the second domain using plan metric is superior to the first one in all scenarios. This is because anytime planners take significantly more time to compute and optimize the plan than the non-anytime planners, and the FF performs much better than the BFWS and plans found by FF are not that worse than those of the two anytime planners compared to the BFWS especially when the problem sizes are small. Therefore, we said that FF with the first classical domain can be considered over the anytime planners on the second domain with plan metric under certain scenarios especially when there is time constraint for the planners. However, after considering the dispatching and execution time in the simulation. We found out that the dominating factor is the dispatching and execution time, which is directly influenced by the plan quality. In comparison, the time taken by planners to compute and optimize plan is just a very insignificant portion. The two anytime planners on the second PDDL domain with plan metric is evidently

[2]A video clip showing part of the simulation using the BFWS planner: https://www.youtube.com/watch?v=xZXIUXKoJVM. After placing all the seeds (after around 0:55), the plan taken by the Farmbot to visit all the position and check the soil moisture is not optimal.

more effective and taking less total time during the simulation because their plans are more optimal. The timeout limit of the two anytime planners is perfectly acceptable because the plan quality is much more crucial to the overall efficiency. In the benchmarking experiment, we show that the timeout limit of 90 seconds is more than enough to optimize plans for problem size up to 100. With this comparatively insignificant time expense in exchange for a much shorter dispatching and execution time, the FF planner on the first pure classical domain now shows no advantage, even when the problem size is relatively small and the h_{FF}, as explained in the benchmarking experiment, is sensitive to the cost. Therefore, we can conclude that the second PDDL domain with action cost plan metric allows us to improve the plan quality significantly and the improvement can compensate the extra planner time cost. Although, during the simulation, we could still observe some occasions where the plans found by the anytime planners were not taking the optimal steps, they were still much better in results.

Furthermore, according to Table IV in the appendix, the actual time taken for both anytime planners to find plans was much smaller than the timeout limit. Both LAMA and Dual-BFWS-LAPKT only took around 11 to 14 seconds to find the last found plan for all their simulation on the large garden layout with 50 position objects, suggesting that there is more room to reduce the total time cost for both anytime planners.

The second observation is that the time taken for other overheads, including the time complexity of the agent planning program (red bar in Figure 35 and Figure 36 and the data of 'Other time taken' in Table D) is also very insignificant and invisible on the chart. In the corresponding data displayed in Table D, even though the fluctuations exist, we did not observe any pattern and trend that differentiate them significantly. Combine with the success of continuous planning and transitions between program states demonstrated in the simulation, this is suggesting that the APP is able to find a solution in the form of online plan realization while not generating additional time complexity that is correlated to the specific planner or domain.

Another observation worth mentioning is that the blue bar, which is the plan dispatching and executing time excluding the move action is the most dominating portion in the stacked bar. This is because the remaining actions also take a significant amount of time to execute, especially with actions like removing weeds, whose animations in the simulation take a considerable amount of time to execute around each plant. However an interesting observation is that the size of these blue portions is seemingly correlated to their corresponding orange portion in the same stacked bar. In theory, the orange portion is the time taken

for dispatching and executing the action move, which is the only optimizable part by the plan metric in our PDDL modelling and correlates to the total plan cost. The blue portion, on the other hand, should stay similar between all trials on the same garden layout because the remaining actions carried out by the Farmbot to complete the same tasks are similar. After carefully examining the simulation process, we found the reason. That is, due to the way we simulate the water and the weeds, if the previous transitions take longer, the later plans computed online for enabling the same transition in the APP will find more weeds spawn around the same plant object and hence, the action to remove weeds will take longer. Similarly, more plants will have their moisture level below the threshold and require watering, taking more time for the Farmbot to reach the same transition goal. The randomness in which we simulate the spawning of weeds also contribute to the main reason of the fluctuations in time taken between the 5 repetitions of the same configuration.

We consider this as a limitation to the current domain modelling, which the domain and the planner fail to capture these external events in the environment. We will discuss it in detail when we propose future extensions later in section IX. In addition, we think the way we simulate the environment in the simulator does not imitate the real world adequately. This is considered another limitation which we will discuss later. However, we still consider our observation and discussion regarding the experiment reliable. Firstly, we combine the results from the benchmarking, and analyze the effectiveness of plan metric from multiple different experiments. Secondly, what is considered adequate for this simulation is a tool that can receive the dispatched actions, provide feedback of executions to enable the updates of the world states and the transitions of the program states that allow us to observe the traversing of the APP and plan realization over our PDDL domains. The realism of the simulation is not critical to this experiment. Thirdly, the plan execution and dispatching in the simulator is accelerated compared to that on a real Farmbot. The real platform moves its gantry much more slowly and hence, the time taken for dispatching and executing plans is much longer. However, other time, such as the time taken by planners and the APP will likely be the same when the system is running on a real Farmbot. Therefore, our discussion and conclusion that the planner time is insignificant is still valid. It is reasonable to spend time and find a perfect plan. Furthermore, in the future, much of the effort should be dedicated into improving the modelling and inventing better plan metric to allow better plans to be generated instead of worrying the complexity it brings to the planners. The evidence we have are sufficient for us to answer our research questions in the next section.

IX Conclusion

In this section, we conclude this research project and summarize the thesis. We will answer our research questions and state the significance and contributions of our project. Finally, we will analyze the limitations of this project and propose future directions that can be studied as the research topics in future projects.

A. *Conclusion and Answering RQs*

Classical planning is both a well-developed and novel area in the field of artificial intelligence. On one hand, theoretical study dates back to 1970s [3] and already becomes mature, on the other hand, compared to other emerging AI topics such as artificial neural network [75], which is developing rapidly and becoming ubiquitous in daily life, real world applications of classical planning and mature tools such as PDDL is relatively limited due to the lack of commercial planners and industry uptake. Planning is a general area of study that aims to give agent autonomy in making decision, this is fundamentally challenging as the real world environment is complex and stochastic, the world state is continuous rather than discrete. These factors limit the scope to apply such a technique, to construct a model, researchers need to consider the level of details of the world state to be captured, the way to represent the world state in the conjunction set of propositions and the change of the world state in the form of STRIPS actions. Any attempt to apply classical planning to a specific real world problem is a step forward in this research area and pushes the uptake of planning technology in real-world problems. In section IV of this research, we proposed the PDDL domain that models the planning problems on Farmbot, and demonstrated how it can be modelled, including how to capture various aspects of the state of the environment and Farmbot in the domain propositions and represent the state change caused by the available actions on Farmbot in the form of domain actions. We also utilized the plan metric, which is an extensive feature of PDDL beyond the pure STRIPS classical planning model in a meaningful way to optimize the solutions of the planning problems on Farmbot and demonstrated its usefulness through experiments in section V and section VIII. In section VII, we applied and adapted the novel concept of agent planning program to overcome the limitation of the conventional planning problems, which is that they can only achieve one goal at a time, and we enabled the Farmbot to form multiple planning problems continuously and autonomously decide the goal for the next planning problem. The resulting software system we implemented in this research was evaluated through a simulation in section VIII and it demonstrated the ability of the system to form meaningful planning problems continuously, which generates valid plans over the proposed domains that enable the Farmbot to accomplish a sequence of tasks.

We provide answers to our research questions as follows:

RQ 1: Can we successfully model the appropriate domain on Farmbot using PDDL to enable the automatic generation of task-accomplishing sequences of actions by domain-independent planners?

The answer is affirmative, the case on Farmbot satisfies all the assumptions of the classical planning. The environment is fully observable. Their actions, if applied successfully, will have deterministic results. Unexpected accidents can happen which will lead to unexpected action effects to the world state, but it will always be failure that will require manual intervention. Therefore, it is not part of the assumptions critical to the success of modelling. Even though the real world environment of the Farmbot is continuous, the parameters to each action can still be represented in discrete propositions, including the coordinates in the garden bed. Through a series of evaluation, our domain modelling enables those sequences of actions to be generated by domain-independent planners (BFWS, FF, LAMA and Dual-BFWS-LAPKT), and these sequences of actions are the expected ones to be used for instructing the Farmbot to accomplish the relevant tasks.

RQ 2: Does the action cost plan metric allow us to improve the plan quality significantly and the improvement can compensate the extra modelling effort and planner complexity?

The answer is affirmative. Through the benchmarking experiment and the simulation, we clearly see the difference between with and without using the plan metric. The plan quality, measured by the total moving distance of Farmbot's gantry to accomplish certain tasks is significantly better if the plan metric is enabled. If the time taken to dispatch and execute the plan is considered, the extra time spent by the anytime planners for solving the domain with plan metric is almost negligible. In additions, the extra modelling effort required to define the metrical functions in the PDDL problem files, more precisely, all the (move-distance ?x ?y) predicates between every pair of the position objects can be easily generated using a script with the provided garden layout and the distance function. Furthermore, for the pure classical domain (the first domain), the FF planner will compute plans with better quality than the BFWS as the h_{FF} heuristic will consider the future cost of the plan when selecting the next search node to expand, contrasting to the novelty, which is less informed and sometimes 'blind', but, BFWS can be used in certain scenarios because of the speed, such as being used as the first search algorithm in Dual-BFWS-LAPKT to quickly find the first plan based on which the following search algorithm can optimize. For the classical domain with plan metric (second domain), we cannot certainly compare the 2 anytime planners, because their results from both the benchmarking and the simulation are very similar. However, this is not a problem since that is not the main aim of this research. They are both top-tier state-of-the-art planners

with close performance in the previous IPC [51] and they have never phased out each other in the current situation.

RQ 3: Can the agent planning program (APP) be adapted for goal reasoning and synthesizing multiple planning problems over the proposed domain on Farmbot?

The answer is affirmative. In this project, we proposed the APP over our domains and demonstrated through the simulation its ability to allow the agent to formulate a sequence of transitions that enables continuous planning. The realization of the APP was found as the concatenation of alternating planning for the next goal and plan execution for the transitioning to the next program state. Our strategy for goal reasoning and tie breaking allows the agent to successfully select the next goals in a meaningful way.

In conclusion, this project is considered a successful research project that proposes a solution to apply classical planning in a new domain. We not only address the aims of this research, but also add values to the field of AI planning and open up many options for the future study that might extend upon the foundation laid by our work.

B. Significance and Contributions

We consider the contributions and significance of this research project as follows. First and foremost, this project introduces a new research problem that has real world significance, which is to apply classical planning technique to automate agricultural process. We successfully utilize an existing platform: Farmbot, which is the most appropriate choice to implement such an application targeting small scale farming. The accomplishment serves as a proof of concept that explores many aspects in such a new research problem, including the approach to model the PDDL, the methods for conducting experiments, the proposed APP to enable autonomous agent to continuously synthesize planning problems and execute the plans, and the architecture and solution to integrate the system to the real physical platform. All these methodologies serve as important references for future study that continue the exploration into this new domain.

In addition to introducing the new problem and being the first work in this area, our work is likely to be of interest to several communities around the world. Firstly, the Farmbot is an open-source hardware platform that has a considerable community dedicated to it. Our work integrates other international open standards such as ROS, PDDL to Farmbot, and demonstrate to this community a way to adopt classical planning for automation on Farmbot. We potentially extend the functionalities of this platform, and present

an openly available automated farming solution to the worldwide communities.

Secondly, goal reasoning has long been a foundation for the research and development of intelligent agents [65]. In this research, we extend the agent planning program in a meaningful way for goal reasoning. Also, the ROS project is essentially an implementation of an interactive goal reasoning program. Our experimentation and implementation in this research adds value to the relevant areas and might be of interest to the research community focused on goal reasoning in AI planning. Furthermore, to the best of our knowledge, this project is the first work that takes the agent planning program beyond theoretical study and applies it to a specific problem. This is important to this new knowledge proposed back in 2017 as our work helps validate the theoretical work and points out the limitations when it is applied to a real world scenario. Many other researchers who are interested in applying agent planning programs might consider the value of our work.

Thirdly, the significance of this project is not just limited to the field of computer science and AI planning. In the area of digital agriculture, researchers are aiming to automate the agricultural process and apply digital technologies on specific platforms including the Farmbot. We already received attention from the departments of food science and digital agriculture in the university of Melbourne, who are interested in utilizing new sensor systems to enable automation on the same platform. They can directly extend our work to incorporate other actions on Farmbot that utilize their sensors.

Last but not least, the infrastructure we created through this research, both the ROS project[3] and the simulator[4] provides a foundation to work on for those communities mentioned above who are interested to extend our work. In particular, the simulator allows fast development and evaluation in the future. Our works enable the future research that aims to perfect such a system to a fully automated farming solution targeting small-scale urban agriculture. We will continue to maintain these projects in the future. Furthermore, through this research, we identify many current limitations of our work and possible directions for future extensions, which open up many options and opportunities for future research topics. We will discuss them in the remaining part of this thesis below.

[3]https://github.com/The-Kharsair-Empire/ROSfarm.git, will be made available as soon as we tidy it up

[4]https://github.com/The-Kharsair-Empire/Farmbot_Simulator.git

C. Weaknesses and Future Works

Even though the significance and contributions of this project are numerous, we still consider this research as an experimentation and a proof of concept and new techniques that focuses on breadth rather than depth. We recognize and identify many limitations of this research that we could not address due to many constraints of this one-year master research project, such as time and resources. These will be the options for possible directions of future extensions and research topics.

Firstly, the original research plan aimed to deploy and evaluate the system on a real Farmbot platform. Prior to the start of the first semester this year, a real Farmbot was installed on the rooftop of Building 170, Parkville, University of Melbourne, of which Figure 1 was taken. However, because of a series of unexpected difficulties accessing it, including the lockdown policy caused by the pandemic, we switched the evaluation to a simulator. Due to the absence of a Farmbot simulator, we had to create our own. However, because of the time constraint, we could not perfect our simulator. There are problems with this simulation tool, especially with respect to the realism compared to the actual platform, as many parameters were selected in an arbitrary way. One of the future extensions would be to deploy and evaluate the system on the real Farmbot, observe the performance in a real world environment and gather evidence to validate the conclusion made in this research project.

Another direction is to continue the improvement of the simulator and put emphasis on realism. This includes proposing better algorithms and techniques for simulating various aspects such as moisture, weed growth, plant growth and even whether system and mechanical failures etc. Therefore, researchers who might be interested in conducting experiment in simulation will have a more realistic and accurate simulator.

Secondly, in the PDDL domain modelling in this project, we only considered the default actions available to the Farmbot. Since Farmbot is an open source platform and its potential can be enlarged through customization, having custom actions to achieve additional functionalities such as installing new custom tools and using custom sensors are allowed. Modern digital agriculture is a complex coordination and integration of many sensors and actuators [28]. To finally achieve a working solution that can autonomously monitor, manage and respond to the various events in the garden bed, the default actions and hardware are not sufficient. In the future, new research might propose new sensors and tools which enable new actions, the planning domains on Farmbot should therefore integrate them.

In addition, PDDL has many extensive features and language standards beyond the pure classical domain, the plan metric in PDDL2.1 [5] is just one of them. Throughout the majority of this year, we considered most of them unnecessary. We tried temporal domain semantics, which is another PDDL2.1 feature that introduces the concepts of duration and temporal dependency to each domain actions. In short, temporal domain will allow the individual actions in the plan to have duration information, and it is useful for time-critical action dispatching and parallelizable actions. It is unnecessary for us as the duration information does not improve plan quality, specially as plans are sequential (only one actuator at a given time) on Farmbot and can be efficiently computed using plan costs, meanwhile the temporal domains require a class of much more complicated planners that have significantly more time complexity. We also tried numeric propositions, which also comes with PDDL2.1 standard, but we did not find meaningful use and hence, removed it from our proposed domains. However, towards the end of this project, especially when we were conducting the simulation, we started to recognize the potential value of numeric propositions, also the continuous processes and events, which are the features introduced in PDDL+ standard [6]. One limitation we mentioned in the discussion of simulation in section VIII-C is that our domains fail to capture the activities that are happening in the environment, these activities are external and commonly referred to as exogenous events. The continuous processes and events introduced in PDDL+ can be used for the purpose of modelling these external activities, such as the spawning of weeds and the change of moisture of each plant object so that the planners will be aware of those factors and make plans accordingly. Furthermore, with numeric propositions, the addition of moisture at each (water_plant) action can be modelled as variable instead of a fixed amount. Our domains proposed in this project are not aware of these factors and hence, can be extended using such PDDL features to improve the expressiveness. These are just some possible future improvements to the domain modelling, due to time constraint, we missed the opportunity to experiment them in this project. The lesson learnt is that we should consider all the aspects of a planning domain more thoroughly, including the aforementioned exogenous events before we start modelling, a more experienced PDDL user would likely propose a domain that will capture those aspects. In this project, we explored the meaningful way to apply the feature of plan metric. In the future, those additional features should be explored, and similar benchmarking experiments to this project should be conducted to evaluate the usefulness of those features and find the suitable planner that can solve the planning domain with those features.

Thirdly, the adaptation to the agent planning program can be unsafe. It is possible that the goal reasoning cannot find an available transition from a program state. Furthermore, even if an available transition is found, because we relax the original APP to allow multiple different world states to correspond to a single

program state, there might not be any valid plan that exists for the current transition. This problem is not crucial to the success of this research as we were not aiming for a safety-critical system, but this might be a problem to the eventual success of the whole automated farming system that must be addressed. The handling of this 'exception' can be a research question on its own in the future.

Furthermore, regarding to the agent planning program, if we apply additional PDDL features such as numeric propositions in our planning domain, the agent planning program over this domain might potentially have infinite world states. Currently, if a valid solution of an APP can be defined, which is a plan realization, the world state must be finite. This give rise to anther future direction, which is to explore the possible solution of plan realization of APP over such a planning domain with numeric propositions and infinite world states.

Last but not least, as we mentioned in the benchmarking experiment, the BFWS can easily be modified to be cost-sensitive. This is another future direction to extend the current work, and test this modified planner on the pure STRIP classical domain. Our hypothesis is that it will perform better than the original BFWS in terms of the plan quality, it might be similar to the FF planner, but it is very unlikely to surpass that of the two anytime planners using the action cost plan metric.

References

[1] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise, "An introduction to the planning domain definition language," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 13, no. 2, pp. 1–187, 2019.

[2] S. Jiménez, J. Segovia-Aguas, and A. Jonsson, "A review of generalized planning," *The Knowledge Engineering Review*, vol. 34, 2019.

[3] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[4] M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner *et al.*, "The 2014 international planning competition: Progress and trends," *Ai Magazine*, vol. 36, no. 3, pp. 90–98, 2015.

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

[6] ——, "Pddl+: Modeling continuous time dependent effects," in *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, vol. 4, 2002, p. 34.

[7] N. Lipovetzky, C. Burt, A. Pearce, and P. Stuckey, "Planning for mining operations with time and resource constraints," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 24, no. 1, 2014.

[8] M. Vallati, D. Magazzeni, B. De Schutter, L. Chrpa, and T. L. McCluskey, "Efficient macroscopic urban traffic models for reducing congestion: A pddl+ planning approach," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] M. Fox, D. Long, and D. Magazzeni, "Automatic construction of efficient multiple battery usage policies," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 21, no. 1, 2011.

[10] M. Helmert and H. Lasinger, "The scanalyzer domain: Greenhouse logistics as a planning problem," in *Twentieth International Conference on Automated Planning and Scheduling*, 2010.

[11] M. Cardellini, M. Maratea, M. Vallati, G. Boleto, and L. Oneto, "A planning-based approach for in-station train dispatching," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 12, no. 1, 2021, pp. 156–158.

[12] E. J. Van Henten, J. Hemming, B. Van Tuijl, J. Kornet, J. Meuleman, J. Bontsema, and E. Van Os, "An autonomous robot for harvesting cucumbers in greenhouses," *Autonomous robots*, vol. 13, no. 3, pp. 241–258, 2002.

[13] T. Bakker, K. van Asselt, J. Bontsema, J. Müller, and G. van Straten, "An autonomous weeding robot for organic farming," in *Field and Service Robotics*. Springer, 2006, pp. 579–590.

[14] Q. Feng, W. Zou, P. Fan, C. Zhang, and X. Wang, "Design and test of robotic harvesting system for cherry tomato," *International Journal of Agricultural and Biological Engineering*, vol. 11, no. 1, pp. 96–100, 2018.

[15] E. Van Henten, B. Van Tuijl, G.-J. Hoogakker, M. Van Der Weerd, J. Hemming, J. Kornet, and J. Bontsema, "An autonomous robot for de-leafing cucumber plants grown in a high-wire cultivation system," *Biosystems Engineering*, vol. 94, no. 3, pp. 317–323, 2006.

[16] K. Tanigaki, T. Fujiura, A. Akase, and J. Imagawa, "Cherry-harvesting robot," *Computers and electronics in agriculture*, vol. 63, no. 1, pp. 65–72, 2008.

[17] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras, "Rosplan: Planning in the robot operating system," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 25, no. 1, 2015.

[18] G. De Giacomo, A. E. Gerevini, F. Patrizi, A. Saetti, and S. Sardina, "Agent planning programs," *Artificial Intelligence*, vol. 231, pp. 64–106, 2016.

[19] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[20] J. Haas, "A history of the unity game engine," *Diss. WORCESTER POLYTECHNIC INSTITUTE*, 2014.

[21] D. UN, "World urbanization prospects: The 2014 revision," *United Nations Department of Economics and Social Affairs, Population Division: New York, NY, USA*, vol. 41, 2015.

[22] F. Nations, *The future of food and agriculture: Trends and challenges*. Food and Agriculture Organization of the United Nations, 2018. [Online]. Available: https://books.google.com.au/books?id=SRFfDwAAQBAJ

[23] D. Buehler and R. Junge, "Global trends and current status of commercial urban rooftop farming," *Sustainability*, vol. 8, no. 11, p. 1108, 2016.

[24] F. Orsini, R. Kahane, R. Nono-Womdim, and G. Gianquinto, "Urban agriculture in the developing world: a review," *Agronomy for sustainable development*, vol. 33, no. 4, pp. 695–720, 2013.

[25] M. Safayet, M. F. Arefin, and M. M. U. Hasan, "Present practice and future prospect of rooftop farming in dhaka city: A step towards urban sustainability," *Journal of urban management*, vol. 6, no. 2, pp. 56–65, 2017.

[26] C. R. Vogl, P. Axmann, and B. Vogl-Lukasser, "Urban organic farming in austria with the concept of selbsternte ('self-harvest'): An agronomic and socio-economic analysis," *Renewable Agriculture and Food Systems*, pp. 67–79, 2004.

[27] E. Appolloni, F. Orsini, K. Specht, S. Thomaier, E. Sanyé-Mengual, G. Pennisi, and G. Gianquinto, "The global rise of urban rooftop agriculture: a review of worldwide cases," *Journal of Cleaner Production*, p. 126556, 2021.

[28] R. R Shamshiri, C. Weltzien, I. A. Hameed, I. J Yule, T. E Grift, S. K. Balasundram, L. Pitonakova, D. Ahmad, and G. Chowdhary, "Research and development in agricultural robotics: A perspective of digital farming," 2018.

[29] P. J. Sammons, T. Furukawa, and A. Bulgin, "Autonomous pesticide spraying robot for use in a greenhouse," in *Australian Conference on Robotics and Automation*, vol. 1, no. 9, 2005.

[30] J. Cruz, S. Herrington, and B. Rodriguez, "Farmbot," 2014.

[31] C. J. C. Moscoso, E. M. F. Sorogastúa, and R. S. P. Gardini, "Efficient implementation of a cartesian farmbot robot for agricultural applications in the region la libertad-peru," in *2018 IEEE ANDESCON*. IEEE, 2018, pp. 1–6.

[32] A. Yeshmukhametov, L. Al Khaleel, K. Koganezawa, Y. Yamamoto, Y. Amirgaliyev, and Z. Buribayev, "Designing of cnc based agricultural robot with a novel tomato harvesting continuum manipulator tool," *International Journal of Mechanical Engineering and Robotics Research*, vol. 9, no. 6, 2020.

[33] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 2002.

[34] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of artificial intelligence tools*. Springer, 1984, pp. 13–13.

[35] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[36] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001.

[37] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[38] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson *et al.*, "Pddl— the planning domain definition language," Technical Report, Tech. Rep., 1998.

[39] J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, pp. 57–57, 2001.

[40] N. Lipovetzky and H. Geffner, "Best-first width search: Exploration and exploitation in classical planning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[41] A. Coles, A. Coles, M. Fox, and D. Long, "Forward-chaining partial-order planning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 20, no. 1, 2010.

[42] S. Richter and M. Westphal, "The lama planner: Guiding cost-based anytime planning with landmarks," *Journal of Artificial Intelligence Research*, vol. 39, pp. 127–177, 2010.

[43] G. Frances, H. Geffner, N. Lipovetzky, and M. Ramirez, "Best-first width search in the ipc 2018: Complete, simulated, and polynomial variants," *IPC2018–Classical Tracks*, pp. 22–26, 2018.

[44] Á. García-Olaya, S. Jiménez, and C. Linares López, "The 2011 international planning competition," 2011.

[45] A. Gerevini and I. Serina, "Lpg: A planner based on local search for planning graphs with action costs." in *AIPS*, vol. 2, 2002, pp. 281–290.

[46] J. Hoffmann, "A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm," in *International Symposium on Methodologies for Intelligent Systems*. Springer, 2000, pp. 216–227.

[47] B. Bonnet and H. Geffner, “Hsp: Heuristic search planner,” 1998.

[48] S. Edelkamp, S. Jabbar, and M. Nazih, “Large-scale optimal pddl3 planning with mips-xxl,” *5th International Planning Competition Booklet (IPC-2006)*, pp. 28–30, 2006.

[49] A. Torralba, V. Alcázar, D. Borrajo, P. Kissmann, and S. Edelkamp, “Symba*: A symbolic bidirectional a* planner,” in *International Planning Competition*, 2014, pp. 105–108.

[50] D. Bryce and O. Buffet, “6th international planning competition: Uncertainty part,” *Proceedings of the 6th International Planning Competition (IPC’08)*, 2008.

[51] I. Cenamor and A. Pozanco, “Insights from the 2018 ipc benchmarks,” in *ICAPS 2019 Workshop on the International Planning Competition (WIPC)*, 2019, pp. 8–14.

[52] M. Helmert, “The fast downward planning system,” *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[53] M. Vallati, L. Chrpa, T. L. McCluskey, and F. Hutter, “On the importance of domain model configuration for automated planning engines,” *Journal of Automated Reasoning*, pp. 1–47, 2021.

[54] M. R. W. P. P. Bevan, D. A. M. F. D. Long, and D. Magazzeni, “Automated planning with goal reasoning in minecraft,” *IntEx 2017*, p. 43, 2017.

[55] J. Wichlacz, A. Torralba, and J. Hoffmann, “Construction-planning models in minecraft,” 2019.

[56] M. Cardellini, M. Maratea, M. Vallati, G. Boleto, and L. Oneto, “In-station train dispatching: a pddl+ planning approach,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 31, 2021, pp. 450–458.

[57] A. Coles, A. Coles, M. Fox, and D. Long, “Temporal planning in domains with linear processes,” in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[58] N. Kushmerick, S. Hanks, and D. S. Weld, “An algorithm for probabilistic planning,” *Artificial Intelligence*, vol. 76, no. 1-2, pp. 239–286, 1995.

[59] S. W. Yoon, A. Fern, and R. Givan, “Ff-replan: A baseline for probabilistic planning.” in *ICAPS*, vol. 7, 2007, pp. 352–359.

[60] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal, and P. Stone, “Task planning in robotics: an empirical comparison of pddl-and asp-based systems,” *Frontiers of Information Technology & Electronic Engineering*, vol. 20, no. 3, pp. 363–373, 2019.

[61] P. Munoz, M. D. R-Moreno, and B. Castano, “Integrating a pddl-based planner and a plexil-executor into the ptinto robot,” in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, 2010, pp. 72–81.

[62] L. Chrpa, N. Lipovetzky, and S. Sardina, “Handling non-local dead-ends in agent planning programs.” in *IJCAI*, 2017, pp. 971–978.

[63] A. Cleeremans, D. Servan-Schreiber, and J. L. McClelland, “Finite state automata and simple recurrent networks,” *Neural computation*, vol. 1, no. 3, pp. 372–381, 1989.

[64] G. De Giacomo, F. Patrizi, and S. Sardina, “Agent programming via planning programs,” in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 2010, pp. 491–498.

[65] D. W. Aha, “Goal reasoning: Foundations, emerging applications, and prospects,” *AI Magazine*, vol. 39, no. 2, pp. 3–24, 2018.

[66] “Open-source cnc farming.” [Online]. Available: https://farm.bot/

[67] C. Muise and N. Lipovetzky, “Keps book: Planning. domains,” in *Knowledge Engineering Tools and Techniques for AI Planning*. Springer, Cham, 2020, pp. 91–105. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-38561-3_5

[68] “Welcome to python.org.” [Online]. Available: https://www.python.org/

[69] “Ubuntu documentation.” [Online]. Available: https://help.ubuntu.com/lts/ubuntu-help/index.html

[70] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[71] KCL-Planning, “Kcl-planning/rosplan: The rosplan framework provides a generic method for task planning in a ros system.” [Online]. Available: https://github.com/KCL-Planning/ROSPlan

[72] R. A. Light, “Mosquitto: server and client implementation of the mqtt protocol,” *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[73] “Farmbot software development.” [Online]. Available: https://developer.farm.bot/v13/Documentation/farmbot-software-development

[74] Unity-Technologies, “Unity-technologies/unity-robotics-hub: Central repository for tools, tutorials, resources, and documentation for robotics simulation in unity.” [Online]. Available: https://github.com/Unity-Technologies/Unity-Robotics-Hub

[75] S.-C. Wang, “Artificial neural network,” in *Interdisciplinary computing in java programming*. Springer, 2003, pp. 81–100.

Appendix

A. The Classical Planning Domain on Farmbot

```
(define (domain classical-domain-farmbot)

(:requirements
    :strips
    :typing
)

(:types
    position
    plant
    seed
    weed
    tool
    toolholder
    seedcontainer
)

(:predicates
    (tool-mount-free)
    (farmbot-at ?x - position)
    (carry-tool ?t - tool)
    (carry-camera)
    (tool-at ?x - position ?t - tool)
    (tool-rack-at ?tr - toolholder ?t - tool ?x - position)
    (no-plant-at ?x - position)
    (plant-at ?x - position ?p - plant)
    (checked-weed-exist ?x - position)
    (weed-at ?x - position ?w - weed ?p - plant)
    (weed-removed ?x - position)
    (no-weed-at ?x - position)
    (carry-seed ?s - seed)
    (seeder-free)
    (container-at ?x - position ?c - seedcontainer)
    (container-has ?c - seedcontainer ?s - seed)
    (match-seed-type-n-plant-type ?s - seed ?p - plant)
```

```
    (checked-moisture ?x - position ?p - plant)
    (need-water ?x - position ?p - plant)
    (not-need-water ?x - position ?p - plant)
    (watered ?x - position ?p - plant)
)

(:constants
    seeder wateringnozzle weeder soilsensor - tool
    seedtray - seedcontainer
    seederrack wateringnozzlerack weederrack soilsensorrack - toolholder
    seederPos wateringnozzlePos weederPos soilsensorPos posSeedTray - position
)

(:action move
    :parameters (?x ?y - position)
    :precondition (and
        (farmbot-at ?x)
    )
    :effect (and
        (not (farmbot-at ?x))
        (farmbot-at ?y)
    )
)


(:action pick_up_tool
    :parameters (?x - position ?t - tool)
    :precondition (and
        (tool-at ?x ?t)
        (tool-mount-free)
        (farmbot-at ?x)
    )
    :effect (and
        (not (tool-mount-free))
        (not (tool-at ?x ?t))
        (carry-tool ?t)
    )
)
```

```
(:action put_down_tool
    :parameters (?x - position ?t - tool ?tr - toolholder)
    :precondition (and
        (carry-tool ?t)
        (tool-rack-at ?tr ?t ?x)
        (farmbot-at ?x)
    )
    :effect (and
        (not (carry-tool ?t))
        (tool-mount-free)
        (tool-at ?x ?t)
    )
)



(:action pick_up_seed
    :parameters (?x - position ?s - seed ?c - seedcontainer)
    :precondition (and
        (container-has ?c ?s)
        (seeder-free)
        (container-at ?x ?c)
        (farmbot-at ?x)
        (carry-tool seeder)
    )
    :effect (and
        (not (seeder-free))
        (carry-seed ?s)
        (not (container-has ?c ?s))
    )
)

(:action place_seed
    :parameters (?x - position ?s - seed ?p - plant)
    :precondition (and
        (no-plant-at ?x)
        (carry-seed ?s)
```

```
        (farmbot-at ?x)
        (match-seed-type-n-plant-type ?s ?p)
        (carry-tool seeder)
    )
    :effect (and
        (not (carry-seed ?s))
        (not (no-plant-at ?x))
        (seeder-free)
        (plant-at ?x ?p)
    )
)

(:action check_need_water
    :parameters (?x - position ?p - plant)
    :precondition (and
        (plant-at ?x ?p)
        (farmbot-at ?x)
        (carry-tool soilsensor)
    )
    :effect (and

        (checked-moisture ?x ?p)
    )

)

(:action water_plant
    :parameters (?x - position ?p - plant)
    :precondition (and
        (need-water ?x ?p)
        (farmbot-at ?x)
        (plant-at ?x ?p)
        (carry-tool wateringnozzle)
    )
    :effect (and
        (not (need-water ?x ?p))
        (watered ?x ?p)
```

```
    )
)

(:action detect_weed
    :parameters (?x - position ?p - plant)
    :precondition (and
        (carry-camera)
        (farmbot-at ?x)
        (tool-mount-free)
        (plant-at ?x ?p)

    )
    :effect (and
        (checked-weed-exist ?x)
    )
)

(:action remove_weed
    :parameters (?x - position ?w - weed ?p - plant)
    :precondition (and
        (farmbot-at ?x)
        (carry-tool weeder)
        (weed-at ?x ?w ?p)
    )
    :effect (and
        (weed-removed ?x)
        (not (weed-at ?x ?w ?p))

    )
)

(:action skip_watering_plant
    :parameters (?x - position ?p - plant)
    :precondition (and
        (not-need-water ?x ?p)
        (plant-at ?x ?p)
    )
    :effect (and
```

```
        (not (not-need-water ?x ?p))
        (watered ?x ?p)

    )
)

(:action skip_removing_weed
    :parameters (?x - position)
    :precondition (and
        (farmbot-at ?x)
        (carry-tool weeder)
        (no-weed-at ?x)
    )
    :effect (and
        (weed-removed ?x)
        (not (no-weed-at ?x))

    )
)



)
```

B. The Classical Planning Domain with Action Cost on Farmbot

```
(define (domain classical-domain-farmbot-plan-metric)

(:requirements
    :strips
    :fluents
    :typing
)

(:types
    position
    plant
    seed
    weed
    tool
```

```
    toolholder
    level
    seedcontainer
    bin
)


(:predicates
    (tool-mount-free)
    (farmbot-at ?x - position)
    (carry-tool ?t - tool)
    (carry-camera)
    (tool-at ?x - position ?t - tool)
    (tool-rack-at ?tr - toolholder ?t - tool ?x - position)
    (no-plant-at ?x - position)
    (plant-at ?x - position ?p - plant)
    (checked-weed-exist ?x - position)
    (weed-at ?x - position)
    (weed-removed ?x - position)
    (no-weed-at ?x - position)
    (carry-seed ?s - seed)
    (seeder-free)
    (container-at ?x - position ?c - seedcontainer)
    (container-has ?c - seedcontainer ?s - seed)
    (match-seed-type-n-plant-type ?s - seed ?p - plant)
    (bin-at ?x - position ?b - bin)
    (checked-moisture ?x - position ?p - plant)
    (need-water ?x - position ?p - plant)
    (not-need-water ?x - position ?p - plant)
    (watered ?x - position ?p - plant)
)

(:constants
    seeder wateringnozzle weeder soilsensor - tool
    seedtray - seedcontainer
    seederrack wateringnozzlerack weederrack soilsensorrack - toolholder
    seederPos wateringnozzlePos weederPos soilsensorPos posSeedTray - position
)
```

```
(:functions
    (move-distance ?x - position ?y - position) - number
    (total-cost) - number
)

(:action move
    :parameters (?x ?y - position)
    :precondition (and
        (not (= ?x ?y))
        (farmbot-at ?x)
    )
    :effect (and
        (not (farmbot-at ?x))
        (farmbot-at ?y)
        (increase (total-cost) (move-distance ?x ?y))
    )
)


(:action pick_up_tool
    :parameters (?x - position ?t - tool)
    :precondition (and
        (tool-at ?x ?t)
        (tool-mount-free)
        (farmbot-at ?x)

    )
    :effect (and
        (not (tool-mount-free))
        (not (tool-at ?x ?t))
        (carry-tool ?t)
    )
)

(:action put_down_tool
    :parameters (?x - position ?t - tool ?tr - toolholder)
```

```
    :precondition (and
        (carry-tool ?t)
        (tool-rack-at ?tr ?t ?x)
        (farmbot-at ?x)
    )
    :effect (and
        (not (carry-tool ?t))
        (tool-mount-free)
        (tool-at ?x ?t)
    )
)


(:action drop_seed
    :parameters (?x - position ?s - seed)
    :precondition (and
        (bin-at ?x seedbin)
        (farmbot-at ?x)
    )
    :effect (and
        (seeder-free)
        (not (carry-seed ?s))
    )
)


(:action pick_up_seed
    :parameters (?x - position ?s - seed ?c - seedcontainer)
    :precondition (and
        (container-has ?c ?s)
        (seeder-free)
        (container-at ?x ?c)
        (farmbot-at ?x)
        (carry-tool seeder)
    )
    :effect (and
        (not (seeder-free))
        (carry-seed ?s)
```

```
        (not (container-has ?c ?s))
    )
)

(:action place_seed
    :parameters (?x - position ?s - seed ?p - plant)
    :precondition (and
        (no-plant-at ?x)
        (carry-seed ?s)
        (farmbot-at ?x)
        (match-seed-type-n-plant-type ?s ?p)
        (carry-tool seeder)
        (not (seeder-free))
    )
    :effect (and
        (not (carry-seed ?s))
        (not (no-plant-at ?x))
        (seeder-free)
        (plant-at ?x ?p)
    )
)

(:action check_need_water
    :parameters (?x - position ?p - plant)
    :precondition (and
        (plant-at ?x ?p)
        (farmbot-at ?x)
        (carry-tool soilsensor)
    )
    :effect (and
        (checked-moisture ?x ?p)
        (not (watered ?x ?p))
    )
)

(:action water_plant
    :parameters (?x - position ?p - plant)
    :precondition (and
```

```
        (need-water ?x ?p)
        (farmbot-at ?x)
        (plant-at ?x ?p)
        (carry-tool wateringnozzle)
    )
    :effect (and
        (not (need-water ?x ?p))
        (not (checked-moisture ?x ?p))
        (watered ?x ?p)
    )
)


(:action skip_watering_plant
    :parameters (?x - position ?p - plant)
    :precondition (and
        (not-need-water ?x ?p)
        (plant-at ?x ?p)
    )
    :effect (and
        (not (not-need-water ?x ?p))
        (not (checked-moisture ?x ?p))
        (watered ?x ?p)
    )
)


(:action detect_weed
    :parameters (?x - position ?p - plant)
    :precondition (and
        (carry-camera)
        (farmbot-at ?x)
        (tool-mount-free)
        (plant-at ?x ?p)
    )
    :effect (and
        (not (weed-removed ?x))
        (checked-weed-exist ?x)
```

```
    )
)

(:action remove_weed
    :parameters (?x - position)
    :precondition (and
        (farmbot-at ?x)
        (carry-tool weeder)
        (weed-at ?x)
    )
    :effect (and
        (weed-removed ?x)
        (not (weed-at ?x))
        (not (checked-weed-exist ?x))
    )
)

(:action skip_removing_weed
    :parameters (?x - position)
    :precondition (and
        (farmbot-at ?x)
        (carry-tool weeder)
        (no-weed-at ?x)
    )
    :effect (and
        (weed-removed ?x)
        (not (no-weed-at ?x))
        (not (checked-weed-exist ?x))
    )
)

)
```

C. Planning Problems for evaluating the correctness of PDDL modelling

1) Scenario 1

```
(define (problem scenario1) (:domain classical-domain-farmbot)
(:objects
    home posA posB posC posD pos1 pos2 pos3 - position
```

```
    A B C D - plant
    seed1 seed2 seed3 seed4 - seed

)

(:init
    (farmbot-at home)
    (carry-camera)
    (tool-mount-free)
    (seeder-free)

    (tool-rack-at seederrack seeder seederPos)
    (tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
    (tool-rack-at weederrack weeder weederPos)
    (tool-rack-at soilsensorrack soilsensor soilsensorPos)

    (tool-at seederPos seeder)
    (tool-at wateringnozzlePos wateringnozzle)
    (tool-at weederPos weeder)
    (tool-at soilsensorPos soilsensor)

    (container-at posSeedTray seedtray)

    (container-has seedtray seed1)
    (container-has seedtray seed2)
    (container-has seedtray seed3)
    (container-has seedtray seed4)

    (match-seed-type-n-plant-type seed1 A)
    (match-seed-type-n-plant-type seed2 B)
    (match-seed-type-n-plant-type seed3 D)
    (match-seed-type-n-plant-type seed4 C)

    (no-plant-at posA)
    (no-plant-at posB)
    (no-plant-at posC)
    (no-plant-at posD)
```

```
)
(:goal (and
    (plant-at posA A)
    (plant-at posB B)
    (plant-at posC C)
    (plant-at posD D)
))
)
```

2) Scenario 2

```
(define (problem scenario2) (:domain classical-domain-farmbot)
(:objects
    home posA posB posC posD pos1 pos2 pos3 - position
    A B C D - plant
    seed1 seed2 seed3 seed4 - seed

)
(:init
    (farmbot-at home)
    (carry-camera)
    (carry-tool weeder)
    (seeder-free)

    (tool-rack-at seederrack seeder seederPos)
    (tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
    (tool-rack-at weederrack weeder weederPos)
    (tool-rack-at soilsensorrack soilsensor soilsensorPos)

    (tool-at seederPos seeder)
    (tool-at wateringnozzlePos wateringnozzle)
    (tool-at weederPos weeder)
    (tool-at soilsensorPos soilsensor)
    (container-at posSeedTray seedtray)

    (plant-at posA A)
```

```
    (plant-at posB B)
    (plant-at posC C)
    (plant-at posD D)

    (need-water posA A)
    (need-water posB B)
    (need-water posC C)
    (need-water posD D)
)

(:goal (and
    (watered posA A)
    (watered posB B)
    (watered posC C)
    (watered posD D)
))
)
```

3) Scenario 3

```
(define (problem scenario3) (:domain classical-domain-farmbot)
(:objects
    home posA posB posC posD pos1 pos2 pos3 - position
    A B C D - plant
    seed1 seed2 seed3 seed4 - seed

)

(:init
    (farmbot-at home)
    (carry-camera)
    (carry-tool weeder)
    (seeder-free)

    (tool-rack-at seederrack seeder seederPos)
    (tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
    (tool-rack-at weederrack weeder weederPos)
    (tool-rack-at soilsensorrack soilsensor soilsensorPos)
```

```
    (tool-at seederPos seeder)
    (tool-at wateringnozzlePos wateringnozzle)
    (tool-at weederPos weeder)
    (tool-at soilsensorPos soilsensor)
    (container-at posSeedTray seedtray)

    (plant-at posA A)
    (plant-at posB B)
    (plant-at posC C)
    (plant-at posD D)
)

(:goal (and
    (checked-moisture posA A)
    (checked-moisture posB B)
    (checked-moisture posC C)
    (checked-moisture posD D)
    (farmbot-at home)
))

)
```

4) Scenario 4

```
(define (problem scenario4) (:domain classical-domain-farmbot)
(:objects
    home posA posB posC posD pos1 pos2 pos3 - position
    A B C D - plant
    seed1 seed2 seed3 seed4 - seed

)

(:init
    (farmbot-at home)
    (carry-camera)
    (carry-tool weeder)
```

```
    (seeder-free)

    (tool-rack-at seederrack seeder seederPos)
    (tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
    (tool-rack-at weederrack weeder weederPos)
    (tool-rack-at soilsensorrack soilsensor soilsensorPos)

    (tool-at seederPos seeder)
    (tool-at wateringnozzlePos wateringnozzle)
    (tool-at weederPos weeder)
    (tool-at soilsensorPos soilsensor)
    (container-at posSeedTray seedtray)

    (plant-at posA A)
    (plant-at posB B)
    (plant-at posC C)
    (plant-at posD D)
)

(:goal (and
    (checked-weed-exist posA)
    (checked-weed-exist posB)
    (checked-weed-exist posC)
    (checked-weed-exist posD)
    (farmbot-at home)
))

)
```

5) Scenario 5

```
(define (problem scenario5) (:domain classical-domain-farmbot)
(:objects
    home posA posB posC posD pos1 pos2 pos3 - position
    A B C - plant
    w1 w2 w3 w4 w5 w6 - weed
)
```

```
(:init
    (farmbot-at home)
    (carry-camera)
    (tool-mount-free)
    (seeder-free)

    (tool-rack-at seederrack seeder seederPos)
    (tool-rack-at wateringnozzlerack wateringnozzle wateringnozzlePos)
    (tool-rack-at weederrack weeder weederPos)
    (tool-rack-at soilsensorrack soilsensor soilsensorPos)

    (tool-at seederPos seeder)
    (tool-at wateringnozzlePos wateringnozzle)
    (tool-at weederPos weeder)
    (tool-at soilsensorPos soilsensor)

    (container-at posSeedTray seedtray)

    (plant-at posA A)
    (plant-at posB B)
    (plant-at posC C)

    (weed-at pos1 w1 A)
    (weed-at pos2 w2 B)
    (weed-at pos3 w3 B)
)
(:goal (and
    (weed-removed pos1)
    (weed-removed pos2)
    (weed-removed pos3)
))
)
```

D. Results Collected From Simulation

TABLE IV: Results of simulation

Garden Layout	Domain	Planner	No. of repetition	Total time (seconds)	Time for plan dispatching and execution (s)	Cost (Time for dispatching and execution move action) (s)	Time for plan dispatching and execution (excl. cost) (s)	Time taken by planner (s)	Other time taken (s)	Time taken by planner (consider the actual time instead of the timeout for the two anytime planner) (s)
Small (15 position objects)	Classical domain (First PDDL domain)	BFWS	1	3198.679	3194.682	1490.965	1703.717	0.0035	3.9936	0.0035
			2	3238.11	3233.717	1409.969	1823.749	0.0031	4.3891	0.0031
			3	3239.502	3234.354	1210.62	2023.734	0.0029	5.1445	0.0029
			4	3149.689	3145.801	1251.148	1894.653	0.0028	3.8852	0.0028
			5	3218.007	3213.016	1338.339	1874.677	0.0038	4.9872	0.0038
		FF	1	2186.774	2183.438	588.3152	1595.123	0.0124	3.3235	0.0124
			2	2115.427	2111.621	955.7946	1155.826	0.0107	3.7951	0.0107
			3	2196.906	2191.687	790.8637	1400.823	0.0127	5.2068	0.0127
			4	2078.443	2074.715	539.9696	1534.745	0.0087	3.7199	0.0087
			5	2077.656	2073.832	937.5889	1136.243	0.0138	3.8107	0.0138
	Classical domain with action cost (Second PDDL domain)	LAMA	1	1798.447	1788.007	804.4536	983.5534	5.3027	5.1375	0.3488
			2	1805.018	1795.794	749.1613	1046.633	5.2453	3.9786	0.4041
			3	1797.618	1787.449	921.6994	865.7492	5.2971	4.872	0.4202
			4	1885.196	1875.097	513.5142	1361.582	5.2249	4.8745	0.3804
			5	1871.828	1860.41	897.8784	962.5314	5.0473	5.3713	0.4065
		Dual-BFWS-LAPKT	1	1842.409	1831.108	602.31	1228.798	5.2505	5.0504	0.3936
			2	1828.599	1817.295	530.8887	1286.406	5.1794	5.4252	0.3305
			3	1895.651	1886.426	899.3244	987.1018	5.2504	3.9747	0.3501
			4	1821.5	1810.851	668.6723	1142.178	5.098	4.551	0.3371
			5	1809.075	1799.511	929.1604	870.3501	5.0221	3.6419	0.3623
Large (50 position objects)	Classical domain (First PDDL domain)	BFWS	1	17489.52	17484.17	5611.732	11872.44	0.0581	5.2877	0.0581
			2	17815.9	17812.06	6964.468	10847.6	0.0683	3.7632	0.0683
			3	17358.43	17352.88	7087.269	10265.62	0.0641	5.485	0.0641
			4	17855.68	17848.68	6834.076	11014.6	0.0644	6.9382	0.0644
			5	17622.23	17618.76	6279.615	11339.15	0.0614	3.4077	0.0614
		FF	1	5970.084	5965.07	2497.327	3467.743	0.1641	4.8499	0.1641
			2	6009.126	6002.524	2232.3	3770.225	0.1388	6.4628	0.1388
			3	5719.54	5714.095	1682.015	4032.08	0.1393	5.3059	0.1393
			4	5833.446	5829.683	2511.209	3318.474	0.1539	3.6084	0.1539
			5	5833.463	5827.496	1392.485	4435.011	0.1337	5.8334	0.1337
	Classical domain with action cost (Second PDDL domain)	LAMA	1	3653.664	3590.064	1265.96	2324.103	50.0516	5.5489	12.5526
			2	3615.45	3549.505	950.5414	2598.963	50.5006	6.4445	12.3189
			3	3505.684	3445.21	858.2616	2586.948	50.9454	5.5283	14.1377
			4	3590.661	3527.38	1546.35	1981.03	50.95	8.3309	11.865
			5	3561.962	3499.337	957.293	2542.044	50.3676	7.2579	11.0586
		Dual-BFWS-LAPKT	1	3910.846	3871.52	1538.479	2333.041	30.9114	6.4149	13.7881
			2	3993.748	3956.194	1401.653	2554.541	30.3655	5.1883	12.4619
			3	3986.237	3943.51	1452.953	2490.557	30.8742	7.8528	13.9769
			4	4078.281	4035.288	1679.914	2355.374	30.5169	6.4768	11.3271
			5	4051.758	4011.938	1734.196	2277.743	30.0112	7.809	13.0854

All time in second, taking at most 4 significant figures after decimal point