Media Engineering and Technology Faculty
German University in Cairo

A Virtual Environment for Partial-Order Planning

Bachelor Thesis

Author:	Mohamed Ayman Tammaa
Supervisors:	Assoc. Prof. Haythem Ismail
Submission Date:	19 May, 2024

Media Engineering and Technology Faculty
German University in Cairo

A Virtual Environment for Partial-Order Planning

Bachelor Thesis

Author: Mohamed Ayman Tammaa
Supervisors: Assoc. Prof. Haythem Ismail
Submission Date: 19 May, 2024

This is to certify that

(i) the thesis comprises only my original work toward the Bachelor degree, and

(ii) due acknowledgement has been made in the text to all other material used.

Mohamed Ayman Tammaa
19 May, 2024

Acknowledgements

Text

Abstract

Abstact

Contents

Chapter 1

Introduction

1.1 Section Name

Some sample text with an **ac!** (**ac!**), some citation [**?**], and some more **ac2!** (**ac2!**).

1.2 Another Section

Reference to Section 1.1, and reuse of **ac!** nad **ac2!** with also full use of **ac2!** (**ac2!**).

Chapter 2

Background

Planning in Artificial Intelligence (AI) is a fundamental aspect in the field that allows agents to formulate sequences of actions and strategies to achieve a specific goal. It is used in a wide range of fields where agents need to make decisions and take actions based on the current state of the environment.

Classical planning is a type of planning that is used in AI to solve problems that can be represented as a set of states and actions. It deals with straightforward actions & with predictable and deterministic environments, where the agent can predict the outcome of its actions. The challenge in classical planning is to construct a sequence of actions that will transform the initial state of the environment into a desired goal state, while dealing with exponential growth in the search space, and dealing with the actions and steps in chronological order.

Among the different types of planning, we have **state space planning** and **plan space planning**. (They will be discussed in more details in the Key Definition section 2.1) In this thesis, we will focus on plan space planning, and more specifically on Partial Order Planning (POP).

2.1 Key Definitions and Concepts

Here we will tackle some key definitions and concepts that are essential to understand the POP algorithm. All definitions and concepts in this section are based on the book "Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig [16], the book "Automated Planning: Theory & Practice" by Malik Ghallab et al. [14], and on the lecture slides of the course "Introduction to Artificial Intelligence" by Prof. Haythem Ismail at the German University in Cairo [10]. Now, in the following list, we will look at some key definitions and concepts in the field of planning:

- **Agent**: An Agent is an entity that can perceive its environment, make decisions, and take actions to achieve a specific goal. An agent can be a robot, a computer program, or a human being. It can be a simple agent that follows a set of rules, or it can be a complex agent that uses AI to learn and adapt to its environment. For example, a self-driving car is an agent that uses sensors to perceive its environment, a specialized robot is an agent that can sense the environment and perform the required steps to achieve a specific goal.

- **State**: A State is a snapshot of the environment at a specific point in time. It represents the current configuration of the environment, including the location of objects, the status of objects, and the relationships between objects. In planning contexts, it can be represented as a set of propositions. For example, consider a robot that needs to navigate through a maze. The state of the robot can be represented as the location of the robot in the maze, the location of the walls, the location of the obstacles, and the location of the goal. The state of the robot changes as the robot moves through the maze, and the robot needs to update its state to reflect the changes in the environment.

- **State Space Planning**: State Space Planning is a type of planning that is used in AI to solve problems searching through a set of states and actions. In this type of planning, the world is represented as a set of states, and the agent can move from one state to another by applying actions. While the agent is searching, it represents the world as a graph, where the nodes are states and the edges are actions. It has the ability to expand any node using the available actions, and it can backtrack if it reaches a dead-end. When the agent reaches any node, it runs a goal test to check if the current state is the goal state. If the goal test is successful, the agent stops and returns the solution. If the goal test fails, the agent continues searching and expanding until it finds a solution.

  For example, consider a robot that needs to pickup a package from one location and deliver it to another location. The robot can represent the world as a set of states, where the initial state that the robot is $At(location_A)$, and the goal state is to satisfy the condition Delivered($package, location_B$). The robot can move from one state to another by applying actions like Move($location_A$, $location_B$) and Pickup($package$, $location_C$) and Deliver($package$, $location_B$). Of course, there are some constraints and preconditions that the robot needs to satisfy before applying any action. For example, the robot cannot deliver a package if it has not picked it up first, and the robot cannot pickup a package from a location if the robot is not at that location. One way to solve this problem is to start from the goal state and work backward to the initial state, applying the actions in reverse order, and this is called backward state space planning. To achieve Delivered($package$, $location_B$), the robot needs to Deliver($package$, $location_B$), and to deliver the package, the robot needs to Pickup($package$, $location_C$) & Move($Location_C$, $Location_B$), and to pickup the package, the robot needs to Move($location_A$, $location_C$). So, the robot needs to apply the actions Move($location_A$, $location_C$), Pickup($package$, $location_C$), Move($Location_C$, $Location_B$), and Deliver($package$, $location_B$) in this order to achieve the goal state Delivered($package$, $location_B$).

- **Total-Order Plan**: A Total-Order Plan is a plan that specifies a total order of actions. This means that it gives you a specific sequence of actions that need to be executed in a specific and strict order to achieve a specific goal. It does not give you the freedom or choices while executing the actions. For example, $LeftSock() \rightarrow RightSock() \rightarrow LeftShoe() \rightarrow RightShoe()$, is a total-order plan that specifies a strict order to achieve *wearing both shoes*. Other valid solutions may involve interchanging the order of the socks or the shoes as long as the socks are worn before the shoes. In Figure 2.1, we can see 6 different total-order plans and each one of them can be considered a linearization of the partial order plan in Figure 2.2.

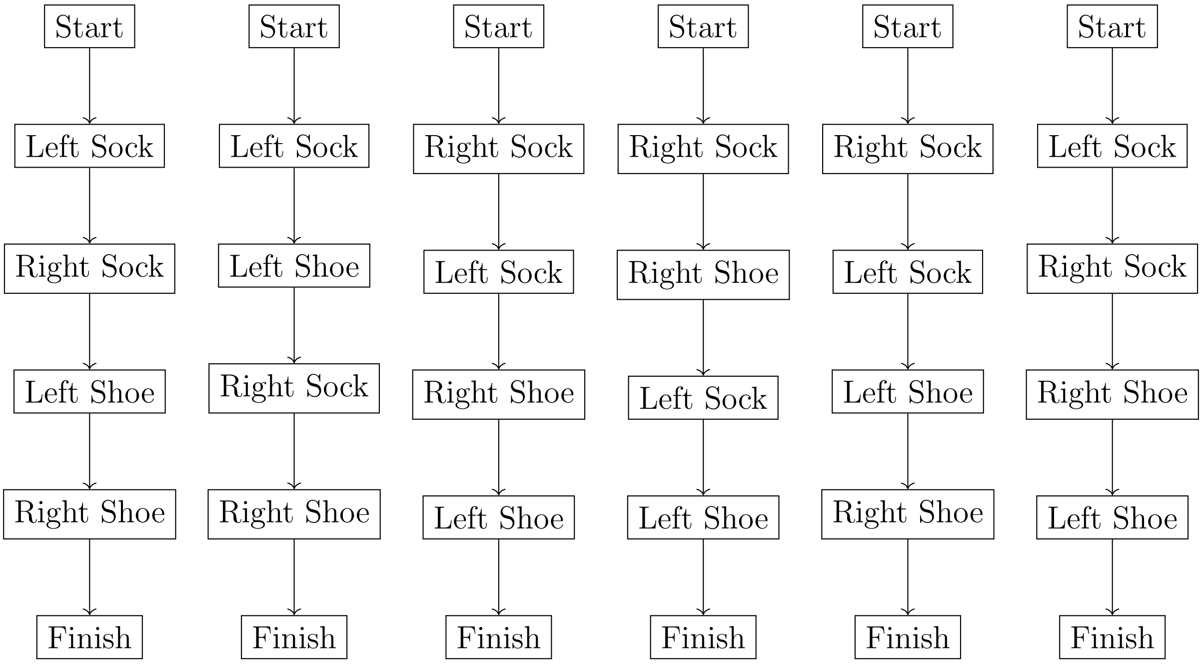

Figure 2.1: Socks & Shoes Total Order Plan solutions.

- **Partial-Order Plan**: A Partial-Order-Plan is a plan that does not specify a total order of actions, but instead specifies a partial order of actions. This means that it gives you a general structure of what needs to be done with some constraints and ordering between them, but some steps/actions are not given a specific order. This gives the agent the freedom to execute some actions in any order, as long as the constraints & general structure are satisfied. For example, consider the problem of putting on socks and shoes. The general structure is to put on the socks first, then the shoes. But, the order in which you put on the left and right sock and shoe can vary. One person might put on the left sock first, then the right sock, then the left shoe, and finally the right shoe. Another person might interchange the order of the 2 socks or the 2 shoes, and so on. The partial plan does not specify the actual order of the final actions. (See Figure 2.2)

- **Plan Space Planning**: Plan Space Planning is a type of planning that is used in AI to solve problems searching through a set of partial plans and actions. In this

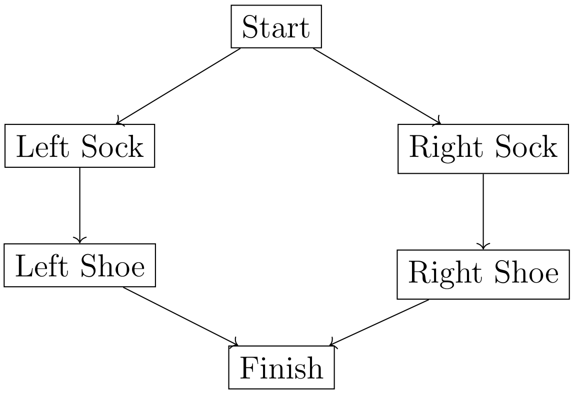

Figure 2.2: Socks & Shoes Partial Order Plan solution.

type of planning, the world is represented as some nodes, each node represents a partial plan, and the edges represent the actions that can be applied to move from one partial plan to another. The agent can explore the search space by expanding the nodes and applying the actions, can also backtrack, and it can explore multiple paths in parallel. The agent can also use problem decomposition to break down the problem into smaller subproblems, solve them independently, and then merge the subplans into a final plan. For example, consider the problem of cooking a meal. The agent can represent the world as a set of partial plans, where each partial plan represents a step in the cooking process. The agent can explore the search space by expanding the partial plans and applying the actions to move from one partial plan to another. The agent can use problem decomposition to break down the cooking process into smaller subproblems, such as chopping vegetables, boiling water, and frying meat. The agent can solve each subproblem independently and then merge the subplans into a final plan to cook the meal.

- **Least Commitment Strategy**: Least Commitment Strategy is a strategy that does not require the planner to commit to a specific order of actions. Instead, the planner can choose to leave some actions unordered, and the planner can choose to order actions only when necessary. This allows the planner to explore a larger space of possible plans, and it allows the planner to find plans that are more flexible and more robust.

- **Operator**: An Operator is a tuple $o = (name, preconds, effects)$ where:
  - $name$ is the name of the operator.
  - $preconds$ is a set of preconditions that need to be satisfied before applying the operator.
  - $effects$ is a set of effects that will be achieved after applying the operator.

  Example: $Buy(item, store)$, $preconds = \{At(store)\}$, $effects = \{Has(item)\}$.

- **Action**: An Action is a partially instantiated operator, (i.e. any ground instance of an operator with some of its variables instantiated). Example: $Buy(item, store)$, $Buy(Oranges, Market)$, $Buy(Phone, store)$, are all actions.

- **Achiever**: An Achiever is an operator that can achieve a specific precondition. For example, if we have a precondition p_j that needs to be satisfied, the achiever of p_j is an operator that has p_j in its effects, and their Most General Unifier (MGU) is consistent with the bindings in the plan.

- **Causal Link**: A Causal Link is in the form of $(a_i \xrightarrow{P_j} a_j)$ where:
  - a_i is an action.
  - a_j is another action linked to a_i.
  - P_j is a precondition of a_j, and at the same time an effect of a_i.

  An action a_i is said to achieve a precondition p_j if $p_j \in$ effects(a_i) and $p_j \in$ preconds(a_j). A causal link is initiated to ensure that there is another action, before the action with this precondition, that can satisfy it.
  Example: $Go(store) \xrightarrow{At(store)} Buy(item, store)$, where $Go(store)$ achieves the precondition $At(store)$ for the action $Buy(item, store)$.

- **Binding Constraints**: Binding Constraints is a set of constraints that bind variables to values. For example, if we have a binding $B = \{x \leftarrow \{1, 5\}, y \leftarrow 2\}$, this means that the variable x is bound to the values 1 or 5, and the variable y is bound to the value 2.

- **Planning Problem**: A Planning Problem is a problem that can be represented as a set of states and actions. The goal of the planning problem is to find a sequence of actions that will transform the initial state of the environment into a desired goal state. The planning problem can be solved using classical planning algorithms, such as state space planning and plan space planning. A planning problem can be represented as a tuple $P = (O, s_0, g)$ [14] where:
  - O is a set of operators.
  - s_0 is the initial state.
  - g is the goal state.

  Example: $Problem = ($
  - Operators: $\{Buy(item, store)$, $Go(somewhere)\}$
  - Initial State: $\{At(Home)\}$
  - Goal State: $\{Has(Oranges)$, $At(Home)\}$

).
  This planning problem represents a simple problem where the agent needs to buy some oranges from the store and go back home.

- **Ordering Constraint ($\prec$)**: An Ordering Constraint is a constraint that specifies the order of actions in a partial order plan. It is in the form of $a_i \prec a_j$, which means that action a_i must be executed before action a_j. The ordering constraint is used to ensure that the actions are executed in the correct order and to prevent conflicts between actions.

- **Partial Plan**: A Partial Plan is a tuple $\pi = (A, \prec, B, L)$ where:
  - A is a set of actions, or partially instantiated Operators.
  - $\prec$ is a set of ordering constraints between actions in the form of $a_i \prec a_j$
  - B is a set of bindings.
  - L is a set of causal links.

  are the components of a partial plan.

- **Consistency of Partial Plans**: A partial order plan $\pi = (A, \prec, B, L)$ is consistent if it satisfies the following conditions: [14]
  - The transitive closure of the ordering constraints $\prec$ is a strict partial order.
  - every subsitution σ which binds a variable x to a value in its allowed domain D_x is consistent with the bindings in all other constraints in B.

- **Threat**: A Threat in a partial plan is an action that could potentially undo the effects of another action. A threat is a potential problem in a partial plan that needs to be resolved to make the plan consistent. An action a_k threatens a causal link $(a_i \xrightarrow{P_j} a_j)$ if it has the following properties:
  - $e_k (\in \text{effect of } a_k)$ unifies with $\neg p_j$.
  - the MGU of e_k and $\neg p_j$ is consistent with the bindings in B.
  - $\prec \cup \{a_i \prec a_k, a_k \prec a_j\}$ is consistent.

  Once the 3 conditions are met, we can say that a_k threatens the causal link $(a_i \xrightarrow{P_j} a_j)$. Example: $Go(Cinema)$ threatens the causal link $(Go(Store) \xrightarrow{At(Store)} Buy(Oranges, Store))$, as it can negate the effect of being $At(Store)$.

- **Partial Plan Completion**: A partial plan $\pi = (A, \prec, B, L)$ is complete if all the preconditions of the operators or actions in A are satisfied and causally linked, and all causal links in L are not threatened by any action in A.

2.2 Partial Order Planning

Partial Order Planning (POP) is a plan-space search algorithm (See definition in Section 2.1) that searches through a set of partial plans. Unlike other planning algorithms, POP gives partial-ordered plans, which means that it does not specify a total order of actions. This gives it the advantage over total order planning algorithms that it can use problem decomposition, work on several subproblems in parallel independently, solve them with several subplans, and then merge the subplans into a final plan.

POP uses least commitment strategy that does not require the planner to commit to a specific order of actions, as defined in Section 2.1. In addition, POP uses closed-world assumption, which means that the planner assumes that everything that is not known or not stated is false. This allows the planner to make assumptions about the world and to make decisions based on the available information. POP is also sometimes represented as a Directed Acyclic Graph (DAG), where the nodes represent actions, and the edges represent the causal links or the ordering constraints between actions. (See Figure 2.2 for an example of a DAG representing a partial order plan).

2.2.1 POP Algorithm

The Partial Order Planning (POP) algorithm works by incrementally building a partial order plan, and then refining the plan by adding operators, ordering constraints, bindings, and causal links to resolve threats and make the plan consistent. The planner keeps track of a set of pairs (a_i, p_i), under the name of *agenda*, where a_i is an action and p_i is a precondition of a_i. The *agenda* is used to keep track of the unsatisfied preconditions in the plan.

Overview of the POP Algorithm Execution

The POP algorithm starts by calling the POP function with the initial state s_0 and the goal state g, and some set of operators O. The POP function then initializes the plan with the initial and goal actions (a_0 & a_∞ respectively). The initial action (often referred to as Start) does not have any preconditions, just have effects, and these effects correspond to the initial state of the world. The goal action (often referred to as Finish) does not have any effects, just have preconditions, and these preconditions correspond to the goal state of the world (the conditions you need to meet to finish). Finally, the POP function initializes the agenda with the preconditions of the goal action a_∞. For each precondition p_i in the agenda, $POP()$ adds the pair (a_i, p_i) to the agenda. Then the algorithm calls the POP1 function to start building the plan. The POP1 function selects any action from the agenda, randomly or based on some heuristic, then removes it from the agenda. It then finds the achievers of the selected action, and if there are no achievers, it returns a failure, as there are some condition that cannot be satisfied. If there are achievers, it nondeterministically chooses one of the achievers and adds it to the plan, as in line 14 of the algorithm 1 (we will talk about nondeterminism in more details in subsection 3.1.1). It then updates the ordering constraints, bindings, and causal links in the plan in lines 15 to 17. $POP1()$ then checks if the selected action is already in the plan, if not, it does three things: adds the new action to the set of actions A, updates the ordering constraints

Algorithm 1 POP Algorithm

Function POP(O, s_0, g)
Ensure: a plan
return POP1($\{a_0, a_\infty\}, \{a_0 \prec a_\infty\}, \emptyset, \emptyset$, $\{a_\infty\}\times$ Preconds(a_∞))

Function POP1($\pi = (A, L, \prec, B)$, agenda)
Ensure: a plan
if agenda $==\emptyset$ **then**
  return π
end if
Select any pair (a_i, p_i) and remove it from agenda
achievers $\leftarrow$ the set of operators achieving (a_i, p_i)
if achievers $==\emptyset$ **then**
  return *failure*
end if
Nondeterministically choose some operator $a_j \in$ achievers
$L \leftarrow L \cup \{\langle a_j \xrightarrow{P_i} a_i\rangle\}$
Update $\prec$ with $a_j \xrightarrow{P_i} a_i$
Update B with binding constraints of this link
if $a_j \notin A$ **then**
  $A \leftarrow A \cup \{a_j\}$
  Update $\prec$ with $a_0 \prec a_j$ and $a_j \prec a_\infty$
  agenda $\leftarrow$ agenda $\cup\{(a_j, p_j)|p_j \in$ preconds$(a_j)\}$
end if
$\pi \leftarrow$ RESOLVE-THREATS(π, a_j, $\langle a_j \prec a_i\rangle$)
return POP1(π, agenda)

Function RESOLVE-THREATS($\pi = (A, L, \prec, B)$, a_l, L)
Ensure: a plan
for each threat a_k on $(a_i \xrightarrow{P_j} a_j)$, where $a_k = a_l$ or $(a_i \xrightarrow{P_j} a_j) = l$ **do**
  Nondeterministically choose one of the following:
  - Update $\prec$ with $a_k \prec a_i$
  - Update with $a_j \prec a_k$
  - Add a binding constraint to B which renders p_i nonunifiable with any threatening effect of a_k.
end for
return π

with the new action to be ordered between the *Start* and *Finish* actions, and adds the preconditions of the new action to the agenda. Finally, it resolves any threats in the plan in line 23 and calls the POP1 function recursively with the updated plan and agenda. The POP1 function continues to build the plan until the agenda is empty, at which point it returns the plan as the solution. If at any point the algorithm encounters a failure, it backtracks and tries a different path. The algorithm continues to explore the search space until it finds a valid plan or exhausts all possible paths.

Threats Resolution

Threats are potential problems in a partial plan that need to be resolved to make the plan consistent. A threat occurs when an action could potentially undo the effects of another action. The POP algorithm uses a threat resolution mechanism to resolve threats and make the plan consistent. The algorithm checks for threats in the plan and resolves them by adding ordering constraints, bindings, or causal links to prevent the threat from occurring. The function **RESOLVE-THREATS** takes a partial plan $\pi = (A, L, \prec, B)$, an action a_l, and a causal link L as input and resolves any threats in the plan. The function gets any threat a_k that threatens the causal link L, or any causal link that a_l itself threatens. Letting a_k be the action that threatens any of the causal links, and $link = (a_i \xrightarrow{P_j} a_j)$ be the causal link that a_k threatens, the function nondeterministically chooses one of the following options to resolve the threat: update the ordering constraints with $a_k \prec a_i$ (called **Demotion**), update with $a_j \prec a_k$ (called **Promotion**), or add a binding constraint to B that renders p_i nonunifiable with any threatening effect of a_k. The function then returns the updated plan with the resolved threats. The threat resolution mechanism is an essential part of the POP algorithm, as it ensures that the plan is consistent and that the actions are executed in the correct order.

2.3 Unity 3D Engine

Unity is a cross-platform game engine developed by Unity Technologies. It was first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. The engine has since been gradually extended to support more than 25 platforms. It is used to create interactive 2D, 3D, VR, and AR applications. The engine can be used to create both three-dimensional and two-dimensional games as well as simulations for its many platforms. Unity's core advantages are its ease of use, and flexibility. Unity is a powerful engine that is used by developers around the world to create a wide range of games and applications. One of the strongest features of Unity is its built-in physics engine, which allows developers to create realistic physics simulations in their games in a short amount of time.

2.3.1 Virtual Reality (VR) in Unity

Unity is a popular game engine that is used to create VR applications. Unity provides a wide range of tools and features that make it easy to create VR experiences. It supports a wide range of VR devices, including the Oculus Rift, HTC Vive, and PlayStation VR. It also provides a set of tools and APIs that make it easy to create VR applications. Unity's VR support includes features such as stereoscopic rendering, head tracking, and motion controllers. Virtual Reality (VR) now has a wide range of applications, from games and simulations to training and education. It is a powerful tool that allows developers to create immersive and interactive experiences for users.

2.3.2 Visualization of the Partial Order Planning (POP) Algorithm in Unity

In this thesis, Unity was used to visualize the POP algorithm. I created a 3D environment that represents the planning problem, and I used Unity's physics engine to simulate the actions and effects of the operators as a self arranging DAG in the plan. I also used Unity's VR support to create an immersive VR experience that allows users to interact with the planning problem in a virtual environment. The goal of this visualization is to provide a more intuitive and interactive way to understand the POP algorithm and how it works for new learners. By visualizing the algorithm in Unity, we hope to make it more accessible and engaging for users, and to help them better understand the concepts and principles behind the POP algorithm.

Chapter 3

Design and Implementation

In this chapter, we will discuss the design and implementation of the POP algorithm, the design of the Virtual Environment and the Unity components that were used to visualize the POP algorithm. The chapter will also discuss the design of the user interface and the interaction techniques that were used in the VR game.

3.1 POP Algorithm

This section will discuss some of the design decisions that were made during the implementation of the POP algorithm. It will include an overview of the implemented algorithms that were used beside the POP algorithm. In addition, it will discuss the data structures that were used to represent the planning domain and the planning problem.

3.1.1 Nondeterministic Achievers & Threat Search

Partial Order Planning (POP) needs to be able to handle some form of nondeterminism in the planning domain. This is because the planner needs to be able to handle situations where there are multiple ways to achieve a precondition of an action. Nondeterminism is a concept that is used in computer science to describe the occurrence of events without a predictable outcome, where multiple outcomes are possible from a given state. In the context of planning, nondeterminism is used to describe the situation where there are multiple ways to achieve a precondition of an action, for example. This cannot be achieved in practice without trying to search all possible ways of the choices that can be made. So, to model nondeterminism in the planning domain, we need to use some form of graph search algorithm to search for all possible ways to achieve a precondition of an action efficiently. In this project, multiple graph search algorithms were implemented to handle nondeterminism in the planning domain. These algorithms are A-Star (A*) Search, Breadth First Search (BFS), and Depth Limited Search (DLS) which is a variation of Depth First Search (DFS). All of these algorithms are discussed in detail in the following sections.

3.1.2 Graph Search Algorithms

Graph search algorithms are widely used in computer science to solve problems that can be represented as graphs. They have many applications in various fields such as artificial intelligence, computer graphics, and computer vision. In this project, graph search algorithms were used to solve the problem of nondeterminism in the planning domain. The following graph search algorithms were implemented in the project:

- **A-Star (A*) Search Algorithm**
  A* Search is a graph search & traversal algorithm that is widely used in computer science due to its efficiency, optimality, and completeness. It is an informed search algorithm that uses a heuristic function to estimate the cost of reaching the goal from a given state. The heuristic function is used to guide the search towards the goal state by selecting the most promising nodes to explore. A* Search is an extension of Dijkstra's algorithm that uses a heuristic function to estimate the cost of reaching the goal from a given state[6]. A* is a Best First Search algorithm that uses a priority queue to store the nodes to be explored. The priority queue is ordered based on a cost function that combines the cost of reaching the node from the start state and the heuristic estimate of the cost of reaching the goal from the node. The cost function is defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of reaching the node from the start state and $h(n)$ is the heuristic estimate of the cost of reaching the goal from the node. The algorithm selects the node with the lowest f value from the priority queue and explores its neighbors until the goal is reached. The algorithm is guaranteed to find the optimal solution if the heuristic function is admissible, i.e., it never overestimates the cost of reaching the goal from a given state. The algorithm is also complete if the search space is finite and the heuristic function is consistent.

  In this project, $g(n)$ is the path (level) cost from the root node to the current node, and $h(n)$ heuristic was chosen to be the number of unsatisfied preconditions, i.e., the count of pairs in the *agenda* discussed in section 2.2.1.

- **Depth Limited Search (DLS) Search Algorithm**
  First, let's discuss the DFS algorithm. DFS is an uninformed search algorithm that is widely used in computer science to explore a graph or tree data structure. It is a depth-first traversal algorithm that explores the nodes deep in the graph before exploring the nodes at the same level. The algorithm starts at the root node and explores the nodes along each branch before backtracking to explore the other branches. The algorithm uses a stack data structure to store the nodes to be explored. The algorithm is not guaranteed to find the optimal solution, but it is complete if the search space is finite. The algorithm is also efficient in terms of memory usage as it only needs to store the nodes along the current path. However, the algorithm can get stuck in infinite loops if the graph contains an infinite deep path. To avoid this problem, a depth limit can be imposed on the search to limit the depth of the search tree. This is known as the Depth Limited Search (DLS)

algorithm. The algorithm is similar to DFS but with an additional depth limit parameter that specifies the maximum depth of the search tree. The algorithm stops exploring a branch when the depth limit is reached and backtracks to explore other branches. The algorithm is complete if the depth limit is greater than the depth of the optimal solution. The algorithm is also efficient in terms of memory usage. In this project, DLS was also provided as an option to be used in the POP algorithm.

- **Breadth First Search (BFS) Search Algorithm**
  BFS is an uninformed search algorithm that is widely used to explore a graph or tree data structure. It is a breadth-first traversal algorithm, meaning that it explores the nodes at the same level before exploring the nodes at the next level. The algorithm starts at the root node and explores the nodes at the same level before moving to the next level. The algorithm uses a queue data structure to store the nodes to be explored. The algorithm is guaranteed to find a solution if one exist, even if the graph is infinite. However, the algorithm is not guaranteed to always find the optimal solution. The algorithm is also inefficient in terms of memory usage as it needs to store all the nodes at the current level. In this project, BFS was also provided as an option to be used in the POP algorithm.

3.1.3 Unification Algorithm

Finding the Most General Unifier (MGU) of two terms is a crucial step in the POP algorithm. The MGU is used to unify two terms by finding a substitution that makes the two terms equal. The MGU is the most general substitution that can be applied to the two terms to make them equal. For example, consider the following terms: $P(x, B)$ and $P(A, y)$. The MGU of these two terms is $\{x/A, y/B\}$, which means that the terms can be unified by substituting A for x and B for y to make them equal to $P(A, B)$. The MGU is used to unify the preconditions of an action with the current state of the world to determine if the action can be applied. The MGU is also used to unify the effects of an action that can be used, for example, to detect threats in the partial plan.

In this project, the MGU algorithm was implemented using a variation of the unification algorithm discussed in Russell and Norvig's book in Chapter 9 [15], and taken from the lecture slides of the course *Introduction to Artificial Intelligence* by Prof. Haythem Ismail [10]. There is a convention used in the algorithm implementation: the variables are represented as strings starting with a lowercase letter, and the constants are represented as strings starting with an uppercase letter. The algorithm described in Algorithm 2 has three main functions: UNIFY, UNIFY1, and UNIFYVAR. The UNIFY function takes two exrpressions as input, listifies them, and then calls the UNIFY1 function with the two listified expressions and an empty substitution set. An example of listifying an expression is converting the expression $P(x, f(y, z))$ to the list $[P, x, [f, y, z]]$.

Algorithm 2 Unification Algorithm

1: **function** UNIFY($E1, E2$) :
2: **return** UNIFY1(LISTIFY($E1$), LISTIFY($E2$), {});
3: ---
4:
5: **function** UNIFY1($E1, E2, \mu$) :
6: **if** $\mu =$ fail **then**
7:   **return** fail
8: **end if**
9: **if** $E1 = E2$ **then**
10:   **return** μ
11: **end if**
12: **if** VAR?($E1$) **then**
13:   **return** UNIFYVAR($E1, E2, \mu$)
14: **end if**
15: **if** VAR?($E2$) **then**
16:   **return** UNIFYVAR($E2, E1, \mu$)
17: **end if**
18: **if** ATOM?($E1$) or ATOM?($E2$) **then**
19:   **return** fail
20: **end if**
21: **if** LENGTH($E1$) != LENGTH($E2$) **then**
22:   **return** fail
23: **end if**
24: **return** UNIFY1(REST($E1$), REST($E2$), UNIFY1(FIRST($E1$), FIRST($E2$), μ))
25: ---
26:
27: **function** UNIFYVAR(x, e, μ) :
28: **if** $t/x \in \mu$ and $t \neq x$ **then**
29:   **return** UNIFY1(t, e, μ)
30: **end if**
31: $t =$ SUBST(μ, e)
32: **if** x occurs in t **then**
33:   **return** fail
34: **else**
35:   **return** $\mu \circ \{t/x\}$
36: **end if**

The UNIFY1 function takes two listified expressions and a substitution set as input and returns the most general unifier of the two expressions. The UNIFYVAR function takes a variable, an expression, and a substitution set as input and returns the most general unifier of the variable and the expression. The main algorithm starts in line 6 by checking if the substitution set is a failure, in which it returns a failure. Then, it checks if the two expressions are equal, in which it returns the curernt substitution set. Then, it checks if $E1$ or $E2$ is a variable, in which it calls the UNIFYVAR function with the first parameter being the variable and the second parameter being the expression. Then, it checks if $E1$ or $E2$ is an atom, in which it returns a failure in line 19. The reason for this is that after making sure that neither $E1$ or $E2$ are variables, and they are not equal, then they must be different atoms, and thus cannot be unified. Then, it checks if the length of $E1$ is not equal to the length of $E2$, in which it returns a failure in line 22, as we cannot unify two expressions with different lengths, like $P(x)$ and $P(x, y)$. Finally, it calls the UNIFY1 function recursively with the rest of the two expressions and the unification of the first elements of the two expressions.

The UNIFYVAR function starts by checking if the variable x is already in the substitution set and bound to a term t that is not equal to x, in which it calls the UNIFY1 function with the term t and the input expression e. If the variable x is not in the substitution set, then it substitutes every term in the expression e with the substitution set μ. Then, it checks if the variable x occurs in the term t, in which it returns a failure, as we cannot unify a variable with an expression that contains the variable. If we had x and $f(x)$, then the unification would fail, since it would result in an infinite recursion. Otherwise, it returns the substitution set μ composite with the substitution $\{t/x\}$, which means that the term t is substituted for the variable x in the substitution set μ.

```
/POP
├── controller
│   └── POPController.cs
└── engine
    ├── Action.cs
    ├── Agenda.cs
    ├── BindingConstraints.cs
    ├── CausalLink.cs
    ├── Helpers.cs
    ├── Literal.cs
    ├── Operator.cs
    ├── ParitalPlan.cs
    ├── Planner.cs
    ├── PlanningProblem.cs
    └── PriorityQ.cs
```

Figure 3.1: Folder Structure of the POP Algorithm Engine

3.1.4 Classes and Folder Structure

The POP algorithm engine was implemented in C#. The algorithm engine was divided into two main folders: the controller folder and the engine folder. The folder structure of the POP algorithm engine is shown in Figure 3.1. The controller folder contains the

POPController class, which is responsible for controlling the execution of the POP algorithm for Unity. The engine folder contains all the classes that implement the POP algorithm. The class diagram of the POP algorithm presented in Figure 3.2 shows the main classes and their relationships in the POP algorithm. In the following list, we will discuss the main classes, their responsibilities, how they are implemented, and what their structure is:

- **PlanningProblem Class:**
  The **PlanningProblem** class is responsible for representing the planning problem, that is the only input to the Partial Order Planning algorithm. The **PlanningProblem** class has the initial state, the goal state, and the set of operators as its members. The initial and goal states are represented as list of *Literal* objects. The operators are represented as a list of *Operator* objects. The class keeps hold of a list of *Literals*, that is not given as an input, but is instantiated automatically once the planning problem is initialized, and it is used to keep track of the existing literals from the initial and goal states in addition to the literals in the preconditions and effects of the operators. This list is to assist the planner.

  The class has only one functional method, which is the **GetListOfAchievers()** method, that takes a *literal* as input. The **GetListOfAchievers()** method is responsible for returning a list of operators that can achieve the input *literal*. The **PlanningProblem** class also has some predefined problems that can be used to test the POP algorithm and the VR game. Among these problems are the *Socks and Shoes problem*, the *Milk, Bananas, and Cordless Drill problem*, the *Groceries Buying problem* which is a simplified variation of the *Milk, Bananas, and Cordless Drill problem*, and some other problems.

- **PartialPlan Class:**
  The **PartialPlan** class is responsible for representing the partial plan that is being constructed by the POP algorithm. The **PartialPlan** class has a set of actions, a set of causal links, an instance of *BindingConstraints* object, and a set of ordering constraints as its members. The class contains some helper functions that are used by the planner like the *GetListOfActionAchievers()* method, and some others that are shown in the class diagram in Figure 3.2.

- **Planner Class:**
  It is the main class that has the core logic of the POP algorithm. The **Planner** class has the Planning Problem, the Partial Plan, and the Agenda as its members. It also has variables counter array to keep track of the generated distinct variables. The following list will discuss the logic and flow of the POP algorithm in the **Planner** class:

  - The **Planner** class has the *POP()* method that implements the POP algorithm. The *POP()* method is responsible for generating the plan by creating a priority queue or stack to store nodes to be used while searching. The *POP()* method also has a loop that iterates over the priority queue or stack until the goal is reached or no nodes are left. For each node, it checks if the node partial plan is not acyclic, if it is not, then it pops the node from the priority queue or stack and continues to the next node. If the partial plan is acyclic, then it checks if the node is a goal node, if it is, then it returns the plan. If the node is not a goal node, then it expands the node by applying the applicable actions to the node.

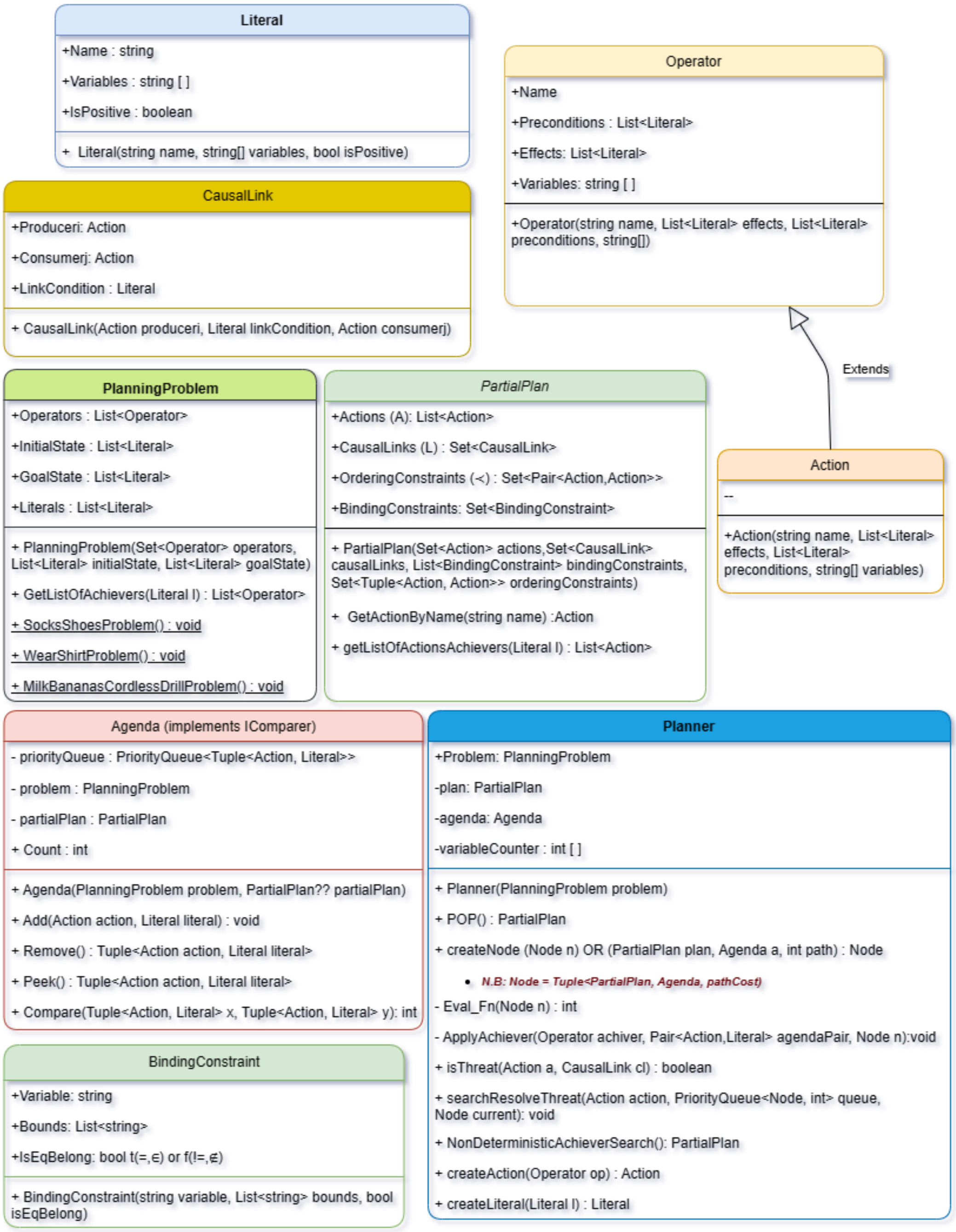

Figure 3.2: Class Diagram of the POP Algorithm

- The *EXPAND()* method is responsible for expanding the node. It first chooses a pair of (*action*, *precondition*) from the *agenda* based on some heuristic. The heuristic that I chose is to prioritize the pair with the precondition that has the least number of achievers, and this will be discussed in more details later in this section in the **Agenda** class implementation. Then, after selecting a pair of (*action*, *precondition*), the *POP()* method gets the achievers by gathering the existing actions and new operators that can achieve this precondition, and for each achiever, it creates a new node applies this achiever to it and then adds this node to the main queue or stack.
- The **POP** method also has a helper method called *ApplyAchiever()* that applies the achiever to the node. The *ApplyAchiever()* method works by unifying the achiever's effect with the precondition of the action, then it adds the binding constraints to the partial plan of the node, and then it adds the causal link to the partial plan, as well as updating the ordering constraints with the achiever being ordered before the action with the precondition we are trying to achieve. It also checks whether the achiever is an existing action or a new action, so that if it is a new action, it adds it to the partial plan, update its ordering constraints to be between the *Start* and *Finish* actions, and adds the new action's preconditions to the agenda.
- After applying the achiever, the *EXPAND()* method checks if the new node has threats before adding it to the priority queue or stack through a helper function, *searchResolveThreats()*. The *searchResolveThreats()* method is responsible for searching for threats in the partial plan and resolving them recursively until no threats are left. The *searchResolveThreats()* method has the new added action achiever and the new causal link as inputs. It first starts to check if any of the causal links in the partial plan is threatened by the new action, and if it is, then it resolves the threat by trying promotion and demotion on new cloned nodes. If no other threats are found, then the new node is added to the priority queue or stack.
- The *searchResolveThreats()* method also has a helper method called *isThreat()* that checks if a causal link is threatened by an action. The *isThreat()* method works by checking the conditions, discussed in the definition in Section 2.1, to be a threat. The *searchResolveThreats()* then checks if the new causal link input is threatened by any of the existing actions in the partial plan. If it is, then it follows the same steps as before to resolve the threat until there is no threats.
- No node is added to the main queue or stack until it passes all the checks and has no threats. All nodes outputted from the resolving of threats caused by the new action are then passed to checking whether the new causal link is threatened by any other action. After extensive testing, the planner was found to output wrong results of the all threats are not resolved recursively, or through an iterative approach of the recursive method, as this is the only stage where threats are checked and resolved.

- **Agenda Class:**
The **Agenda** class is responsible for representing the agenda that is used by the POP algorithm. The agenda is a list of pairs of (*action*, *precondition*) that are used to keep track of the preconditions that need to be achieved. The class internally uses a priority queue to store the pairs of (*action*, *precondition*). The chosen heuristic is to prioritize the pair with the precondition that has the least number of achievers. This is because the precondition with the least number of achievers minimizes the branching factor of the search tree, which leads to a more efficient search [17]. This leads to detecting special cases faster, like the case where the precondition has only one achiever, in which the planner has no any other choice but to apply this achiever. The **Agenda** class has a custom comparer method that compares with the number of achievers of the preconditions. Finally, in the special case where the number of achievers is zero, the class throws an exception, indicating that there is no solution to the problem, and something is wrong with the input problem.

- **Operator Class:**
The **Operator** class is responsible for representing the operators that are used in the planning domain. The operators has a name, a list of preconditions, a list of effects, and a string variables array to represent the unbonded variables and their relations in the effects and the preconditions of the operator.

- **Action Class:**
The **Action** class is responsible for representing the actions that are used in the partial plan. The *Operator* class is the superclass of this class. The **Action** class inherits the name, preconditions, effects, and variables array from the *Operator* class. The class also has a method that checks if the action has conflicts in the effects or the preconditions. The conflicts that *hasConflictingPreconditionsOrEffects()* method checks is of of that format: for example if $P(x), \neg P(y) \in$ preconditions, where x and y are bound to the same variable, and P is the same predicate. This is a conflict because it is impossible for x and y to be the same variable and different at the same time.

- **Literal Class:**
The **Literal** class is responsible for representing the literals, the conditions or the predicates that are used in the planning domain. The literals, in this project, are used to represent the preconditions, the effects, the initial state, and the goal state. The **Literal** class has a name, a list of arguments, and a boolean variable to represent the negation of the literal.

- **BindingConstraints Class:**
The **BindingConstraints** class is responsible for representing the binding constraints that are used in the partial plan. The binding constraints contains all variables that are bound and equal (or not equal) to a constant term or another variable. As mentioned in subsection 3.1.3, there is a convention that variables are lowercase strings and constants are uppercase strings. In this class, the focus os on four

main methods: *SetEqual()*, *SetNotEqual()*, *getBoundEq()*, and *getBoundNE()*. The *SetEqual()* and *SetNotEqual()* methods are used to set the constraints of the variables to be equal or not equal to each other. The *getBoundEq()* and *getBoundNE()* methods are used to get the variables that are bound equal or not equal to a term or another variable.

In this project, for the **non-equality constraints**, I used a regular *Hash Map* or *Dictionary* data structure to store the constraints. The key of the *Dictionary* is the variable, and the value is a list of variables not equal to the key variable. There is no relation between the variables in the list, and the list is not ordered.

As for the **equality constraints** between the variables, I used a *Disjoint Sets* representation. **Disjoint Sets** data structure is a set of elements partitioned into a number of non-overlapping subsets[7][8]. The main reason for using a *Disjoint Set* is that it is efficient in terms of time complexity. The implementation used in this project uses the *Union-Find* algorithm to merge the sets and find the representative of the set, in other words, the constant term that is bound to the variable, or a variable if no constant term is bound to it. The *Union-Find* implementation guarantees a time complexity of $O(\log n)$, where n is the number of variables in the partial plan. This number may be large for some problems as the planner uses an incremented counter to generate new variables in action's parameters, as well as in the effects and preconditions of the actions. This number continues to grow as the planner decides to backtrack and generate new actions. Besides the time complexity, the *Disjoint Set* was chosen because for all equal variables in a set, it will always have the same root, which is the representative of the set, which can make things easier to trace for humans.

Using both data structures, the *Dictionary* and the *Disjoint Set*, the planner can easily check if there exists some logical contradiction in the binding constraints. For example, if there is a variable x that is bound equal to a variable y, and y is bound equal to a third variable z, then another constraint that makes x not equal to z is added directly or indirectly, then the **BindingConstraints** class can easily detect this contradiction and stop the search.

- **CausalLink Class:**
  The **CausalLink** class is responsible for representing the causal links that are used in the partial plan. The exact definition of a causal link is discussed in the definition in Section 2.1. The **CausalLink** class has a *LinkCondition* of type *Literal*, a *Produceri* of type *Action*, and a *Consumerj* of type *Action*. The *Produceri* is the action that has the effect that is the *LinkCondition*, and the *Consumerj* is the action that has the precondition that is also the *LinkCondition*.

- **PriorityQ Class:**
  The **PriorityQ** is a class taken from the Microsoft documentation and GitHub repository under the MIT license[13][12]. The class is used to mimic the internal original implementation of the *PriorityQueue* in C#. The reason for creating this class is that the latest Unity Long Term Support (LTS) version at the time of

writing this thesis uses an older version of C# language (C# 9.0) that does not have the *PriorityQueue* class.

3.1.5 POP Terminal Output

The POP algorithm was tested using the *Socks and Shoes problem*, the *Milk, Bananas, and Cordless Drill problem*, the *Groceries Buying problem*, the *Spare Tires problem*, and some other custom problems. The POP algorithm was able to find the optimal solution for all the problems that were tested. Next, we will show the output for the *Milk, Bananas, and Cordless Drill problem* and the *Spare Tires problem.*

Milk, Bananas, and Cordless Drill Problem

The *Milk, Bananas, and Cordless Drill problem* is a problem where the agent needs to buy milk and bananas from the supermarket and go to the hardware store to buy a cordless drill. Now let's see the input to the planner in Listing 3.1:

```
PlanningProblem milkBananasCordlessDrill = new PlanningProblem(
    operators: new HashSet<Operator> {
        new Operator("Buy", variables: new []   { "x" },
            preconditions: new List<Literal> { new ("Sells", new
               []{ "store", "x" }), new ("At",new[]{ "store" })
               },
            effects:     new List<Literal> { new ("Have", new[]{"
               x"}) }
        ),
        new Operator("Go", variables: new[]   { "there" },
            preconditions:  new List<Literal> { new ("At", new[]{
                "here" }),        new ("At", new[]{"there"}, false
               ) },
            effects:        new List<Literal> { new ("At", new[]{
                "here" }, false), new ("At", new[]{"there"}) }
        )
    },
    initialState: new List<Literal>{ new("At",new []{"Home"}),
       new("Sells", new []{"SM", "Milk"}), new("Sells", new []{"
       SM", "Bananas"}), new("Sells", new []{"HWS", "Drill"})
                    , new("At",new []{"HWS"}, false), new("At",
                      new []{"SM"}, false)},
    goalState: new List<Literal> { new("At", new string[] { "Home
       " }), new("Have", new[] { "Milk" }), new("Have", new[] { "
       Bananas" }), new("Have", new[] { "Drill" }) }
);
```

Listing 3.1: Milk, Bananas, and Cordless Drill Problem Input to the Planner

The planner was able to find the optimal solution for the *Milk, Bananas, and Cordless Drill problem* in runtime of 00.260 seconds. The output of the planner is shown in Listing 3.2:

```
    Searching.....................................

Plan  found:


Linearized Steps:
*************
Start() -> Go(HWS) -> Buy(Drill) -> Go(SM) -> Buy(Bananas) -> Buy
   (Milk) -> Go(Home) -> Finish()
************

Actions: Start(), Finish(), Go(Home), Buy(Drill), Buy(Milk), Go(
   SM), Buy(Bananas), Go(HWS)

Links:
Go(Home) --At(Home)--> Finish(),
Buy(Drill) --Have(Drill)--> Finish(),
Start() --Sells(HWS, Drill)--> Buy(Drill),
Buy(Milk) --Have(Milk)--> Finish(),
Start() --Sells(SM, Milk)--> Buy(Milk),
Go(SM) --At(SM)--> Buy(Milk),
Buy(Bananas) --Have(Bananas)--> Finish(),
Start() --Sells(SM, Bananas)--> Buy(Bananas),
Go(SM) --At(SM)--> Buy(Bananas),
Go(HWS) --At(HWS)--> Go(SM),
Go(HWS) --At(HWS)--> Buy(Drill),
Go(SM) --At(SM)--> Go(Home),
Start() --At(Home)--> Go(HWS),
Start() -- ¬At(HWS)--> Go(HWS),
Start() -- ¬At(SM)--> Go(SM),
Go(HWS) -- ¬At(Home)--> Go(Home)

Binding Constraints:
{
Equal:
g0 = Home = a4,
b1 = Drill,
s1 = HWS = g4 = a2,
b3 = Milk,
s3 = SM = g2 = s4 = a0,
b4 = Bananas,

Not Equal:
```

```
}


Ordering Constraints: (Start() < Finish()), (Start() < Go(Home)),
    (Go(Home) < Finish()), (Start() < Buy(Drill)), (Buy(Drill) <
   Finish()), (Start() < Buy(Milk)), (Buy(Milk) < Finish()), (
   Start() < Go(SM)), (Go(SM) < Finish()), (Go(SM) < Buy(Milk)),
   (Go(SM) < Go(Home)), (Buy(Milk) < Go(Home)), (Start() < Buy(
   Bananas)), (Buy(Bananas) < Finish()), (Go(SM) < Buy(Bananas)),
    (Buy(Bananas) < Go(Home)), (Start() < Go(HWS)), (Go(HWS) <
   Finish()), (Go(HWS) < Go(SM)), (Go(HWS) < Buy(Drill)), (Buy(
   Drill) < Go(SM)), (Go(HWS) < Go(Home))

RunTime: 00.260 seconds
```

Listing 3.2: Milk, Bananas, and Cordless Drill Problem Output of the Planner

Spare Tires Problem

The *Spare Tires problem* is a problem where the agent needs to change a flat tire with a spare tire. Now let's see the input to the planner in Listing 3.3:

```
PlanningProblem spareTires = new PlanningProblem(
operators: new HashSet<Operator> {
        new Operator("Remove",
            variables:      new []{"obj", "loc"},
            preconditions:  new List<Literal>{ new ("At", new []{
               "obj", "loc"}), new("Tire", new []{"obj"})},
            effects:        new List<Literal>{ new ("At", new []{
               "obj", "Ground"}), new("At", new []{"obj", "loc"},
                false)}
        ),
        new Operator("PutOn",
            variables:      new []{"t", "Axle"},
            preconditions:  new List<Literal>{ new ("At", new []{
               "t", "Ground"}), new("At", new []{"Flat", "Axle"},
                false), new("Tire", new []{"t"})},
            effects:        new List<Literal>{ new ("At", new []{
               "t", "Axle"}), new("At", new []{"t", "Ground"},
               false)}
        ),
            new Operator("LeaveOvernight",
            variables:      new string []{},
            preconditions:  new(),
            effects:        new List<Literal>{ new ("At", new []{
               "Spare", "Axle"}, false), new("At", new []{"Spare"
```

```
              , "Trunk"}, false), new("At", new []{"Spare", "
              Ground"}, false)
                   ,new("At", new []{"Flat", "Axle"}, false),
                      new("At", new []{"Flat", "Trunk"}, false),
                       new("At", new []{"Flat", "Ground"}, false
                      )}
        )
    },
initialState: new List<Literal> { new("At", new[] { "Flat", "Axle
   " }), new("At", new[] { "Spare", "Trunk" }), new("Tire", new[]
    { "Spare" }), new("Tire", new[] { "Flat" }) },
goalState: new List<Literal> { new("At", new[] { "Spare", "Axle"
   }), new("At", new[] { "Flat", "Ground" }) }
);
```

Listing 3.3: Spare Tires Problem Input to the Planner

The planner was able to find the optimal solution for the *Spare Tires problem* in runtime of 00.520 seconds. The output of the planner is shown in Listing 3.4:

```
Searching............

Plan  found:

Linearized Steps:
*************
Start() -> Remove(Spare, Trunk) -> Remove(Flat, Axle) -> PutOn(
   Spare, Axle) -> Finish()
************


Actions: Start(), Finish(), PutOn(Spare, Axle), Remove(Flat, Axle
   ), Remove(Spare, Trunk)

Links:
PutOn(Spare, Axle) --At(Spare, Axle)--> Finish(),
Start() --Tire(Spare)--> PutOn(Spare, Axle),
Remove(Flat, Axle) --At(Flat, Ground)--> Finish(),
Start() --Tire(Flat)--> Remove(Flat, Axle),
Start() --At(Flat, Axle)--> Remove(Flat, Axle),
Remove(Spare, Trunk) --At(Spare, Ground)--> PutOn(Spare, Axle),
Start() --Tire(Spare)--> Remove(Spare, Trunk),
Start() --At(Spare, Trunk)--> Remove(Spare, Trunk),
Remove(Flat, Axle) -- ¬At(Flat, Axle)--> PutOn(Spare, Axle)

Binding Constraints:
{
Equal:
```

```
aa1 = Ground,
a2 = Flat = r1,
p0 = Spare = r3,
pp0 = Axle = rr1,
rr3 = Trunk,

Not Equal:
}


Ordering Constraints: (Start() < Finish()), (Start() < PutOn(
   Spare, Axle)), (PutOn(Spare, Axle) < Finish()), (Start() <
   Remove(Flat, Axle)), (Remove(Flat, Axle) < Finish()), (Start()
    < Remove(Spare, Trunk)), (Remove(Spare, Trunk) < Finish()), (
   Remove(Spare, Trunk) < PutOn(Spare, Axle)), (Remove(Flat, Axle
   ) < PutOn(Spare, Axle))

RunTime: 00.520 seconds
```

Listing 3.4: Spare Tires Problem Output of the Planner

3.2 Virtual Reality Game

The POP algorithm was visualized using a VR game that was developed using the Unity game engine. The VR game was developed to allow the user to interact with the POP algorithm and try to be the planner himself, or just watch the algorithm looping through the actions and causal links to find the optimal plan. The VR game was developed to be used with the HTC Vive VR headset and the Vive controllers. In this section, we will discuss the design and implementation of the VR game, how every object of the algorithm was visualized, and how the user can interact with the game.

3.2.1 Plugins, Devices and Setup

The VR game was developed using the Unity game engine version 2020.3.20f1 LTS. The game was targeted to be used with the HTC Vive Pro VR headset and the Vive controllers. The Vive Pro was released in 2018 as an upgrade to the original HTC Vive headset, and has a resolution of 1440 x 1600 pixels per eye[3].

Figure 3.3: HTC Vive Pro [9]

The VR game was developed using the SteamVR plugin version 2.8.0, which is a plugin developed by Valve Corporation that allows the Unity game engine to interact with the HTC Vive VR headset and the Vive controllers. The SteamVR plugin provides the necessary scripts and prefabs to interact with the VR headset and the controllers. The plugin also provides the necessary scripts to interact with the VR camera rig, which is the main camera that is used to render the VR scene. SteamVR uses the OpenVR SDK, which is a software development kit developed by Valve Corporation that serves as the interface between the VR hardware and the game engine[4].

3.2.2 Visualization of the Elements

The VR game was developed to visualize the elements of the POP algorithm, which are the actions, the causal links, the binding constraints, the ordering constraints, and the whole plan as a Directed Acyclic Graph (DAG). The visualization of the elements was done using the Unity game engine, the SteamVR plugin and some external assets and scripts. The following list will discuss how each element was visualized in the VR game:

- **Actions/Operators:**
  The actions or the operators were visualized as children building blocks. Each action was represented as a building block with the name of the action written on the block, and the preconditions and effects of the action were written on the sides of the block. The *Start()* and *Finish()* actions were represented as a yellow block. New actions were represented as a green block, and existing actions were represented as a red block. When the user is creating the plan, he can create new actions, interact with them, and move them around in the scene.

Figure 3.4: Actions and Causal Links in VR

- **Causal Links:**
  The causal links were visualized as lines connecting the producer action to the consumer action. The links were represented as chains with an arrow-like object in the middle of the chain that had the link conditions written on it. A link sample is shown in Figure 3.4.

Figure 3.5: Agenda in VR

- **Agenda:**
  The agenda was visualized as an actual agenda that the user can interact with.

The agenda had the actions and the preconditions that need to be achieved. The user can interact with the agenda by selecting the action that he wants to achieve, and then it will get the list of possible achievers for this action, whether they are existing actions or new actions.

- **Ordering Constraints:**
The ordering constraints were visualized as lines connecting the actions with a blue arrow-like object in the middle of the line that had the ordering constraints written on it. An ordering constraint sample is shown in Figure 3.6.

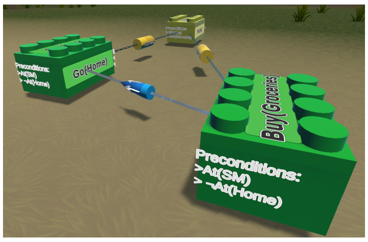

Figure 3.6: Ordering Constraints in VR

- **Threats:**
The threats were always being checked in the background, and if a threat was found, the user would be notified with an emergency sound and a red alarm[1] light placed on a computer screen in the scene. The user can then interact with the computer screen and a keyboard to see the threats and resolve them.

Figure 3.7: Threats Computer Desk

Figure 3.8: Threats Alert

- **Auto Graph Layout:**
The DAG of the plan was visualized using the *Force Directed Graph* algorithm. *Force Directed Graph* is an algorithm that is used to visualize graphs in a 2D or 3D space. The algorithm uses forces to simulate the physical behavior of the nodes and the edges of the graph. The algorithm was used to show the user the final plan

and the intermediate steps that the planner went through as a floating graph in the scene.

The implementation of the *Force Directed Graph* algorithm was mainly inspired by the implementation in the GitHub repository, *Force-Directed-Graph*, of the Ph.D. candidate, *Omar Addam*, that is under the MIT license[2]. The code was modified to fit the needs of the VR game and the POP algorithm, as the original implementation was visualized using a 2D view in Unity3D and was designed to use the mouse to interact with 2D image nodes. The modified implementation was designed to use the Vive controllers to interact with the 3D nodes in the VR game and to carry all needed data to synchronize the POP algorithm with the graph. A snapshot of the *Force Directed Graph* in the VR game is shown in Figure 3.9.

Figure 3.9: Floating Force Directed Graph (Spectator View)

- **Game Starting Environment:**
  The game starting environment was designed to look like an old temple with some ancient artifacts. The user starts the game in the temple, and then he can teleport to the different options in the game, where he can choose some properties of the planner before starting the game. The exact options that the user can choose are discussed in later sections. A top view of the game starting environment is shown in Figure 3.10.

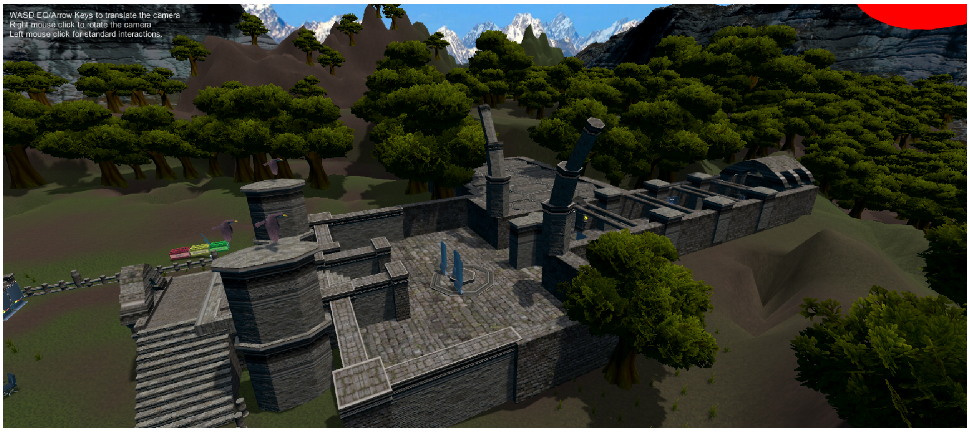

Figure 3.10: Game Starting Environment Top View

- **Main Game Environment:**
  The main game environment was designed to look like a grass open area with some trees and mountains in the background. The graph area, where the user can see or even construct the plan, was marked by yellow sand color. The environment has a computer desk with a screen that shows the threats, the agenda, two menus to delete links or ordering constraints, and other options. A top view of the main game environment is shown in Figure 3.11.

Figure 3.11: Main Game Environment Top View

3.2.3 User Interaction

The user can interact with the VR game using the HTC Vive controllers or the mouse and the keyboard in case of using a simulated VR environment. The main interactions that the user can do in the game are as follows:

- **Teleportation:**
  The user can teleport to different locations in the game by pressing the touchpad

of the Vive controller. The user can point anywhere, where teleport markers are put in the game, then leave the teleport button to teleport there. The teleportation was implemented using the SteamVR plugin. In the case of using the simulated VR environment, the user can either use the arrow keys to move around or use the button T to mimic the Vive controller teleportation.

- **Interacting with Buttons:**
  The user can interact with buttons in the game by approaching the button and pressing the trigger button of the Vive controller when the button is highlighted. In the case of using the simulated VR environment, the user can use the mouse to click on the buttons.

- **Interacting with Interactables:**
  The user can interact interactables in the game by approaching the interactable and pressing the trigger button of the Vive controller when the interactable is highlighted. If the interactable can be held or moved, the user can hold the trigger button and move the interactable around in the scene. Usually, there are limitations from the SteamVR plugin that the user cannot hold the interactable and teleport at the same time, so the user can simply hold the interactable with one controller and teleport with the other. In the case of using the simulated VR environment, the user can use the mouse to click on the interactables and drag them around. The right mouse button can be used to rotate the camera view.

3.2.4 Game Walkthrough

This section will discuss the walkthrough of the VR game, the options that the user can choose, how the user can interact with the game, and how the game can guide the user. This section will also discuss the different modes that the user can choose to interact with the graph.

User Path Guidance

When the user starts the game, he is guided to the temple environment, where he can choose some options before starting the game. The user is guided to the temple environment by a path of green arrows that are placed on the ground. The green arrows are placed in the direction that the user should follow to reach the next step in the game. The arrows guide are shown when the user presses the teleport button of the Vive controller, and they disappear when the user releases the teleport button. The main environment also has some green arrows that guide the user to the agenda, the threats and so on. There are also some yellow arrows that indicate an optional path.

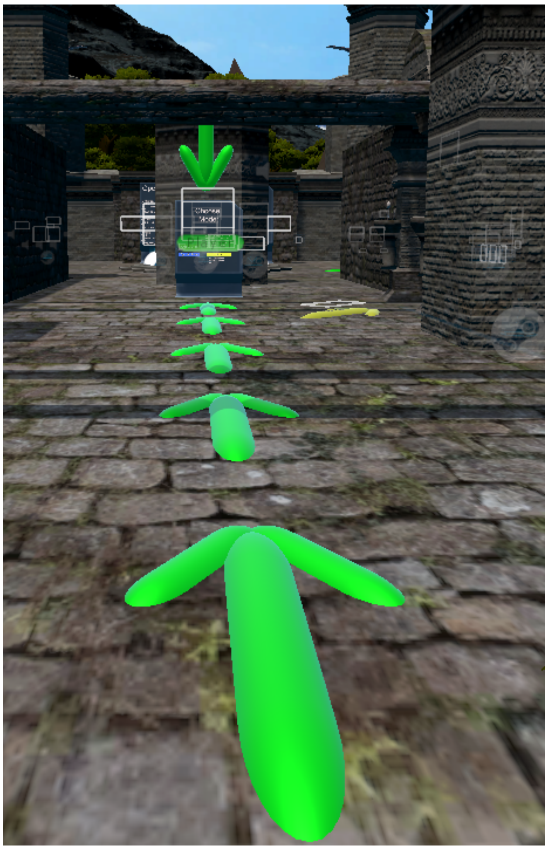

Figure 3.12: User Path Guidance in the Environment

Pre-Game Options

The user starts the game in the temple environment, where he can choose some options before starting the game. This stage is called the *Pre-Game Options*. The user can choose the following options:

Figure 3.13: Name Entering Menu in VR

- **Name Entering Menu:**
  The first menu that the user encounters is the *Name Entering Menu*, where the user can enter his name. The user can enter his name using the keyboard that is placed in front of him inside the scene. The user can then press the *Done* button to save his name.

Figure 3.14: Mode Selection Menu

- **Mode Selection Menu:**
  The second menu that the user encounters is the *Mode Selection Menu*, where the user can choose the mode that he wants to play in. The user can choose between the *Creative Mode* and the *Spectator Mode*.

  The *Creative Mode* allows the user to be the planner, where he can check the agenda, create new actions and delete them, create new causal links and delete them, and check the threats and resolve them. The user has full control over the planner in this mode, while given some guidance from the planner, helping with the bindings and suggesting achievers.

  The *Spectator Mode* allows the user to watch the planner working on the plan, applying achievers, resolving threats, backtracking from wrong decisions, and finally finding the optimal plan. The user can only watch the planner in this mode, and the only interactions that the user can do are starting or pausing the planner to check the new steps, and teleporting around the scene to see the plan from different angles.

- **Planning Problem Selection Menu:**
  The third menu that the user sees is the *Planning Problem Selection Menu*, where the user can choose the planning problem that he wants to solve or watch the

planner solving. The user can choose between the *Socks and Shoes problem*, the *Milk, Bananas, and Cordless Drill problem*, the *Groceries Buying problem* and the *Spare Tires problem*. When the user chooses the planning problem, the initial state, the goal state, and the operators of the problem with their preconditions and effects are shown to the user to the right and left of the *Planning Problem Menu*. The user can explore the problem and understand it before starting the planner.

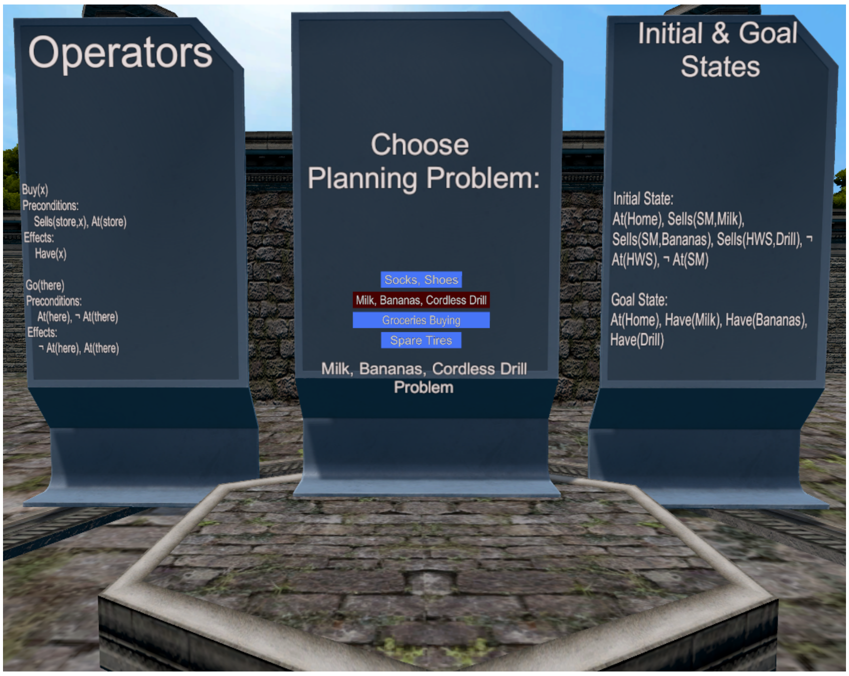

Figure 3.15: Planning Problem Selection Menu

- **Search Algorithm Selection Menu:**
  The fourth menu that the user sees **if he chooses the *Spectator Mode*** is the *Search Algorithm Selection Menu*, where the user can choose the search algorithm that he wants to watch the planner using it. The user can choose between *DFS*, *BFS* and *A* Search*. The user see a brief explanation of each search algorithm and how it works. The user can then choose the search algorithm that he wants to watch the planner using it. If the user chooses the *DFS*, another menu will appear and the user will be asked to enter the maximum depth of the search tree that the planner can go through. Entering the maximum depth is important to avoid infinite loops in the search tree. Moreover, the game will suggest a maximum depth based on the planning problem that the user chose.

Figure 3.16: Search Algorithm Selection Menu in VR

- **Ready to Start Menu:**
  The last menu that the user sees before starting the game is that he is asked if he is ready to start the planner. When the user confirms that he is ready, the game will start and the planner will start working on the plan, or initializes the environment for the user to start planning.

Spectator Game Environment

The user is instructed to go to the main game environment, where he can watch the planner working on the plan. The user can teleport around the scene to see the plan. There are two buttons used in the *Vive Controller* to interact with the planner:

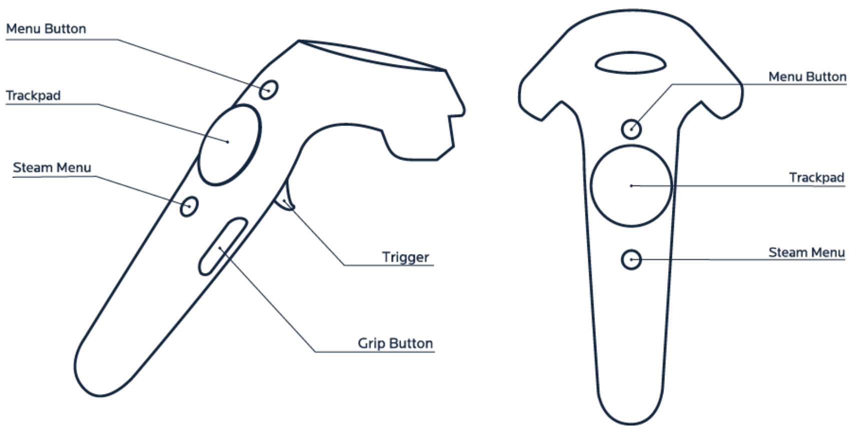

Figure 3.17: Vive Controller[5]

- **Menu Button:**
  The user can press the menu button of the Vive controller to let the planner show the next step of the plan. After the planner shows the next step, it will stop and wait for the user to press the menu button again to show the next step.

- **Grip Button:**
  The user can press the grip button of the Vive controller to shuffle the graph nodes. The reason why the user can shuffle the graph nodes is to avoid the nodes or edges from overlapping each other, as the *Force Directed Graph* algorithm may not always give the best layout of the graph. The user can press the grip button multiple times to shuffle the graph nodes until he gets a better layout of the graph.

Creative Game Environment

When the user chooses the *Creative Mode*, he is instructed to go to the main game environment, where he can start planning. The user can check the agenda, create new actions, throw actions in the trash and delete them, create new causal links, delete causal links and ordering constraints, check the threats, and resolve the threats. The user can also interact with the graph and move the nodes around to get a better layout of the graph. In this mode, the graph is not floating in the scene, but it is placed on the yellow sand area in the scene to be more easier for the user to interact with the graph. The following list will discuss the interactions that the user can do in the *Creative Mode*:

- **Agenda:**
  The user can check the agenda by approaching the agenda and then the agenda will automatically open. The user can then select the pair of action and preconditions that he wants to achieve, and then the planner will suggest the possible achievers for this action. The user can then choose between creating a new action or selecting an existing action to achieve the preconditions. The agenda will give the feedback to the user if he chose an existing action that cannot unify or bind the preconditions, or if he chose to link an action that will cause the graph to have a cycle. (See Figure 3.5)

- **Actions Spawn Point**
  If the user chooses to create a new action as an achiever for the preconditions, he can go to the *Actions Spawn Point* where the new action will fall from the sky. The user is then asked to take the action and place it in the graph for it to be linked automatically. The user can then move the action around in the scene, or throw it in the trash if he does not want to use it. (See Figure 3.18)

- **Trash:**
  If the user wants to delete an action, he can go to the *Trash* and throw the action in the trash. The action will then be deleted from the graph. If the user decides to delete an action that has causal links, the causal links will also be deleted. The

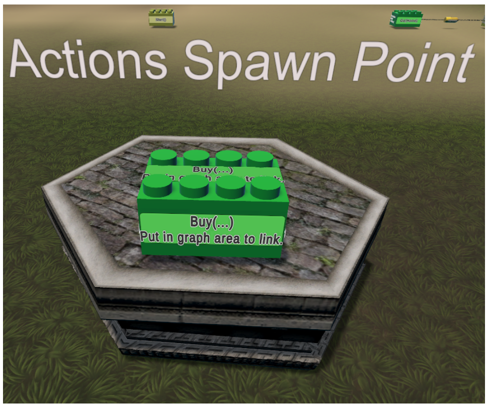

Figure 3.18: Actions Spawn Point

Figure 3.19: Actions Trash

Start() and *Finish()* actions will be respawned back in the scene if they are thrown in the trash, as they are essential for the planner to work. (See Figure 3.19)

- **Causal Links and Ordering Constraints Deletion:**
  The user can delete causal links and ordering constraints by going to any of the causal links or ordering constraints menus and selecting the link or constraint that he wants to delete. The user can then press the delete button to delete the link or constraint. (See Figure 3.20)

Figure 3.20: Causal Links and Ordering Constraints Menu

- **Threats Desk:**
  When the user is notified with a threat, he can go to the *Threats Desk* and interact with the computer screen to see the threats and resolve them. The user is notified with an emergency sound and a red alarm light placed on the computer desk. The user is asked to resolve the threats before continuing with the planner. All agenda

actions will be disabled until the threats are resolved. There are four ways to resolve the threats, the user can either delete the action that causes the threat, delete the causal link that causes the threat, promote the action that is threatening the link, or demote the action that is threatened by the link. The user can use the in-game keyboard to either press P to promote the action or D to demote the action. (See Figures 3.7 and 3.8)

Chapter 4

Evaluation Results

Following the system's implementation, we needed to evaluate how visualizing the Partial Order Planning algorithm in a VR environment enhances its learning effectiveness. This chapter presents the results of the evaluation process. We first describe the sample population and the evaluation process. We then present the results of the evaluation, which include the participants' feedback and the results of their surveys.

4.1 Participants

We recruited 20 participants for the evaluation process. The participants were students from the Media Engineering and Technology (MET) program at the German University in Cairo (GUC). All participants had a Computer Science (CS) background, and were in their third, fourth, or fifth year of study. The ages of the participants ranged from 21 to 25 years old. An important selection criterion was that the participants had no prior knowledge of the Partial Order Planning (POP) algorithm. This criterion was essential to ensure that the participants' feedback was not biased by their prior knowledge of the algorithm or their practice and experience with it.

4.2 Evaluation Process

The evaluation process was divided into two main parts: an explanation session of the POP algorithm and a test session. Since the participants had no prior knowledge of the POP algorithm, the explanation session was essential to introduce them to the algorithm and its concepts. The test session was then conducted to evaluate the participants' understanding of the algorithm and their ability to solve problems using it. Finally, the participants were asked to fill out a survey to provide feedback on their experience with the VR environment and the POP algorithm.

4.2.1 Explanation Session

The explanation session was conducted in a classroom setting, where the participants were introduced to the POP algorithm. The session was divided into two parts: a quick theoretical explanation of the algorithm and a theoretical example of the algorithm that is different from the one used in the VR environment, but with a similar level of complexity.

4.2.2 Test Session

The test session was conducted in the VR environment, where the participants were asked to solve a problem using the POP algorithm. The participants were given a brief introduction to the VR environment and the controls used to interact with it. The planning problem that the participants were asked to solve was a simplified version of the *Milk, Bananas, and Cordless Drill* problem. The problem is called the *Groceries Buying* problem, and it involves buying buying groceries from a store and returning home, while keeping in mind the constraints and avoiding the threats. Since the explanation session was a quick explanation to the algorithm, the participants were allowed to ask questions and request clarifications during the test session.

4.2.3 Survey

After the test session, the participants were asked to fill out a survey to provide feedback on their experience. The survey included questions about the participants' understanding of the POP algorithm, their experience with the VR environment in the test session, and a System Usability Scale (SUS) questionnaire to evaluate the usability of the system. The survey was conducted using Google Forms to be easily accessible to the participants.

System Usability Scale (SUS)

The SUS is a widely used questionnaire for evaluating the usability of a system. The questionnaire consists of 10 questions, each with a 5-point Likert scale. The scores of the odd-numbered questions are subtracted by 1, and the scores of the even-numbered questions are subtracted from 5. The scores are then summed up and multiplied by 2.5 to get the final SUS score, which ranges from 0 to 100. The SUS score is used to evaluate the usability of the system, with higher scores indicating better usability. An average SUS score is considered to be 68[11]. The SUS questionnaire is shown in Listing 4.1.

```
1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical
   person to be able to use this system.
```

```
    5. I found the various functions in this system were well
       integrated.
    6. I thought there was too much inconsistency in this system.
    7. I would imagine that most people would learn to use this
       system very quickly.
    8. I found the system very cumbersome to use.
    9. I felt very confident using the system.
    10. I needed to learn a lot of things before I could get
       going with this system.
```

Listing 4.1: System Usability Scale (SUS) Questionnaire

Feedback Questions

The survey included the following questions to gather feedback from the participants:

1. I think the VR game helped me to understand the POP Algorithm more (than theoretically). *(10-point Likert scale)*
2. I felt the VR game made the algorithm more interesting or engaging. *(5-point Likert scale)*
3. I would recommend the game as a learning tool to others. *(5-point Likert scale)*

4.3 Results

The results of the evaluation process are presented in this section. The results include the participants' feedback on the POP algorithm and the VR environment, as well as the results of the SUS questionnaire.

4.3.1 Participants' Feedback

The participants' feedback on the POP algorithm and the VR environment was collected through the survey. The participants were asked to rate their understanding of the POP algorithm, the effectiveness of the VR environment in enhancing their understanding of the algorithm, and the engagement level of the VR environment. The participants were also asked if they would recommend the game as a learning tool to others. The results of the feedback questions that were discussed in section 4.2.3 are shown in Figure 4.1.

The results show that the majority of the participants found the VR environment helpful in understanding the POP algorithm. More than 90% of the participants answered 7 or above on the 10-point Likert scale question. The participants also found the VR environment engaging, with 95% of the participants answering 4 or above on the 5-point

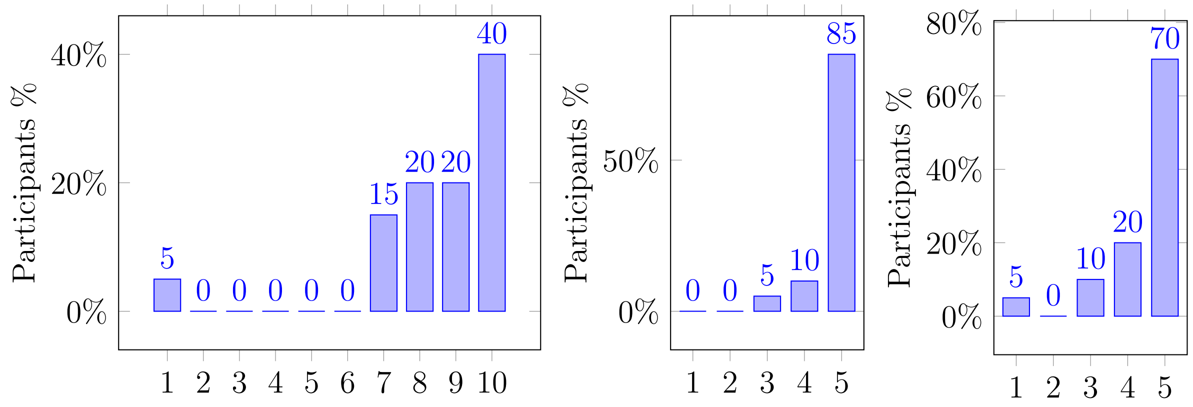

Figure 4.1: Participants' response on the POP Algorithm and the VR Environment Feedback Questions 1, 2, and 3 respectively

Likert scale question. Finally, 90% of the participants said they would recommend the game as a learning tool to others. These results indicate that the VR environment was effective in enhancing the participants' understanding of the POP algorithm and engaging them in the learning process. Moreover, in the open-ended feedback section, the participants expressed their satisfaction with the VR environment and the learning experience, and provided suggestions for improvement, which will be considered and added in the future work section.

4.3.2 System Usability Scale (SUS) Results

The results of the SUS questionnaire are shown in Figure 4.2. The average SUS score was 79.125, which is higher than the average SUS score of 68. This indicates that the VR environment had good usability, and the participants found it easy to use and engaging. The standard deviation of the SUS score was 11.363, which indicates that the participants' opinions were not significantly high or low, and the variance was 129.128, which indicates that the participants' opinions were not significantly different from each other.

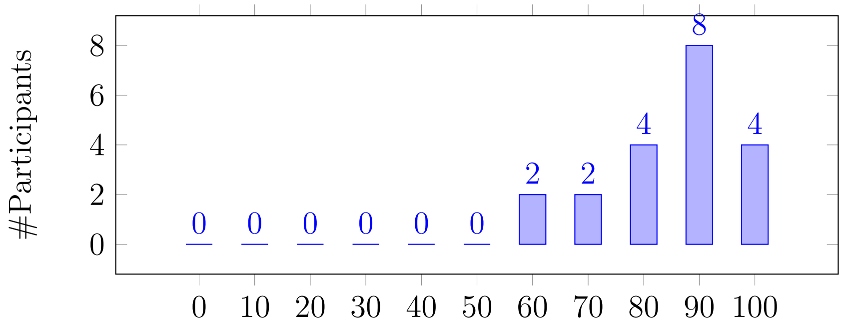

Figure 4.2: Participants' response on the System Usability Scale (SUS) Questionnaire

Chapter 5

Conclusion

Conclusion

Chapter 6

Future Work

Text

Appendix

Appendix A

Lists

POP	Partial Order Planning
AI	Artificial Intelligence
MGU	Most General Unifier
DAG	Directed Acyclic Graph
A*	A-Star
DFS	Depth First Search
BFS	Breadth First Search
DLS	Depth Limited Search
VR	Virtual Reality
LTS	Long Term Support
GUC	German University in Cairo
MET	Media Engineering and Technology
CS	Computer Science
SUS	System Usability Scale

List of Figures

Bibliography

[1] 5CNG5. Alarm light. https://sketchfab.com/3d-models/alarm-light-0e009c47066d4ab4b4329fdd4820f141. ”Alarm Light” (https://skfb.ly/oE9CF) by 5CNG5 is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).

[2] Omar Addam. Force directed graph. https://github.com/omar-addam/Force-Directed-Graph.

[3] Multiple Contributors. Htc vive. https://en.wikipedia.org/wiki/HTC_Vive.

[4] Multiple Contributors. Openvr. https://en.wikipedia.org/wiki/OpenVR.

[5] Multiple Contributors. Vive controller rotation axis to skeletal mesh head bone. https://forums.unrealengine.com/t/vive-controller-rotation-axis-to-skeletal-mesh-head-bone/466537.

[6] Multiple Contributors. A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm, 2024. Accessed: 8 May 2024.

[7] Multiple Contributors. Disjoint-set data structure. https://en.wikipedia.org/wiki/Disjoint-set_data_structure, 2024. Accessed: 16 May 2024.

[8] Geeks for Geeks. Disjoint set data structures. https://www.geeksforgeeks.org/disjoint-set-data-structures/, 2023. Accessed: 16 May 2024.

[9] HTC. Vive pro full kit. https://www.vive.com/uk/product/vive-pro-full-kit/.

[10] Haythem Ismail. Introduction to artificial intelligence course lecture slides, 2023. Lecture slides provided at the German University in Cairo.

[11] PhD Jeff Sauro. Sus: A quick and dirty usability scale. https://measuringu.com/sus/, 2011.

[12] Microsoft. Priorityqueue¡t¿ class. https://github.com/dotnet/runtime/blob/5535e31a712343a63f5d7d796cd874e563e5ac14/src/libraries/System.Collections/src/System/Collections/Generic/PriorityQueue.cs, 2022.

[13] Microsoft. Priorityqueue class. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.priorityqueue-2?view=net-8.0, 2024.

[14] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 9, pages 326–328. Prentice Hall Press, 2009. Section 9.2.2, Figure 9.1.

[16] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[17] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*, chapter 11. Pearson, 2020.