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

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

---

Mohamed Ayman Tammaa
19 May, 2024

Acknowledgments

Text

Abstract

Abstact

Contents

Acknowledgments **V**

1 Introduction **1**
1.1 Section Name . . . 1
1.2 Another Section . . . 1

2 Background **3**
2.1 Key Definitions and Concepts . . . 3
2.1.1 Planning . . . 3
2.2 Partial Order Planning . . . 8
2.2.1 POP Algorithm . . . 9
2.3 Unity 3D Engine . . . 11
2.3.1 Virtual Reality (VR) in Unity . . . 12
2.3.2 Visualization of the Partial Order Planning (POP) Algorithm in Unity . . . 12

3 Design and Implementation **13**
3.1 Introduction . . . 13
3.2 Design of the Algorithm . . . 13
3.2.1 Nondeterministic Achievers & Threat Search . . . 13
3.2.2 Graph Search Algorithms . . . 14
3.2.3 Unification Algorithm . . . 15

4 Conclusion **17**

5 Future Work **19**

Appendix **20**

A Lists **21**
List of Abbreviations . . . 21
List of Figures . . . 22

References **23**

Chapter 1

Introduction

1.1 Section Name

Some sample text with an Acronym Without Citation (AC), some citation [1], and some more Acronym With Citation [4] (AC2).

1.2 Another Section

Reference to Section 1.1, and reuse of AC nad AC2 with also full use of Acronym With Citation [4] (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.

2.1.1 Planning

Planning is the process of formulating a sequence of actions and strategies to achieve a specific goal. It is a fundamental aspect of AI that allows agents to make decisions and take actions based on the current state of the environment.

- **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**: 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**: 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**: 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 & strict order to achieve a specific goal. It does not give you the freedom or choices while executing the actions. For example, Buy(Ticket) → Go(Airport) → Board(Plane) → FlyWith(Airplane) → Land(Airplane) → Call(Taxi) → Go(Home) → Sleep(Bed). This is a total-order plan that specifies a strict order to achieve *Sleeping in Bed*. (See Figure 2.1)

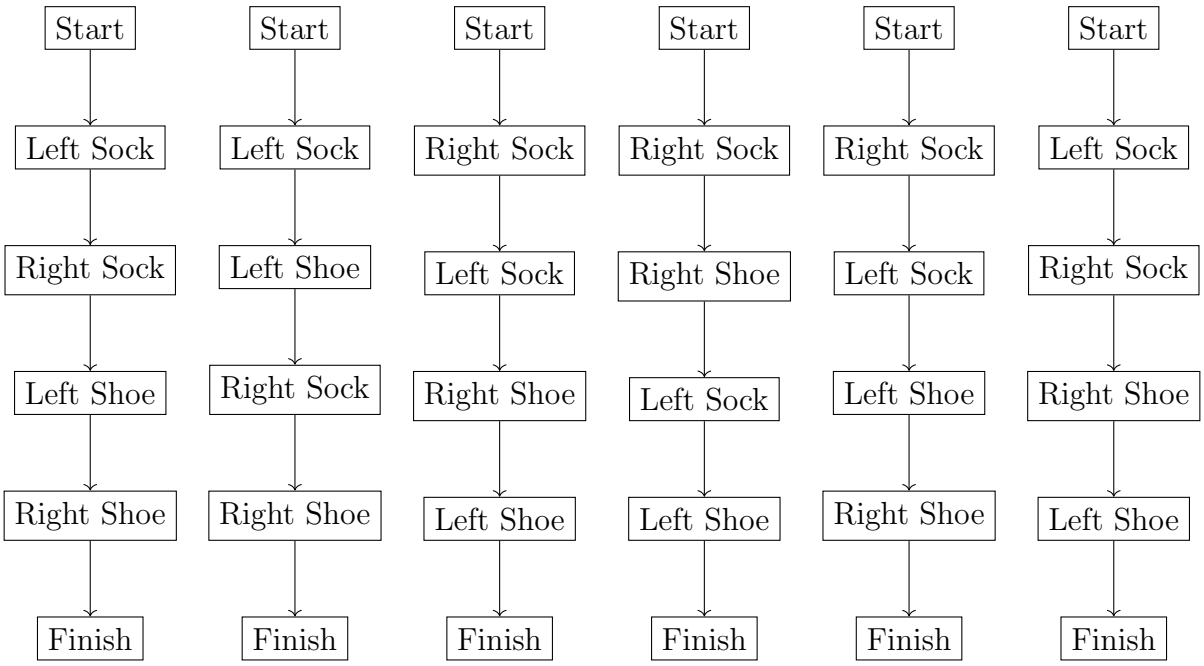

Figure 2.1: Socks & Shoes Total Order Plan solutions.
(Each one of them can be considered a linearzation of the partial order plan in Figure 2.2)

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

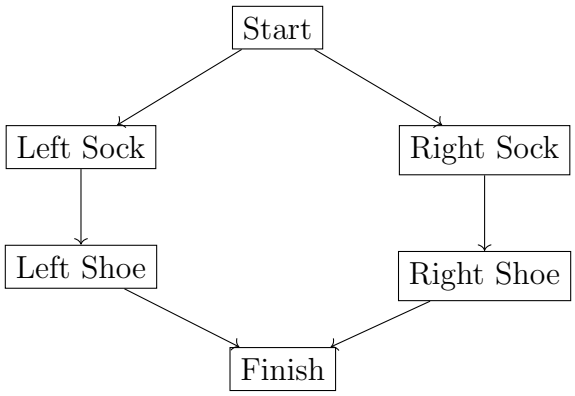

Figure 2.2: Socks & Shoes Partial Order Plan solution.

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**: 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**: 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**: 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**: 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**: 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 $p_j \in \text{effects}(a_i)$ and $p_j \in \text{preconds}(a_j)$. 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**: 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**: 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 $P = (O, s_0, g)$ [2] 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 ($\prec$)**: 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.

- **Formal Definition of Partial Plans**: A partial order plan (2.1.1) is a tuple $\pi = (A, \prec, B, L)$ where: [2]

  - 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: [2]

  - 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**: 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: [2]

  - $e_k (\in \text{effect of } a_k)$ 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]

2.2 Partial Order Planning

Partial Order Planning (POP) is a plan-space search algorithm (See plan-space planning definition in 2.1.1) 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 (See least commitment strategy definition in 2.1.1). 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 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 3.2.1). 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

3.1 Introduction

In this chapter, we will discuss the design and implementation of the POP algorithm. We will start by discussing the design of the algorithm, and then we will discuss the implementation of the algorithm in C#.

3.2 Design of the Algorithm

3.2.1 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.

3.2.2 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.

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. 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.

3.2.3 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. 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 [3]. The algorithm is described as follows:

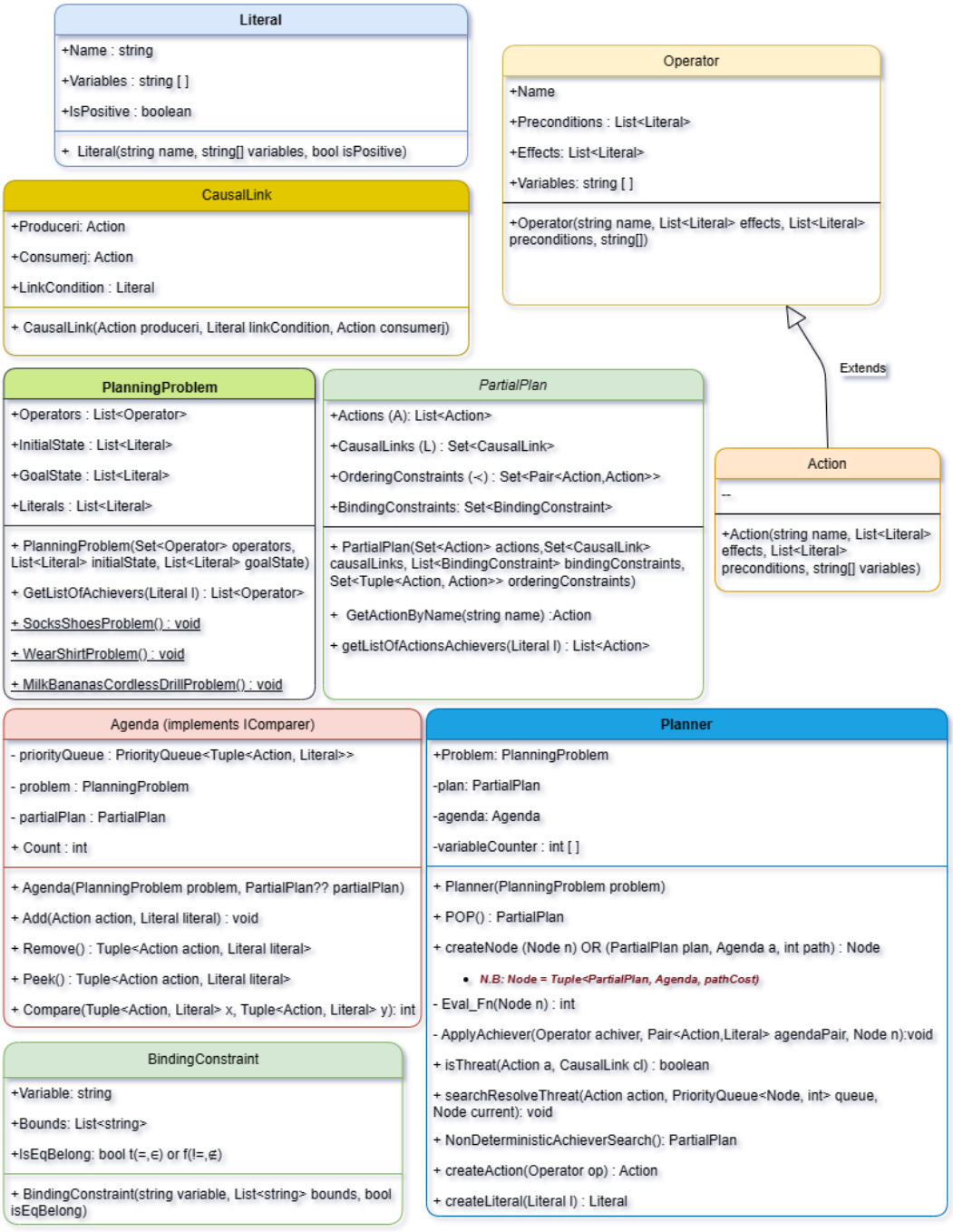

Figure 3.1: Class Diagram of the POP Algorithm

Chapter 4

Conclusion

Conclusion

Chapter 5

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
AC	Acronym Without Citation
AC2	Acronym With Citation [4]

List of Figures

2.1 Socks & Shoes Total Order Plan solutions. (Each one of them can be considered a linearzation of the partial order plan in Figure 2.2) 5

2.2 Socks & Shoes Partial Order Plan solution. 6

3.1 Class Diagram of the POP Algorithm 16

Bibliography

[1] W.G. Campbell. *Form and style in thesis writing.* Houghton Mifflin, 1954.

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

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*, chapter 9, pages 326–328. Pearson, 2020. Section 9.2.2, Figure 9.1.

[4] S. Wenkang. An analysis of the current state of English majors' BA thesis writing [J]. *Foreign Language World*, 3, 2004.