Creation of an Interactive Blocksworld Tool

Elia Hänggi

2. August 2023

Abstract

Contents

Chapter 1

Introduction

This thesis is about the creation of an interactive tool showcasing techniques of classical planning. The tool shows the Blocksworld domain in a 3D environment. The goal of the tool is to serve as a introduction into planning, state spaces and heuristics.
Blocksworld is a famous planning domain. It consists of a set of blocks laying on a table. These blocks can have different colors or labels to distinguish them from each other. Blocks can be used to form stacks. To do this, a block can be picked up and placed either on another block or on the table. However, only one Block can be moved at a time. Blocksworld starts with a given initial block structure. The goal is to convert the initial structure to the goal state with the moves mentioned.
In the tool we use a slightly modified version of Blocksworld. Instead of laying on a table, the blocks can be placed on 4 predefined stacks. Blocks can thus only be moved back and forth between these 4 stacks. The creation of new stacks is not possible. Another modification to the standard Blocksworld is an additional height constraint. Each stack has a predefined height that must not be exceeded. This means there is a maximum number of blocks that can be in a stack.

The created tool can be seen as an extension of an already existing project. For the Fantasy Basel 2019, a demo video was created with an environment containing the Blocksworld domain. In the video a harbor is used to visualize the domain. In the center of this port is a docked cargo ship. In this case, blocks are represented as containers with different colours and labels. These can be stacked on three different stacks on the cargo ship. To move the containers between the stacks, a crane which is located on land can be used. In addition, there is another stack at the port. This is for unloading containers and changing the order of containers on the ship. Thus, the Blocksworld variant in the video corresponds to the one described earlier with 4 fixed stacks. In the video, the containers are moved by the crane until a certain stack structure is reached. The goal of the video was to visualize an application of classical planning for people who are not familiar with it.
The procedure in the video will now be expanded to an interactive tool. This will be done while using the same environment as in the video. The user should be able to control the crane and thus move containers by himself. The resulting state space should be

displayed to the user as a graph. Another goal is the integration of the Fast Downward Planner (Helmert, 2006). The user should have the possibility to ask the planner for the solution. In addition, it should be possible to receive further information about the plan as well as about the current state space in the form of heuristics or explored states. This should demonstrate how a planner behaves and what methods are used to find a plan.
The interactive tool will be generated using the Unreal Engine. Unreal Engine is a 3D graphic engine published by Epic Games. Therefore, the implementation will be written in C++, or in the visual scripting system "Blueprints". The tool should work on Windows as well as on Unix operating systems.

This thesis starts with a short presentation of the Unreal Engine. Furthermore, there is a introduction into classical planning where basic definitions and notations are introduced. Additionally, the Fast Downward Planner and its functionalities will be explained.
In the main part, Blocksworld is defined in the Planning Domain Definition Language (Aeronautiques et al., 1998). Furthermore, the solutions to the domain for different algorithms are investigated. Subsequently, the implementation of the tool itself is discussed. Finally, all the techniques the tool uses to show how the planner functions are analyzed.

Chapter 2

Background

2.1 Unreal Engine

Unreal Engine is a game engine developed by Epic Games. Epic Games (2023) It was published in 1998 and is widely used for creating video games and other 3D applications. The engine is very versatile, it runs on Windows, Unix and MacOS operating systems. In order for the engine to be used across platforms, many modules in the engine have their own platform-dependent implementation.
The engine is written in C++. Therefore, the default programming language utilized to create content is C++ as well. However, Unreal Engine includes an additional scripting system called "Blueprints". The Blueprint system in Unreal Engine is a visual scripting tool to create game logic and behavior by connecting nodes in a graph-like interface. It is designed so that no deep programming knowledge is required to use Blueprints. C++ implementation can be used in Blueprints and vice versa. While C++ represents the low-level programming approach, Blueprints are much more abstract.
Unreal Engine works exclusively object-oriented. Each class inherits from the default object class UObject. However, there are many already predefined classes that derive from UObject. A famous example is the Actor class. Actors are objects that can be spawned on a map.
The engine also has its own implementations of data structures. This includes custom array or string structures. These offer additional functionality for manipulation compared to the C++ standard implementation. These structures also ensure that C++ and Blueprints are consistent and compatible with each other. Thus, as a rule, structures from outside the engine (e.g. C++ standard library) should be avoided.

2.2 Planning Theory

This sections introduces state spaces, search algorithms for these spaces and classical planning. This is necessary to understand the functionality of the Fast Downward planner. Besides, this is the theory that the tool is supposed to convey.

2.2.1 State Spaces

State spaces are a way of defining an environment. These spaces contain different states and actions that can be executed. The goal in a state spaces is to find a path from an initial state to a goal state. All actions and states are predefined in the definition of a state space.

Definition 1 (State Space)**.** *A state space $\mathcal{S}$ is defined as a tuple $\mathcal{S} = \langle S, A, cost, T, s_I, S_* \rangle$ with*

- *a finite set of States S,*
- *a finite set of Actions A,*
- *action costs $cost : A \to \mathbf{R}_0^+$,*
- *a transition relation $T \subseteq S \times A \times S$,*
- *an initial state $s_I \in S$, and*
- *a set of goal states $S_* \subseteq S$.*

As we can see there can be multiple goal states but only one initial state. Additionally it holds that every transition T must be **deterministic**. This means that for transitions $\langle s, a, s' \rangle \in T$ it is forbidden to have $\langle s, a, s_1 \rangle$ and $\langle s, a, s_2 \rangle$ with states $s_1 \neq s_2$.
The solution for a state space $\mathcal{S}$ is a sequence of actions. Such a sequence is also called a solution path. We can sum up the cost of the actions in the sequence. This gives us the total cost to get from start to goal. An **optimal** solution has the minimal costs of all solutions that exists.

2.2.2 Search Algorithms

Search algorithms try to find solutions in a given state space using various methods. Essential algorithms, which are used later in the planner are elaborated here.
The most common way to do a state space search is to start at the initial state. Then actions are applied one after the other and new states are created. The search ends when a Goal state is found. Finding an optimal solution for every problem cannot be done efficiently. Thus, optimal state space search belongs to the **NP-hard** problems. However, there are algorithms that compute good solutions efficiently.
In order to apply search algorithms on state spaces, methods to represent the space are needed. There are two main ways how state space search can be represented: tree search and graph search.

Tree Search

In tree search, the structure of a tree is used to search for goal states. The root of the tree is the Initial Node. Now the tree can be explored from the root. Each path of the tree represents a single sequence of actions. It is possible that the same state occurs

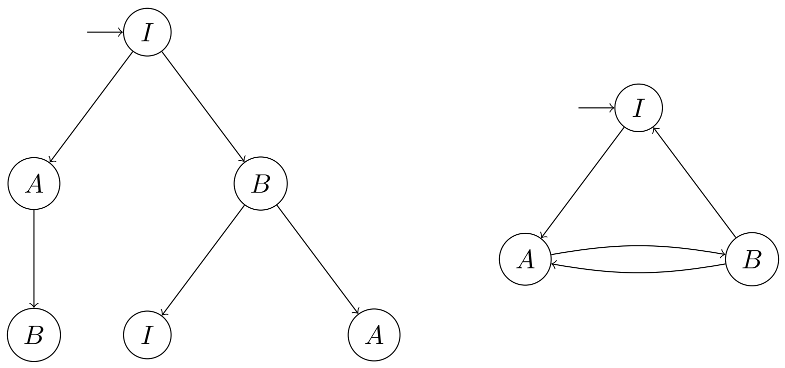

Figure 2.1: Example of the same search state represented with tree search (left) and graph search (right). Nodes labeled with the same letter should indicate equal states. I is the initial node.

several times in the tree. This is the case if there are different paths to the same state. Since each state can occur more than once, there is no limit to the size of a tree.
The only thing that needs to be stored in tree search is the so-called **open list**. This list contains all leaves of the tree. This corresponds to all states whose successors have not been explored yet. When a state in the list is explored, its successors are added to the open list and the original state is deleted.
Usually, in addition to the current state, other information is stored in a tree node, for example the current costs or the parent node in the tree. Therefore, strictly speaking, the tree does not consist of states but of nodes that contain information about the state.

Graph Search

Graph search is the other well-known method of state space search. The search is started again at the initial state. Unlike tree search, each node represents exactly one state. Since there cannot be more nodes than states, there is a maximum size for the graph. This is one of the main advantages of graph search.
Besides the open list, graph search also has a **closed list**. This list contains all nodes that have already been explored. The closed list ensures that each state occurs only once in the graph. Because of the additional list graph search needs a higher memory capacity than tree search.

Heuristics

Both tree and graph search look for solutions by exploring nodes one by one. However, the order in which this is done depends on the used algoritms. To have an efficient algorithm, states that are close to the target should be explored first. Exactly this is implemented with the help of heuristics.
A heuristic is a function that assigns a number to a state. This number is supposed to estimate approximately how far away the state is from the goal. The estimate can be

used to determine the order of state exploration.
In state space search, heuristics are individually designed for each problem. Rules of thumb or other observations are usually used to try to make a good estimate. This can be problematic since for each problem a new heuristic must be constructed. Furthermore, there may be little or no information available about a problem. Accordingly, it can be difficult to make an accurate evaluation of the states.

A* search

Heuristics can be used to explore suitable states early on to reach the goal faster. If the states are simply selected according to the lowest heuristic, the algorithm is called Greedy-best-first search. This strategy is used to find solutions as quickly as possible. Most of the time, however, the cost of the solutions are suboptimal.
Instead, it makes sense to not only look at the heuristic, but also at the cost of the current path. This will disadvantage paths that are close to the goal but very expensive. The result is that discovered solutions have lower costs. However, this can make the search take a longer time.
The most well-known algorithm that uses both costs and heuristics is A*. Here, the costs up to the current state are added with the heuristic. The state with the smallest value is explored.
Both Greedy-best-first search and A*, are typically implemented in a graph search representation. However, there are implementation that use tree search. If the memory capacity of the algorithm should be small, tree search would be the optimal representation.

2.2.3 Classical Planning

There are many parallels between state space search and classical planning. In planning, however, an additional level is abstracted compared to state space search. Solution methods are now independent of the problem itself, no problem specific knowledge is needed to perform planning. Thus, planning can be applied equally to any state space. Search algorithms such as A* are still used to achieve this. In planning, however, heuristics are completely independent of the state space. Thus, solutions can be found efficiently without having information about the space.

Planning can be used to find solutions for very large state spaces. However, the current definition of state spaces (definiton 1) are not well defined for large spaces. Each state and action of the space must be specified individually. In addition, the associated transitions must also be listed. For large state spaces it makes sense to introduce a new way of describing problems in general.

Planning Domain Definition Language

Definition of problems with state variables. Short introduction into PDDL. Laying foundations to later define Blocksworld in PDDl.

Planning Heuristics

Introduction into Planning Heuristics. Only essential ideas should be shown here such that section does not get too long.

2.3 Fast Downward Planner

Description of the fast downward planner. Show how the planner is called and what the output looks like. Part should be similarly structured as Unreal Engine part.

Chapter 3

Main Part

3.1 Blocksworld implementation

Show implementation of Blocksworld in PDDL. How do different planning algorithms behave in domain. Analyze solution costs of different heuristics. (Reference to Search algorithms in background)

3.2 Implementation in Unreal Engine

Discussion of the code and its structure in the Engine (concentrate on interesting implementation). Use diagrams to prevent description of code. Part should not be too long.

3.3 Visualization of Planning Techniques

Should be essential part of thesis. Analyze what methods are used to show planning techniques and how planner behaviour is demonstrated. Which representation methods are used and why.

Chapter 4

Conclusion

Discuss if goal was reached (Reference to Introduction).
Analyze how tool can be further expanded and developed

Bibliography

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

Epic Games (2023). Unreal engine.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.