Creation of an Interactive Blocksworld Application showcasing Planning Techniques

Elia Hänggi

2. August 2023

Abstract

Fast Downward is a classical planner using heuristical search. It is using many advanced planning techniques that are not easy to teach, since they usually rely on complex data structures. This interactive application introduces mentioned techniques to the user with an illustrative example: The Blocksworld planning domain.
A special form of the blocks world was designed for the application, which allows a simple in game representation of a state space. It is implemented in the Unreal Engine and provides an interface to the Fast Downward planner. Users can explore a state space themselves or have Fast Downward generate plans for them. Heuristics as well as the state space are explained and made accessible to the user. The user experiences how the planner explores a state space with the information available. Furthermore, different visualization methods of search algorithms and state spaces are discussed.

Contents

Chapter 1

Introduction

This thesis is about the creation of an interactive application showcasing techniques of classical planning. The application shows the Blocksworld domain in a 3D environment. It uses the Fast Downward planner to discover solutions in the domain. The goal is to explain how the planner behaves by introducing heuristics and state spaces to the user.
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. Normally, Blocksworld talks about a hand that can hold or drop blocks. However, only one Block can be held 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 our application 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 interactive 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, 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. There was a fixed camera position which let the user monitor the whole process.
The procedure in the video will now be expanded to an interactive application. 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. Additionally, the application should have an interface to the Fast Downward planner (Helmert, 2006). The user should have the possibility to ask the planner for specific information about the state space. This could be that the Planner shows certain states that are interesting candidates to explore further. The planner could also make the discovered solution available to the user. Another goal is that the planner does not function as black box but its behaviour is introduced to the user. This can be done by explaining the heuristic used or displaying the created state space to the user.
The application will be generated using the Unreal Engine. Unreal Engine is a 3D engine published by Epic Games (Epic Games, 2023). Since the Engine uses C++ as programming language, this will be the default language used. However, Unreal Engine does provide the scripting system called "Blueprints". Some functionality can also be implemented in Blueprints. Target operating systems are Windows and Unix.

This thesis starts with a section introducing the Unreal Engine and important concepts of planning theory. This lays all the foundations that need to be known for other parts. With the Unreal Engine, various features and conventions are elaborated. These will later play an integral role in the design of the implementation. Furthermore, there is an introduction into the theory of state spaces and classical planning where basic definitions and notations are introduced. Those definitions play a key part in the integration of the Fast Downward planner into the application. The exact usage and functionality of the planner is stated as well.
Afterwards, Blocksworld is defined in the Planning Domain Definition Language (Aeronautiques et al., 1998). This is needed to use the planner for this specific problem domain. Furthermore, we define the search algorithm that will be used. For this purpose, a heuristic is designed that has the desired properties for the application. Subsequently, the implementation of the tool itself is discussed. Important design decisions in the engine are explained. Finally, we analyze all planning concepts introduced in the application. We justify the choice of specific visualization methods and compare them with other visualizations.

Chapter 2

Background

This chapter introduces various topics that serve as background in further work. There is a short introduction of the Unreal Engine. This serves to be able to discuss later about the implementations in the engine. Different mathematical notations are introduced with which Blocksworld can be described. Furthermore the Fast Downward planner and its usage is introduced. This is important as we later want to pass our Blocksworld domain to the planner.

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 and classical planning. All subjects are necessary to understand the functionality of the Fast Downward planner. Besides, these theoretical concepts should later be communicated to the user.

2.2.1 State Spaces

State spaces are a way of defining an environment. State 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, Russell, 2010)**.** *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 $T \subseteq S \times A \times S$, 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.
A state space search is usually started at the initial state. Then actions are applied one after the other and new states are explored. The search ends when a goal state is found. There are various attributes for search algorithms that are advantageous. One such attribute is that if a solution exists, the algorithm should also return a solution. An additional property is that algorithms should terminate if no solutions exist. In the following, we are particularly interested in algorithms where both properties hold. Furthermore, it would be desirable if the solution found by the algorithm would be optimal.
In order to apply search algorithms on state spaces, methods to represent the state 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. Then we put all neighbours of the initial node in the **open list**. This list should contain all possible candidates that might be explored in the next iteration. From there on, we start exploring nodes in the open list. All neighbours of the explored node are again added to the list. We can call this procedure an **expansion**. Each path of the tree represents a single sequence of actions. However, this means that there can be cases where we look at the same state twice. This is exactly the case if two different action sequences lead to the same state. Even worse would be if the state space contains a loop. This is the case if a state $a \in S$ is reachable from a state $b \in S$ and vice versa. In this case, tree search could explore the corresponding nodes infinitely many times.
The goal of a search algorithm is usually not only to find a solution but to output more information. This might be the solution path or the cost of this path. For this, the node needs to consist of a special data structure. Besides the state itself, the parent node and the action used to get to the state should therefore be saved in this data structure.

Graph Search

Graph search is another well-known state space search method. The basic structure of graph search is the same as tree search. The search is started again at the initial state. However, another list is used in addition to the open list. This list is called the **closed list**. The closed list consists of all nodes that already were explored. This ensures that the same state is never explored twice. Therefore if one node was expanded, this node is added to the close list. The result is that each node in the graph corresponds to one unique state. All edges represent one unique transition of the state space. Since there are a finite number of states, it is ensured that unlike tree search, graph search terminates. All in all, the built graph with graph search is much more compact than the tree with tree search (Figure 2.1). However, the distance from a node to the initial node can be determined more intuitively with tree search. This is because there are never multiple paths between initial and all other nodes.
The fact that in graph search, each state is only visited once has a positive effect on the time complexity. There are much fewer expansions. However, graph search has the disadvantage that the closed list must constantly be maintained. Thus, the memory usage of graph search is higher than that of tree search. However, in most cases it is more viable to prefer graph search. This is especially true for state spaces, which often contain loops.

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 algorithms. 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

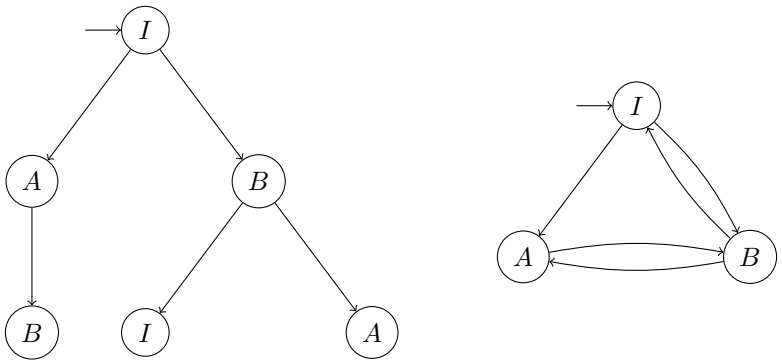

Figure 2.1: Example of the same state space represented with tree search (left) and graph search (right). Nodes labeled with the same letter should indicate equal states. I is the initial node.

estimate approximately how far away the state is from the goal. The estimate can be used to determine the order of state exploration.

Definition 2 (Heuristic)**.** *A heuristic h for a state space S is defined as function*

$$h : S \to R_0^+ \cup \{\infty\}$$

In addition, we define the heuristic h^* as the function that assigns the optimal solution cost to each state. In this sense, h^* is the perfect heuristic. Furthermore we define that the perfect heuristic assigns the value infinity to states from which the goal cannot be reached.
There are several properties that are desirable for heuristics, which are now presented. The first property is **admissibility**. For an admissible heuristic h it holds for every state s that $h(s) \leq h^*(s)$. In other words, the distance to the goal is never overestimated in an admissible heuristic. Another key property is **consistency**. A heuristic h is consistent if $h(s) \leq cost(a) + h(s')$ for all transitions $\langle s, a, s' \rangle \in T$. For a heuristic it is advantageous if both properties hold. We can use these properties later to show that discovered solutions are optimal for certain heuristic with a suitable search algorithm. There are many other heuristic attributes that are not covered here.
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. States in the open list can be ordered with respect to their heuristic value. This ensures that good states are explored first. Usually, the open list is ordered according to an evaluation function f. This function, like heuristics, assigns every state a numerical value.

Different algorithms calculate f in different ways. If the states are simply selected according to the lowest heuristic, the algorithm is called **Greedy-best-first search**. In this case, it holds that $f(s) = h(s)$ for a state s. This strategy is used to find solutions very quickly. However, this algorithm does not take into account the costs to get to the states with low heuristics. This can result in solutions with high costs.
Instead, it makes sense to not only look at the heuristic, but also at the cost of the current path. We call the sum of the costs on the current path g. Taking g into account to calculate the value f disadvantages 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 g and heuristics h in the evaluation function is A* (Hart et al., 1968). A* is a search algorithm using the open list evaluation function $f(s) = g(s) + h(s)$ for a state s, where g denotes the cost on the current path and h a heuristic. An example of an implementation of A* is given in Algorithm 1. However, the same algorithm can be used for arbitrary search algorithms that use an evaluation function f. Greedy-best-first search for example can also be implemented with this algorithm. The only difference is the ordering of the open list.
The quality of the solutions found by A* highly depend on the heuristic used to calcu-

Algorithm 1 A* algorithm

```
1:  open := new list ordered by f
2:  if h(init)) < ∞ then
3:      open.insert(root)
4:  closed := new list
5:  while not open.isEmpty() do
6:      n := open.pop()
7:      if n ∉ closed then
8:          closed.insert(n)
9:          if n is goal then
10:             return path
11:         for n' ∈ succ(n) do
12:             if h(s') < ∞ then
13:                 open.insert(n')
14: return unsolvable
```

late f. It can be proven that A* discovers only optimal solutions if a suitable heuristic is used.

Theorem 1. *Solutions discovered by A* are optimal if the used heuristic h is admissible and consistent.(Hart et al., 1968)*

Both Greedy-best-first search and A*, are typically implemented in a graph search representation, using an additional closed list to detect duplicates. However, there are implementation that use tree search. If the memory capacity of the algorithm should be small, tree search would be the more viable representation.

2.2.3 Classical Planning

Planning is a special field of state space search. In planning, an additional level is abstracted compared to state space search. Solution methods are 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* can still be 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 state space.

Planning can be used to find solutions for very large state spaces. However, the current definition of state spaces (Definition 1) is 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

Planning Domain Definition Language (PDDL) is a standardized language to describe planning tasks or state spaces in general (Aeronautiques et al., 1998). PDDL uses the concept of **state variables** to define states. State variables are binary variables that can be assigned true or false. A state consists of an assignment of each state variable to either true or false. With this definition of states, it is possible to represent 2^n different states with n state variables. This allows the representation of much bigger states, without specifically list all states.
PDDL can be separated into two parts, the **domain** and the **problem**. The domain contains all information that hold for all problem instances. **Predicates** of the problem are listed there. Predicates can be seen as types of state variables that can take objects from the problem file as parameters. Furthermore, **actions** are specified in the domain. Actions consist of preconditions and effects. Preconditions state which conditions have to be satisfied for the action to be applicable. Effects are conditions that hold after applying the action. In general, actions have the same functionality as transitions in the classical state space definition (definition 1). The notation with preconditions and effects allows us to define transitions without listing all of them. In addition, different type of objects can be specified in the domain.
The problem file contains all instance specific information. **Objects** are defined here together with their associated type. Combined with the predicates in the domain file, objects represent state variables. In the problem file, **initial state** and **goal state** are specified. The initial state consists of an assignment of state variables. The goal state consists of state variables that must hold for the goal to be reached.
The solution of a PDDL problem instance usually consists of a sequence of actions used to get from the initial state to the goal state.

2.3 Fast Downward Planner

The Fast Downward planner is a planning system based on heuristical search (Helmert, 2006). It was published in 2006. The planner supports many different planning heuristics as well as search algorithms. This includes Greedy-best-first search as well as A*. Fast Downward uses Python as well as C++ as programming language. To start the planner, the python file *fast-downward.py* needs to be called. Additionally, the domain file is specified followed by the problem file. The –search flag can be used to set the search algorithm and heuristic to be used. At first, the planner translates the problem into a different problem representation. From there, the search starts. Finally, the planner returns a plan containing the sequence of actions leading to the goal state. The solution also includes the total cost of the solution. With another additional flag it is possible to specify the destination folder of the solution.

Chapter 3

Blocksworld Implementation

This chapter is about the definition of the Blocksworld variant used in the game. The state space is defined at first with the classical state space definition (Definition 1) where all states are listed in a set. Furthermore, the implementation of Blocksworld in PDDL is discussed. In addition, we define a heuristic specifically for Blocksworld that is used in the application.

3.1 State Space Definition

In the application, we use a special variant of Blocksworld. There are in total 4 stacks, three on the ship and one on land. Stacks on the ships can have a maximal height of 4 while the land stack can be up to three containers high. There are in total 9 distinguishable containers in the scene. Additionally, we do not only consider different stacks as states but also if a container is attached to the crane. In other words, the container can be either removed from a stack, or put on a stack. With this information we can define the state space in the scene according to definition 1:
$\mathcal{S} = \langle S, A, cost, T, s_I, S_* \rangle$ with

- $S = \{\langle L_1, ..., L_m, A_1, ..., A_n, B_1, ..., B_n, C_1, ..., C_n, D\rangle | L_n, A_n, B_n, C_n, D \in \{-1, 0, ..., 8\} \wedge m = 3 \wedge n = 4\}$,
- $A = \{unstack, stack\}$,
- $cost(unstack) = cost(stack) = 1$,
- $\{\langle S, unstack, S'\rangle \in T|$
  $S = S'$ except for elements $X_k \in S, X'_k \in S', S_D, S'_D$ where $X_k = S'_D \wedge$
  $X'_k = S_D = -1 \wedge X_{k+1}, X_{k+2}, ... = -1$ for $X \in \{L, A, B, C\}\}$

  $\{\langle S, stack, S'\rangle \in T|$
  $S = S'$ except for elements $X_k \in S, X'_k \in S', S_D, S'_D$ where $X'_k = S_D \wedge$
  $X_k = S'_D = -1 \wedge X'_{k+1}, X'_{k+2}, ... = -1$ for $X \in \{L, A, B, C\}\}$,

- $s_I = (-1, -1, -1, 7, 3, 0, -1, 6, 1, 2, -1, 4, 5, 8, -1, -1)$, and
- $S_* = \{(-1, -1, -1, 2, 1, 0, -1, 5, 4, 3, -1, 8, 7, 6, -1, -1)\}$.

To define a state we use a tuple containing all 4 stacks concatenated plus the block that is currently held. The tuple contains the ids of the blocks from 0 to 8. An empty slot in a stack is indicated by using the id -1.
The given initial state contains all block ids once. Theoretically, it would be possible that the same id is present twice in one tuple. However, the transition as we defined them do only change the order in the given tuple, no new numbers are added in a transition. In the initial state, every id is present once. Therefore, all states using the same id multiple times are unreachable from the initial state. In this definition, we used the default initial state while creating a new scene. There is the possibility for the user to change the initial state by saving the current progress. In this case, the s_I would change to the one saved by the user.
As it can be seen, it is very difficult to define a large and complex state space with this type of definition. For this reason, the transitions were rather described by words to increase the understandability of the definition. However, we can also describe a state space in PDDL format using state variables. This way we can create a definition in a more compact way.

3.2 PDDL Implementation

In this section we discuss the implementation of Blocksworld in PDDL. As stated earlier, PDDL can be separated into a domain and a problem. The domain is always the same while the problem changes for different instances. We first state the implementation of the PDDL domain (Listing 3.1).

```
(define (domain BLOCKS)
(:requirements :strips)
(:types block height)

(:predicates (on ?x ?y - block)
(clear ?x - block)
(handempty)
(holding ?x - block)
(on-height ?x - block ?y - height)
(SUCC ?hx ?hy - height))

(:action stack
:parameters (?x ?y - block ?hx ?hy - height)
:precondition (and (holding ?x) (clear ?y)
(on-height ?y ?hy) (SUCC ?hy ?hx))
:effect
(and (not (holding ?x))
(not (clear ?y))
(clear ?x)
(handempty)
(on ?x ?y)
(on-height ?x ?hx)))

```

```
 (:action unstack
:parameters (?x ?y - block ?hx - height)
:precondition (and (on ?x ?y) (clear ?x)
(handempty) (on-height ?x ?hx))
:effect
(and (holding ?x)
(clear ?y)
(not (clear ?x))
(not (handempty))
(not (on ?x ?y))
(not (on-height ?x ?hx)))))
```

Listing 3.1: Blocksworld PDDL Domain

There are in total 6 predicates in the Blocksworld domain. Predicates *on* and *clear* set the stack structure. *on* states that a block is positioned on another block. *clear* marks if the block is the top of a stack. *handempty* specifies if the hand is currently empty and *holding* states which block is currently held. Those 4 predicates are also present in the standard Blocksworld problem.
The last two predicates *on-height* and *SUCC* are newly added. These predicates enforce the height constraints for each stack. The used PDDL version does not allow the use of numerical values. Therefore we cannot just define heights as integers. Instead we regard heights as discrete objects. The *on-height* predicate assigns a block the corresponding height object. *SUCC* defines which heights follow each other. According to this definition, the height limit is reached if there exists no height object for the container to be placed.
A difficulty we have to address in PDDL is that we want to have a fixed number of 4 stacks. This is another difference to the standard Blocksworld problem where it is allowed to freely place blocks and therefore creating new stacks. We ensured the fixed stack size with the definition of the stack action. The stack as well as the unstack action require two block objects: The block that is moved and the block underneath. With this definition, it is not possible to create new stacks since there is no block underneath in this case. However, this would mean that unstacking the lowest block is not possible. Since there is no block underneath, unstack would not be an applicable action. We can address this problem in the PDDL problem file (Listing 3.2).

```
(define (problem BLOCKS)
(:domain BLOCKS)
(:objects stackland stack2 stack1 stack3 c0 c1 c2 c3 c4 c5 c6 c7 c8 -
    block h0 h1 h2 h3 n0 n1 n2 n3 n4 - height)

(:INIT (handempty) (SUCC h0 h1) (SUCC h1 h2) (SUCC h2 h3) (SUCC n0 n1)
    (SUCC n1 n2) (SUCC n2 n3) (SUCC n3 n4)(on-height stackland h0)
    (on-height stack1 h0) (on-height stack2 h0) (on-height stack3 h0)
    (clear c0) (on c0 c3) (on-height c0 n3) (on c3 c7) (on-height c3 n2
    ) (on c7 stack1) (on-height c7 n1)
    (clear c2) (on c2 c1) (on-height c2 n3) (on c1 c6) (on-height c1 h2
    ) (on c6 stack2) (on-height c6 h1)
    (clear c8) (on c8 c5) (on-height c8 h3) (on c5 c4) (on-height c5 h2
    ) (on c5 stack3) (on-height c4 h1)
    (clear stackland))

(:goal (and (on c0 c1) (on c1 c2) (on c2 stack1)
```

```
    (on c3 c4) (on c4 c5) (on c5 stack2)
    (on c6 c7) (on c7 c8) (on c8 stack3) (clear stackland))))
```

Listing 3.2: Blocksworld PDDL Problem

As we can see, we defined several objects for this problem including 9 blocks which all represent containers. Additionally, we did introduce 4 other blocks. These blocks represent the base of each existing stack. Since our actions do not allow the movement of the lowest blocks, these 4 blocks are stationary. All other blocks can now be moved on top of those stationary bases. Therefore, we have our 4 fixed stacks (Figure 3.1).
Furthermore, we defined different height objects that allow us to enforce the height constraint. Objects starting with the letter *h* represent the height on the land while objects starting with *n* correspond to the height on the ship. This distinction allows us to define different heights for land and ship. As we can see, on the land the maximal height is 3 and the maximal ship height is 4 (excluding stationary blocks).
Like mentioned before, the problem file can vary depending in which state the scene is. However, the object definition stays the same for every instance. This is because it is not possible to alter the amount of containers or to change the height constraints during the scene.
For the problem file, we used the same initial state like the previously stated initial state in the classical state space definition. A sketch of the defined initial state is displayed in Figure 3.1.
We can now use both the domain and the problem file as input in the Fast Downward

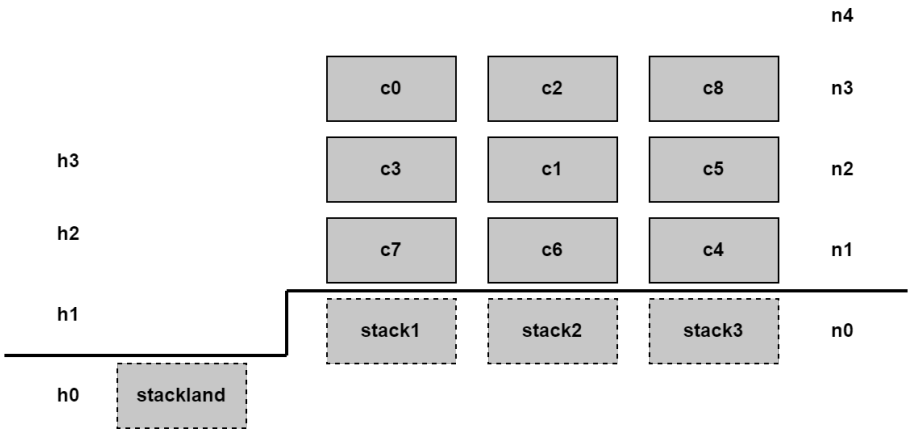

Figure 3.1: Sketch of the defined initial state in Listing 3.2. Blocks with dashed lines symbolize stationary base blocks. Assigned height objects are marked on the sides of land and ship stacks.

planner. The output is a file stating the sequence of actions as well as the total cost to reach the goal. Since we did not provide any cost for the stack and unstack action, a default cost of 1 for each action is assumed.

3.3 Blocksworld Heuristic

In order to start planning with those PDDL files, we need to indicate a search algorithm as well as a heuristic. Those can be appended to the command calling the planner. Both Greedy-best-first search and A* we can consider as viable search algorithms. Although Greedy-best-first search might give us fast results, we would like to find optimal plans. It should not be possible that the user might discover better solutions than the planner. Therefore, A* is the most reasonable choice as search algorithm. However, for A* to be optimal, an admissible and consistent heuristic is needed. Additionally, the heuristic should be a good estimation for the real goal distance as this would reduce the planning time. Last but not least, the calculation of the heuristic should be easy to understand such that people not familiar with planning can comprehend it.
We will define a heuristic on our own trying to design it with the attributes mentioned. Since this heuristic is created specifically for our application, we will address blocks as containers. Furthermore, we do not talk about a hand that is moving containers but a crane. The self designed heuristic for this task is stated in Definition 3:

Definition 3 (Blocksworld heuristic)**.** *We define a heuristic $h(s)$ for a Blocksworld state s with*

$$h(s) = \sum_{c \in c^*} 2 + d$$

where c^* is the set of all wrongly placed containers in s and

$$d = \begin{cases} 1 \textit{ if a container is attached to the crane} \\ 0 \textit{ else.} \end{cases}$$

We additionally define that for all goal states it holds that no container is attached to the crane. Since we already have defined a goal state with no attached container that cannot be altered, this is always true.
In the heuristic definition we sum over all wrongly placed containers. A container counts as wrongly placed if and only if it is not placed at its goal location, or there is a container underneath that is wrongly placed. In other words, if one container is not at its goal location, the container itself and all above are in $c*$. We assume that for each wrongly placed container, 2 actions are needed to place it correctly. For the container attached to the crane, at least one action is needed to place it correctly.
Now we can find out which properties apply for this heuristic.

Theorem 2. *The heuristic $h(s)$ (Definition 3) is admissible.*

Proof. We prove that h is an admissible heuristic. A heuristic is admissible iff $h(s) \leq h^*(s)$ for all states s. We previously defined that $cost(a) = 1$ for all actions a. We need to prove the following two properties to show that h is admissible:

- At least 1 action is needed to place a container attached to the crane at its goal location, and

- at least 2 actions are needed to place a container in c^* at its goal location.

If both properties hold, it is ensured that h never overestimates the distance to the goal, which means $h(s) \leq h^*(s)$ for all states s. The first property is trivially true. Since a container attached to the crane never is at its goal location, we at least need one stack action to place it correctly.
For the second property, it holds that the goal location for a container must be on a stack. Thus, for a container in c^*, the container has to be unstacked and stacked at least once to be correctly placed. At least 2 actions are needed.
Both properties are true, this implies that $h(s) \leq h^*(s)$ holds for all states s. h is admissible. □

Theorem 3. *The heuristic $h(s)$ (Definition 3) is consistent.*

Proof. We want to show that h is consistent. For all transitions $\langle s, a, s' \rangle \in T$ of a consistent heuristic, it applies that $h(s) \leq cost(a) + h(s')$. We know that $cost(a) = 1$ for all actions a. To prove consistency we make a case distinction between both actions:

Case 1: $a = unstack$:
If a is an unstack action, it means that at first the crane is empty and after the action a container is attached. If this container is $\in c^*$ at the beginning, it holds that the cardinality of c^* decreases by 1, however the heuristic increases by 1 because of the non-empty crane. We can say for the new heuristic $h(s') = h(s) - 2 + 1 = h(s) - 1$.
If the container is $\notin c^*$, this implies that $h(s') = h(s) + 1$.
For *Case 1* we can conclude that $h(s) \leq h(s') + 1$.

Case 2: $a = stack$:
In this case it holds that the crane is not empty before the action and is empty after the action. If the container is stacked correctly, the cardinality of c^* stays the same. This implies that $h(s') = h(s) - 1$. Otherwise, the cardinality of c^* increases by 1. Then it holds that $h(s') = h(s) + 2 - 1 = h(s) + 1$.
This means for *Case 2*, $h(s) \leq h(s') + 1$.

Since the condition $h(s) \leq h(s') + 1$ holds for both cases, we can conclude that h is consistent. □

We now have proven that the heuristic stated in Definition 3 is admissible and consistent. Together with Theorem 1, this implies that if we use A* as search algorithm and the heuristic in Definition 3, discovered solutions are optimal.
Another important fact is that this heuristic can be implemented efficiently. Incorrectly placed containers can be counted by simply going through the stacks from bottom to top. Thus, the run time of the heuristic calculation is linear to the number of containers. A possible implementation is given in Algorithm 2.

Algorithm 2 Heuristic Implementation

```
1: h := 0
2: State := list containing all stacks in state.      ▷ Stacks ordered from bottom to top
3: GoalState := list containing all goal stacks.      ▷ Same number of stacks as State
4: while not State.isEmpty() do
5:     Stack := State.pop()
6:     GoalStack := GoalState.pop()
7:     IsStackWrong := false
8:     while not Stack.isEmpty() do
9:         Container := Stack.pop()
10:        if GoalStack.IsEmpty() then
11:            h = h + 2
12:            continue
13:        GoalContainer := GoalState.pop()
14:        if IsStackWrong or Container != GoalContainer then
15:            IsStackWrong = true
16:            h = h + 2
17: if State.craneNotEmpty() then
18:     h = h + 1
19: return h
```

Chapter 4

Implementation in Unreal Engine

This chapter discusses the implementation in the Unreal Engine. Key design decisions are presented. For example, the division between C++ and Blueprints is described. Furthermore, the integration of the Planner into the application is reported.

4.1 Division in Blueprints and C++

Blueprints is the visual scripting system that allows coding on at a higher abstraction level. This abstraction level does allow programming with less coding experience. Concepts like pointers or garbage collection are hidden from the user in Blueprints.
All in all, it can be said that code written in C++ is clearer and allows programming on a deeper level. Therefore as rule of thumb, default implementations are in C++. Another design goal is that frequent changes between languages were avoided. Blueprint and C++ components should be as decoupled from each other as possible. This makes a clear project structure possible, since Blueprints and C++ classes are stored at different places. However, there are cases where using Blueprints is a good or even better alternative to C++. For this reason code is partially written in Blueprints.
An application where Blueprints are easy to handle is the creation of graphical user interfaces. The Blueprint system does have a specific user interface object called user widget. Those objects allow the design of custom user interfaces with a specific editor. The creation of graphical user interfaces is thus massively simplified and also easier to modify compared to a text-based design editor. All graphical user interfaces were therefore written in Blueprints. Inputs of the user are registered and handled in user widgets. From there, those inputs are handled in Blueprints, and afterwards forwarded to the corresponding C++ classes. There exists a class specifically for connecting C++ implementations with user interfaces (Figure 3.2). This class is called **UserInputManager**. In this structure, Blueprints can be seen as the frontend where everything exposed to the user is managed. C++ can be seen as the backend where all other calculations where made.

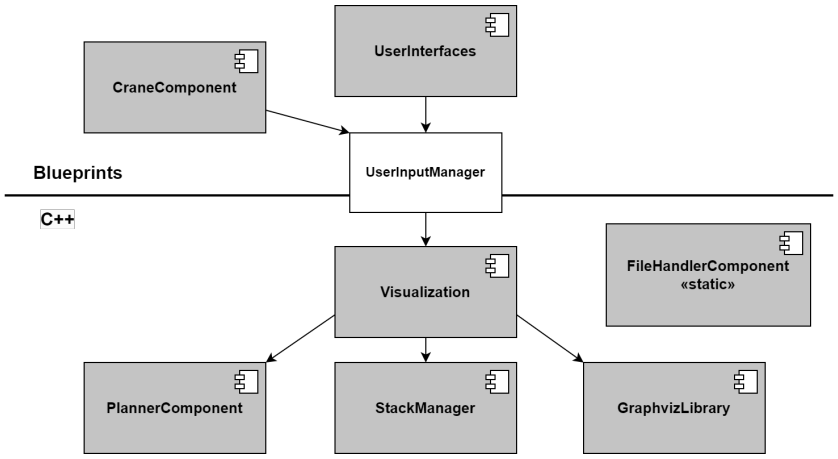

Figure 4.1: Diagram showing interactions of components in the project. The separating line indicates the division in C++ and Blueprints.

The C++ components contain data structures to manage states and plans. Furthermore, the planner component is also written in C++.

Another component that only consists of Blueprints is the crane component. This has to do with the history of the project. As it was stated before, this implementation is built on top of an existing project. The goal of this project was to create a video that shows different crane moves. It was used to represent different states of a state space. The project contained the whole map where the scene takes place including the harbor, crane and containers. Another preexisting part is the component responsible for moving the crane and containers. This whole component was written in Blueprints only. Apart from a few changes, all existing functionality could be embedded in the newly created application. To keep the dependency between Blueprints and C++ low, extensions of this component were also written in Blueprints. One such extension is that the user can manually move containers.
Another possibility would have been to translate the entire crane component into C++. The biggest advantage of this would have been that calculations of the crane position or movement would all take place in C++. Especially arithmetic calculations can be represented better in text based form. However, this would be associated with a certain effort. In addition there are some auxiliary functions in Blueprints, which facilitate the representation of movements (in this case of the crane). For future extensions, it may be helpful to translate this part into C++. The user interfaces would then be the last remaining Blueprints component. This would further decouple the two languages, as all the game logic would then be contained in the C++ backend.

4.2 Integration of the Planner

In the application, the user can call the Fast Downward planner which returns an executable plan for the current state. In order to use Fast Downward, it needs to be downloaded externally. To connect the planner, the user has to enter the paths to the installed python interpreter as well as to the **fast-downward.py** file. Both strings are stored in a configuration file in yaml format. Python and Fast Downward are additional dependencies which allow the usage of the planner in the application. Otherwise, calling the planner automatically fails.
When the planner is called, the first thing that is done is the creation of the PDDL domain and problem file. The domain file is a fixed string that is written into a temporary directory. The current state is translated into PDDL and written afterwards together with the remaining problem file into the same directory. Then the planner is called. We use A* as search algorithm and our constructed heuristic (definition 3) as search parameters. A* is already part of the planner. The heuristic had to be added manually to the planner by extending the source code. The output of the planner is again saved in the temporary directory. From there, the plan is read, parsed and made accessible to the user.
There are two possible errors that can occur while planning. The first one is a time limit exceeded error. The planner component does have an integrated timer which measures the total planning time. The maximal time is set to 90 seconds. After those 90 seconds, the planning is stopped and reset. This does not happen in the vast majority of cases. The time limit is only exceeded in the absolute worst case. The worst case is that the distance from initial state to goal state is near to the maximum. The second error occurs if the arguments in the planning command are incorrect. This is exactly the case if the specified data in the configuration file is missing or wrong.
From an implementation perspective, the planner component was designed to work completely independently of Blocksworld. Thus, the component can be used for any planning domain. The component must be connected to an actor for this purpose. The actor has to set the domain and problem file correctly when calling the component. When the planning is finished, different events are triggered. These events can be managed in the actor. Thus, for example, it is possible to react differently to errors during planning.

Chapter 5

Visualization of Planning Techniques

The goal of the application is to introduce the planner and its functionality to the user. Therefore, different techniques that are used in classical planning are presented in the application. However, no previous knowledge should be necessary to be able to use the application. For this reason, the concept of states and state spaces must be explained.
The application can be divided into two different modes: The tutorial and the load mode. The tutorial is used to show and explain the functions available in the application to the user. This initially refers only to game controls. Afterwards, some theoretical backgrounds for planning and state spaces are explained. All functionalities that exist in the game are gradually presented to the user. However, not all game functions are available during the time in the tutorial. This allows the user to get to know the game little by little and not to be overwhelmed by all functions at once.
The load mode provides all functions to the user from start to end. The one and only goal is to reach the goal state. It is possible to save the current state and to play on later. When saving, the current state can be given a name. This state is stored externally, where the name acts as an ID. When starting the mode, either a new game state can be created or a saved game state can be selected. In this case the saved state corresponds to the new initial state. The default initial state was chosen in such a way that solving the problem is a challenge, but the planning duration is not too high. It is also possible to create very difficult initial states where the planning time is accordingly high. Overall, different levels of difficulty can be defined with the load functionality.
During the game, additional information about the state is displayed to the user. This information is shown on a clipboard. The user carries the a virtual clipboard with him. It appears on the screen at the push of a button and can also be hidden again with the same button. The clipboard consists of 3 different pages that contain different representations. There is a state page, an action page and a graph page. These pages are introduced in the tutorial in this exact order. The following planning techniques are introduced in the application: the heuristic, the state space and the search in this state space. The state page and the graph page of the clipboard are particularly relevant for

this. The visualization of those techniques is discussed in the following.

5.1 Visualization of the Heuristic

The heuristic is visualized both on the state and the graph page. On the state page, the concept of a heuristic is shown where on the graph page, the usage of the heuristic can be seen. We already did define the heuristic used in the application (Definition 3). However, this heuristic is problem specific. This means that assumptions were made for the heuristic that only apply to this problem domain. We know in Blocksworld that if a block is placed on a wrongly placed block, this block has to be moved in any case. We used this knowledge in our heuristic to get more accurate estimates of the distance to the goal. In other words we did expose the nature of a stack to improve the search. This he heuristic actually does not fall into the category of planning heuristics.
The vast majority of good planning heuristics are based on a complicated calculation procedures. Most of them do not use the actual state space. Instead, the actions of a state space are slightly adjusted, which should simplify the calculation of the heuristic (e.g. delete relaxation). Besides, there are often some calculation steps needed until the heuristic value is found. This makes explaining how the value is obtained very difficult. Additionally it must be considered that the concept of a heuristic is unknown to the user at the beginning. Therefore, explaining heuristics with an easier example is advantageous. Our heuristic is defined in a way that no special data structures must be used to obtain the heuristic. The value of the heuristic can be understood by simply comparing the current state with the goal state.
The state page contains everything that is needed to understand the calculation of the heuristic. It shows the current state as well as the goal state. Furthermore, the current heuristic value is displayed. In the current state, different colors indicate which containers are correctly placed and which are not. Containers with red edges are wrong place. This corresponds to an increase of the heuristic by 2. A container attached to the crane is marked yellow. This increases the heuristic by 1 (Definition 3). Correctly placed containers are indicated in green. In addition to the colored container borders, the page contains an extra information button. If the user hovers over this button, the meaning of the colors are explained. All in all, the state page can be used for two purposes: as an overview over the current and goal state, and as an introduction into the heuristic. The heuristic is used again on the graph page where the state space is visualized.

5.2 Visualization of the State Space

The state space is visualized on the graph page of the clipboard. The graph is the last page that the tutorial deals with. Therefore the concept of a heuristic and how it is calculated was already introduced to the user. There are two possibilities how a state space can be represented: tree search and graph search. For this application, graph search was selected as visualization method. Graph search allows a more compact and clearer representation. Graph search also is the more intuitive way to explain a

state space since each state corresponds to exactly one node. In a tree search graph, many duplicates of the same state would coexist at different places. This is becoming increasingly difficult to display in large state spaces.
The idea of the graph is that the user steadily explores the state space by applying actions. In the graph each node corresponds to a state the user has generated and each edge to a transition the user has applied. The initial state is indicated by an arrow pointing at the initial node. Additionally, all successors of the current graph nodes are displayed as well. These successors are the equivalent to the open list.
Since the space where the graph can be displayed is rather small, it is not possible to directly represent the state in the node. Instead the node consists of a small circle. If the circle is hovered by the user, an illustration of the corresponding state is displayed.
The user has, instead of just applying actions on his own, the ability to call the planner. Two different things can be done with the received plan. We can either apply an action of the found solution. This function allows the user to find the solution as quickly as possible. However, it is not really clear to the user how the planner has determined this solution. It is rather meant to show what the Planner is capable of, without going into the exact approach.
The more interesting function is that the planner can choose which state is expanded next. This way, the planner can help us finding the solution by ourselves by expanding one of the successors. However, it should be explained to the user for which reason that successor was chosen. This is done by a colored circle surrounding each node. The color of the circle stands for the f value of the sate. Since we are using A* as search algorithm, f is the sum of the heuristic and the path length from the initial state. Another information button explains how the evaluation function f is calculated. The user thus learns which successors are promising and with which information the planner works with.
The graph page visualizes the state space for the user. We can also explain how the planner works by using colored notes to indicate the f value of a node. It is the user that can decide if they want to expand promising nodes by themselves or let the planner do it. We now want to compare our graph representation with other examples of state space visualization.

5.2.1 Comparison with other Visualization

We will deal with a visualization tool for heuristic state space search called Web Planner (Magnaguagno et al., 2017). The purpose of the visualization tool is very similar to the goal of our graph representation. Major differences are that Web planner uses tree search instead of graph search. An advantage of tree structure is that although more nodes are needed, the distance from the initial state can be better represented. The reason for this is that there is exactly one path from the initial node to each other node. Another difference is that Web Planner is using Greedy-best-first search as algorithm. This simplifies the evaluation function $f(s)$ calculation since it holds that $f(s) = h(s)$ for a state s. Therefore, the concept of the g value does not have to be introduced. However, found plans with Greedy-best-first search are not optimal. Despite using another color scheme, the usage of color in both visualization tools are very similar.

Chapter 6

Conclusion

We created an application with the setting of the Blocksworld domain. Blocksworld is illustrated with 9 containers that can be placed on 4 stacks. For this we defined our own Blocksworld domain in PDDL. We had to make small adjustments to set a fixed number of stacks as well as a height constraint for each stack. The user in the application can execute actions to the corresponding state space on his own. The two actions are stacking or unstacking a container from a stack with the help of a crane. Additionally, the application contains a connection to the Fast Downward planner. The planner can be called by the user. The current state is then translated into PDDL. The PDDL serves as input for the planner. In Fast Downward, we use the A* algorithm and our self defined heuristic to discover plans. The heuristic is easy understandable and specifically designer for Blocksworld. The heuristic counts wrongly placed containers which result in a higher heuristic value. We did prove that the heuristic is both admissible and consistent. This shows that together with A*, the discovered solutions always are optimal. The user can later apply the found plan in the application. There is the possibility to simulate a state space search. For this the planner shows the user which state looks promising. The corresponding node can be expanded. However, it is also possible to let the planner solve the problem by itself. A click on a button executes the next action from the found solution.
The user not only has the possibility to use the functionality of the planner. The concepts of techniques used by Fast Downward are explained with the help of different representation. These representations can be found on a clipboard that the in game character carries with him. The used heuristic is explained on this clipboard. The current state and the goal state are also displayed. Incorrectly placed containers are marked with colors. An additional info box explains how the heuristic value is calculated.
The state space and the search in the state space are also represented. For the representation of a state space, graph search is used. In the graph itself, the evaluation function f is marked with different colors for each node. Another info box shows the user which information the planner uses to choose the next expansion. Thus, the user can understand at any time why the planner has explored a certain node.
All in all, heuristics, state spaces and the search in state spaces are introduced to the user. There is also a tutorial that describes the entire control of the application. Further

information and explanations about the planner and different concepts can be found on the clipboard.
The visualization of the state space is compared with another visualization tool called Web Planner (Magnaguagno et al., 2017). Some similarities were noted such as the use of colors to represent the f value of a state. But there are also differences. In Web Planner, tree search was used as representation method. Also, with greedy-best-first search a different heuristic was used.

There are goals that could not be implemented. The self-defined heuristic is simple to present and explain, but it has been constructed specifically for Blocksworld. Thus, it is not a planning heuristic. A possible extension would be that planning heuristics are also part of the application. Potential planning heuristics could be for example h^{add} and h^{max}. Those heuristics could possibly replace the current heuristic. However, since these heuristics are more difficult to explain, it would be a harder challenge to introduce them to the user.
Further extensions could be the implementation of additional search algorithms such as greedy-best-first search. It would also be conceivable to choose an algorithm that does not use heuristics such as uniform cost search. Thus, the improvement of using a heuristic could be worked out.
Besides adding more functionality to the existing application, there is also the possibility to reuse parts of the current implementation in other projects. This is especially true for the planner component. The component was designed so that it runs independently of Blocksworld. So projects that use other problem domains can also use the Planner. The implementation of the graph can also be reused. However, the visualization of the individual states would then have to be adapted.

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, I. (2023). Unreal Engine. `https://www.unrealengine.com/`. Version 5.1.1.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

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

Magnaguagno, M. C., FRAGA PEREIRA, R., Móre, M. D., and Meneguzzi, F. R. (2017). Web planner: A tool to develop classical planning domains and visualize heuristic state-space search. In *2017 Workshop on User Interfaces and Scheduling and Planning (UISP@ ICAPS), 2017, Estados Unidos.*

Russell, S. J. (2010). *Artificial intelligence a modern approach*. Pearson Education, Inc.