Interactive Blocksworld Application Showcasing Planning Techniques

Elia Hänggi

2. August 2023

Abstract

Fast Downward is a classical planner using heuristical search. The planner uses many advanced planning techniques that are not easy to teach, since they usually rely on complex data structures. To introduce planning techniques to the user an interactive application is created. This application uses an illustrative example to showcase planning techniques: Blocksworld
Blocksworld is an easy understandable planning problem which allows a simple 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. The concept of 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 and which techniques the planner uses.

Chapter 1

Introduction

Fast Downward is a planning system which searches for solutions in problems. In planning, problems are also called domains. The planner uses different concepts and algorithms to find solutions in a short time. For people not familiar with computer science, it is difficult to imagine in which areas a planner is used and what benefits it has. Concepts used in planning have a mathematical background and are therefore not always easy to understand. Search algorithms are complex solution methods that require some programming experience to comprehend how they work. These are all factors that make the functioning of a planner not easily accessible. The interactive application explains techniques and functioning of planning. Concepts are presented with simple representations and additionally explained. Furthermore, an illustrative example of a planner application is used: the Blocksworld domain.

Blocksworld is a famous planning domain. It consists of a set of blocks that can have different colors or labels to distinguish them from each other. Blocks can be used to form stacks. There are 4 fixed stacks available for blocks to be placed. The two moves possible are picking a block up and placing it on a stack. However, only one block can be moved at a time. The goal is to convert the initial state to the goal state with the moves mentioned.

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 (Figure 1.1). 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.

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 process.

The procedure in the video is expanded to an interactive application. This is done while using the same environment as in the video. The user is able to control the crane

Figure 1.1: Scene to represent the Blocksworld domain. The scene contains three container stacks on the ship and one on land. The crane is used to move containers.

and thus move containers by himself. Additionally, the application has an interface to the Fast Downward planner (Helmert, 2006). The user is able to ask the planner for specific information about the state space. The planner shows certain states that are interesting candidates to explore further. The planner also makes 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 is done by explaining the heuristic used and displaying the created state space to the user.
The application is generated using the Unreal Engine. Unreal Engine is a real-time 3D creation tool (Epic Games, Inc., 2023). Since the Engine uses C++ as programming language, this will be the language used in the application. However, Unreal Engine does provide the scripting system called "Blueprints". Some functionality is also be implemented in Blueprints.

This thesis starts with an introduction of the Unreal Engine and important concepts of planning theory. This lays all the foundations that need to be known for other parts. This includes features of the engine that are used to connect the planner to the application. Furthermore, an introduction into the theory of state spaces and classical planning where basic definitions and concepts are provided. These concepts are conveyed to the user with the application. The exact usage and functionality of the Fast Downward planner is stated as well.
After that, the didactic concept of the application is explained. The order in which different concepts are introduced is discussed. We analyze all visualizations of planning concepts introduced in the application. We justify the choice of specific visualization methods and compare them with other visualizations.
The implementation of the tool itself is discussed. We elaborate how the Blocksworld domain is defined to use it as input in Fast Downward. Important design decisions in the engine are explained.

Chapter 2

Background

We first introduce the Unreal Engine. Unreal Engine has its own classes and data structures, which differ from the C++ standard implementation. We discuss peculiarities of the engine. We state mathematical definitions of state spaces and heuristics. Furthermore the Fast Downward planner and its usage are 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, Inc., 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. 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 allow to easily design simple program parts or to test existing implementations.
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 frequently used class is the Actor class. Actors are objects that can be spawned on a map. The in game character can interact with actors.
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, for data structures and types exposed to C++ and Blueprints, Unreal Engine implementations should be used.

2.2 Planning Theory

We introduce state spaces, search algorithms and classical planning. All subjects are necessary to understand the functionality of the Fast Downward planner. The goal of the application is to convey the introduced definitions and concepts using examples and illustrative explanations.

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.

Definition 1 (State Space, Russell and Norvig, 2010)**.** *A state space $\mathcal{S}$ is 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 may be multiple goal states but only one initial state. Additionally it holds that for every state, if we apply an action, the outcome must be **deterministic**. This means that for a state s and an action a, there cannot be transitions $\langle s, a, s_1 \rangle \in T$ and $\langle s, a, s_2 \rangle \in T$ with states $s_1 \neq s_2$.
In the following we will focus on the search in a state space. The goal in state space search is to find a path from the initial state to a goal state. The solution of the search in a state space $\mathcal{S}$ consists of 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 different algorithms and data structures. Essential algorithms, which are used later in the planner are elaborated here.
In state space search, the basic idea is to iteratively generate a search space. The search space contains all states that were generated in the search process.
The search is started with the initial state. From there, new states are explored by adding the successors of the state to the search space. We call this procedure an **expansion**. The search ends when a goal state is expanded. In this case a solution was found. The search ends as well if all states in the search space were expanded but no goal state was found. In this case no solution exists.

In order to apply search algorithms on state spaces, data structures to represent the search space are needed. There are two main classes of state space searches: tree search and graph search.

Tree Search

In tree search, the structure of a tree is used for the search space representation. Tree search uses an **open list** as data structure. The open list contains all states that can be expanded next. If a state is expanded, its successors are added to the open list. Each path of the tree represents a single sequence of actions. However, this means that there can be cases where the same state is expanded twice. This is the case if two different action sequences lead to the same state. If the state space contains a loop, this can be problematic. A loop is present if a state $a \in S$ is reachable from a state $b \in S$ and vice versa. Tree search could expand states in a loop infinitely many times.
The goal of a search algorithm is usually not only to discover 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 the node data structure.

Graph Search

Graph search is the second class of search algorithms. The basic structure of graph search is the same as tree search. States that are expansion candidates are contained in the open list. 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 expanded. This ensures that the same state is never expanded twice. Therefore if one node was expanded, this node is added to the closed 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 is a finite number of states, it is ensured that the search terminates. All in all, the search space in graph search is much more compact than the tree in tree search (Figure 2.1). However, the distance from a node to the initial node is not easy to determine. This is because there are can be multiple paths between the initial node and all other nodes.
The fact that in graph search, each state is only visited once has a positive effect on the time complexity. It leads to a fewer number of 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. If the state space contains many loop, graph search is more viable.

Heuristics

Both tree and graph search are searching for solutions by expanding 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 on an optimal path should be expanded first. Heuristics help us in finding such states.

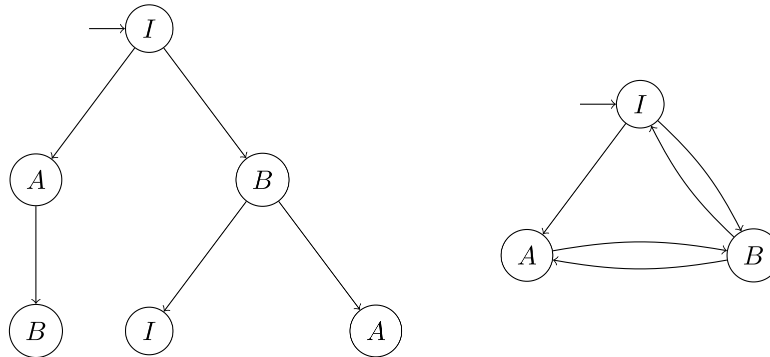

Figure 2.1: Example of the same search space represented with tree search (left) and graph search (right). Nodes labeled with the same letter should indicate equal states. I is the initial node.

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.

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

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

In addition, we define the heuristic h^* as the function that assigns the optimal solution cost to each state and ∞ to states from which the goal cannot be reached. This means h^* is the perfect heuristic since it perfectly estimates the distance from the state to the goal.
There are several properties that are desirable for heuristics, which are now presented. The first property is **goal-awareness**. A heuristic h is goal-aware if for all goal states s it holds that $h(s) = 0$. Another 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 all three 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 connections between the introduced properties (Theorem 1).

Theorem 1. *Let h be a goal-aware and consistent heuristic. Then h is an admissible heuristic.*

Therefore, we do not need to prove admissibility if a heuristic is goal-aware and consistent.

In state space search, heuristics are individually designed for each problem. The behavior or a structure of the problem is exploited to make a good estimation. 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.

Best-First Search

Heuristics are used to explore promising states early on to reach the goal faster or to find optimal solutions. In Best-first search, states in the open list are ordered with respect to an evaluation function f. f assigns each node a numerical value. The lower the value is the more promising is the node. Therefore, nodes with the smallest f value in the open list are expanded first. The pseudo code of Best-first search is stated in Algorithm 1.

If the evaluation function simply equals the heuristic value, 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 $g(s)$ for a state s. 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 tend to 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 of the current path and h a heuristic. A* and Greedy-best-first search both belong to Best-first-search algorithms. 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

Algorithm 1 Best-first Search

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

calculate f. It can be proven that A* discovers only optimal solutions if an admissible and consistent heuristic is used (Theorem 2). For this reason used heuristics in A* are almost exclusively admissible.

Theorem 2 (Hart et al., 1968)**.** *Solutions discovered by A* are optimal if the used heuristic h is admissible and consistent.*

Both Greedy-best-first search and A* are part of the graph search algorithms, using an additional closed list to detect duplicates.

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. For planning heuristics, no special properties of the problem are exploited. Instead, the problem is considered as a black box.
In planning, formalisms are used to define a planning task. The planning task induces the actual state space. The planning formalism that the Fast Downward planner uses is the Planning Domain Definition Language.

Planning Domain Definition Language

Planning Domain Definition Language (PDDL) is a standardized language to describe planning tasks (Ghallab et al., 1998). PDDL uses first order logic to define states. A state is a set of atoms where each atom is the value true or false assigned. An atom consists of a predicate that can take objects as parameters. With this definition of states, it is possible to represent an exponential number of different states with n different atoms.
PDDL can be separated into two parts, the **domain file** and the **problem file**. The domain file contains all information that hold for all instances of the same problem. **Predicates** of the problem are listed there. Predicates are relations to which objects are assigned. Furthermore, **actions** are specified in the domain file. 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. Actions in PDDL induce transitions in the classical state space definition. The notation of actions with preconditions and effects allows inducing transitions without listing all of them. To distinguish and group objects, object types are defined in the domain file.
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 atoms. In the problem file, **initial state** and **goal state** are specified. The initial state consists of a set of atoms assigned either true or false. The goal state consists of atoms that must hold for the goal to be reached.
The solution of a PDDL problem instance 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). The planner supports many different planning heuristics as well as search algorithms. This includes Greedy-best-first search as well as A*. 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 is 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 an additional flag it is possible to specify the destination folder of the solution.

Chapter 3

Didactive Concept

The goal of the application is to introduce the planner and its functionality to the user. Therefore, the concept of state spaces, heuristic and state space search are presented in the application. However, no previous knowledge should be necessary to be able to use the application.
The application can be divided into two different modes: the tutorial and the free play mode. The structure and purpose of both modes will now be discussed from an user perspective.

3.1 Tutorial Mode

The tutorial is used to show and explain the functions available in the application to the user. This is done by means of displayed texts. Game mechanics and functionalites are explained one by one. The goal is that the user understands all functionalities and visualizations that the application offers and can start with the free play mode afterwards.

First of all, the game controls are introduced. It is possible for the in game character to move with the W, A, S and D keys. Furthermore, the mouse can be used to control the field of view. This is shown to the user in a graphic that displays all controls. Now the user can look around and move. The user sees a ship at a harbor at first. On the ship there are 3 stacks with a total of 9 containers. Containers are distinguishable by their color and numbering. The user is now introduced to the problem by an overlay. The containers have to be brought into a given structure with the help of a crane. Subsequently, the container move mechanic is explained. This is done by highlighting the stacks to be clicked on. Meanwhile, the user is shown the stack located on land, on which containers can also be placed. With additional text the maximum height of each stack is specified. These are all controls that are available to the user to change the scene with the help of the crane.
After that, the clipboard is introduced. The clipboard is a surface that the in game player carries. When the C key is pressed, the clipboard can be picked up or put aside. The clipboard consists of 3 different pages, which are introduced one after the other in

the tutorial. The first page is the states page. Both the current state and the goal state are depicted. The tutorial explains here that the game is successfully completed when current and goal states are equal. Additionally, the concept of a heuristic is explained on this page.

3.1.1 Blocksworld Heuristic

To illustrate the concept of a heuristic, the heuristic used in the application serves as an example. The selection of a heuristic depends on a number of attributes that should hold for the heuristic in the application. It is important that the calculation process of the heuristic is easy understandable. As soon as complex data structures are necessary to calculate the heuristic, programming knowledge is required. The heuristic should also be a good estimate of the actual distance to the goal. This reduces the planning time and thus leads to a lower waiting time for the user. Another goal is that the heuristic is both admissible and consistent. This leads to the fact that only optimal solutions are discovered when using A*. This is desirable, because this means that the Planner's capabilities are not fully exploited. Otherwise the user might find better solutions than the planner.
The vast majority of good planning heuristics are based on complicated calculation procedures. Most of them use a slightly adjusted state space. This should simplify the calculation of the heuristic (e.g. delete relaxation). There are often multiple calculation steps needed until the heuristic value is found. This makes explaining how the value is obtained very difficult. Moreover, planning heuristics are based on abstract concepts that are difficult to teach. It is easier to explain the heuristics using a Blocksworld specific example.
Therefore, a heuristic is defined specifically for Blocksworld. This simplifies the explanation of the heuristic, since no additional concepts are needed.

Heuristic Definition

We define a heuristic on our own trying to design it with the attributes mentioned. For this we use the concept of a wrongly placed container. A container is wrongly placed if it is not at its goal location. In this case, the container itself and all containers above it are considered wrongly placed. Furthermore, we define for our state space that in a goal state there is no container attached to the crane. The self designed heuristic for this task is stated in Definition 3:

Definition 3 (Blocksworld heuristic). *We define a heuristic $h_{Blocks}(s)$ for a Blocksworld state s with*

$$h_{Blocks}(s) = 2 \cdot |c'| + d$$

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

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

The heuristic takes advantage of the fact that at least two actions must be performed if a container is placed incorrectly on a stack. If a container is attached to the crane, at least one action must be performed.
Now we check which of the defined properties apply to the heuristic.

Theorem 3. *The heuristic $h_{Blocks}(s)$ is goal-aware.*

Proof. We prove that h_{Blocks} is a goal-aware heuristic. A heuristic is goal-aware iff $h_{Blocks}(s) = 0$ for all goal states s.
Since s is a goal state, it holds that all containers are correctly placed. This implies that $|c'| = 0$. We also know that in a goal state there are no containers attached to the crane. Therefore $d = 0$. We can conclude that for all goal states s it holds that $h_{Blocks}(s) = 0$. Thus, h_{Blocks} is goal-aware. □

Theorem 4. *The heuristic $h_{Blocks}(s)$ 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 h, 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 d value increases by 1 because of the non-empty crane. We can say for the new heuristic $h_{Blocks}(s') = h_{Blocks}(s) - 2 + 1 = h_{Blocks}(s) - 1$. If the container is not in c', this implies that $h_{Blocks}(s') = h_{Blocks}(s) + 1$.
For *Case 1* we can conclude that $h_{Blocks}(s) \leq h_{Blocks}(s') + 1$.

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 $h_{Blocks}(s') = h_{Blocks}(s) - 1$. Otherwise, the cardinality of c' increases by 1. Then it holds that $h_{Blocks}(s') = h_{Blocks}(s) + 2 - 1 = h_{Blocks}(s) + 1$.
This means for *Case 2*, $h_{Blocks}(s) \leq h_{Blocks}(s') + 1$.

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

We now have proven that the heuristic h_{Blocks} is goal-aware and consistent. Together with Theorem 1, this implies that h_{Blocks} is admissible. If we use A* as search algorithm and the Blocksworld heuristic h_{Blocks}, 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.

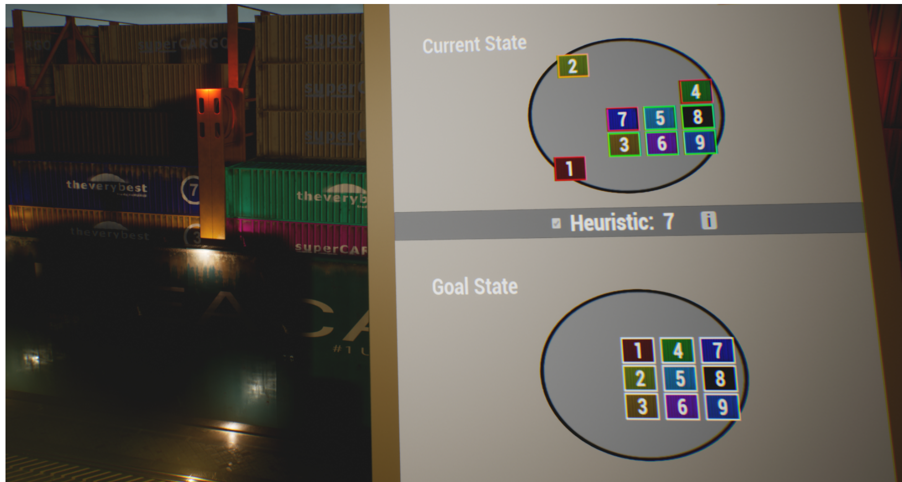

Figure 3.1: Cut-out of the clipboard state page. The current state and the goal state are displayed. Colors indicate which containers are correctly placed and which not. Heuristic value and legend button are located between the two states.

Visualization of the Heuristic

The state page of the clipboard 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 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 wrongly placed. This corresponds to an increase of the heuristic by 2. A container attached to the crane is marked yellow. This increases the heuristic h_{Blocks} by 1. Correctly placed containers are indicated in green (Figure 3.1). In addition to the colored container borders, the page contains a legend button. If the user hovers over this button, the meaning of the colors as well as the purpose of a heuristic is 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.

3.1.2 Visualization of the Plan

Next in the tutorial, the planner is introduced. This is done on the second page of the clipboard: the action page. The user has the possibility to call Fast Downward with the current state to generate a plan. All actions performed so far are listed in a scroll box (Figure 3.2). In addition, actions in the scroll box are clickable buttons. The click on an action restores the state where the selected action was executed. On the right side of the clipboard is the plan and the solution button. The plan button executes the planner command. When a plan is found, the entire action sequence is added to the scroll box. The solution button executes the plan in the scene. When the button is pressed, the next action of the plan is applied.

Figure 3.2: Action page of the clipboard. The sequence of actions discovered by the planner is shown. Additionally, the plan and the solution button are on the right side of the page.

Both buttons can be used on all pages of the clipboard. However, the action page is specifically responsible for visualizing the plan. Therefore, the tutorial introduces the planner at this point.

3.1.3 Visualization of the Search Space

The last concept introduced in the tutorial is the search space. The search 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 search 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. Another advantage is that the search space matches the state space, since in graph search each node can be assigned exactly one state.
The search algorithm used in the search space is A* search. This has the advantage that all discovered solutions are optimal. Another advantage is that the concept of g values is introduced. It shows the user that it does not always make sense to expand the node with the lowest heuristic but also to consider the costs of the current path.
The idea of the graph is that the user steadily explores the state space by applying

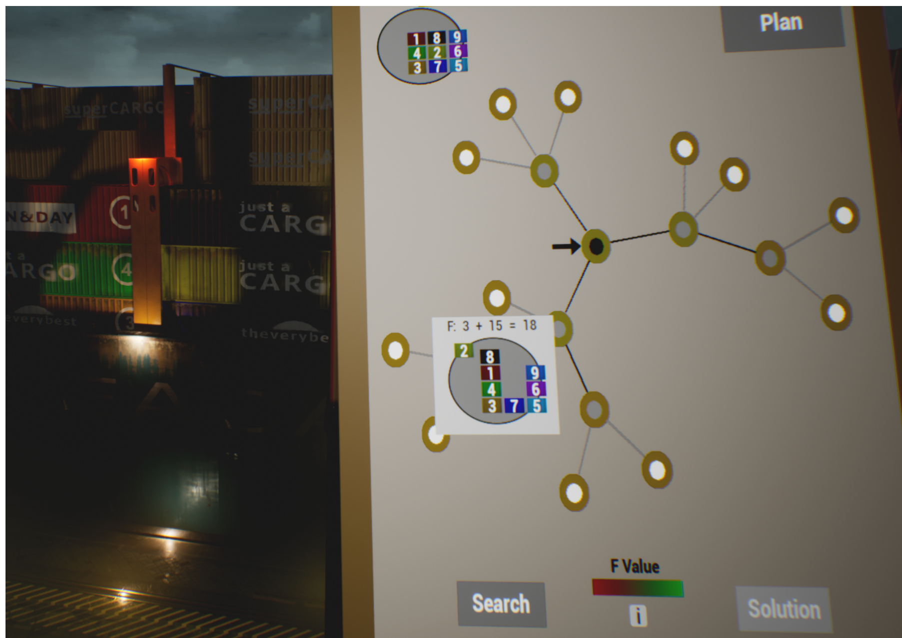

Figure 3.3: Search space visualization on the clipboard. Additional information about a state is displayed. Nodes in the open list are indicated with white color. The initial state is marked with a horizontal arrow. The search button is at the bottom of the page.

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 towards the initial node. Additionally, all candidates of the current search space which could be expanded next are displayed. These candidates are the equivalent to states in the open list.

Since the space where the graph is displayed is rather small, it is difficult to directly represent the state in the node. Instead the node consists of a small circle. If the user hovers over the circle, an illustration of the corresponding state is displayed (Figure 3.3).

A colored circle is surrounding each node. The color of the circle stands for the f value of the state. Since we are using A* as search algorithm, f is the sum of the heuristic and the path length from the initial state. Another legend button explains how the evaluation function f is calculated. The calculation of the f value for each state is visible if the user hovers over the node. The search button at the bottom of the clipboard offers the functionality to simulate a search. The node with the lowest f value is expanded.

The tutorial uses an initial state that is close to the goal state. In the tutorial, it is possible to fully simulate a search until the goal is found. For other initial states this may not be possible since the number of states to expand might be too large.

We now want to compare our graph representation with other examples of search space

visualizations.

Comparison with other Visualization

There are also other tools that visualize search spaces. We compare the visualization in the application with another visualization tool called Web Planner (Magnaguagno et al., 2017). The purpose of Web Planner 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.

3.2 Free Play Mode

The load mode provides all introduced functionality 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. 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 possible to create very difficult initial states where the planning time is accordingly high. The goal of this function is that the level of difficulty can be adjusted to the user. The aim of the mode is that the user utilizes the available functions to find the goal without any guidance.

Chapter 4

Implementation in Unreal Engine

This chapter discusses the implementation in the Unreal Engine. The implementation of the Blocksworld domain presented. Furthermore, the division between C++ and Blueprints is described. Furthermore, the integration of the Planner into the application is reported.

4.1 Blocksworld Implementation

We discuss the implementation of Blocksworld in PDDL. As stated earlier, PDDL can be separated into a domain file and a problem file. The domain file stays the same for every problem instance while the problem file changes for different instances. We first state the implementation of the PDDL domain file for Blocksworld (Listing 4.1).

```
(define (domain BLOCKS)
(:requirements :strips)
(:types block height)
(:predicates (on ?x ?y - container)
    (clear ?x - container)
    (craneempty)
    (holding ?x - container)
    (on-height ?x - container ?y - height)
    (SUCC ?hx ?hy - height))

(:action stack
    :parameters (?x ?y - container ?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)))
```

```
22 (:action unstack
23     :parameters (?x ?y - container ?hx - height)
24     :precondition (and (on ?x ?y) (clear ?x)
25                   (craneempty) (on-height ?x ?hx))
26     :effect
27         (and (holding ?x)
28         (clear ?y)
29         (not (clear ?x))
30         (not (craneempty))
31         (not (on ?x ?y))
32         (not (on-height ?x ?hx)))))
```

Listing 4.1: Blocksworld PDDL Domain File

There are in total 6 predicates in the Blocksworld domain. Predicates *on* and *clear* set the stack structure. *on(x,y)* states that a container *x* is positioned on another container *y*. *clear* marks if the container is the top of a stack. *craneempty* specifies if the crane is currently empty and *holding* states which container is currently attached to the crane.
The last two predicates *on-height* and *SUCC* enforce the height constraints for each stack. Classical planning 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 container 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. We ensure the fixed stack size with the definition of the stack action. The stack as well as the unstack action require two block objects: The container that is moved and the container underneath. With this definition, it is not possible to create new stacks since there is no block underneath. However, this would mean that unstacking the lowest container 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 4.2). To keep the file clearer, we will use an example instance only containing 4 containers.

```
1  (define (problem BLOCKS)
2  (:domain BLOCKS)
3  (:objects stackland stack2 stack1 stack3 c0 c1 c2 c3 - container h0 h1
       h2 h3 n0 n1 n2 n3 n4 - height)
4
5  (:INIT (craneempty) (SUCC h0 h1) (SUCC h1 h2) (SUCC h2 h3) (SUCC n0 n1)
        (SUCC n1 n2) (SUCC n2 n3) (SUCC n3 n4)
6      (on-height stackland h0) (on-height stack1 h0) (on-height stack2 h0
       ) (on-height stack3 h0)
7      (clear c0) (on c0 c3) (on-height c0 n2) (on c3 c7) (on-height c3 n1
       )
8      (clear c2) (on c2 c1) (on-height c2 n2) (on c1 c6) (on-height c1 n1
       )
9      (clear stack3)
10     (clear stackland))
11
12 (:goal (and (on c0 c1)  (on c1 stack1)
13     (on c2 c3) (on c3 stack2))))
```

Listing 4.2: Blocksworld PDDL Problem File

We defined several objects for this problem including 4 containers. Additionally, we did introduce 4 other containers. These containers represent the base of each existing stack. Since our actions do not allow the movement of the lowest containers, these 4 containers are stationary. All other containers can now be moved on top of those stationary bases. Therefore, we have our 4 fixed stacks (Figure 4.1).
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. On the land the maximal height is 3 since $h3$ is the highest height object. The maximal ship height is 4 with the maximal height object being $n4$.
Like mentioned before, the problem file varies depending in which state the scene is. However, the object definition stays the same for every instance. A sketch of the initial state in Listing 4.2. is displayed in Figure 4.1
We now use both the domain and the problem file as input to Fast Downward. The

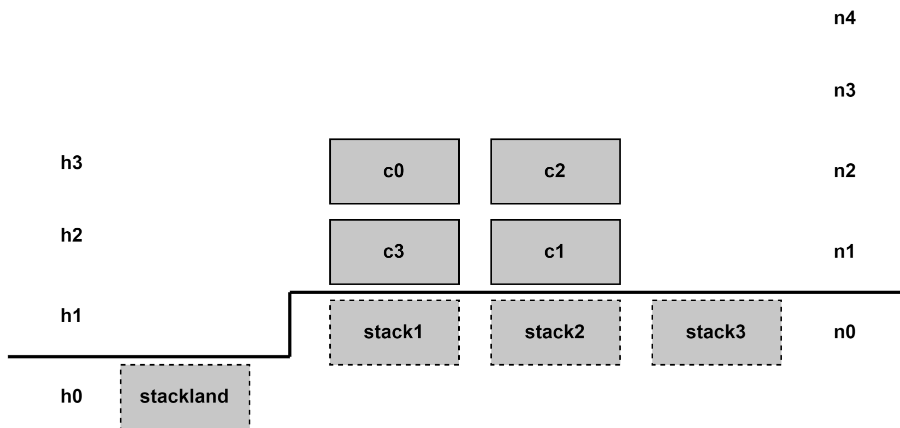

Figure 4.1: Sketch of the initial state in Listing 4.2. Containers with dashed lines symbolize stationary base blocks. Assigned height objects are marked on the sides of land and ship stacks.

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.

4.2 Division in Blueprints and C++

Blueprints are the visual scripting system that allows programming with less coding experience. Due to the visual representation, no knowledge about the syntax of C++ is necessary.
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

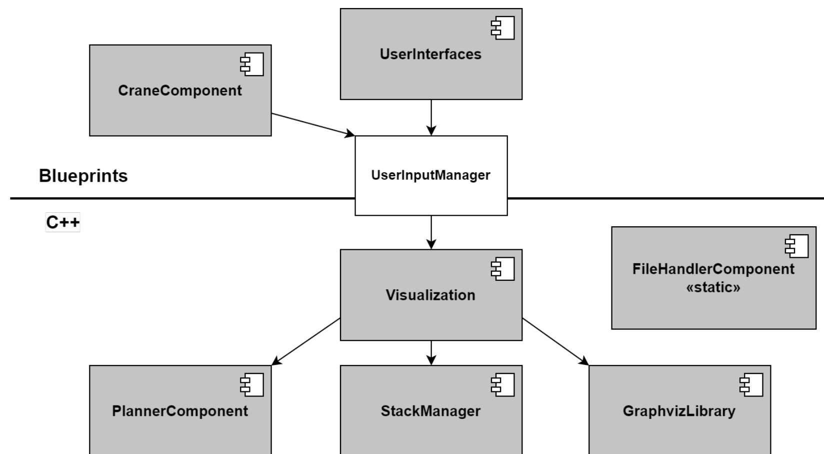

Figure 4.2: Diagram showing interactions of components in the project. The separating line indicates the division in C++ and Blueprints.

C++. Furthermore, 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, some components are implemented in Blueprints. A component is a group of classes that interact with each other to provide one or more functionalities.
One component where Blueprints are used is at 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 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. The application contains a class specifically for connecting C++ implementations with user interfaces (Figure 4.2). 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 were made.
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.

4.3 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 user calls the planner by pressing the plan button, 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 command is called. We use A* as search algorithm and our constructed heuristic h_{Blocks} 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 man-

aged in the actor. Thus, for example, it is possible to react differently to errors during planning.

Chapter 5

Conclusion

Our goal was to introduce concepts and techniques of planning with the application to people with no programming knowledge. At the same time the application should describe the functionality of the Fast Downward planner and illustrate an application area of the planner.
Planning techniques were taught using the Blocksworld domain. Blocksworld represents a real-world problem that is easy to understand and yet cannot be solved straight forward. Concepts explained with the application are heuristics, state spaces and associated search spaces. For explanations colored illustrations were used which adapt to the current state. The application uses a heuristic that can be derived from the state. Legends are used to convey additional information in short sentences.

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 for the application 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. Other projects that use other problem domains can use the planning component as interface to the Fast Downward planner. The implementation of the graph can also be reused. However, the visualization of the individual states would then have to be adapted.

Bibliography

Epic Games, Inc. (2023). Unreal Engine. https://www.unrealengine.com/. Version 5.1.1; accessed: 2023-05-30.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D., et al. (1998). PDDL — the planning domain definition language. *Technical Report CVC TR-98-003/DCS TR-1165. Yale Center for Computational Vision and Control.*

Hart, P. E., Nilsson, N. J., and Bertram, R. (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. (2017). Web planner: A tool to develop classical planning domains and visualize heuristic state-space search. In *Workshop on User Interfaces and Scheduling and Planning (UISP @ ICAPS)*.

Russell, S. and Norvig, P. (2010). *Artificial intelligence a modern approach*. Pearson Education, third edition.