Beer Routing - a different approach to teach routing algorithms

By Jonas Kraus, Matthias Mak, Philipp Speidel, Fabian Widmann

1 Introduction

In the following document we are going to describe our process of making a game that is intended to help students refine their understanding of various routing algorithms. According to Zyda, a serious game can be considered as a mental contest played with a computer with a specific ruleset that tries to provide education, training or other objectives via education [3, p. 26]. Thus we strived to entertain the players as well as provide a training for commonly used algorithms that are used in different areas of computer science. Additionally, he mentioned “Pedagogy must be subordinate to story—the entertainment component comes first.” [3, p. 26]. We tried to create a game that we would enjoy playing by just adding extra detail like little animations and generally staying in a two dimensional world, to be able to easily find suitable graphics or make our own. In addition to that, the chosen algorithms are fundamental in networking or even artificial intelligence courses and can benefit students in computer sciences.
In the following chapters we briefly give an overlook over the complete project from the concept phase to implementation, problems. Additionally we explain how to reuse the project and expand it by adding new levels and a brief documentation of our scripts. At last we will discuss the general outcome and give an outlook into the future of the project.

2 Concept

In the first part of this chapter we show some other work about teaching aspects of computer science via games. Then we describe how our game connects with the player. The main goal of our game is to teach the player some basic skills about routing in computer networks. The player should gain knowledge about how routing algorithms work, what concepts can be used in routing and why they might be useful in a scenario and not useful in other scenarios and that routing can be based on different routing metrics. To teach these aspects, we have developed four different game modes which are presented in the following paragraphs. The first game mode introduces a widely used routing algorithm, the Dijkstra algorithm. The second one should deepen the understanding of Dijkstra’s algorithm using the so called uniform cost search. Game mode three leads the player into another direction, introducing routing based on local information with greedy forwarding. The last game mode shows a common routing policy, namely hot potato based routing.

2.1 Related Work

When looking at serious games that cover topics regarding computer science related material, we mainly found games that should help the user understand programming. One earlier example was published in 2012 by Muratet et al. where the underlying genre was a real-time-strategy game [1]. Players had to code in the game to solve little ai problems. It required assistance by peers or teachers to write those code snippets. Snippets in turn were modeled in a simple programming language that would modify behaviour of units. Another game that aims to teach computational thinking and learning was developed by Kazimoglu et al. [2] in which the goal was to enable students, that started a computer science degree, to learn programming basics in a controlled environment. The gameplay consists of the player trying to help a robot escape. This is done by planning the escape with commands for movement and general control structures like conditions and loops. The goal of the player is to guide the robot to the goal in each level. In the described version of the game there were six levels that tried to teach different programming concepts. The participants liked playing the game, additionally they also thought that this approach can enhance the problem solving abilities of the players.

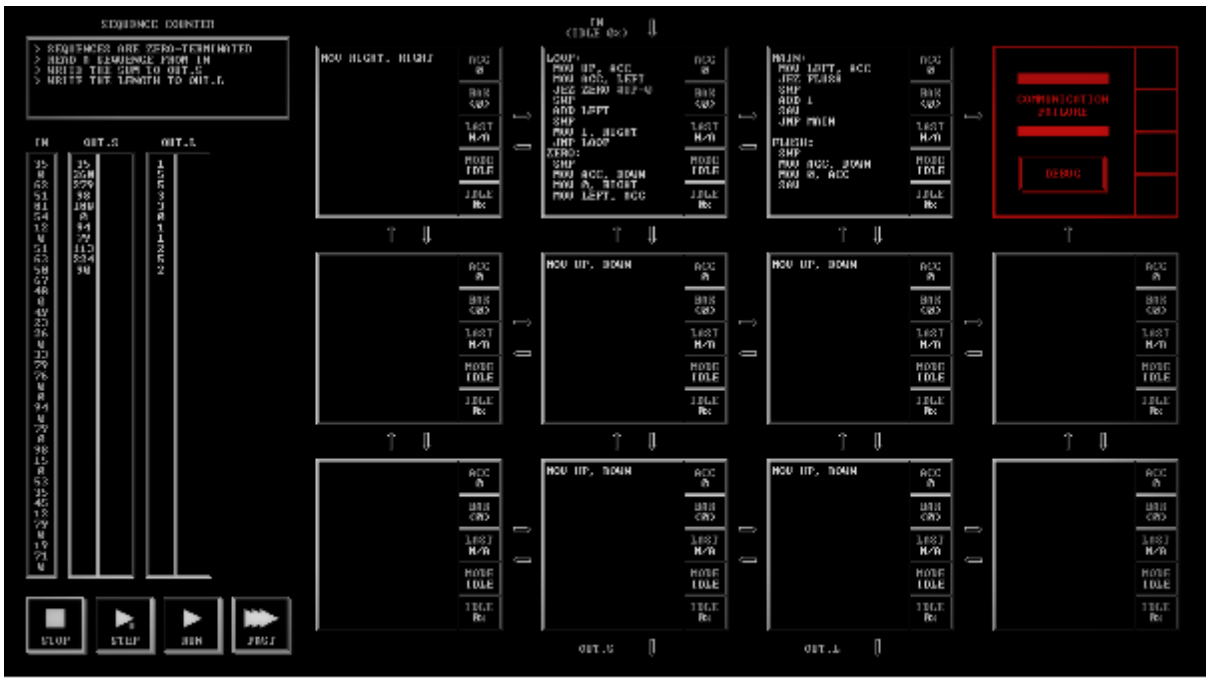
In recent years, other games were released to the public that could rather be considered programming puzzles by companies. The first one is “Human Resource Machine” by Tomorrow Corporation [4]. The theme of the game is the automation of boring office jobs. Those automation jobs get gradually harder and unlock the next level and sometimes more commands which grants a sense of progression.

Generally the game loop consists of three steps. First the player needs to get things from the inbox. After that, he can modify the input and then put the data into the outbox. This is done one after another. Additionally, the player can store the input in various storage fields.

Programming is done graphically by dragging in commands from a sheet of available commands. They consist of I/O, modification, storage and loop operations. The player then is able to test the solution at any time, either by running it completely at various speeds or performing it step by step in order to debug. Below, we can see the in- and outbox as well as options to get help. On the bottom are controls to test the solutions and on the right side we see the available operators. Further right the task is placed and space to write the program is provided.

Human Resource Machine [5] - a typical scene from an advanced level.

"Human Resource Machine" puts the focus on making the game accessible for non programmers, by including a graphical interface that allows visualization of loops and various other constructs.
"TIS-100" by Zachtronics [5] went in another direction by mostly giving up graphical elements to remind the user of plain command line interfaces as seen below.



TIS-100 [6] - typical level.

The goal of the game is to repair the corrupted TIS-100 computer to unlock it's secret. This can be done by completing several puzzles using a customized version of

assembly for the imaginary TIS-100 machine. It does not try to offer help in-game as the user should either have the manual, that contains all commands, printed or should open the document on their computer.
While the visualization is lacking, the player is able to parallelize certain operations as a level consists of several segments (rectangles with in and outputs to other Segments in the screenshot above). It also has an open sandbox mode that lets users create own puzzles. Thus the game generally caters to another audience that mostly can already program, at least a bit, in turn to solve puzzles.

Most of the games in this genre of programming puzzles tried to add progression via levels and visuals that were interesting for the target audience. This in turn inspired us to keep an eye on the difficulty of the levels and let the user unlock levels. Additionally we decided that we do want the user to have an in-game tutorial to introduce the game mechanics and the various metaphors. We want to cater to people that do not know how routing algorithms work and try to teach those concepts in a way that is both, interesting as well as entertaining.

2.2 Guiding the User

At first, we started working on an option to introduce players to our game. We quickly decided, that one tutorial level per game mode was a decent way to do this and thought of a figure that could mentor the player while playing.
We quickly decided on the persona of Professor Wahnsinn, a retired computer science professor who also possesses several pubs and teaches the specifics of delivering beer efficiently to each of his employees himself.

The tutorial explains the basic algorithm while letting the user play the level. It also encourages making a mistake and explains why the move was not correct. This explanation can be opened up again by using a designated button that repeats the last statements of the professor.

2.3 Player motivation

To provide a great game play experience and keep the player motivated we developed the following game elements.

2.3.1 Professor

The professor is fully animated and reacts to the player's actions - when making an error, he will shake his hand angrily and his face seems to be disappointed, while he seems happy when the player is doing things as planned. This allows us to give the player subtle feedback in a graphical way.
In addition, the professor has the ability to speak. The mouth is animated and an easy but fun "blah blah" voice is produced. The professor's voice is a nuance deeper when making errors. The voice doesn't change much, but still enough to recognize the difference to when the professor is happy.
Furthermore, the texts of the professor are not only informative, but sometimes very funny. In combination with the different animations it's pretty fun to interact with the professor.
You can read more about the professor's animations and voice in chapter 3.

2.3.2 Barkeeper

Like the professor, the barkeeper (player character) can change his face according to the latest user input. In general the user looks happy. On correct actions the barkeepers face changes to ultra happy. In contrast, the barkeeper looks very angry if the user makes a mistake.

2.3.3 Score and stars

Each level is assessed by the amount of collected points. You get points for correct actions and lose points for mistakes. As a visual representation there is a score beer in the status bar on the right side of the screen. The beer shows the current score in percentage terms. Furthermore, the content of the beer glas is animated. The beer gets filled or emptied through player actions.

At the end of the level the player gets graded. The grade is represented by stars. You get no stars if you collected less than 33% of the points, one star between 33% and 65% and two stars between 66& and 99%. If all points are collected you achieved 100% and you're rewarded with three stars. Full stars play a success sound while empty stars don't.

The main menu shows all levels. The levels are shown as locked or unlocked. The unlocked levels are depicted with zero, one, two or three stars. This overview shows the player instantly if there is a missing star. This incompleteness may annoy the player and therefore motivates him/her to master all levels.

2.3.4 Progress

In order to play a new level, the previous level must be passed with a minimum amount of success. As a special motivation there is a bonus fun level at the very end. To unlock this level, all other levels must be passed with three stars (100%).

2.3.5 Sounds

A lot of game elements are provided with sounds. Most of them are used to give auditive feedback to the user. For example, there is a positive sound when completing a subtask. Others are used only to enhance the various animations and thereby enrich the game play.

2.3.6 Graphics

Last but not least, we created lots of graphics and animations. The comic style art creates a great game experience. There are a lot of animations which are fun to watch. For example, there are several animations which show the professor with a beer bottle, a huge beer glas or a big bag of money. The player jumps, drinks beer and occasionally throws the beer glas in the air. You can read more about the animations in chapter 3.

2.4 Dijkstra algorithm

The Dijkstra algorithm is an algorithm for the shortest path problem in graphs. The algorithm is named after the computer scientist Edsger W. Dijkstra. There are different points of view that categorize the algorithm either as a greedy algorithm or as a dynamic programming algorithm. The Dijkstra algorithm works on a connected graph with nodes and edges. The edges are assigned with numerical values that represent the path costs for the corresponding path between two nodes. No negative path costs are allowed. In a given graph, shortest paths from one node to all other nodes of the graph can be found, i.e. the result of the Dijkstra algorithm is a shortest-path tree. The algorithm is widely used in network routing protocols, e.g. in the Open Shortest Path First (OSPF) protocol. In a routing context, the nodes are routers and the edges are connections between routers. The path costs usually depend on the speed of the connection.

2.4.1 Learning objective

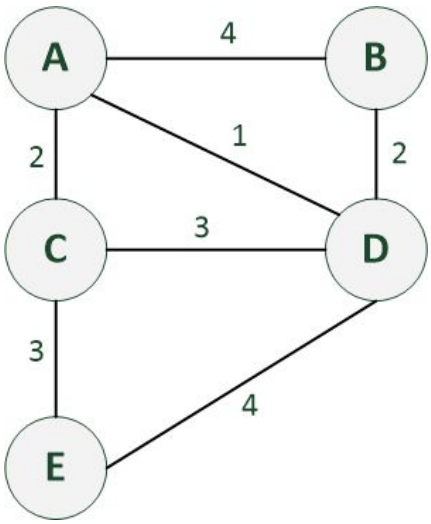
We have chosen this algorithm since it is a relatively simple and intuitive algorithm and also due to its relevance in existing routing protocols. The aim is to teach the player the approach of the algorithm. The goal is reached if the player is able to perform the single steps of the algorithm on an arbitrary graph after learning the approach with our game. Furthermore, the player should be supported in understanding the overall concept of the algorithm. Why does the algorithm extract shortest paths to all other nodes in a graph using the specific approach? Why can there be no other shorter path to a node that has been marked as handled by the algorithm? The next paragraph takes a look at the approach of the algorithm.

2.4.2 Algorithm

As already mentioned, the algorithm requires a source node. In an initialization phase, the source node is initialized with a distance value of 0 while all other nodes are assigned with a distance value of infinity. Furthermore, the algorithm keeps track of already visited and not yet visited nodes and thus initializes a list of unvisited nodes which contains all nodes except the source node at the beginning.
After the initialization phase, the algorithm starts running. One node of the graph is handled at a time, this node is referred to as the current working node. At the beginning, this is the source node. The algorithm considers all neighbour nodes of the working node which have not yet been marked as visited. The path costs of the path to a neighbour node are added to the distance value that is assigned to the current working node. If this calculated distance value is smaller than the current distance

value of this specific node, then the algorithm has found a shorter path to this node and assigns the new distance value. The current working node is memorized as the predecessor node on the shortest path to this node. However, if the calculated distance value is not smaller, then no action is taken for this node. If all unvisited neighbours of the current working node are checked, the algorithm needs to determine the next working node. This is done by checking the list of unvisited nodes. The node with the smallest distance value at that time is chosen as the next working node. The Dijkstra algorithm thus could be classified to the category of greedy algorithms. It takes the node which seems optimal at a certain point in time. The chosen node is marked as visited. A visited node will never be checked again, i.e. the algorithm never takes back any decisions it has made. The Dijkstra algorithm finishes if the list of unvisited nodes is empty.

The following example shows the approach of the Dijkstra algorithm. The given graph is shown in the figure below. The Dijkstra algorithm will be used to determine the shortest paths from the node A to all other nodes.



Step	Working node	A	B	C	D	E
0	init	0	(inf, -)	(inf, -)	(inf, -)	(inf, -)
1	A	0	(4, A)	(2, A)	(1, A)	(inf, -)
2	D	0	(3, D)	(2, A)	(1, A)	(5, D)
3	C	0	(3, D)	(2, A)	(1, A)	(5, D)
4	B	0	(3, D)	(2, A)	(1, A)	(5, D)
5	E	0	(3, D)	(2, A)	(1, A)	(5, D)

Step	List of unvisited nodes
1	{B, C, D, E}
2	{B, C, E}
3	{B, E}
4	{E}
5	{}

The upper table shows the single steps the algorithm performs. In the initialization phase, the node A gets a distance value of 0 while all other nodes are assigned a value of infinity. The algorithm starts with the source node which is A. The lower table shows the list of unvisited routers after each step. All nodes except A are marked as unvisited at the beginning, so the neighbours B, C and D are checked. The calculated distance, for instance 0 + 2 for node C, is smaller than the current value (infinity) for all three neighbours and thus all three nodes get a new distance value and the predecessor node A. The next working node is highlighted in blue in the table, here it is node D for the second step. Node D will be marked as visited. So far the algorithm has recognized that node B can be reached via A with costs of 4. However, while checking the unvisited neighbours, it is recognized that node B can be reached with costs 1 + 2 = 3 via node D. A shorter path is found, i.e. the distance value of B gets updated and the node D is stored as the predecessor node. This is highlighted in red in the table. Furthermore, node D now reaches node E which is not a direct neighbour of A, but can be also reached from A with costs of 5 via node D now. The next working node will be router C since it has the smallest distance cost of the unvisited routers at that time. The algorithm continues in the described way and finishes after node E is handled as the working node.

Although the Dijkstra algorithm can be seen as a greedy algorithms, it always results in an optimal solution. The reason for this stems from Bellman's Principle of Optimality. The principle says that for some optimization problems every optimal solution itself consists of optimal partial solutions. This applies to the shortest path problem. A shortest path P between two nodes A and B which leads over nodes X and Y implies that the path between X and Y is also the shortest possible path. If there was a shorter path between X and Y, then also the path between A and B could be shortened by using exactly this path between X and Y. This is a contradiction to the assumption that the path P between A and B is the shortest path between these two nodes. The optimality principle is used by dynamic programming algorithms. From this

point of view, the Dijkstra algorithm can also be considered to be a dynamic programming algorithm which calculates its result using optimal partial solutions.

2.4.3 Dijkstra game mode

We decided that the best way to teach the Dijkstra algorithm is to show each step of the approach by having the player perform the single steps successively. Each level in this game mode represents a graph on which the algorithm needs to be executed. In the following, nodes are referred to as routers and edges are referred to as paths. The routers are depicted as tables. The bottles on a table depict the paths that lead from this router to neighbour routers. The character on the bottle indicates the target router of the path. The player starts at a router, the source router. Beginning from this router, all shortest paths to all other routers need to be found. The path costs are assigned to the path instances. However, the path costs are hidden at the beginning, i.e. the player doesn't know the exact path costs of a path. The situation is shown in the picture below.

To mimic the behaviour of the Dijkstra algorithm, the player should then first discover all the paths between the source node and the neighbour routers. A path can be discovered by clicking on the corresponding beer bottle at the current table or on the one located at the neighbour router. The player object will then walk along the path and automatically return to the current working router after arriving at the neighbour router. During the walking process, beer puddles are spawned to indicate the player that the path is now discovered. After a successful return to the current working router, the path costs of this path will be displayed to the player. As mentioned, the path costs of a path usually depend on the speed of the link. Here, we rather take the length of the path as an indicator for the path costs as it is easier to depict. Letting the player object walk the path back and forth shall indicate the player that the path costs

are determined. The displaying of the path costs is done using the SpriteFontRenderer. The picture shows the status after the path C has been discovered.

The router, A in this case, remains the current working router until all paths to the neighbour routers are discovered. After all paths have been discovered, the router will be marked as handled. This mimics the behaviour of the algorithm of marking a router as visited. The player then needs to decide which router should be the next working router by walking to an unmarked router via paths that are already discovered. The following picture shows the status of the game after all paths to the neighbour routers have been discovered.

The decision on the next working router needs to be made by considering the path costs of the paths to the individual routers. The total path costs of a path to a specific

router is calculated by summing up the costs of the individual path segments that form the shortest path. The next working router is the unvisited router, i.e. a router that is not marked as handled, with the least distance to the source router regarding the total path costs. If the player is uncertain about the next step, he can display the routing table by clicking on the button in the status bar. The routing table contains entries for every router and shows the costs of the currently known shortest path from the source router to the individual routers including the predecessor router on this shortest path. The routing table is updated every time a path is discovered. The routing table also shows which routers are already marked as handled by highlighting them in a different colour. The routing table should support the player in taking the right decision on the next working router. However, the routing table is just an optional aid and is thus normally hidden. The picture below shows the routing table after handling the current working node A.

The player gets instant feedback on his actions by displaying score-texts like +5 points on a successful path discovery, +10 points on a successful decision regarding the next working router or negative scores on errors. An error can occur if a player walks to another router before having discovered all paths to the neighbours, or if the player simply chooses the wrong next working router. In this cases, professor Wahnsinn shows up and tells the player what he did wrong, so that he can correct his error.

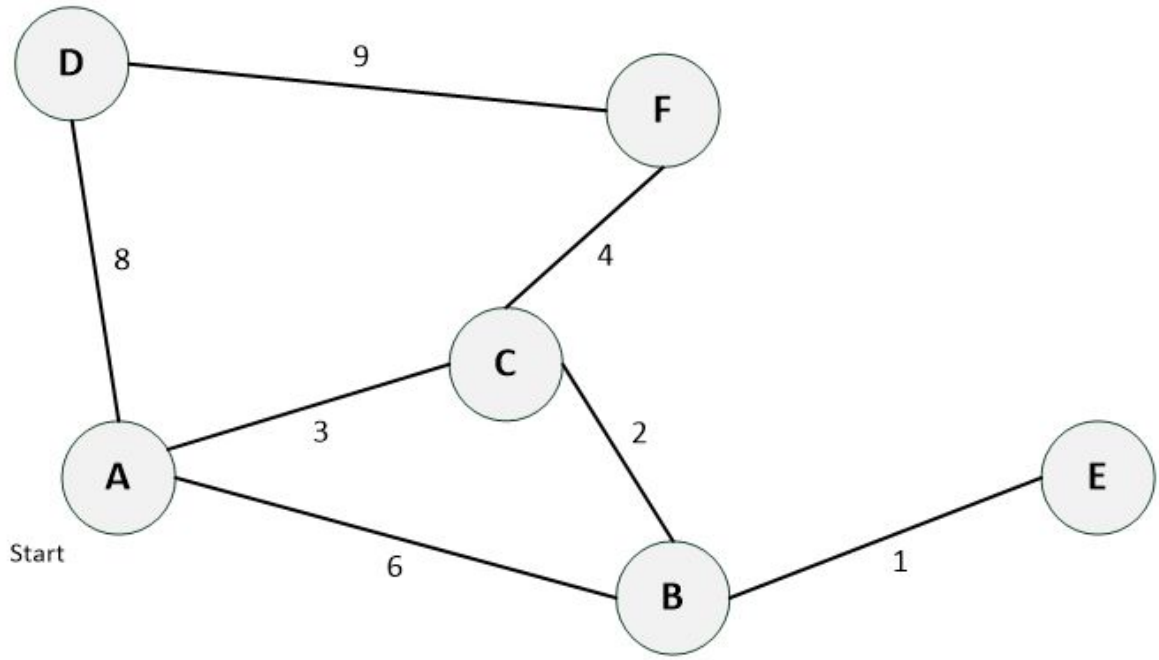
The level is finished if all routers are handled successfully. Professor Wahnsinn shows up and tells the player about the achieved score. The game mode itself and its rules are explained to the player in a separate tutorial level. The tutorial level is followed by five levels with different degrees of difficulty. The levels are described in the next paragraph.

2.4.4 Levels

Before starting to create levels, we needed to decide on criteria which define the difficulty of a level. We quickly agreed on an obvious criterion, namely the size of a level, i.e. the amount of routers and paths, but we required more than this criterion to make an actual difference in the level difficulty. We decided on further criteria like the amount of shorter path updates, i.e. how often are consisting paths to routers updated due to a discovered shorter path, the amount of possible solutions due to equal distances regarding the choice of the next working router at a time and the amount of hops the player needs to perform, especially while moving from the current working router to the next working router. In the following, we show the underlying graphs for the different levels, it's assumed degree of difficulty and the possible solutions.

2.3.4.1 Level 1

The first level is a less difficult level. It consists of six routers and seven paths. There is only one update of a consisting path during the whole execution process, namely a shorter path to B is found via router C in step 2. There are no situations with equal distances at a time, thus there is only one possible solution which is shown in the table below.



Step	Working node	A	B	C	D	E	F
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(6, A)	(3, A)	(8, A)	(inf,-)	(inf,-)
2	C	0	(5, C)	(3, A)	(8, A)	(inf,-)	(7, C)
3	B	0	(5, C)	(3, A)	(8, A)	(6, B)	(7, C)
4	E	0	(5, C)	(3, A)	(8, A)	(6, B)	(7, C)
5	F	0	(5, C)	(3, A)	(8, A)	(6, B)	(7, C)
6	D	0	(5, C)	(3, A)	(8, A)	(6, B)	(7, C)

The next working node is in most cases the direct neighbour of the current working router. This implies that the player doesn't need to perform many hops to get to it. The only case that involves more hops is in step 4 from router E to F.

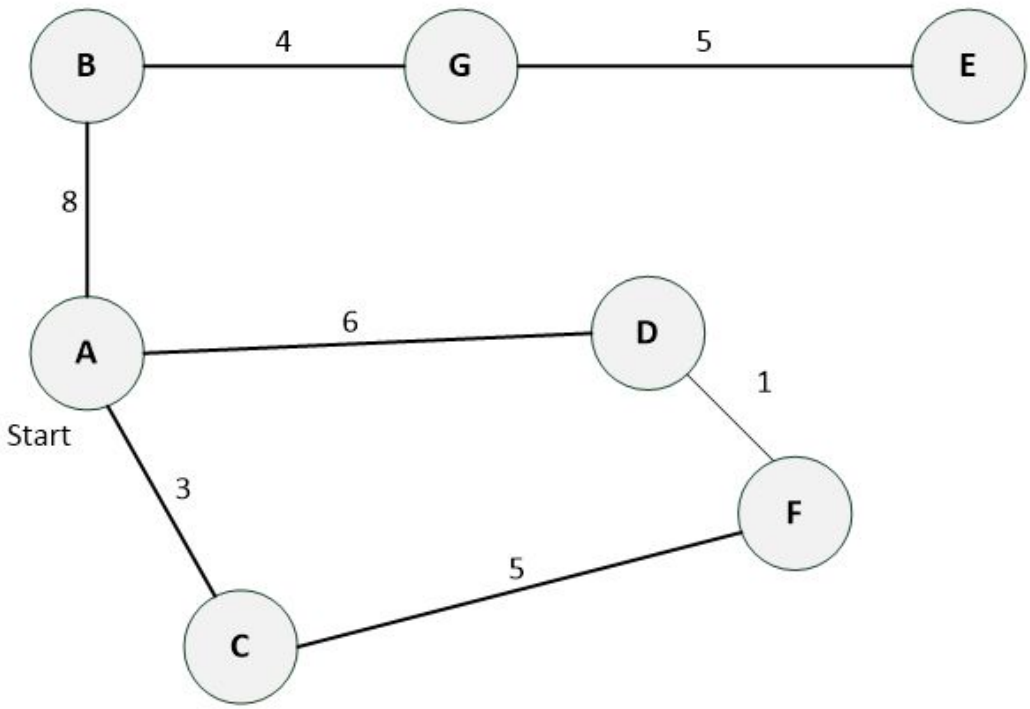
2.3.4.2 Level 2

The second level still has a low degree of difficulty. Just like in the first level, there is only one situation where a shorter path is found. In this level it is from A to F via D instead of the path via C. There is no situation with equal distances regarding the choice of the next working router at a time, thus there is only one possible solution.

The next working router can be reached with one hop in most cases, except from F to B in step 5.

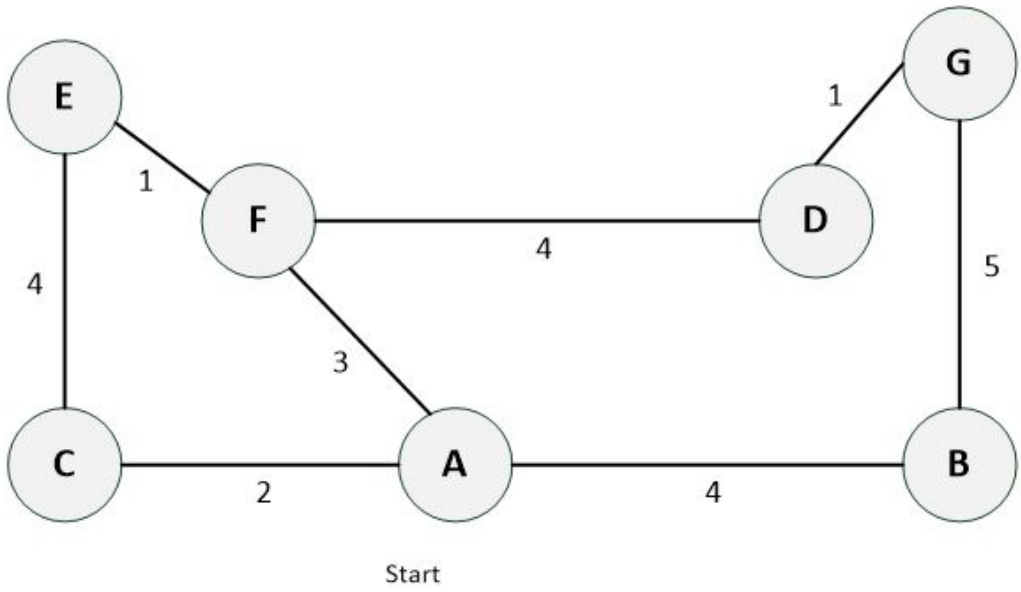
The picture shows the underlying graph of the level and the table below lists the steps for the solution.

Step	Working node	A	B	C	D	E	F	G
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(8, A)	(3, C)	(6, D)	(inf,-)	(inf,-)	(inf,-)
2	C	0	(8, A)	(3, C)	(6, D)	(inf,-)	(8, C)	(inf,-)
3	D	0	(8, A)	(3, C)	(6, D)	(inf,-)	(7, D)	(inf,-)
4	F	0	(8, A)	(3, C)	(6, D)	(inf,-)	(7, D)	(inf,-)
5	B	0	(8, A)	(3, C)	(6, D)	(inf,-)	(7, D)	(12, B)
6	G	0	(8, A)	(3, C)	(6, D)	(17, G)	(7, D)	(12, B)
7	E	0	(8, A)	(3, C)	(6, D)	(17, G)	(7, D)	(12, B)



2.3.4.3 Level 3

The third level has a higher degree of difficulty than the first two levels. It consists of seven routers and eight paths. During the execution of the algorithm, two path updates will be discovered. There is a shorter path to router E via F instead of the path via C and there is a shorter path to router G via D instead of the path via B. Furthermore, we now have a situation with equal distances and thus two possible choices for the next working router. In step 3 the current node is F and there are paths to E and to B with costs of 4. Router E and router B are the ones with the smallest distance value at that time, so the player can either choose E or B as his next working router. Depending on this choice, the rest of the steps adapt accordingly. The whole level thus has two possible solutions. The fact that the next working router in this level is often not a direct neighbour of the current working router increases the level of difficulty as well. For instance, at the beginning the player discovers all paths to the neighbour routers and then needs to choose C as the next working router. After handling router C, the next working router will be router F and the player needs to walk to F via router A. The underlying graph instance is depicted below as well as the two possible solutions for the level.



Solution 1:

Step	Working node	A	B	C	D	E	F	G
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)

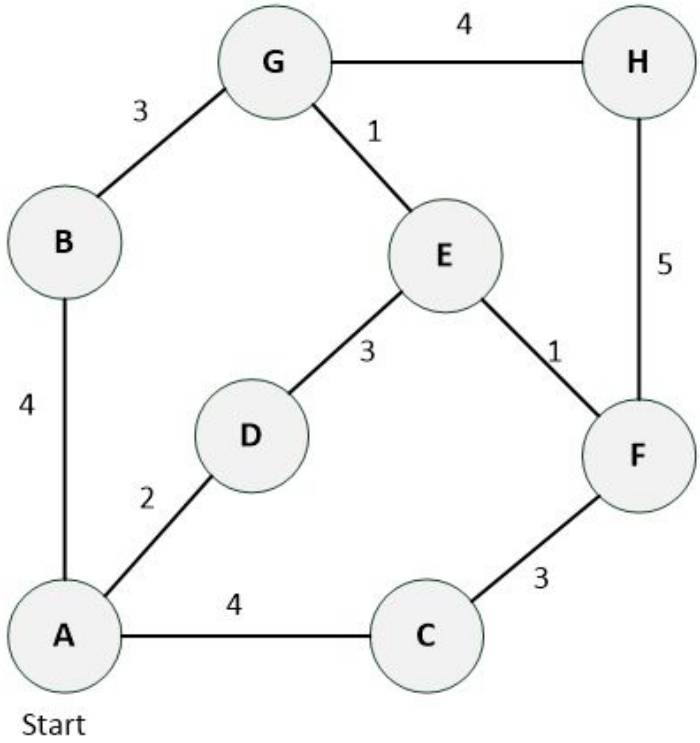
1	A	0	(4, A)	(2, A)	(inf,-)	(inf,-)	(3, A)	(inf,-)
2	C	0	(4, A)	(2, A)	(inf,-)	(6, C)	(3, A)	(inf,-)
3	F	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(inf,-)
4	E	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(inf,-)
5	B	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(9, B)
6	D	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(8, D)
7	G	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(8, D)

Solution 2:

Step	**Working node**	**A**	**B**	**C**	**D**	**E**	**F**	**G**
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(4, A)	(2, A)	(inf,-)	(inf,-)	(3, A)	(inf,-)
2	C	0	(4, A)	(2, A)	(inf,-)	(6, C)	(3, A)	(inf,-)
3	F	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(inf,-)
4	B	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(9, B)
5	E	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(9, B)
6	D	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(8, D)
7	G	0	(4, A)	(2, A)	(7, F)	(4, F)	(3, A)	(8, D)

2.3.4.4 Level 4

In order to make the level more difficult, we once again increased the number of routers and paths. The amount of updates due to shorter paths depends on the execution order of the routers in this level. It ranges between two and three updates. The execution order can vary in this level since there are two situations with equal distances of possible next working routers. This implies that there are four different solutions for this level. Increasing the amount of possible solutions seems to make choices easier at a first glance, however, we rather made the experience that the player needs to consider more routers at a time and is more likely to lose sight on one of them. More solutions thus often make the execution more error prone. The underlying graph and the possible solutions are depicted below.



Solution 1:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(4, A)	(4, A)	(2, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	D	0	(4, A)	(4, A)	(2, A)	(5, D)	(inf,-)	(inf,-)	(inf,-)
3	B	0	(4, A)	(4, A)	(2, A)	(5, D)	(inf,-)	(7, B)	(inf,-)
4	C	0	(4, A)	(4, A)	(2, A)	(5, D)	(7, C)	(7, B)	(inf,-)
5	E	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(inf,-)
6	G	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)
7	F	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)
8	H	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)

Solution 2:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(4, A)	(4, A)	(2, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	D	0	(4, A)	(4, A)	(2, A)	(5, D)	(inf,-)	(inf,-)	(inf,-)
3	C	0	(4, A)	(4, A)	(2, A)	(5, D)	(7, C)	(inf,-)	(inf,-)
4	B	0	(4, A)	(4, A)	(2, A)	(5, D)	(7, C)	(7, B)	(inf,-)
5	E	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(inf,-)
6	G	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)
7	F	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)
8	H	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)

Solution 3:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(4, A)	(4, A)	(2, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	D	0	(4, A)	(4, A)	(2, A)	(5, D)	(inf,-)	(inf,-)	(inf,-)
3	B	0	(4, A)	(4, A)	(2, A)	(5, D)	(inf,-)	(7, B)	(inf,-)
4	C	0	(4, A)	(4, A)	(2, A)	(5, D)	(7, C)	(7, B)	(inf,-)
5	E	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(inf,-)
6	F	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(11, F)
7	G	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)
8	H	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)

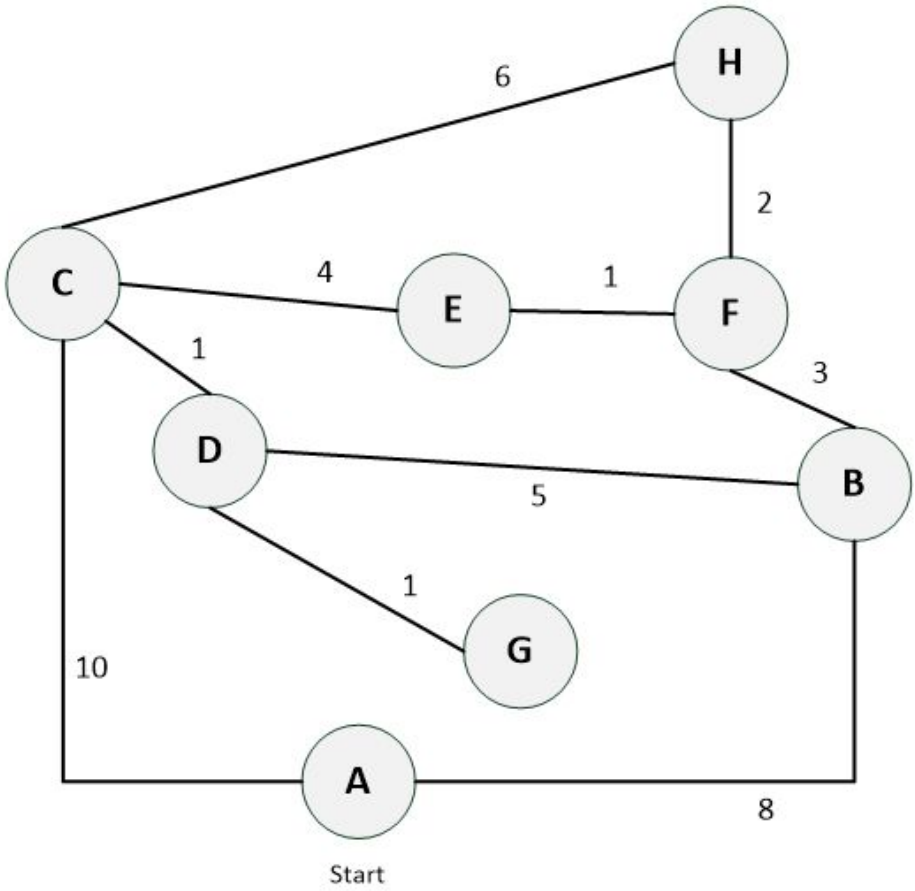
Solution 4:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(4, A)	(4, A)	(2, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)

2	D	0	(4, A)	(4, A)	(2, A)	(5, D)	(inf,-)	(inf,-)	(inf,-)
3	C	0	(4, A)	(4, A)	(2, A)	(5, D)	(7, C)	(inf,-)	(inf,-)
4	B	0	(4, A)	(4, A)	(2, A)	(5, D)	(7, C)	(7, B)	(inf,-)
5	E	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(inf,-)
6	F	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(11, F)
7	G	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)
8	H	0	(4, A)	(4, A)	(2, A)	(5, D)	(6, E)	(6, E)	(10, G)

2.3.4.5 Level 5

The fifth level is the most difficult of the levels. It is not only the largest, it also has long paths and requires a lot of hops in the transition process from one working router to another working router. The level consists of 8 routers and 10 paths. There are three updates due to shorter paths that are discovered during the execution of the algorithm. The level has two situations where possible next working routers have the same distance and thus it provides four different solutions. An additional difficulty was added in form of “traps”. As we can see in the depicted graph below, the node D is connected to G via a path with costs of 1. When discovering this path from D, the player might think that G needs to be handled next. However, the router G is not necessarily the next working router in every possible execution order. This requires the player to keep track of all unmarked routers. The underlying graph and the possible solutions are shown below.



Solution 1:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(8, A)	(10, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	B	0	(8, A)	(10, A)	(13, B)	(inf,-)	(11, B)	(inf,-)	(inf,-)
3	C	0	(8, A)	(10, A)	(11, C)	(14, C)	(11, B)	(inf,-)	(16, C)
4	D	0	(8, A)	(10, A)	(11, C)	(14, C)	(11, B)	(12, D)	(16, C)
5	F	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
6	E	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
7	G	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
8	H	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)

Solution 2:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(8, A)	(10, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	B	0	(8, A)	(10, A)	(13, B)	(inf,-)	(11, B)	(inf,-)	(inf,-)
3	C	0	(8, A)	(10, A)	(11, C)	(14, C)	(11, B)	(inf,-)	(16, C)
4	D	0	(8, A)	(10, A)	(11, C)	(14, C)	(11, B)	(12, D)	(16, C)
5	F	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
6	G	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
7	E	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
8	H	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)

Solution 3:

Step	Working node	A	B	C	D	E	F	G	H
0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(8, A)	(10, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	B	0	(8, A)	(10, A)	(13, B)	(inf,-)	(11, B)	(inf,-)	(inf,-)
3	C	0	(8, A)	(10, A)	(11, C)	(14, C)	(11, B)	(inf,-)	(16, C)
4	F	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(inf,-)	(13, F)
5	D	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
6	G	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
7	E	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
8	H	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)

Solution 4:

Step	Working node	A	B	C	D	E	F	G	H

0	-	0	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
1	A	0	(8, A)	(10, A)	(inf,-)	(inf,-)	(inf,-)	(inf,-)	(inf,-)
2	B	0	(8, A)	(10, A)	(13, B)	(inf,-)	(11, B)	(inf,-)	(inf,-)
3	C	0	(8, A)	(10, A)	(11, C)	(14, C)	(11, B)	(inf,-)	(16, C)
4	F	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(inf,-)	(13, F)
5	D	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
6	E	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
7	G	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)
8	H	0	(8, A)	(10, A)	(11, C)	(12, F)	(11, B)	(12, D)	(13, F)

The Dijkstra algorithm was used to provide an example for an algorithm which determines the shortest paths from one source router to all other routers in the current network topology. To provide a deeper knowledge of the approach of this algorithm, we developed another game mode. Instead of discovering the shortest paths from one source router to all other routers, the shortest path between two individual nodes needs to be found. We call this game mode uniform cost search.

2.5 Uniform Cost Search

The uniform cost search is quite similar to the approach of the Dijkstra algorithm. The difference between the two algorithms is rather based on implementation aspects than on conceptual aspects. The Dijkstra algorithm is often implemented using a priority queue which is initially filled with all nodes of the graph. During the execution, the distance values of the nodes can be updated if a shorter path is discovered. In these cases, the priority queue is reorganized. If the algorithm needs to handle a large graph with many nodes, this restructuring process can take up some time. Although nodes are removed from the priority queue after they have been marked as handled, the queue contains a lot of items, especially at the beginning. In the uniform cost search algorithm, the priority queue only contains one item at the beginning. Nodes are then added to the queue if they are discovered and not yet contained in the queue. Nodes are removed again if they have been marked as handled. Using this approach, the priority queue can be kept smaller during the execution process compared to the Dijkstra approach. The uniform cost algorithm thus can also be applied for graphs that are too large to be represented in memory at a time.

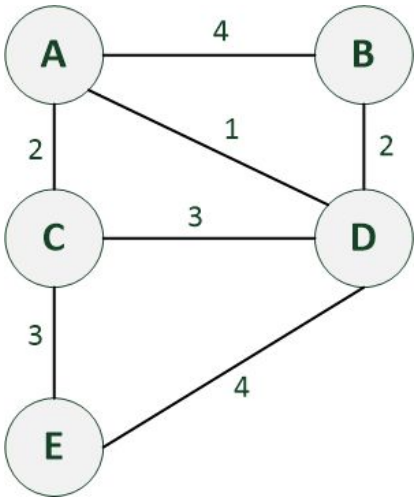
2.5.1 Learning objective

The goal of this game mode is to deepen the player's understanding of the underlying concept of the Dijkstra algorithm. In the previous game mode, the player has performed the single steps like discovering paths and choosing the next working router. However, there was only one source router and the player performed the steps until no unhandled router was left. Using this game mode, the player should be taught to transfer the learned knowledge and apply it to retrieve a specific shortest path between two routers. The goal is reached if the player is able to adapt the learned approach to tasks of this game mode and is able to determine the shortest path between two arbitrary routers of the network topology.

2.5.2 Algorithm

As already mentioned, the algorithm takes the same approach as the Dijkstra algorithm. However, the implementation is different. In the initialization phase the source router will be applied a distance value of 0 and it will be inserted into the priority queue. The algorithm also keeps track of the routers that are already handled by storing those in a list structure. The list is empty at the beginning. After the initialization phase, the algorithm starts running. The priority queue is built up using a priority value for the items which represents the current distance value to the source router. A lower priority value implies a higher priority. Since the queue provides the item with the highest priority on top, the next working router can be easily determined. The algorithm takes the front element out of the queue and handles it as the next working router. First, it is checked whether this next working router is already the target router. If it is, then there can be no shorter path between the source and the target router than the one that is currently memorized as the shortest path. The algorithm finishes in this case. Otherwise, all neighbour routers which are not marked as handled need to be checked. The distance calculation is done similar to the Dijkstra approach, i.e. the distance value of the current working router and the path cost of the path to the neighbour router together form a distance value that needs to be checked against the currently memorized value. This is done by checking whether the neighbour router is stored in the queue anyway. If not, it is added to the queue using the calculated distance value. If it is contained, it is checked whether the distance value needs to be updated. If the calculated distance value is shorter than the memorized one, a shorter path has been found and the update is triggered. Finally, the current working router is added to the list of handled routers and the next working router is taken from the queue. As already mentioned above, the algorithm continues until the target router is retrieved as the next working router.

The example below shows the execution of the uniform cost search algorithm for the given graph structure. The shortest path between router A and router B should be found.



Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{D → 1, C → 2, B → 4}
2	D	{C → 2, B → 3, E → 5}
3	C	{B → 3, E → 5}
4	B	{E → 5}

Step	List of visited nodes
1	{A}
2	{A, D}
3	{A, D, C}
4	{A, D, C, B}

In the initialization phase, the router A is initialized with a distance value of 0 and added to the priority queue. The algorithm starts by taking the front element out of the priority queue. In this case, the router A is the only item in the priority queue. Router A's unhandled neighbours are checked and since they are not contained in the priority queue, they are added with their calculated distance value. We know there is path from A to B with a distance of 4. However, the algorithm is not yet finished. There could be another path to B with less path costs. The algorithm continues with D as the next working router since D is the top element of the queue at that time. Router E is discovered and added to the priority queue. In this step, the algorithm calculates the distance value of B and recognizes that B can be reached with costs of 3 via D, i.e. a shorter path is found. The distance value for B is updated in the priority queue. In the table below, this updating process is marked in red. However, the algorithm still does not terminate. In the next step, router C is chosen as the working router. No updates are performed in this step. Now, the next working router is router B and the algorithm finally finishes. By handling router B, it is known that there can be no shorter path than the one that is currently memorized. The result of the algorithm will be path A – D – B.

2.5.3 Uniform cost game mode

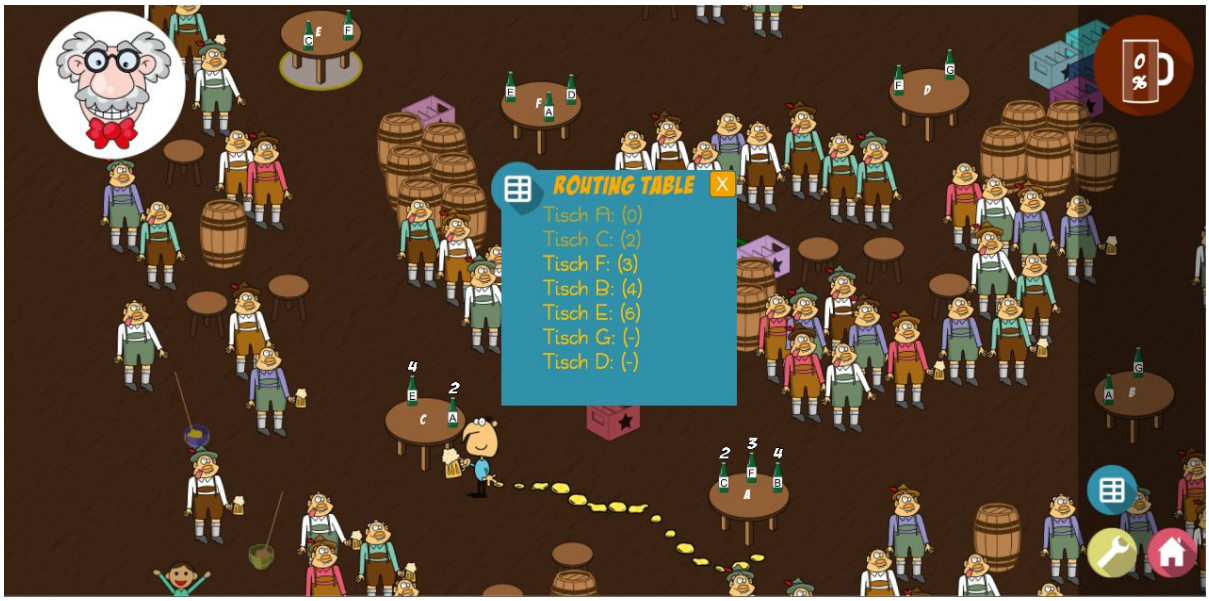
The uniform cost game mode requires the player to find the shortest path between arbitrary routers in the network topology. One level always consists of several runs where each run represents the search of a shortest path between two routers.
The objective of the game mode is to deepen the knowledge about the approach used by the Dijkstra algorithm. However, the player should not perform the exact same steps as in the previous game mode, otherwise the levels would be just further levels of the Dijkstra game mode. The target is rather to figure out, whether the player is able to transfer his acquired knowledge from the Dijkstra game mode and to apply it in a different scenario.

The player thus starts at a source router where the path costs to the neighbour routers are already displayed. No path discovery needs to be performed. The target router of a run is highlighted. The path costs of other paths are hidden at the beginning. The picture below shows this situation. The source router is A and the target router is E in this run.

The player needs to decide which of the neighbour routers will be the next working router. He needs to consider all unhandled routers at that time. The required calculation of the distance from the source router to a specific router is done just like in the Dijkstra algorithm. At the beginning, the only selectable paths are the ones to the neighbour routers whose path costs already represent the distance value since the source router has a distance value of 0. The player therefore can take the path with the least path costs. The path is once again selected by clicking on the beer bottles which represent this path. The player object will walk directly to the chosen

neighbour router. As already mentioned, there is no path discovery or similar mechanisms. However, after arriving at the specific router, the path costs of the router's paths will be revealed. The situation is shown below. As it can be seen, there is no highlighting or similar graphical indicators for marking the source router as handled. The player needs to remember which routers have been already handled in the current run. However, the player object spawns beer puddles when performing valid hops.

Each hop to an unhandled router represents the choice to select this router as the next working router. Hops to already handled routers can be used to get to an unhandled router which is not a direct neighbour of the last working router. The spawning of beer puddles and the score texts are an indicator for the player whether he made a correct choice or not. Also professor Wahnsinn will appear if a player has made a mistake. The mistake is then explained and the player can recover from it. A wrong choice can always be reverted by walking back along the wrong path to the last valid router. If the player is uncertain about the next working router, he can display the routing table for the current network topology. The routing table is adapted compared to the Dijkstra game mode. It lists all routers of the current network topology together with their current distance to the source router of the latest run. If the distance value is unknown for a router, it simply prints a dash character ("-"). The routing table is updated every time costs of paths are revealed. It can help the player to choose the correct next working router. Similar to the Dijkstra game mode, the routing table is normally hidden and needs to be activated by a click on the button in the sidebar.



A run is finished if the target router is chosen as the next valid working router. If there is a next run, it will be started after the player object has arrived at the target router. During the preparation of the next run, the target router is set as the source router for the new run and its highlight from the previous run is disabled. The target router of the next run is highlighted and the routing table is updated. All revealed path costs are hidden again except the ones that origin from the new source router. The algorithm starts again with the new context. The picture below shows the situation after the player has arrived at router E. Although the network topology stays the same, the player is forced to adapt to the new situation. The source and target routers are different and thus also other paths are relevant. This enables us to determine whether the player can apply the knowledge from the Dijkstra game mode to varying situations.

A level is finished if all defined runs are handled successfully by the player. Professor Wahnsinn shows up and tells the player about the achieved score.
The game mode itself is introduced in a tutorial level. The tutorial level is followed by five levels with different degrees of difficulty. The levels are introduced in the following paragraph.

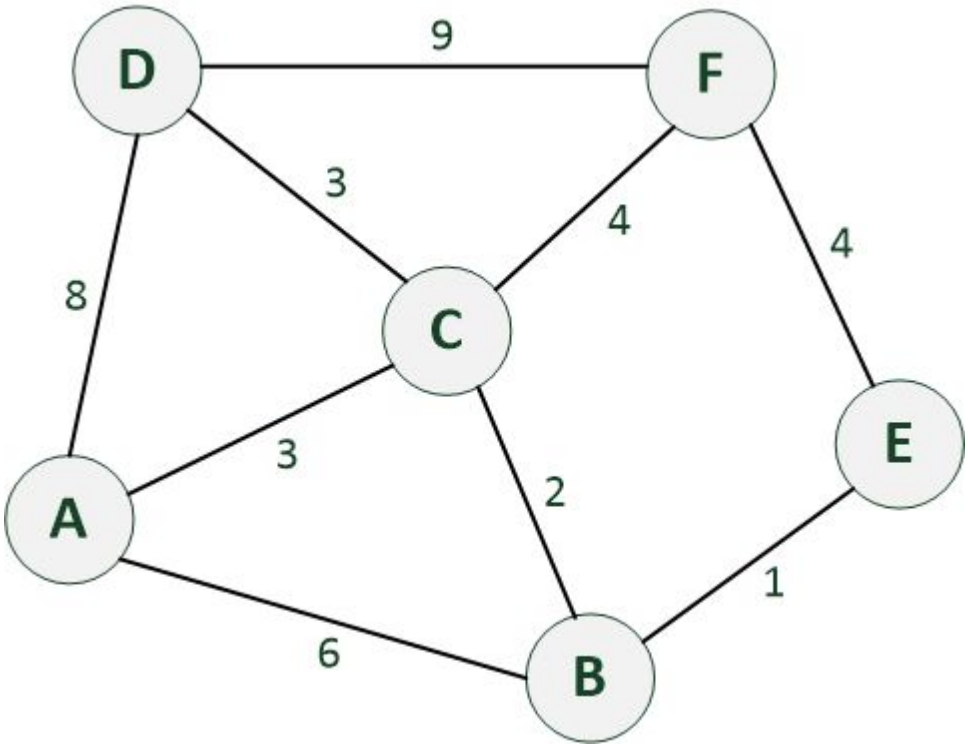
2.5.4 Levels

To determine the difficulty of levels for this game mode, we have decided on several criteria. The first criterion is the level size, i.e. the amount of routers and paths. However, since the algorithm doesn't necessarily require handling all routers of the topology we decided that we should take also into account how many routers actually need to be visited for a certain run. A further criterion is whether the choice of the next working router is rather obvious or if it is difficult due to a larger number of candidates. This can also imply situations where candidate routers have the same distance and thus result in multiple possible solutions. In this game mode, the choice of the next hop can be confusing, e.g. if the target router of a run is a direct neighbour of the source router and still other routers need to be visited first. In the situation that is depicted below, the player needs to choose router E first as their might be a shorter path from E to router C. This needs to be done although the player can already see that router E leads to a dead end.

The levels, their assumed degree of difficulty and their solutions are presented in the following.

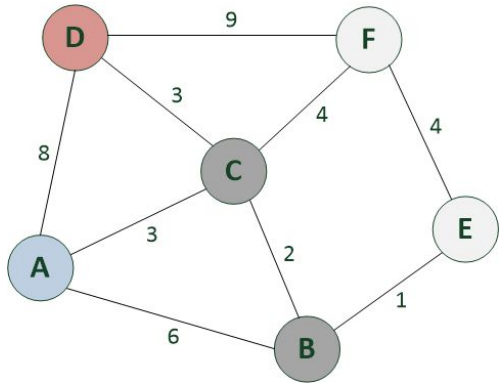
2.4.4.1 Level 1

The first level is an easy level. The topology of the underlying network structure, i.e. the graph, is nearly the same as in the tutorial of the uniform cost game mode. The player thus should already be familiar with it. Despite the low difficulty of the level, some mentioned aspects like counterintuitive path decisions can occur. The underlying graph is depicted below.



There are four runs in this level. The figure for each run show the starting node highlighted in blue, the destination node highlighted in red and nodes that need to be explored during the execution in dark gray. For each run, a table is defined which shows the status of the priority queue after each step. Sometimes different execution possibilities of a run exist. This is the case if nodes in the priority queue have the same priority value, i.e. the same distance to the starting router and thus appear as valid next hops. Updates of distance values in the table are highlighted in red.

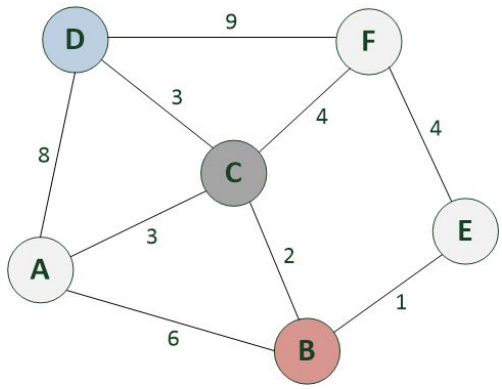
Run 1



Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{C → 3, B → 6, D → 8}
2	C	{B → 5, D → 6, F → 7}
3	B	{D → 6, E → 6, D → 7}
4	D	{E → 6, D → 7}

The first run has some of the mentioned counterintuitive situations, e.g. at the beginning the router D is directly reachable via A, however, the player needs to explore router C first due to the smaller distance value. After that, there is still a router with a better distance value, so B needs to be explored in the next step. After this is done, the player can finally finish the run and go to router D.

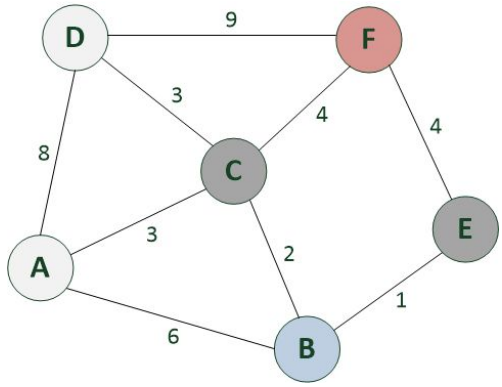
Run 2



Step	Working node	Priority Queue
0	init	{D → 0}
1	D	{C → 3, A → 8, F → 9}
2	C	{B → 5, A → 6, F → 7}
3	B	{A → 6, E → 6, F → 7}

The second run is simple. The player can directly walk to the destination.

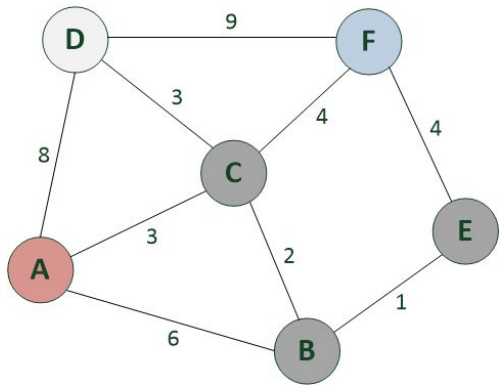
Run 3



Step	Working node	Priority Queue
0	init	{B → 0}
1	B	{E → 1, C → 2, A → 6}
2	E	{C → 2, F → 5, A → 6}
3	C	{F → 5, A → 5, D → 5}
4	F	{A → 5, D → 5}

This level offers three possible next hops in step 3, however, heading towards node F finishes the run.

Run 4



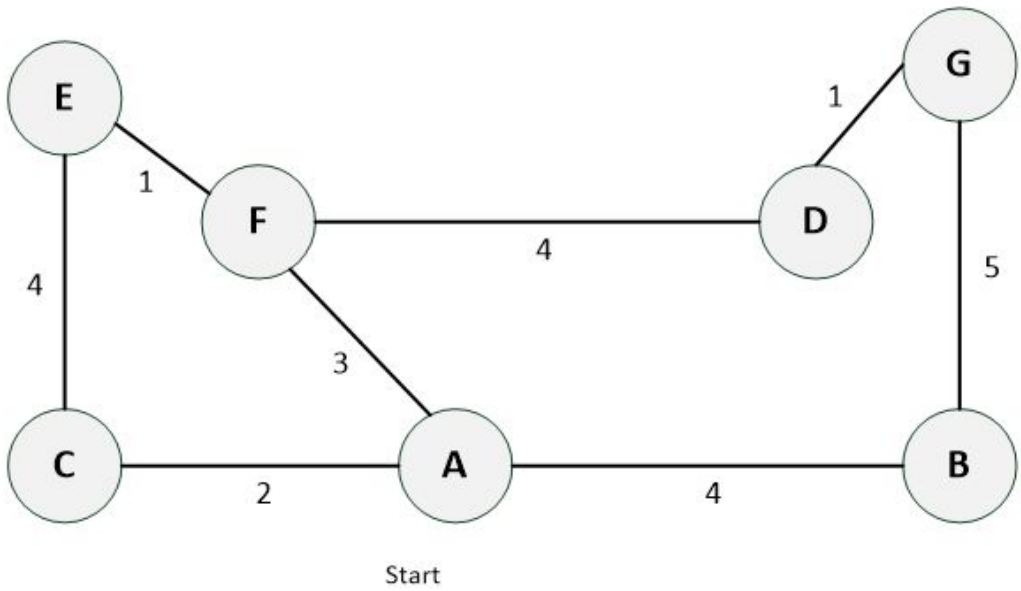
Step	Working node	Priority Queue
0	init	{F → 0}
1	F	{C → 4, E → 4, D → 9}
2	E	{C → 4, B → 5, D → 9}
3	C	{B → 5, A → 7, D → 7}
4	B	{A → 7, D → 7}
5	A	{D → 7}

The fourth run is actually a run with different execution possibilities. The table shows one of them. Instead of choosing node E in step 1, we could have chosen router C which then leads to another possible solution of this run.

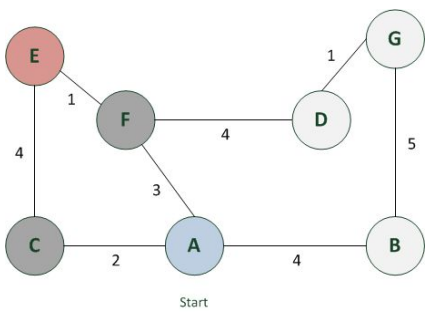
2.4.4.2 Level 2

The second level uses the same topology as the third Dijkstra level. There are four defined runs. The first run requires the player to find the shortest path from A to E, the second run from router E to G, afterwards the shortest path from G to A needs to be discovered and in the last run the path between A and B. The level is considered to

be relatively easy as it requires few hops in most runs. Still two of the four runs require more than four working routers. The choice of the next working router is in most cases quite obvious. There are few situations with equal distances of candidate routers at a time. However, there are a few situations where the player is not allowed to move to the target router although it is a direct neighbour of the current working router. The underlying graph and the possible solutions are presented below.



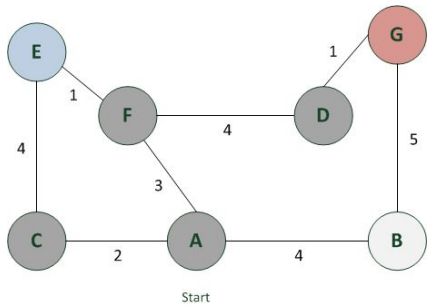
Run 1



Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{C → 2, F → 3, B → 4}
2	C	{F → 3, B → 4, E → 6}
3	F	{B → 4, E → 4, D → 7}
4	E	{B → 4, D → 7}

Alternatively router B can be visited first in step 4. B and E both have a distance of 4 at that time.

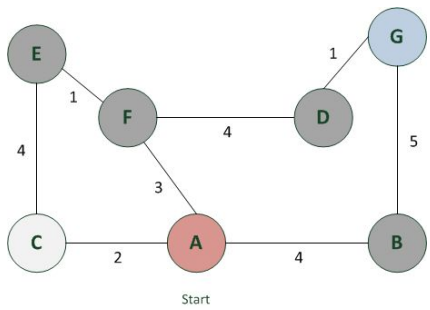
Run 2



Step	Working node	Priority Queue
0	init	{E → 0}
1	E	{F → 1, C → 4}
2	F	{A → 4, C → 4, D → 5}
3	C	{A → 4, D → 5}
4	A	{D → 5, B → 8}
5	D	{G, → 6, B → 8}
6	G	{B → 8}

Alternatively, in step 2 the router A can be chosen first. Router C would then be chosen in step 3.

Run 3

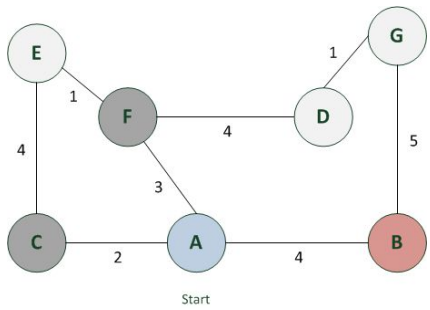


Step	Working node	Priority Queue
0	init	{G → 0}
1	G	{D → 1, B → 5}
2	D	{F → 5, B → 5}
3	B	{F → 5, A → 9}
4	F	{E → 6, A → 8}
5	E	{A → 8, C → 10}
6	A	{C → 10}

At step 2 there are two nodes with the same distance. An alternative solution is presented below.

Step	Working node	Priority Queue
0	init	{G → 0}
1	G	{D → 1, B → 5}
2	D	{F → 5, B → 5}
3	F	{B → 5, E → 6, A → 8}
4	B	{E → 6, A → 8}
5	E	{A → 8, C → 10}
6	A	{C → 10}

Run 4

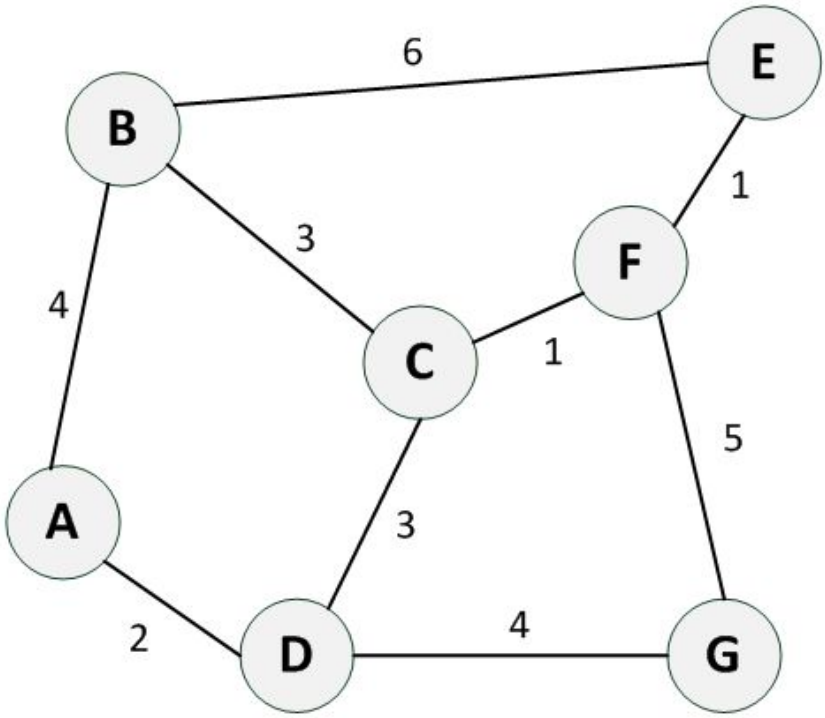


Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{C → 2, F → 3, B → 4}
2	C	{F → 3, B → 4, E → 6}
3	F	{B → 4, E → 4, D → 7}
4	B	{E → 4, D → 7, G → 9}

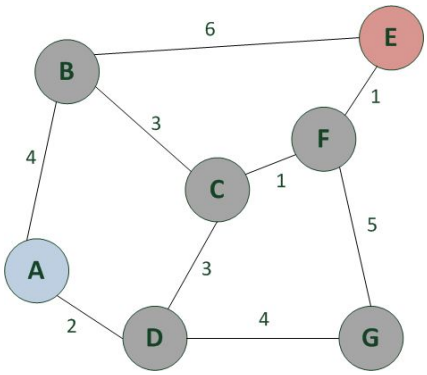
Alternatively, in step 3 the router E can be chosen first. Router B would then be chosen in step 4.

2.4.4.3 Level 3

The third level has a higher degree of difficulty compared to the second level even if they are about the same size. This is mainly based on the positions of the source and destination routers for the different runs. The amount of hops that need to be done is high for each run of the level. This makes it harder to keep sight of all possible candidates. Just like in the previous level, there are also situations with equal distances of candidate routers. There are four runs in total for this level. The underlying graph and the possible solutions are depicted below.



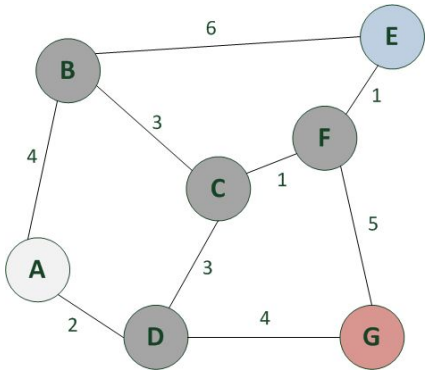
Run 1



Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{D → 2, B → 4}
2	D	{B → 4, C → 5, G → 6}
3	B	{C → 5, G → 6, E → 10}
4	C	{G → 6, F → 6, E → 10}
5	G	{ F → 6, E → 10}
6	F	{E → 7}
7	E	{}

Alternatively router F can be visited first in step 5. Router G would then be the working router in step 6.

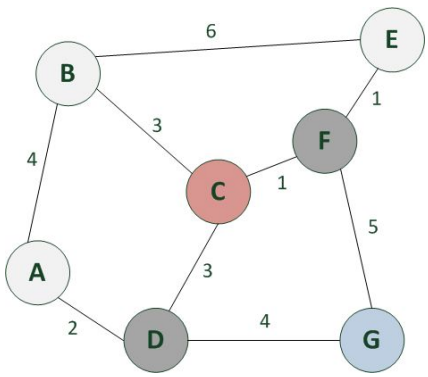
Run 2



Step	Working node	Priority Queue
0	init	{E → 0}
1	E	{F → 1, B → 6}
2	F	{C → 2, B → 6, G → 6}
3	C	{B → 5, D → 5, G → 6}
4	B	{D → 5, G → 6, A → 9}
5	D	{ G → 6, A → 7}
6	G	{A → 7}

Alternatively router D can be visited first in step 4. Router B would then be the working router in step 5. The path update for the path to router A then won't take place since the shorter path is found first.

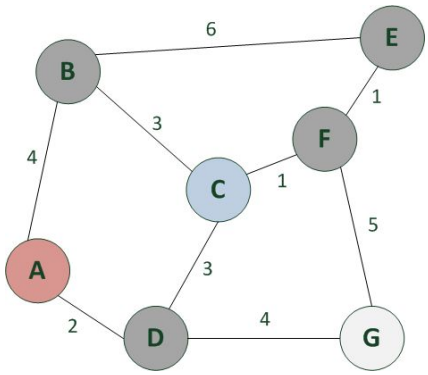
Run 3



Step	Working node	Priority Queue
0	init	{G → 0}
1	G	{D → 4, F → 5}
2	D	{F → 5, A → 6, C → 7}
3	F	{E → 6, A → 6, C → 6}
4	C	{E → 6, A → 6, B → 9}

In step 3, also E and A could be selected as the next working router. These are valid hops, however, moving to C will finish the run directly.

Run 4

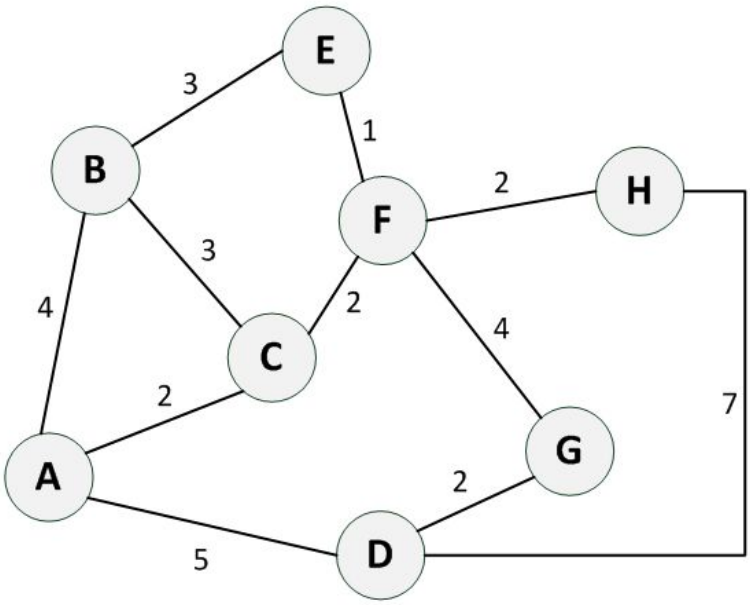


Step	Working node	Priority Queue
0	init	{C → 0}
1	C	{F → 1, B → 3, D → 3}
2	F	{E → 2, B → 3, D → 3, G → 6}
3	E	{ B → 3, D → 3, G → 6}
4	B	{D → 3, G → 6, A → 7}
5	D	{A → 5, G → 6}
6	A	{G → 6}

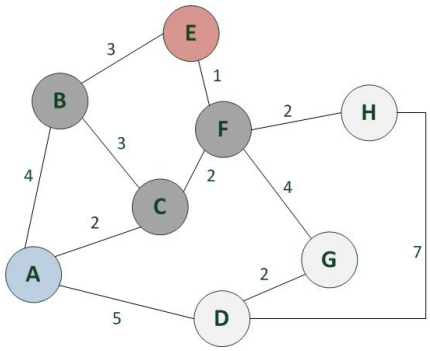
Alternatively, node D could be chosen in step 4 instead of B. Router B would be the working router in step 5 in this case. The update of the path to A won't take place since the shorter path is discovered first.

2.4.4.4 Level 4

To increase the difficulty further, the fourth level has more nodes and paths. This makes it more difficult to keep track of all candidate routers. The level itself consists of 5 runs. The amount of hops for each run is high. Furthermore, there are a lot of situations where candidate routers have the same distance. The level has thus several possible solutions. The underlying graph and the possible solutions are shown below.



Run 1

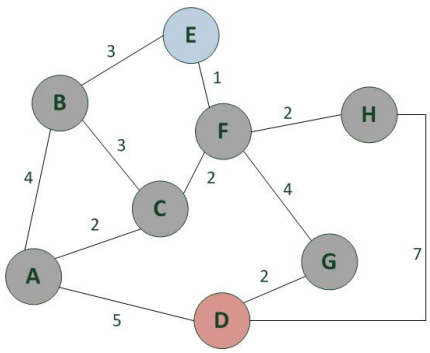


Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{C → 2, B → 4, D → 5}
2	C	{F → 4, B → 4, D → 5}
3	B	{F → 4, D → 5, E → 7}
4	F	{D → 5, E → 5}
5	E	{D → 5}

Alternatively, router F could be selected as the next working router in step 2. The router B would then be selected as the next working router in step 3. In this case, no update of the path to E will take place since the shorter path is discovered first. In the last step, the router D could also be chosen as the working router, however, taking router E finishes the run.

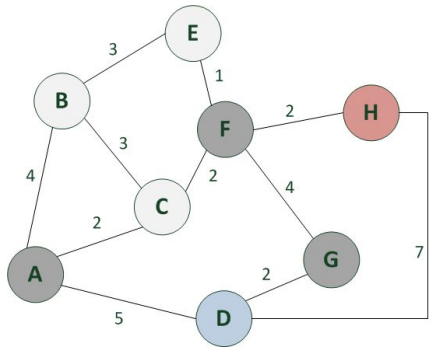
Run 2

The second run offers many possible solutions. There are 3 possible next working routers in step 2. In step 5 are two routers with the equal distance. For obvious reasons we won't list all possible solutions for this run. The following table shows one possible solution. However, independently of the chosen execution order, all nodes need to be processed before the actual shortest path is unambiguously found.



Step	Working node	Priority Queue
0	init	{E → 0}
1	E	{F → 1, B → 3}
2	F	{C → 3, H → 3, B → 3, G → 5}
3	C	{B → 3, A → 5, G → 5 D → 10}
4	B	{A → 5, G → 5, D → 10}
5	A	{G → 5, D → 10}
6	G	{D → 7}
7	D	{}

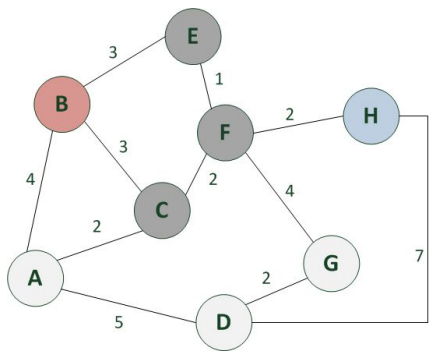
Run 3



Step	Working node	Priority Queue
0	init	{D → 0}
1	D	{G → 2, A → 5, H → 7}
2	G	{A → 5, F → 6, H → 7}
3	A	{F → 6, C → 7, H → 7, B → 9}
4	F	{C → 7, H → 7, E → 7, B → 9}
5	H	{C → 7, E → 7, B → 9}

In step 4, three candidate routers have the same distance. However, the run is finished if the player walks directly to router H.

Run 4

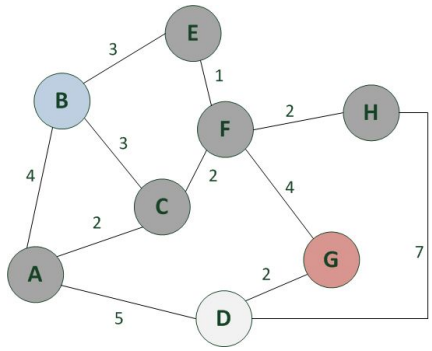


Step	Working node	Priority Queue
0	init	{H → 0}
1	H	{F → 2, D → 7}
2	F	{E → 3, C → 4, G → 6, D → 7}
3	E	{ C → 4, G → 6, B → 6, D → 7}
4	C	{G → 6, B → 6, A → 6, D → 7}
5	B	{G → 6, A → 6, D → 7}

There are several possible next working routers in step 4. However, walking directly to B finishes the run.

Run 5

There are several possible solutions for run 5. In step 1, either router E or C can be chosen as the next working router. In a later step, routers F and A have a distance value of 4. In the table below, one possible solution is shown.



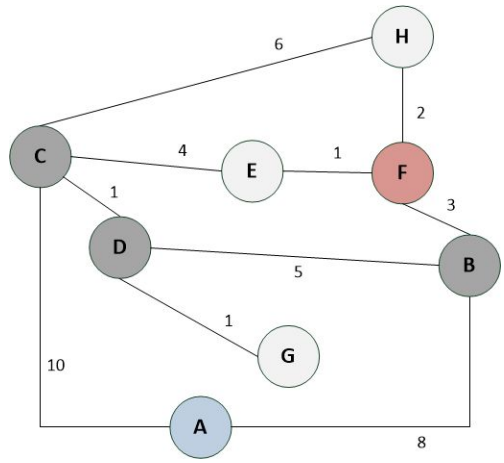
Step	Working node	Priority Queue
0	init	{B → 0}
1	B	{E → 3, C → 3, A → 4}
2	C	{E → 3, A → 4, F → 5}
3	E	{ F → 4, A → 4}
4	A	{F → 4, D → 9}
5	F	{H → 6, G → 8, D → 9}
6	H	{G → 8, D → 9}
7	G	{D → 9}

2.4.4.5 Level 5

The last and hardest level of the uniform cost game mode is based on the topology of the fifth level of the Dijkstra game mode. The level is large and the paths between the nodes are long. There are a lot of situations where the next valid router can be considered to be counterintuitive, so decisions need to be made carefully. The latter is also true regarding the vast extent of the level. Five runs need to be performed in total for this level.

Run 1

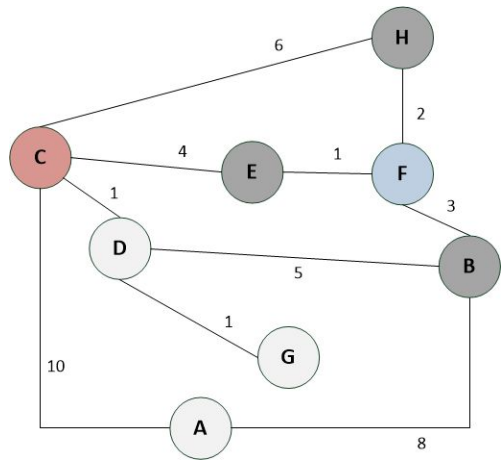
There are several possible solutions for the first run. The solution presented in the table below requires one additional step regarding the optimal solution. However, exploring router D in step 4 has the advantage of being able to walk to B directly instead of walking via router A. The path is thus shorter and the player requires less time to walk. From an algorithm perspective both solutions are accepted and provide full score.



Step	Working node	Priority Queue
0	init	{A → 0}
1	A	{B → 8, C → 10}
2	B	{C → 10, F → 11, D → 13}
3	C	{ F → 11, D → 11, E → 14, H → 16}
4	D	{F → 11, G → 12, E → 14, H → 16}
5	F	{G → 12, E → 12, H → 13}

Run 2

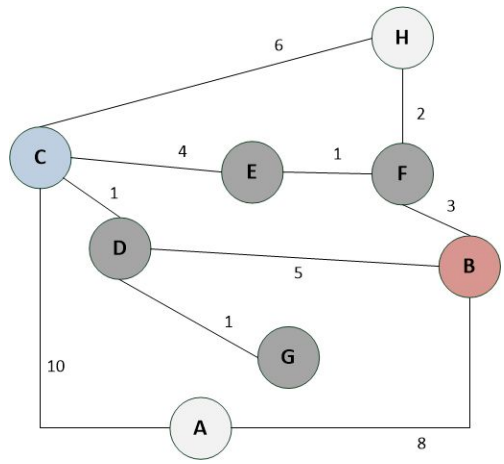
The second run has only one possible solution. However, instead of heading towards C directly from router E, the player needs to explore H and B first.



Step	Working node	Priority Queue
0	init	{F → 0}
1	F	{E → 1, H → 2, B → 3}
2	E	{H → 2, B → 3, C → 5}
3	H	{ B → 3, C → 5}
4	B	{C → 5, D → 8, A → 11}
5	C	{D → 6, A → 11}

Run 3

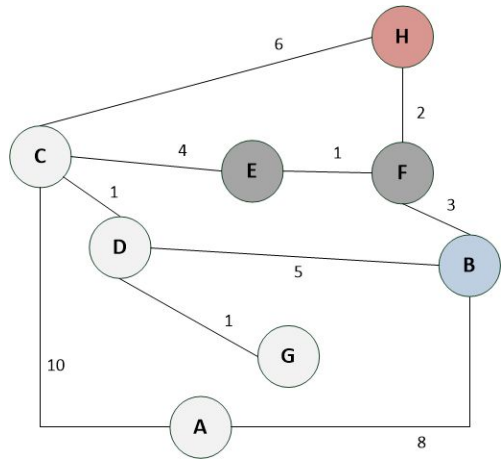
There is a hop in run 3 from router G to E which may seem counterintuitive to the player at the first glance. The remainder of the run is more or less straight forward.



Step	Working node	Priority Queue
0	init	{C → 0}
1	C	{D → 1, E → 4, H → 6, A → 10}
2	D	{G → 2, E → 4, B → 6, H → 6, A → 10}
3	G	{ E → 4, B → 6, H → 6, A → 10}
4	E	{F → 5 , B → 6, H → 6, A → 10}
5	F	{B → 6, H → 6, A → 10}

Run 4

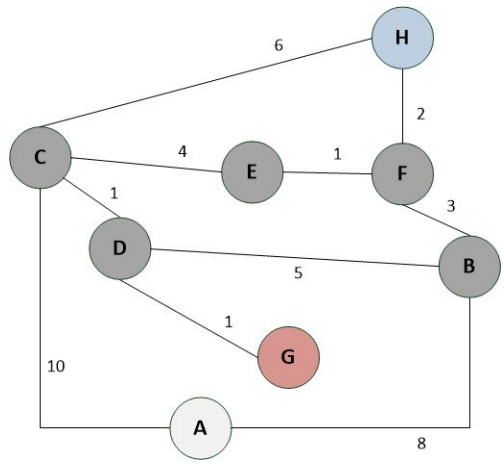
The fourth run is one of the easiest of the level. Instead of heading towards H in step 4, router D could also be visited. However, exploring router H finishes the run.



Step	Working node	Priority Queue
0	init	{B → 0}
1	B	{F → 3, D → 5, A → 8}
2	F	{E → 4, H → 5, D → 5, A → 8}
3	E	{H → 5, D → 5, C → 8, A → 8}
4	H	{D → 5, C → 8, A → 8}

Run 5

The last run of the level includes all nodes except A. However, there is only one possible solution which is shown in the table below.



Step	Working node	Priority Queue
0	init	{H → 0}
1	H	{F → 2, C → 6}
2	F	{E → 3, B → 5, C → 6}
3	E	{B → 5, C → 6}
4	B	{C → 6, D → 10, A → 13}
5	C	{D → 7, A → 13}
6	D	{G → 8, A → 13}
7	G	{A → 13}

2.6 Greedy (Shortest Path First via Heuristic Values)

As an example on geographic routing, we use greedy forwarding. This relies on bringing a message closer to the destination in each step using local information. Every time a new node is chosen, we select the locally best one as our next hop. The locally best one is the adjacent node that has the lowest heuristic value. Normally this algorithm can lead into dead-ends as there might be no neighbors closer to the destination. In this case, we find another node where greedy forwarding can be resumed. We rely on the straight line distance between the nodes to calculate the heuristic values.

2.6.1 Learning Objective

The player should be able to understand why greedy forwarding does not necessarily lead to an optimal path. He should see that the algorithm is simplistic as determining the next node just depends on a simple heuristic value.

2.6.2 Greedy Game Mode

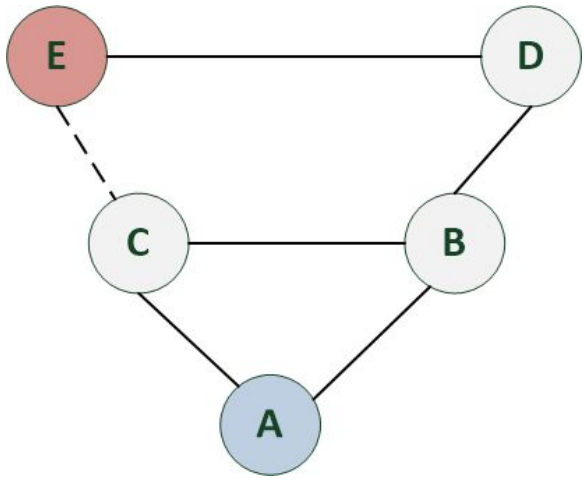
Similar to the game modes before, the user needs to click their way through different runs of the level instead of having only one objective. The player again needs to

deliver beer to specific tables. The destination table is highlighted just like in the uniform cost game mode. The decision on the next hop needs to be made using the heuristic values that are placed on every table. The heuristic value represents the local information indicating the distance of the specific table to the destination. In the picture below, it can be seen that table F has a heuristic value of 13, D has a value of 9 while E has a value of 0. The latter is the case since the table E is the destination of this run.

Points are awarded upon arriving at a target router (+20). Points can be lost when making mistakes. Wrong hops are directly indicated using the score-text pop-up. The player's face is also changed to indicate the wrong hop and there will be no spawning beer puddles. Professor Wahnsinn will show up and tells the player about what should be done instead. To recover from a wrong hop, the player has to go back to his last position.

One specific modification for greedy allows the player to always reach targets. Normally, when using only greedy forwarding, situations could appear that would lead to deadlocks as the player doesn't have a further option to move closer to the target.

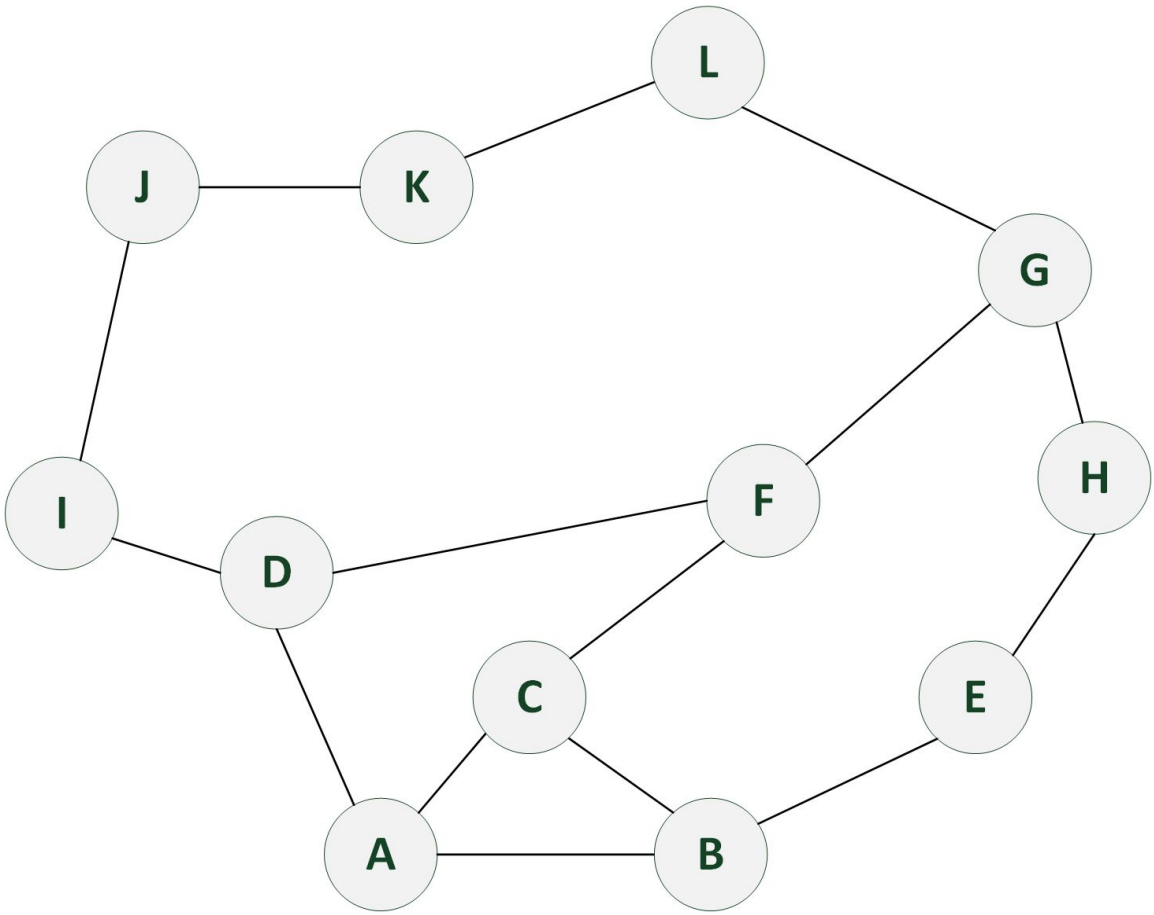
Imagine the following situation: The player is currently at node A. His destination is E. Greedy now expects him to move from A to C. However, there is no direct path from C to E, the dotted line simply indicates that a potential path would be shorter. Without the mentioned modification we, would now be in a deadlock state as C→E would be the natural choice. Nevertheless, the player has to reach the destination and can move from C to B, then to D and, after that, finally to E.



2.6.3 Levels

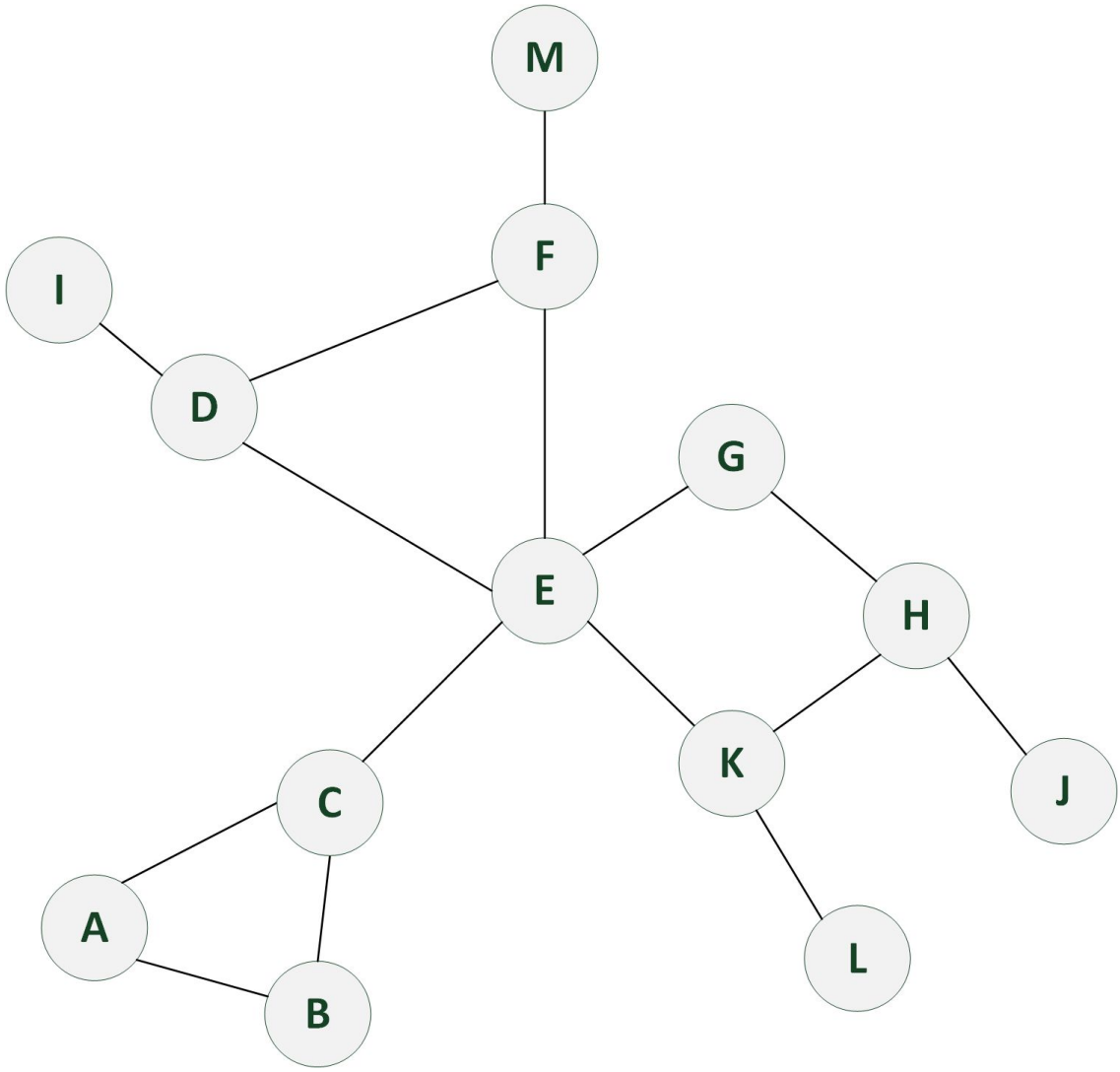
The following section shows the underlying graph structures of the different levels. Furthermore, the start and destination router for each run is presented. As decisions depend on heuristic values which are calculated by the algorithm at run time, these values are not depicted. Sample solutions are not provided for the same reason.

2.5.3.1 Tutorial:



Run No.	**1**	**2**	**3**	**4**	**5**
Start	A	F	B	L	E
Goal	F	B	L	E	D

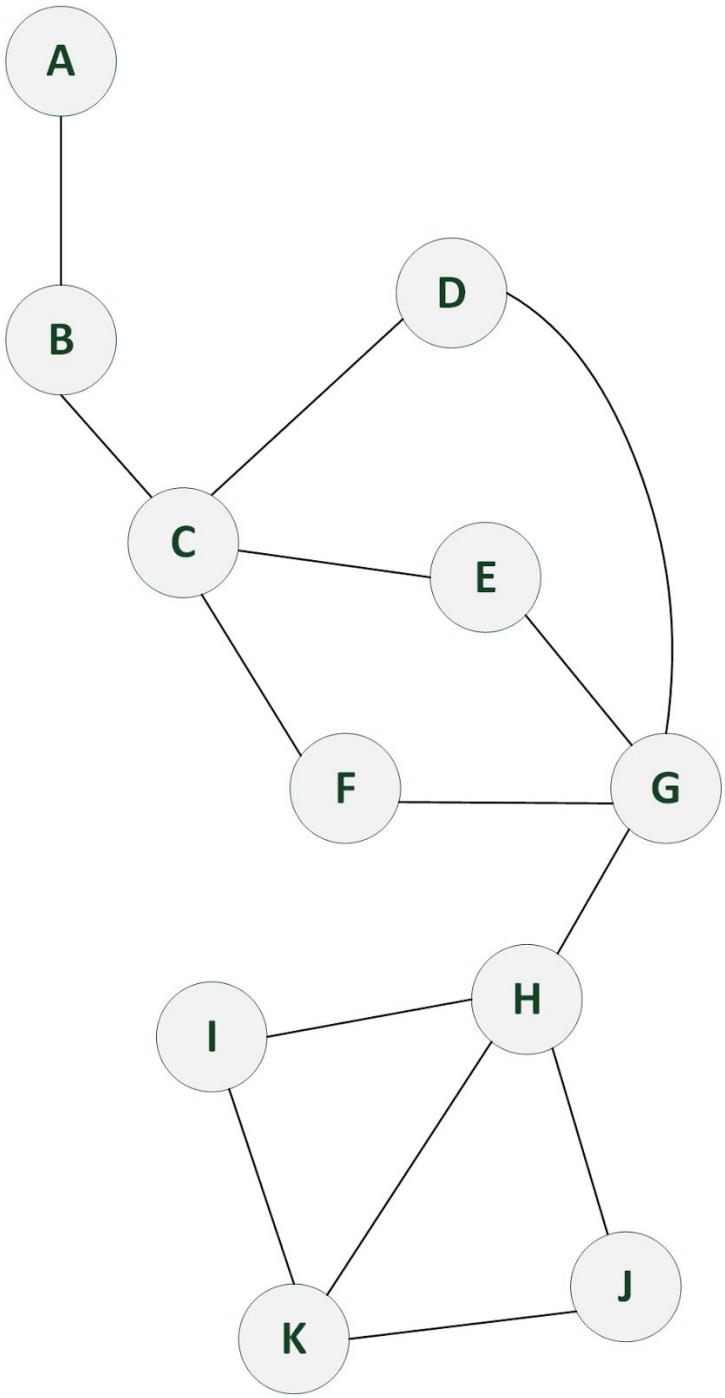
Level 1:



Despite having many connections, the level is straightforward regarding the single runs.

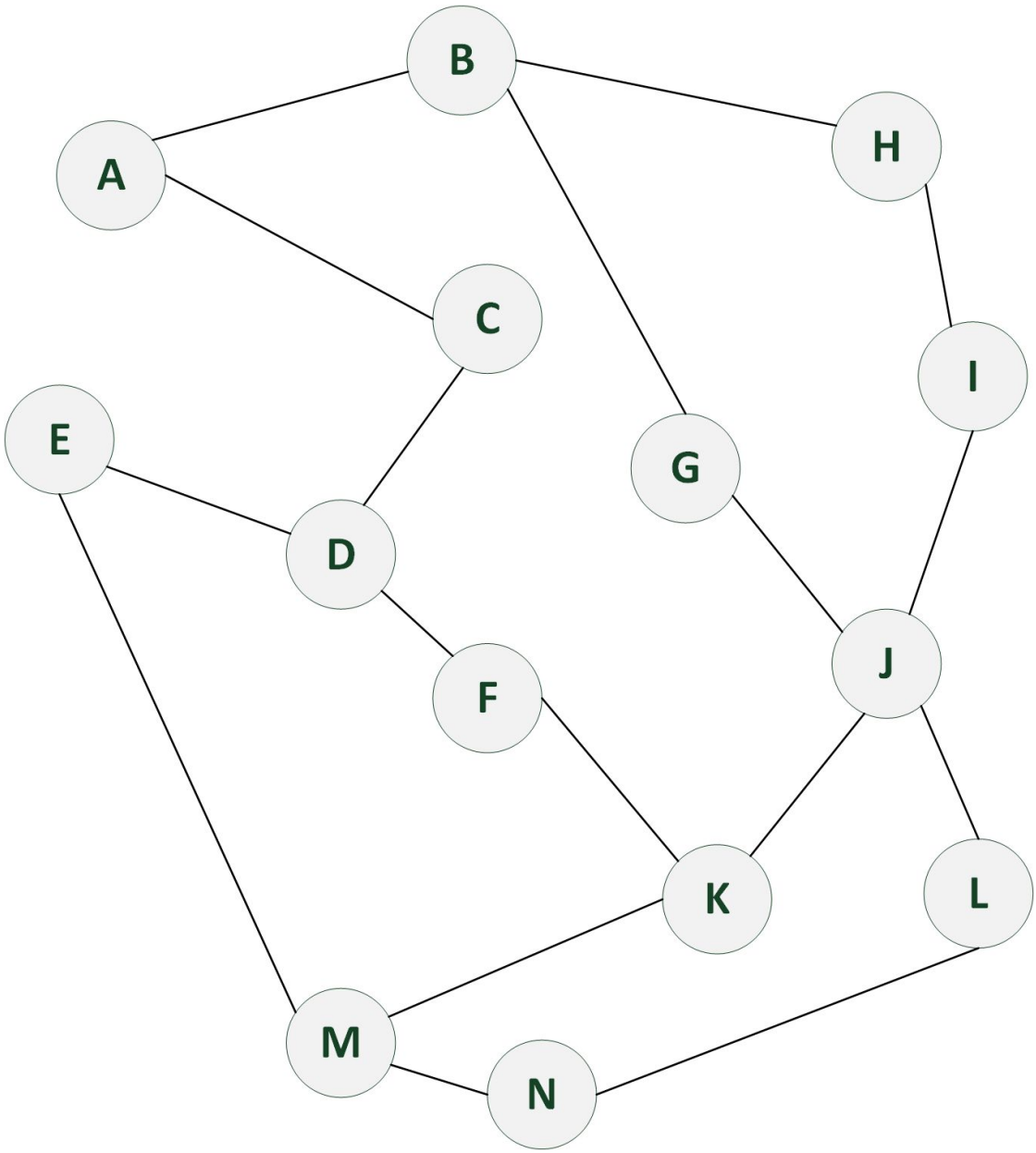
Run No.	**1**	**2**	**3**	**4**	**5**
Start	A	H	M	L	B
Goal	H	M	L	B	I

2.5.3.2 Level 2:



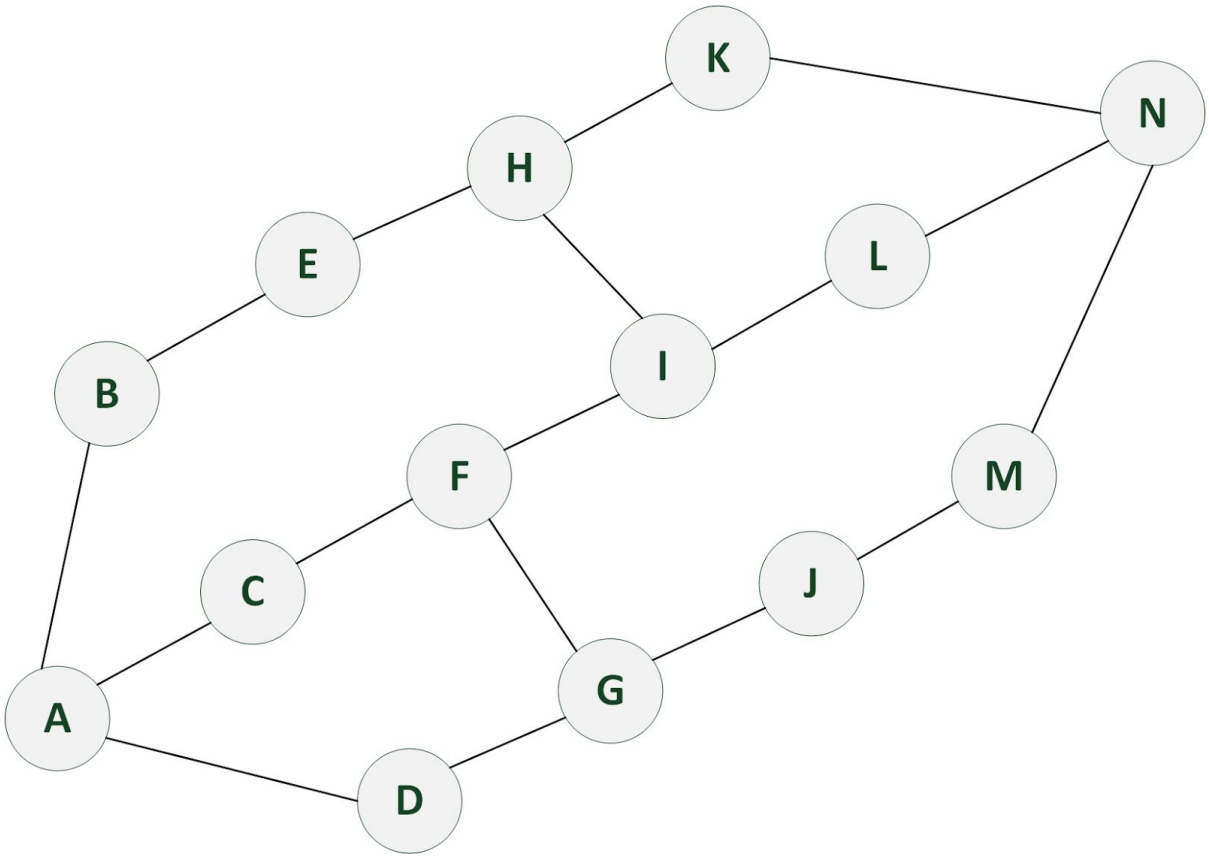
Run No.	**1**	**2**	**3**	**4**	**5**	**6**
Start	A	I	D	J	B	K
Goal	I	D	J	B	K	A

2.5.3.3 Level 3:



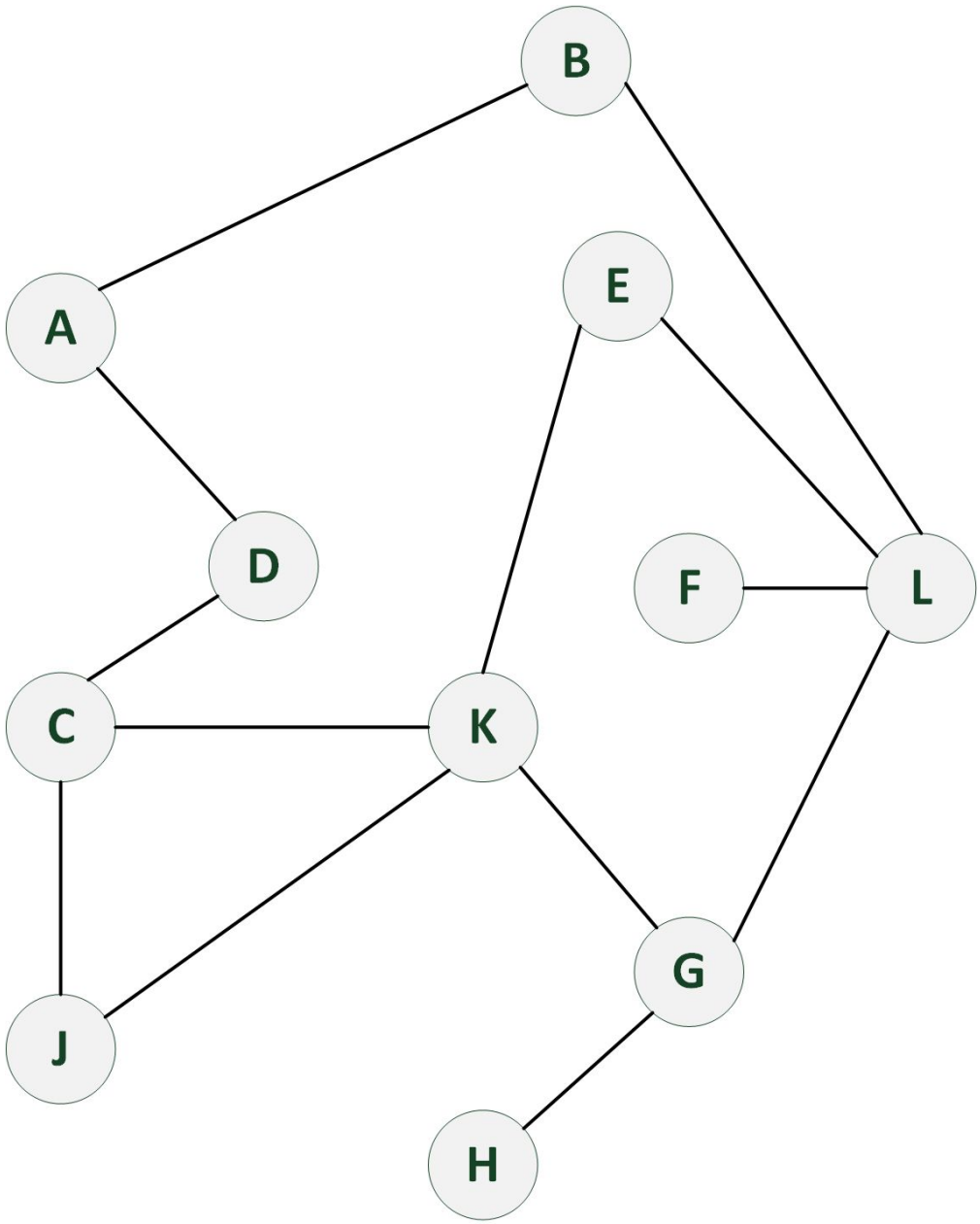
Run No.	**1**	**2**	**3**	**4**	**5**
Start	A	G	E	N	I
Goal	G	E	N	I	D

2.5.3.4 Level 4



Run No.	**1**	**2**	**3**	**4**	**5**	**6**
Start	A	K	J	E	G	K
Goal	K	J	E	G	K	D

2.5.3.5 Level 5



Run No.	**1**	**2**	**3**	**4**	**5**
Start	A	F	D	H	B
Goal	F	D	H	B	K

2.7 Hot Potato Hop Based Routing

For the last game mode, we decided to show a common routing policy, the Hot Potato based routing. A large network infrastructure is intrinsically segmented into smaller sub-networks. Individual networks are managed by network administrators. Different networks are usually managed by different administrators. Those networks can thus be categorized into different administrative domains. To reach their target, packets often pass several of those networks and pass through different administrative domains. Network administrators of one domain can decide that they want to route incoming traffic through their network into adjacent administrative domains as fast as possible. They thus define the policy of sending the packet along the fastest path out of their network towards the packet's destination.

2.7.1 Learning objective

The game mode pursues several objectives. The player should be taught how a routing policy can be defined in order to influence routing behaviour in a network. The game mode tries to point out how such a policy might look like. We decided on the hot potato policy due to its simplicity and intuitiveness. Hot potato can be used with different routing metrics. However, taking the amount of hops as a reference for the optimal path is easy to understand and reasonable. After performing the levels, the player should be able to determine the paths packets would take using the mentioned policy. Furthermore, the player should be taught other aspects. The player should recognize that networks are divided into different administrative domains, so called Autonomous Systems (AS). It is pointed out that the player is solely responsible for his part of the global network. The network topology of his network is known to him while the topologies of networks managed by other administrators are hidden. Nevertheless, packets need to be routed across network borders. The chosen policies for routing between different AS are kept simple in this game. However, the player should be given a hint of how complex routing policies can get in real world scenarios.

2.7.2 Algorithm

Hot potato is rather a routing policy than an actual algorithm. The approach is described briefly in this section.

The hot potato policy tells routers to handle packets as quickly as possible. The goal is to move a packet out of the own network with the least effort and in a fast manner. According to the metaphor, packets should be treated similar to a hot potato. The

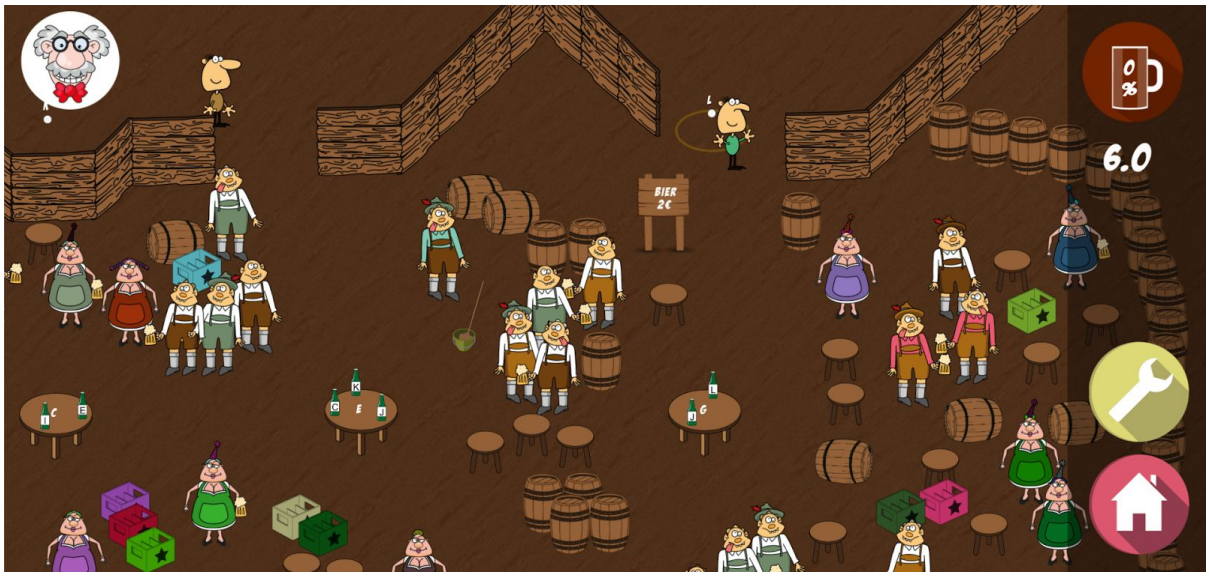
policy can also imply to forward packets on a non-optimal path if the desired output port is blocked by other packages. The advantage of hot potato based routing is the low calculation effort and a quite good network utilization. The traffic with a destination in another network is routed out of the managed network as quickly as possible, i.e. the network is not unnecessarily burdened with traffic that is only passing through. A disadvantage of the approach is that an increasing load of the network leads to non-optimal routing decisions. This could also imply that packets are sent within a circle and pass through the same router several times.

2.7.3 Hot potato game mode

When we started to plan the hot potato game mode, we thought about possibilities to depict different Autonomous Systems and the connections between them. We finally agreed on using the metaphors of rooms. The idea fits to our bar scenario. Professor Wahnsinn has earned a lot of money with his bar. Hence, he has extended it and divided it into several rooms. The rooms are separated using wooden walls. Every room has one or several barkeepers which represent the administrative entities of the AS. There are one or several transition links from one AS to another. In the picture below, a room can be seen which has two barkeepers, one on every transition link between the player's network and the AS.

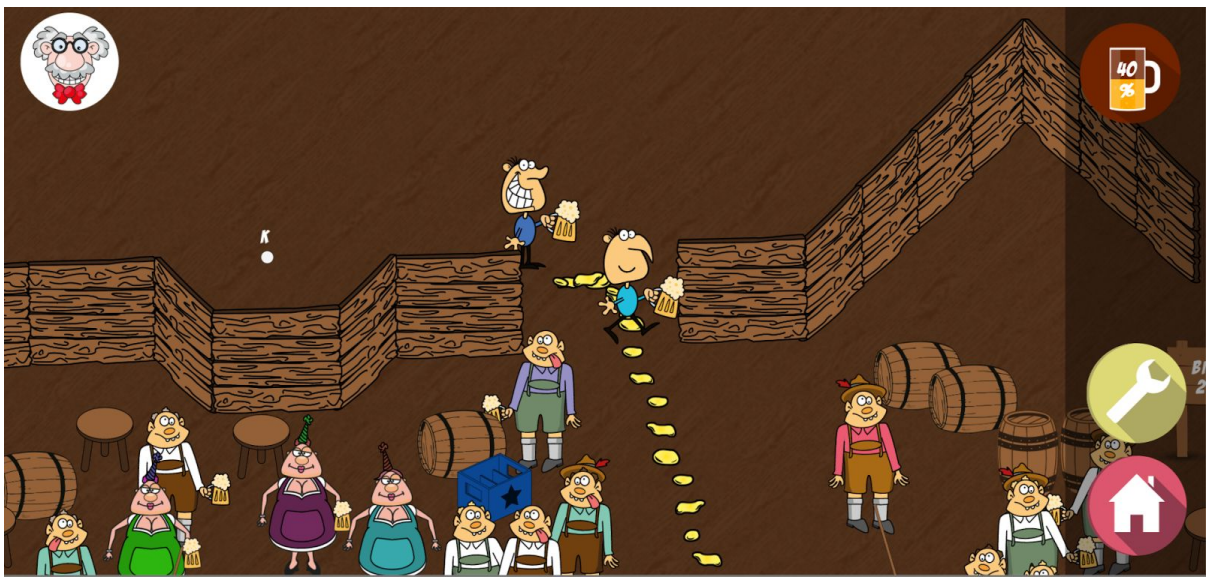
A barkeeper is solely responsible for his room. The network topology of other rooms is hidden for the player. However, the player is barkeeper in his room and knows his network topology. The goal of the player is to deliver beer from his network to other AS according to the hot potato policy. As already mentioned, the hop count is used as a metric for the optimal path. The target room of a run is indicated using a highlight similar to the ones used in the previous game modes. Furthermore, the barkeepers of

the affected AS start to wave. It is worth noting that the beer can be delivered to any of the barkeepers of the affected room, however, usually only one is the endpoint of the optimal path. The picture below shows the highlight of a room indicating the destination of a run.



Like in the previous game modes, a level consists of several runs. Beer is only transported between AS in this game mode, i.e. there are no deliveries to tables in the own network. Since the handling of packets in routers should be done as quickly as possible, we decided to add a timer which restricts the decision time for the next hop. At the beginning of the level, the timer is set to 20 seconds, so that the player can get an overview of the level. However, for every further hop, the player has only 5 seconds to make a decision. On each new run, the timer is set to 20 seconds again to provide enough time for an initial overview of the situation for this run. The timer is shown below the score beer as can be seen in the above picture. If the timer value is expired and the player has not performed an action, the system randomly selects a next hop and performs it. Professor Wahnsinn will appear and advise the player to decide faster next time.

A difference to the other game modes is that there is no direct feedback on wrong hop decisions. The player can perform hops in the level completely free of any restrictions. We only count how many hops he requires until he finally reaches the destination AS. This number is then compared to the amount of hops our algorithm implementation determined for the optimal path. For every unnecessary hop the player is fined with 5 minus points. A maximum of +20 points can be obtained for the optimal path. However, the player can also get 0 points if he has required too many hops. The achieved points for a run are shown when the run is finished, i.e. the player arrives at the destination AS. When arriving at the AS, the beer is passed to the barkeeper waiting at the transition link to indicate the transfer. This is shown in the picture below.



The game mode is explained to the user in a separate tutorial level. Just like in the other game modes, there are five levels which are described in the following paragraph.

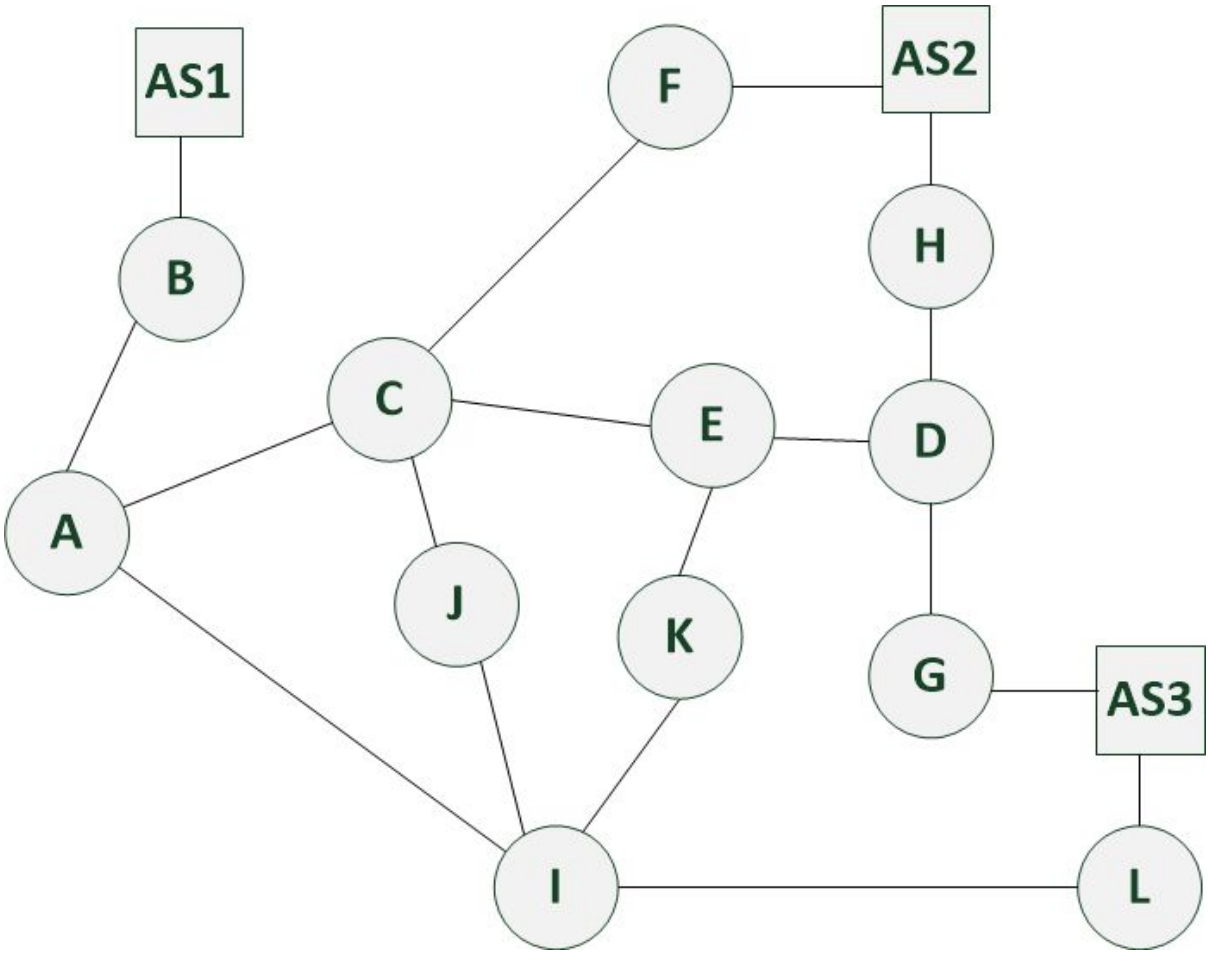
2.7.4 Levels

The criteria for the difficulty of a level in the hot potato game mode are rather simple. The size and clarity of the level are the crucial properties in this context. The player needs to make quick decisions which is rather difficult in a large and confusing level. A second criteria is the possibility of several paths leading to an AS. This makes the decision process of choosing the optimal path more difficult. The game mode is also quite appropriate to install dynamic events like a topology change due to a router failure. Dynamic events are described in a later section.

2.6.4.1 Level 1

The first level consists of four runs. The player starts at router I and has to perform the four deliveries subsequently. There are three different AS that can be reached via the player's network. The topology is depicted below.

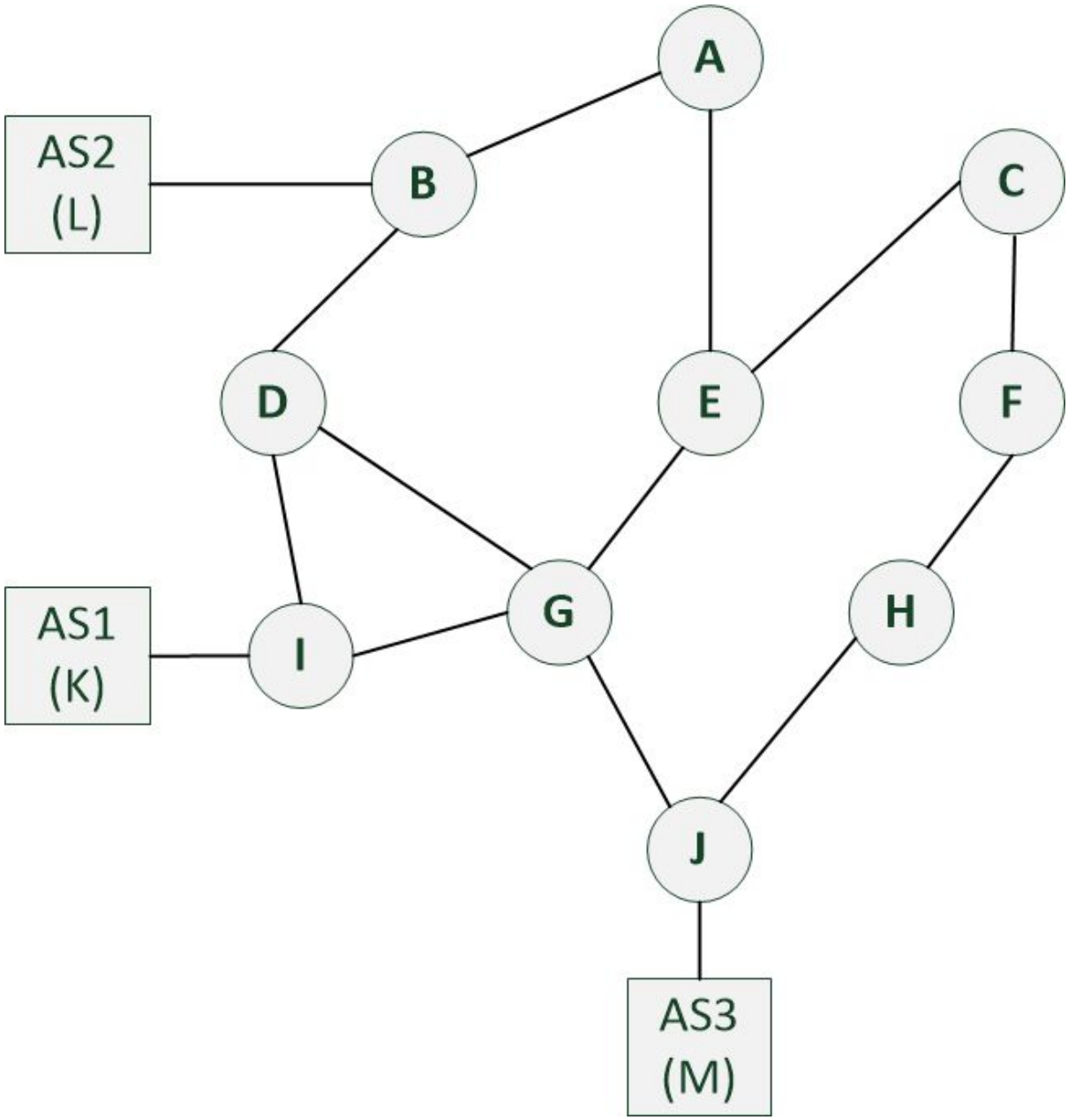
The first target is AS2 and the optimal path is I-J-C-F. If the player chooses the path I-K-E-D-H, he will only be rewarded with +15 points. The next target is AS3 and, of course, the optimal path now depends on whether the player is located on F or H regarding the previous run. Our implementation handles this by always calculating the optimal path depending on the player's current position. If the player is located on router F, the optimal path will be F-C-E-D-G or F-C-J-I-L since both require the same amount of hops. The third run leads to AS1. After finishing this run, a dynamic event will take place and the router C drops out. The destination of the last run is AS2 and the player now needs to find a new way in the adjusted network topology. In this case, the optimal path is B-A-I-K-E-D-H.



2.6.4.2 Level 2

The second level also comprises four runs. The player is located at router A at the beginning. There are three different AS and all are connected to the player's network via a single link. The topology is depicted below.

In the first run, the player needs to walk to AS3. The shortest path is straightforward, namely A-E-G-J. The next destination will be in AS2. There are several paths to AS2, but the shortest is J-G-D-B. The next run is a short one. The target AS1 can be reached via the path B-D-I. However, a dynamic event will deactivate router G after the player has arrived at AS1. In the fourth run, the player needs to go to AS3. Instead of the easy path I-G-J, the player now needs to walk a long path, namely I-D-B-A-E-C-F-H-J. He thus passes through all routers of the network.

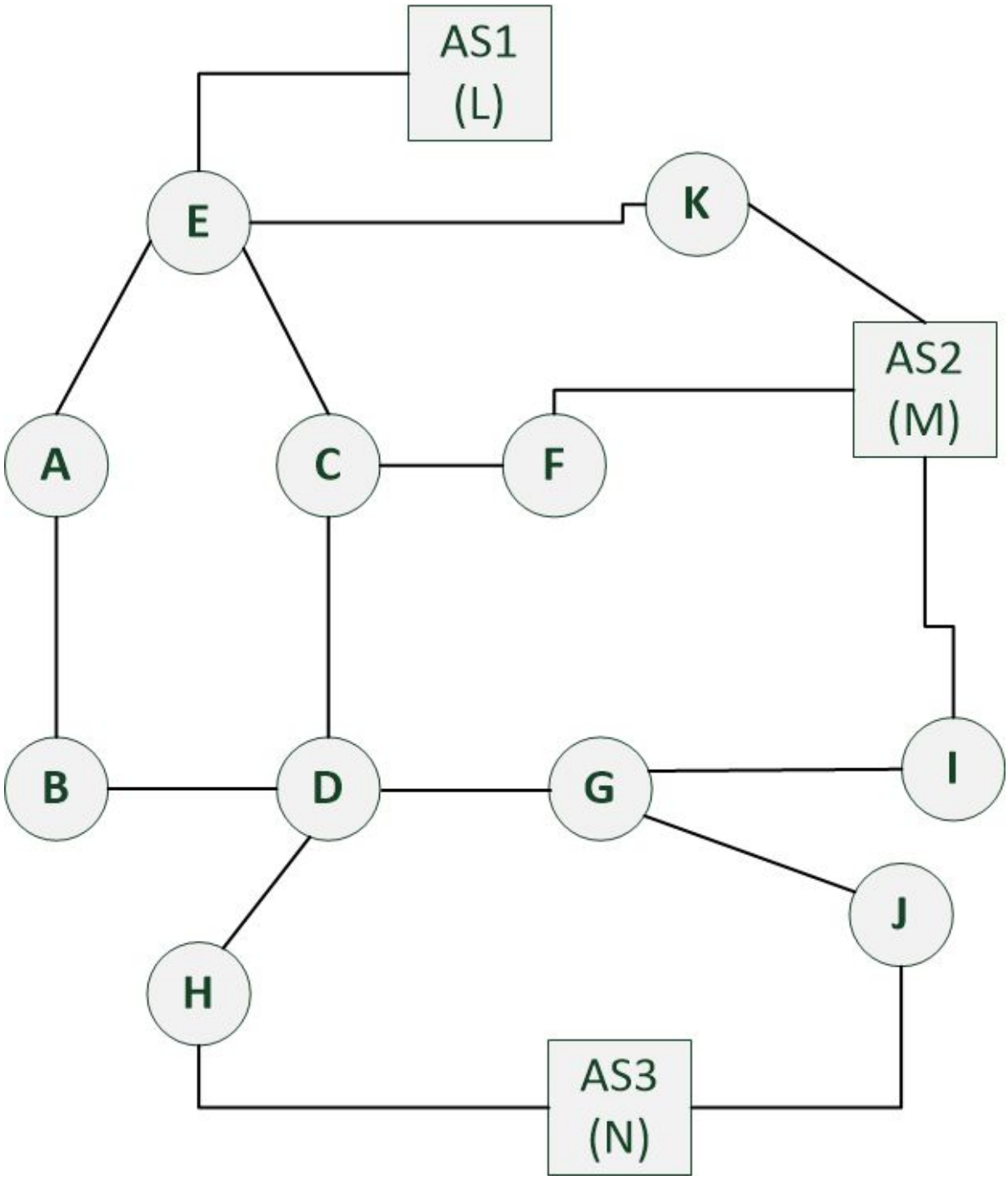


2.6.4.3 Level 3

In the third level, there are three AS connected to the player's network. It is notable that one AS can be reached via three links. The player is located at router A at the beginning. The topology of level 3 is shown below.

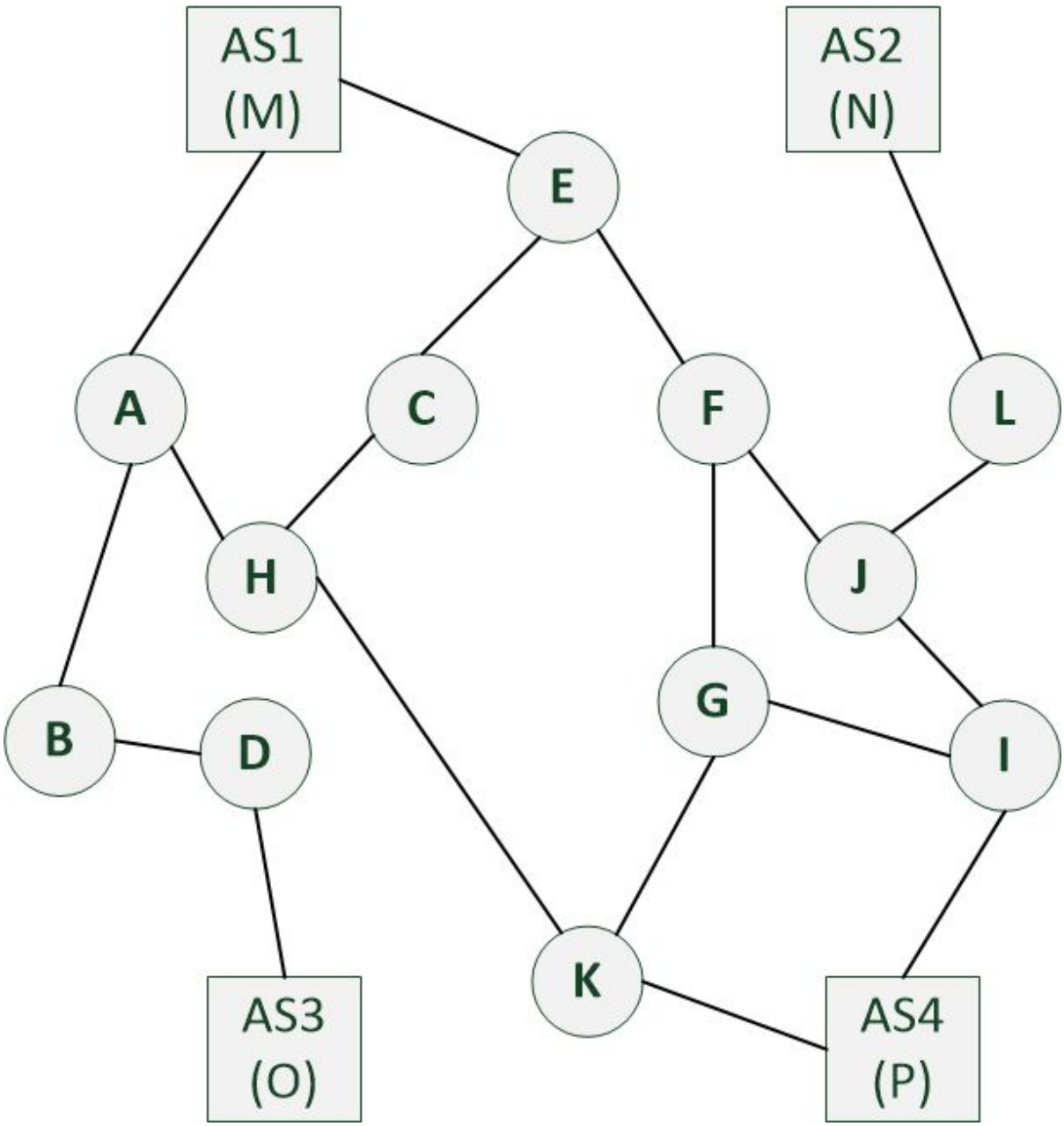
The player needs to perform five runs. The first will lead from router A to AS3. The shortest path is A-B-D-H. In the second run, the target is AS2. There are two optimal paths for this run, assuming the player is located on H and not on J. The possible paths are H-D-C-F and H-D-G-I. This has impacts on the third run. The target of the third run is AS1. If the player is located on router F, then the optimal path is F-C-E. If the player is located on router I, then the path is I-G-D-C-E. The next run leads to AS2

and is quite short. The optimal path is E-K. In the last run, the player needs to walk to AS3 and the optimal path for doing this is K-E-C-D-H. No dynamic event will take place in this level.



2.6.4.4 Level 4

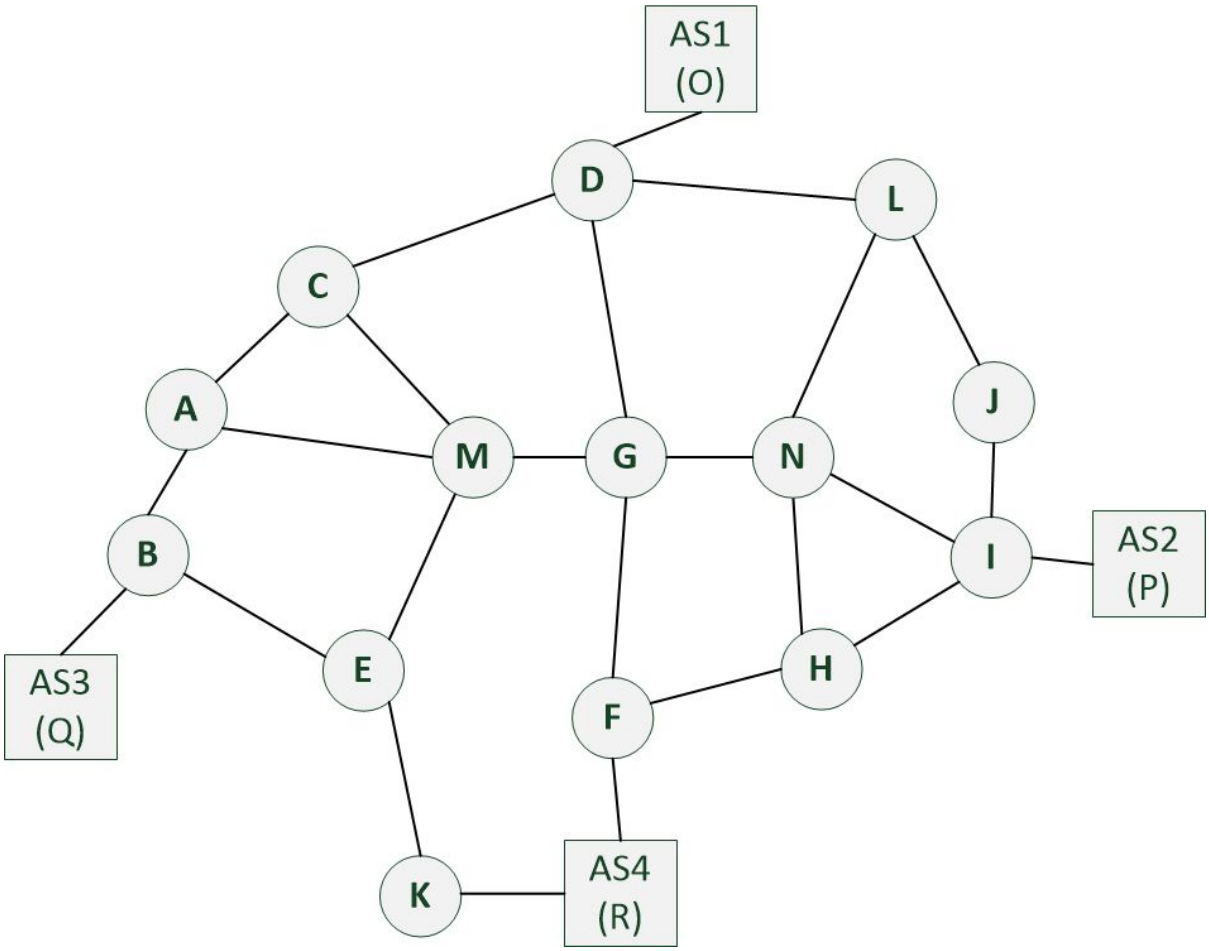
The fourth level has four connected AS. The player starts at router A. Five runs need to be completed in order to finish the level. The network topology is depicted below.



There are several possible solutions for the first run of the level. The target is AS2. The paths A-H-C-E-F-J-L, A-H-K-G-F-J-L and A-H-K-G-I-J-L comprise the same amount of hops and are thus all considered to be optimal. This makes the first run rather difficult to solve. However, the second run is quite easy. The player needs to walk to AS4. The optimal path is L-J-I. Assuming the player is located on I, the next run can be solved with an optimal path I-G-F-E to AS1. Also path I-J-F-E is possible. The target of the fourth run is AS3. There is only one optimal path, namely E-C-H-A-B-D. Before the fifth run, a topology change takes place as router F drops out. To get to AS2 in the last run, the player can take the optimal path D-B-A-H-K-F-I-J-L.

2.6.4.5 Level 5

The last level is the largest of all levels of this game mode. It consists of four AS and 14 routers. The topology is shown below.



The player is located on router A at the beginning. The first run leads to AS2. The optimal path is straightforward, namely A-M-G-N-I. The target of the second run is AS1. There are several optimal paths for this run. The player can choose I-N-G-D, I-N-L-D or I-J-L-D. These three paths result in full credits for the run. The third run is relatively simple again. The player needs to walk to AS4. The optimal path is D-G-F. However, the level gets more difficult as router G drops out after the third run. The next destination is AS3. The optimal path is now F-H-N-L-D-C-A-B. The last run leads to AS2. The optimal path for this run is B-A-C-D-L-J-I. Instead of router J, the player could also choose router N.

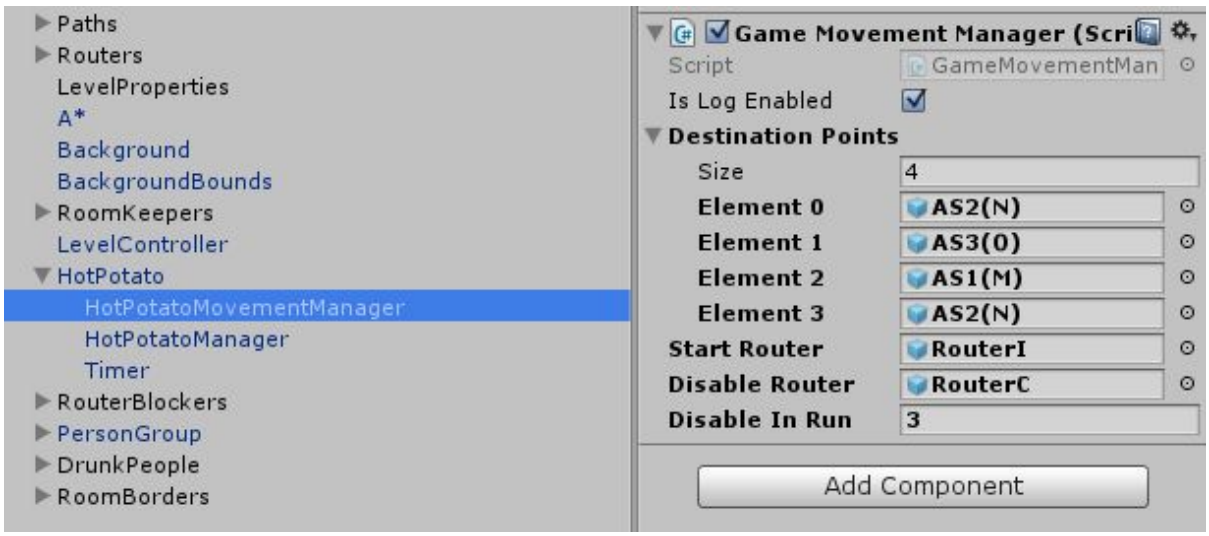
2.8 Dynamic Events

After finishing the game modes, we started to implement dynamic events. We came up with two different types of events that symbolize problems that can occur while routing in networks. All events are introduced to the player via one in-game tutorial by our professor.

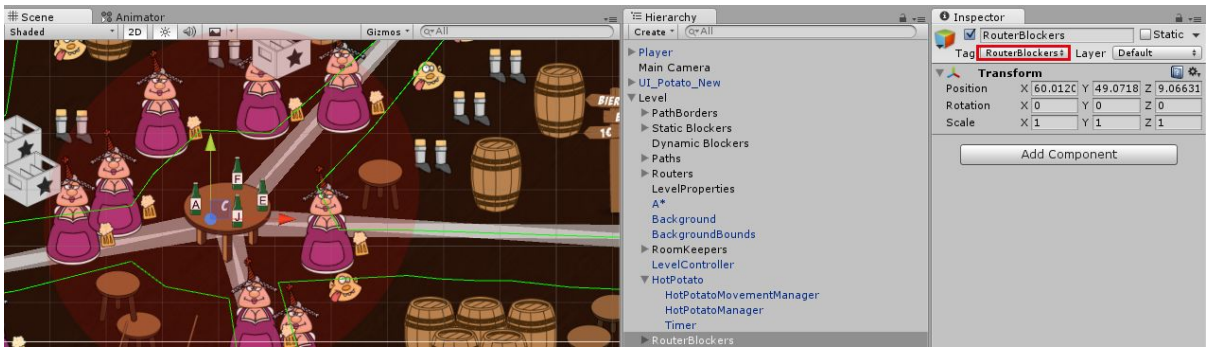
2.8.1 Router Malfunctions

This dynamic event is presented visually by showing a large group of people around a table. This symbolizes the absence of paths to the specific router and alters the paths the player can use to get to the target. In reality this might make routing more difficult as routes may be blocked permanently. In our case, we select the malfunctioning router carefully to keep all routers in the network reachable. When a router malfunctions it won't affect the possibility of finding a solution for the given situation. The player should always be able to complete a level.

The event that spawns people around the tables is triggered on a specific level. It can be used in any game modes except for Dijkstra's algorithm.



To enable this feature, it has to be declared in the specific MovementManager object. We then specify the router to disable and an index indicating the run in which the persons should spawn around the table. In the case of the screenshot, C will be blocked starting with the third run.
The blockers are realized using a group of objects that have one parent gameobject with the tag "RouterBlockers". If the router malfunction event is triggered, these objects spawn in the level. This allows easy adaptation if a new visualization would be preferable.



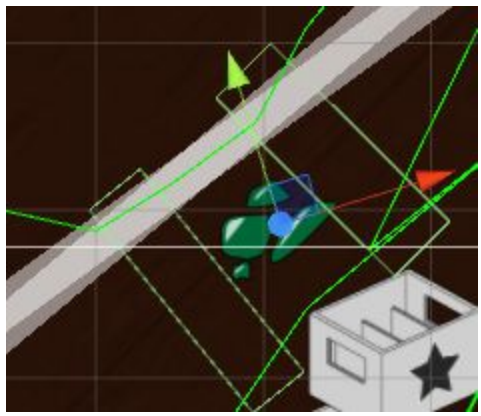
In the first level that contains this type of dynamic event the professor will notify the player that there is no longer a router through the area as too many people block paths.

2.8.2 Unsafe routes

Additionally we thought of a way to map unsafe routes. The game takes place in a bar, mistakes happen so we visualize those unsafe routes by glas shards from broken beer bottles that lay on the ground. The player then has to jump over those each time he needs to pass the route.

When the player enters one of the colliders a quick time event is triggered. This in turn checks whether the player jumps before reaching the glass shards. When the player manages to do this, both colliders that enable the event are disabled until the player reaches his destination. In the game a small help in the form of a graphic overlay will be shown as soon as the event starts.

When the player fails the event, he returns to his last router to get a cached version of the message and has to try it again. It would be even possible to enable the shards in a later run in one level. Until now this is not implemented.
Usage is simple, just drag the glass shard prefab onto the playing field and rotate it. If necessary adjust the two box collider children.



3 Development Process

This chapter describes the development process and milestones we deemed important enough to mention.

3.1 Project management

We opted for an iterative development process, that allowed us to integrate or discard features we planned. Our plan then was to meet at least once per week to present our features and merge those in one working Unity project. We first started using Git but quickly noticed, that a shared drive containing the unity packages was a smoother way for us to develop together.

We used a simple Excel sheet to manage the outstanding dues. In the project meetings, we worked together on complex problems which required a lot of discussions and decisions. At the weekly task allocation, we decided who will work on specific tasks which could be processed by one person individually.

Each team member worked on their tasks and presented the results at the next meeting. Then the merging process started all over again.

Below you can see a small excerpt of the task allocation sheet.

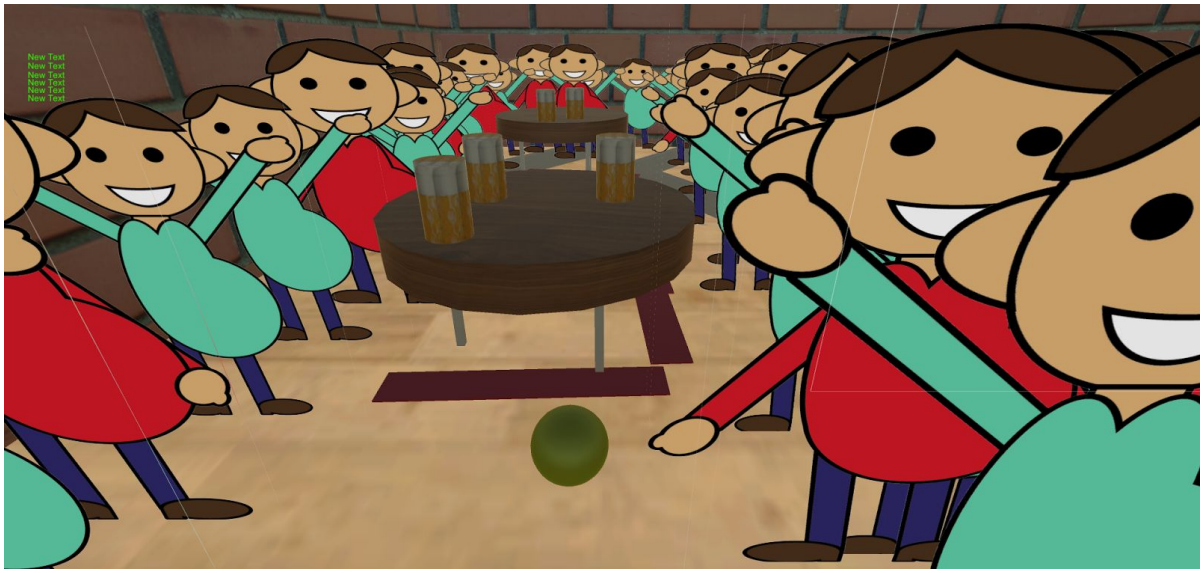
	A	B	C	D	E	F	G
54	**Treffen**	**Problem**	**Matthias**	**Jonas**	**Philipp**	**Fabian**	**Fertig**
55	11.4.	Level füllen (Personen, Hocker, Fässer, etc) -			x		x
56		Prof Hot Potato	x				x
57		Dynamische Elemente (Scherben)			x	x	x
58		Springanimation	x				x
59		Hotpotato Timers fixed (stopped when prof is			x	x	x
60		Spielmodi testen (1-2, 3-4)			x	x	x
61		Neue Barelemente (siehe Signs package in DB)	x				x
62		#if UNITY_EDITOR // #endif vor alle				x	x
63		Prof Mute in den Einstellungen	x	x			x
64		DOKUMENTIEREN (GDOCS)	x	x	x	x	
65		Hilfescreen aktualisieren (Player, Person	x	x			

3.2 Early prototyping

After starting the project we decided that we should split up while learning the basics of the Unity engine. Thus we created three different prototypes. First we could not decide whether a two dimensional game would suffice or if the game would gain benefits from having a third dimension. Thus we created two prototypes with the

same basic principles and rules but different means of movement and underlying systems.

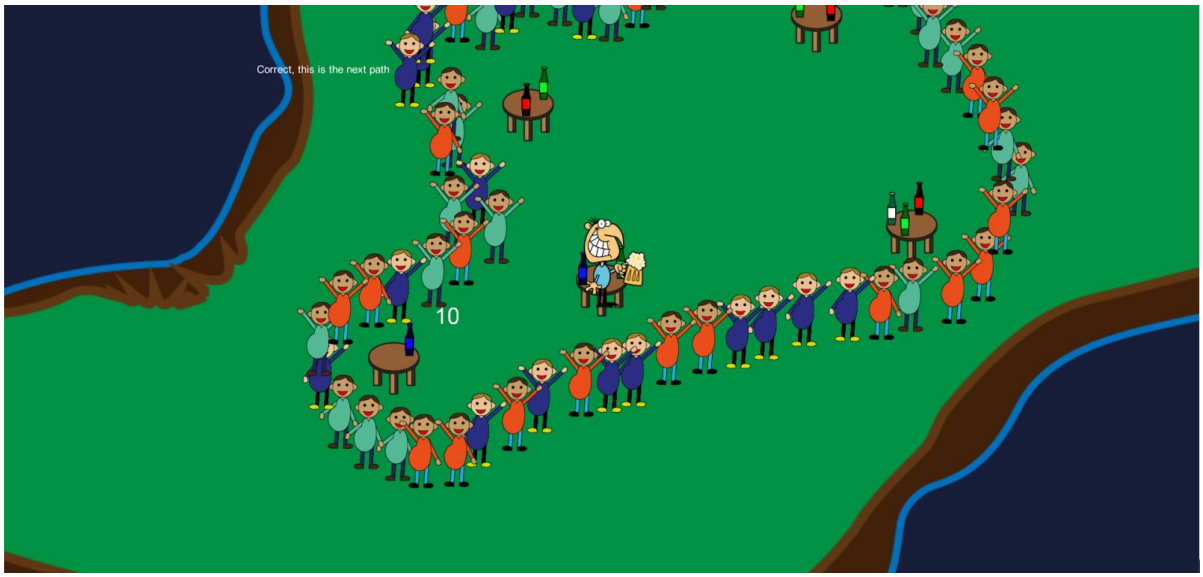
3.2.1 3D Prototype



The screenshot above is taken from the 3D Prototype. The yellow sphere simulates the player and is controlled by the arrow keys of the keyboard. To check which table is reached we used game objects - the fat dark brown strokes on the ground. These game objects trigger events when the user passes. By these events it is possible to locate the player and derive which way he went.

3.2.2 2D Prototype / Dijkstra Prototype

At the same time, we tried making a mouse-movement based prototype that allowed the player to see the level from above. Movement was then in turn coded to be a click on the beer bottles, where the color decided the way. Furthermore, we added guests as blocking elements for the movement. This would in turn be used to shape paths between the tables. The number displayed shows the costs of a given path and it appeared when a table was clicked, to mimic the discovery process of Dijkstra's algorithm. The player was animated and had an idle and a walking animation that adapted to the speed of the object.



3.2.3 Merging the projects

We quickly decided that our game does not benefit from the third dimension as it distracts the user from the core of the game. As we would need to provide a means for the user to see more of the playing field than just the narrow sight our prototype offered to make decent routing decisions. Additionally it was easier for us to create our own graphics for a two dimensional game world.

We then merged the Dijkstra prototype with additional things we had tested in our 2D prototype and added more features, such as error recognition and recovery as well as A* pathfinding. The pathfinding allowed us to have a more realistic way of moving and to disable movement in areas we want to keep blocked. Additionally, we allowed clicking bottles on either side of the path to go to the other router.

3.3 Graphics

We had the intention to create a nice looking game where the users won't get distracted by bad graphics. For this, a consistent style and color scheme is necessary. To achieve this, we decided to draw almost every graphic in this game by our self. It also holds the advantage that animating objects is easier.

Thinking about the large range of our potential target group and what style they could like, we decided to make use of a comic theme. Using a soft color scheme and flat design for ui elements gives it a modern touch.

For creating images we used Adobe Illustrator which adds the benefit of flawless scalable graphics. After adapting images to the perfect size, they can be exported as .png files with a perfectly suited resolution - this saves memory and raises performance.

In the following figure you will find some button graphics as an example.

As you can see, a button has four different states, these are from top down: Initial, highlighted, pressed, disabled. To follow design principles the initial-state has a slight shadow which gives the user the affordance to push the button. When hovering the shadow disappears and by pressing it gets an overall shadow, which gives the user a visual feedback. The disabled state is visualized by graying-out and reducing the brightness of the buttons' colors. This is a commonly used constraint for UI design.

3.3.1 Styleguide for buttons

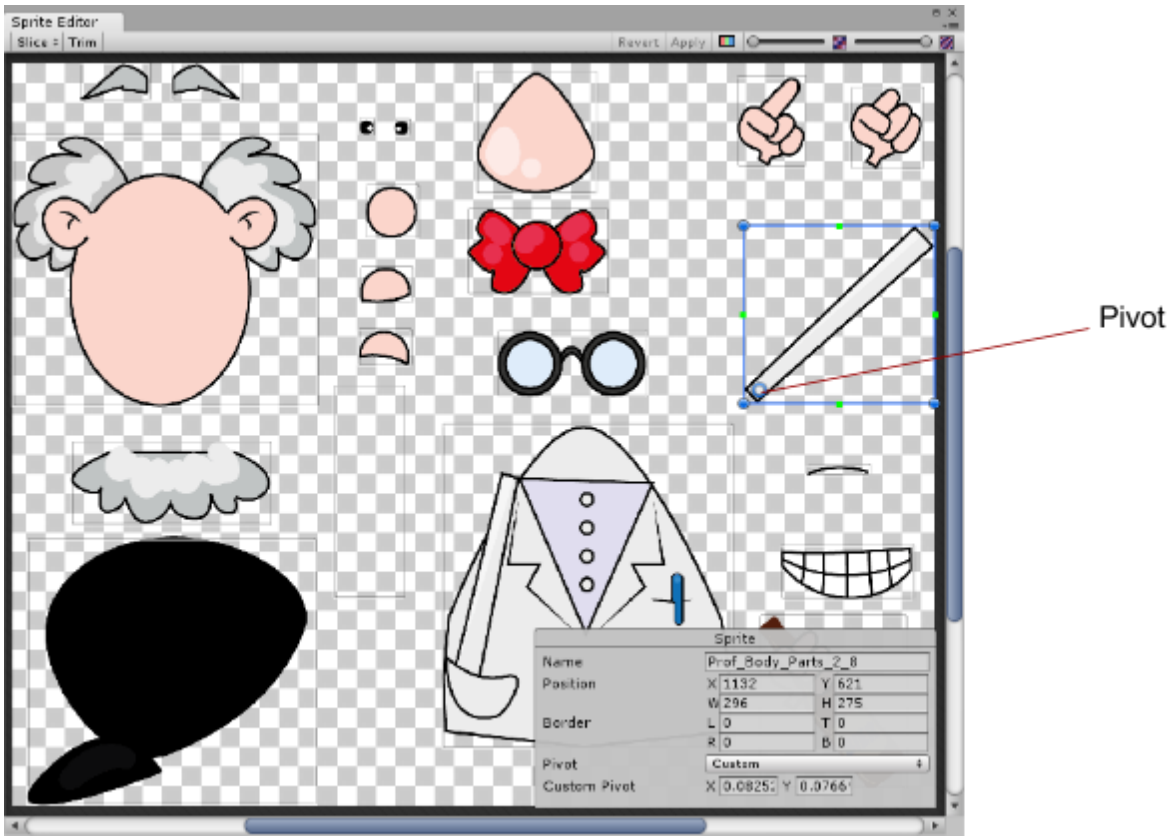
Buttons always taking a soft color as background with a narrow shadow on the left side. In its middle there is a white icon with a long shadow at an 45 degree angle. The edges of shapes are always radiused.

3.4 Animating

There are basically two ways of animating 2D sprites. First, animating with sprite sheets. With this method you create several animation states with any graphics

program and save each image. Then in code you only have to play one frame after another. The Second option is bone-based animation. Here you split your game elements into pieces. You are able to manipulate (e.g. rotate) each piece individually. In code you must define how the pieces should be manipulated over time to create an animation.
In Unity, you do not have to actually code the manipulations of the graphic pieces. The editor provides a convenient way of doing this.

We decided to use the Unity animation editor since it is convenient and the bone-based animation is more flexible and doesn't consume as much storage as the sprite sheet animation.

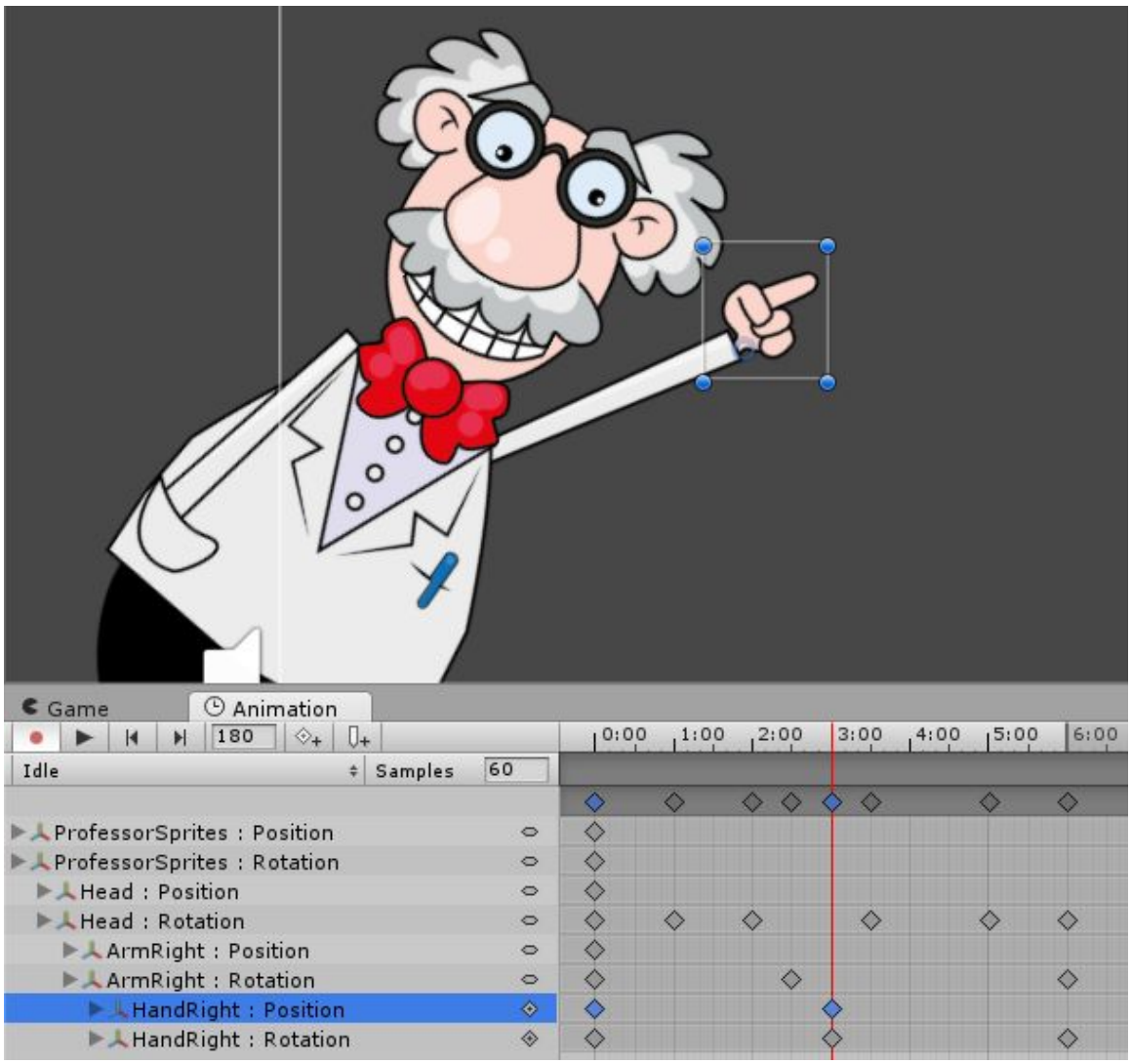


Actually, we didn't use a real bone-based animation. To create real bone-based animations special programs are required. These programs create a skeleton for the character which is used to animate it. Since these programs aren't free to use and we didn't require this kind of complex animation feature, we used the Unity animation editor.

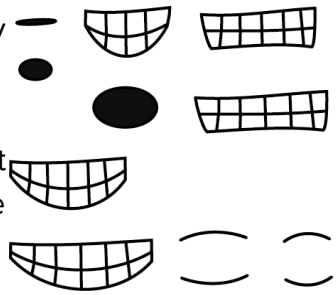
In Unity, you manipulate the single graphic pieces which are not part of a skeleton. To do that, you have to think about how to slice your character. Each moving part has to be a separate image. Another important point is to place the image's pivot point at the

correct place. This point defines how the image is rotated and scaled. In the image above, the pivot point of the arm is set to the shoulder. The arm will be rotated around this point.

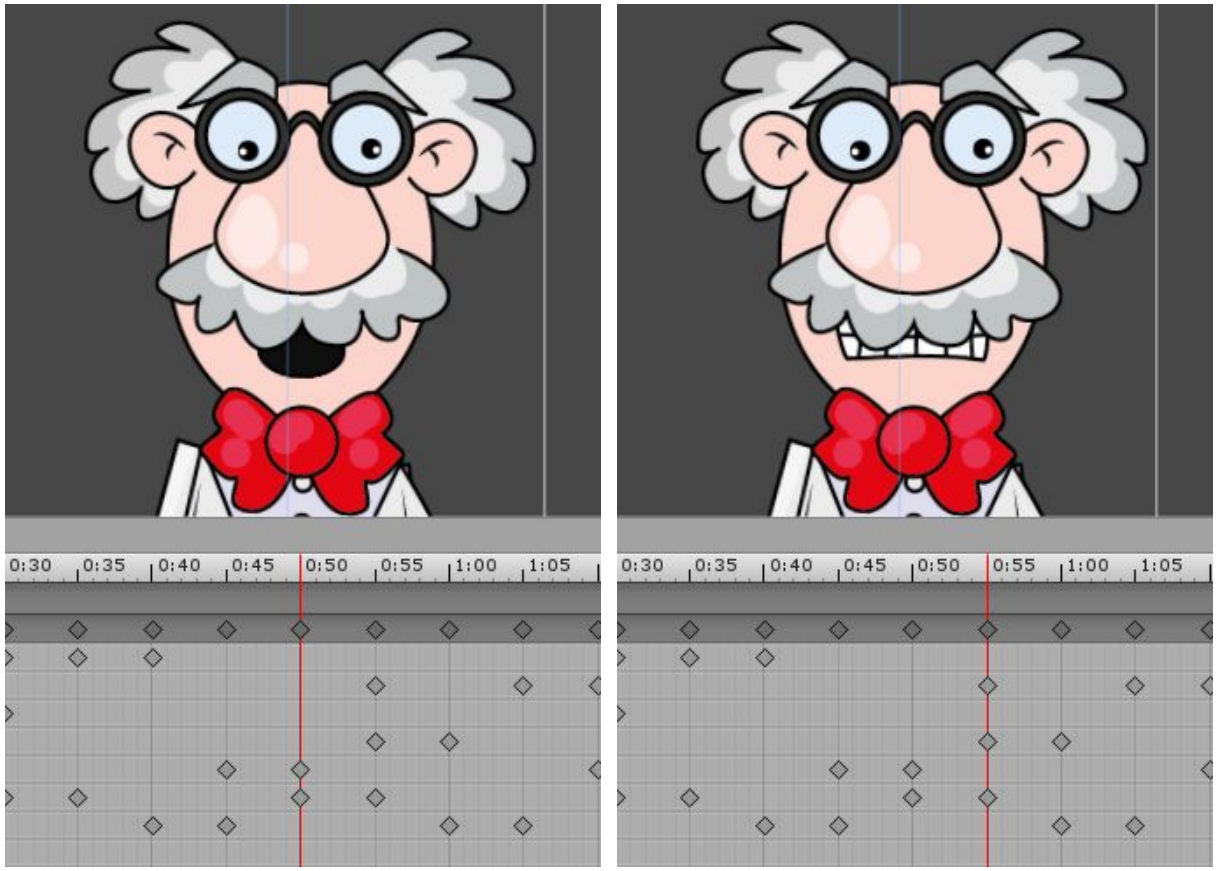
The sliced images are combined in a game object. First, you need to place each piece at the correct position to compose the original image. Then you're able to set keyframes for each piece. A keyframe defines the image's position, scale and rotation at a certain time. To create an animation, you set a keyframe at a certain time and another keyframe at a distinct time with different properties. The images properties (e.g. rotation) will change over time from one keyframe to the other.



The professor's speaking animation was created by quickly showing and hiding different mouth images.



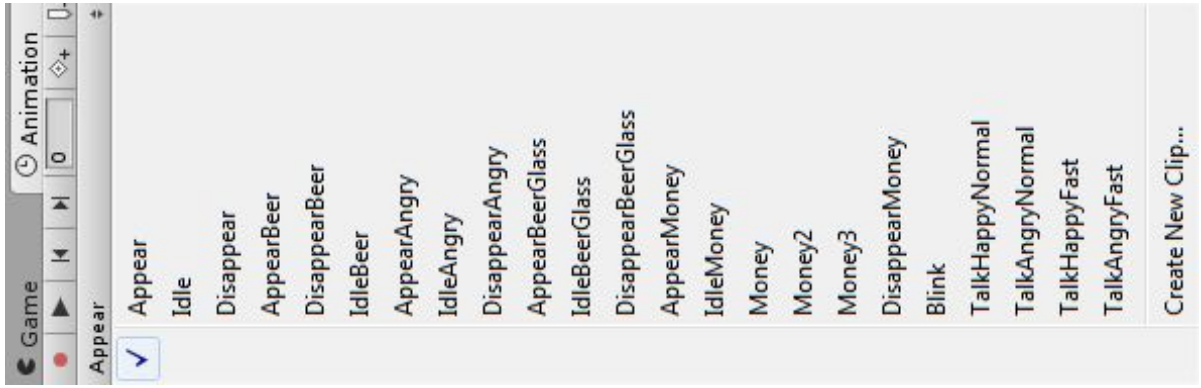
Each keyframe defines which mouth image is shown at that point. To create a relatively smooth talking animation we changed the mouth image every 5 frames.

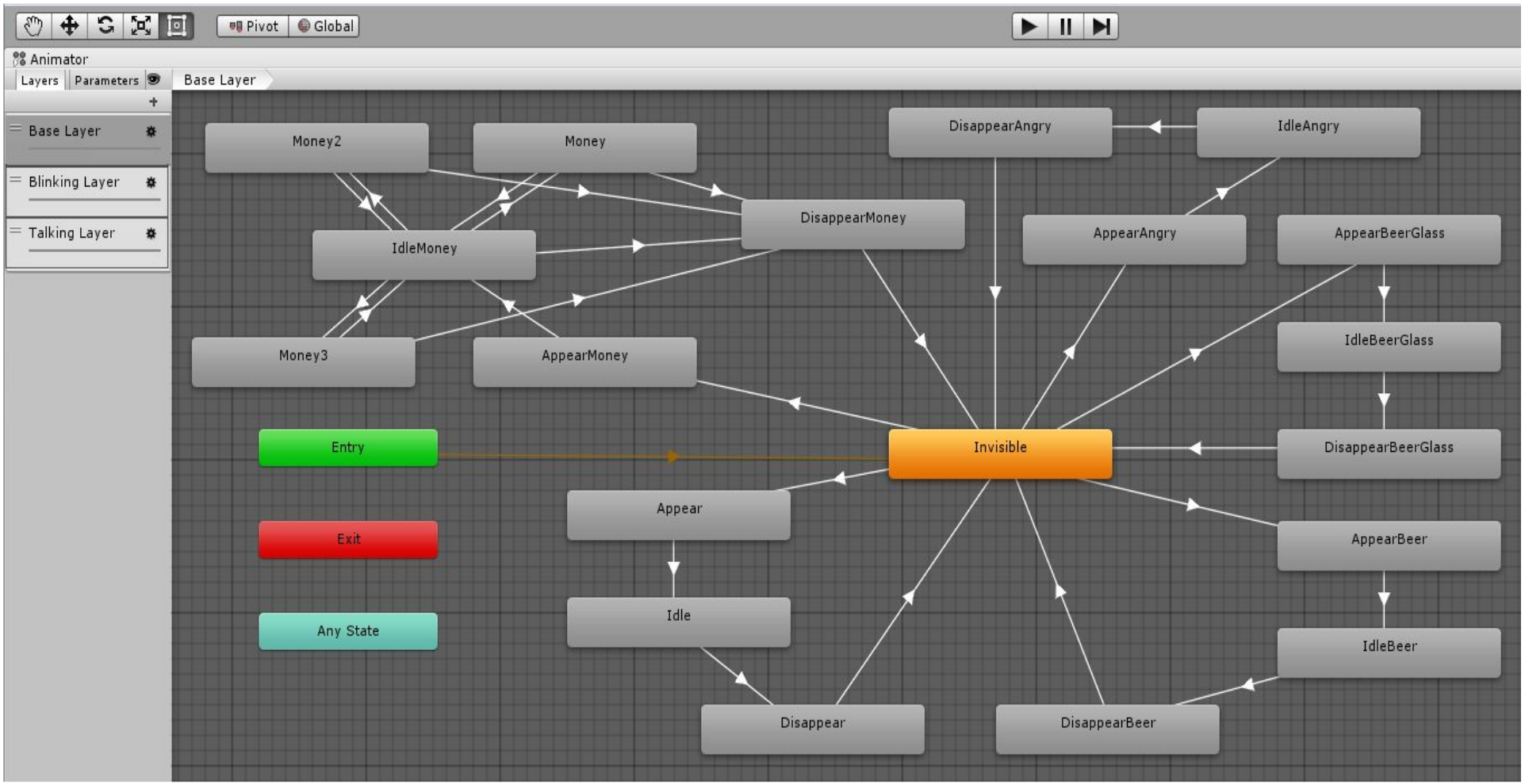


To control the different animations and transition between them, Unity uses an animator. The animator shows all animations and the transitions.

Animation layers are used to play animations above other animations. We need this to play the talking animations while the professor is already playing the idle animation. Otherwise, we would have to set each talking frame in each animation in which we wanted to use the talking animation. This would be very inconvenient. Multiple animation layers are also used for the professors random blinking animation.

On the next page you can see the professor's animator states of the base layer. On the left you can see the two other animation layers for talking and blinking. The second image shows all professor animations.


Animation
Game
Appear
Appear
Idle
Disappear
AppearBeer
DisappearBeer
IdleBeer
AppearAngry
IdleAngry
DisappearAngry
AppearBeerGlass
IdleBeerGlass
DisappearBeerGlass
AppearMoney
IdleMoney
Money
Money2
Money3
DisappearMoney
Blink
TalkHappyNormal
TalkAngryNormal
TalkHappyFast
TalkAngryFast
Create New Clip...

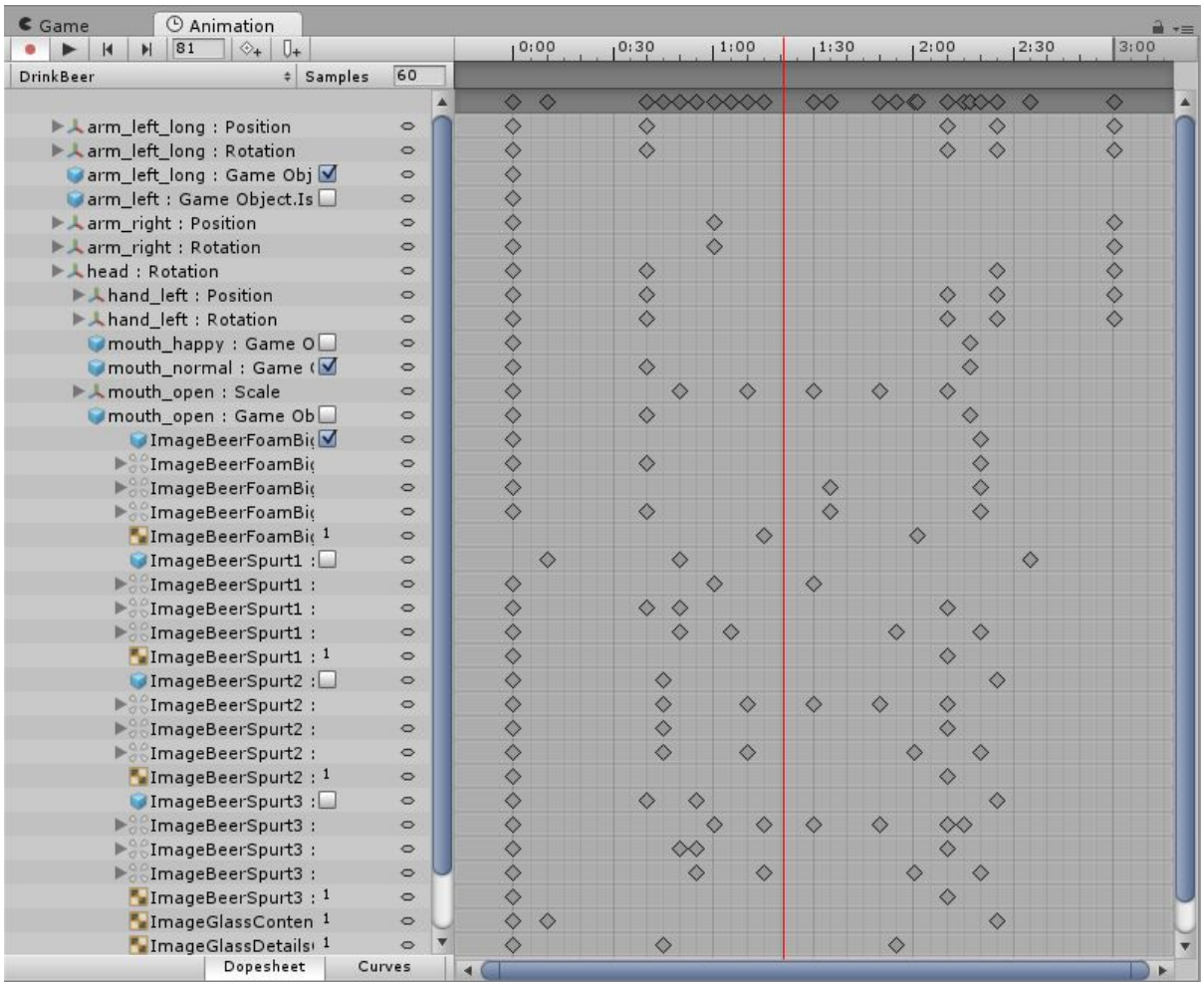

Pivot
Global
Animator
Layers
Parameters
Base Layer
Base Layer
Blinking Layer
Talking Layer
Money2
Money
DisappearAngry
IdleAngry
DisappearMoney
IdleMoney
AppearAngry
AppearBeerGlass
Money3
AppearMoney
IdleBeerGlass
Entry
Invisible
DisappearBeerGlass
Appear
Exit
AppearBeer
Idle
Any State
IdleBeer
Disappear
DisappearBeer

In a game about beer delivery, a drinking animation is of course mandatory. At first glance, it seems to be easy to create. However, creating a smooth drinking animation requires a lot of work.

On the right in the above image, you can see the object tree of the player. You can recognize, that we used a Canvas within the other player sprites. The canvas is used to compose the beer out of many different images. The canvas is used because the UI Image provides the ability to cut (not scale) an image. This method is utilized to create an animation to empty the glass from the top to the bottom. For that, the beer content image needs to be drawn in advance at the correct angle. The same is true for the notches of the beer glass.

There are a variety of slightly different images of the same object. For example, there are three images to animate the notches. One for the border, one for the yellow content and one for the empty content which is about 50% transparent.
Furthermore, there are images for the beer floating into the player's mouth, the mouth itself, the beer foam and the long arm.

The complex keyframe arrangement of the drinking animation is depicted in the image on the next page.



In conclusion, you can see that the animation process is not trivial.

On the one hand, it's an easy and straightforward procedure. You only need to set correct keyframes for different image parts.

On the other hand, you notice that it actually takes a lot of effort to create a fancy and smooth animation. You have to think about the final animation as early as you begin to draw the images. Then you need to

- slice the images
- set pivot points for each image part
- compose the parts in Unity again
- use appropriate Unity image types (Sprite vs UI Image)
- arrange the parts on different layers
- manipulate the images
- set keyframes at different times

Despite the costly animation process, it is a worthwhile task, which enriches the game play significantly.

3.5 Professor voice

In addition to the talking animation we created a talking voice for the professor to make the user interaction more natural. We decided to use a simple “blah blah” voice which can be used for every text. In code the voice recording is splitted into separate parts. These parts are composed to several arrangements. The arrangements are created in a special way to sound natural. Based on the text length of the professor's speech bubble, one or more voice arrangements are chosen to accompany the talking animation.

3.6 Creating in game windows

For showing the routing table or the menu for setting the audio, we didn’t want to start a new scene so we decided to show a panel on the screen. You can see the panels in the following two figures.

Both panels are toggled through the corresponding buttons on the right hand side.

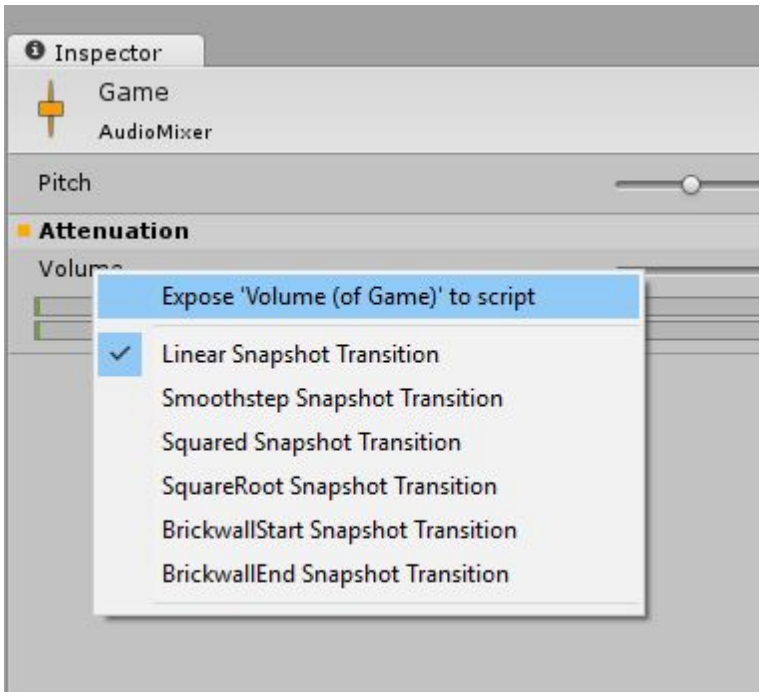
They are created basically the same way. A canvas object that holds an image as background - which is set invisible by default. The content of the panel is handled in form of children of the image - like shown below.

The canvas element has the script to control the content as well as the function to toggle the visibility of the image. With that, the panel will be present or non present on the screen.

3.7 Adjusting the audiomixer within the game

The audiomixer holds all audio sources of the game. By creating meaningful audiomixer groups, these files can be categorized as you can see in the figure above chapter 3.6. Creating in game windows, we made five different groups (Master, Game, Background, Professor and UI). Every group controls the volume of its audio sources. The Master mixer sets the total value. Also more effects can be handled.

To manipulate these groups by scripting, you need to select a group and expose e.g. volume or an other effect for scripting - see screenshot below.



Now you are able to manipulate these effects by scripting. To store adjusted values we used a simple Key-Value Store called PlayerPrefs.

3.8 First user impressions

To get a first user feedback, we carried out a study with 3 participants. A first prototype of the Dijkstra tutorial was used. They all had no prior knowledge about routing algorithms and they were just told that this game should teach them how network links are discovered and packages are forwarded. No further introduction was given and they started to play the Dijkstra tutorial level.
Hereinafter are the protocols of the participants:

3.8.1 First Participant

Prior knowledge: none
Gender: female
Age: 26 years
Degree: Bachelor of arts (BWL im Gesundheitswesen)

Asked questions:
Is it possible to interact and play the game while the professor is visible?
What is the starting table?

Mistakes:
Clicked the wrong bottle to get to another table

Feedback:

Negative:

- The labels above the bottles are hard to read
- Same distances between tables are confusing
- Routing table is confusing and hard to read and also unnecessary

Positive:

- There are very descriptive explanations and metaphors
- The course of the game is highly logical

Improvement proposals:

- none

Grading:

Time: 14 min

Score: 75%

3.8.2 Second Participant

Prior knowledge: none

Gender: male

Age: 24 years

Degree: Bachelor of arts (Informationsmanagement und Unternehmenskommunikation)

Asked questions:

- When will a table be discovered completely?
- What is the starting table?

Mistakes:

- After completely discovering the second table he chose the wrong path

Feedback:

Negative:

- To much text - should be read aloud

Positive:

- Quite descriptive explanations and metaphors which easily can be tracked

Improvement proposals:

- Score-system should be replaced by “tips” or “refund”

Grading:

- **Time:** 10 min
- **Score:** 50%

3.8.3 Third Participant

Prior knowledge: none

Gender: female

Age: 20 years

Degree: High-school graduate

Asked questions:

- none

Mistakes:

- none

Feedback:

Negativ:

- None

Positiv:

- Highly clear depiction
- The numbers above the bottles and the routing table are a great help

Improvement proposals:

- none

Grading:

- **Time:** 7 min
- **Score:** 100%

We can derive from this small study that the visualization still had some issues - like the too small font of the bottle numbers and the hard to read routing table as well as the lack of a marker for the starting router and a completely discovered table. We also see that the tutorial helps to accomplish the level. According to the feedback, we resolved these issues.

4 Reusing the project

This section shortly summarizes aspects relevant for the reuse of our project. More information about the implementation can be found in chapter 6.

4.1 Software and Hardware specs

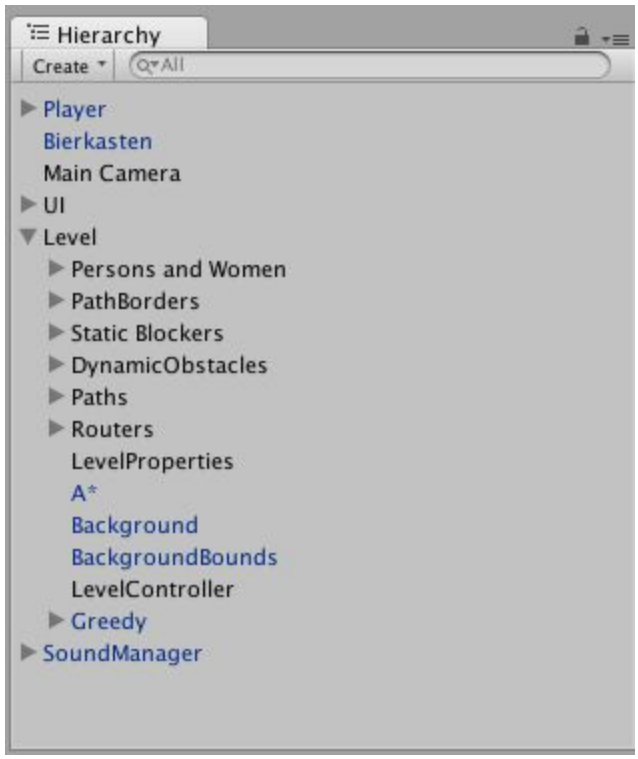
We used the newest unity version available at the time (5.3.2f1) on different windows machines including windows 7 to 10. Additionally one Mac was used with the same version of Unity. We used Visual Studio for developing the C# scripts under windows and MonoDevelop on Mac.
We made the graphics with Adobe Illustrator and sometimes Photoshop. When we had to cut, compose and edit sounds, we used Audacity and Adobe Audition.

4.2 Shortcuts that are usable in the game

- Escape allows us to skip the professor. This can be used to test level boundaries and the player's pathing
- Alt shows all items we can interact with, by displaying a border, additionally it highlights direct paths between nodes.
- WASD can be used to move the camera, as well as left clicking and dragging around
- Clicking 7 times on the "MENU" Tag in the main menu enables all levels

5 Creating new Content

5.1 Building the Level



When building a new level, the best approach is to open an existing scene, save it under a new name and start editing. This enables us to save time from setting up the whole project. When doing so, you should end up with the following hierarchy. The following paragraphs describe steps that need to be done in order to set up a level.

First, select the UI prefab and drag it into the level. Next, create an empty game object “level”. Then, to further structure the level, create the same structure depicted on the right. Alternatively, use one of the existing scenes to define a level to speed up the process.

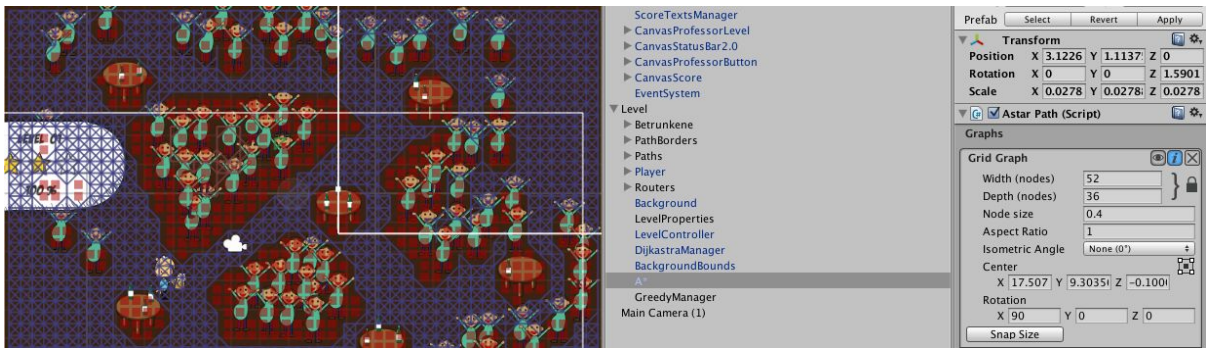
In this example you can see the hierarchy. Ignore the UI game object for the first part and start by opening *Persons* and *Women*. Delete all but one “*Woman*” object and open up the “*PersonGroup*” and delete all but one person.
This allows you to simply copy these objects to fill your level later on. Open up “*Static Blockers*” and delete everything. You can refill this group by dragging in prefab when you setup the routers and paths.

The scene should now consist of three things: *Routers*, *Paths* and *Pathborders*. The next Step would be deleting all but one path and router object.

5.1.1 A* Pathfinding and Player objects

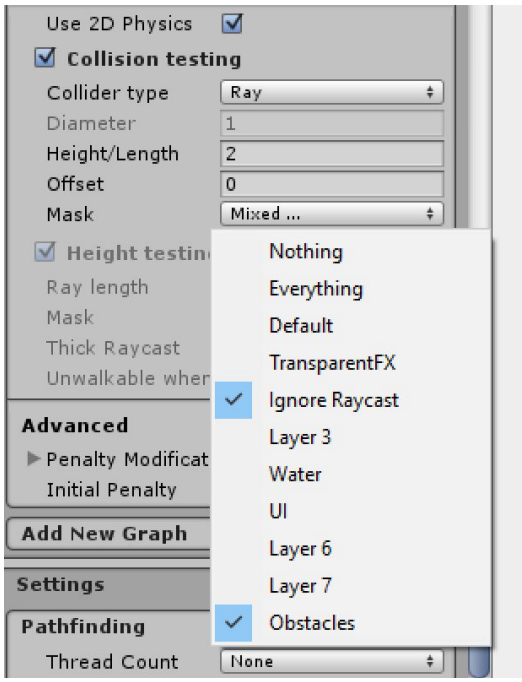
We use the A* pathfinding project for the movement of the player object along the paths between routers. The paragraph briefly describes what steps are required for setting up the pathfinding for a level.

The level requires a game object which provides the A* implementation. If it is not already placed in the level's game object hierarchy, add the A* game object from the prefabs folder. If you click on it, you will see the grid of the A* pathfinding drawn on your scene.



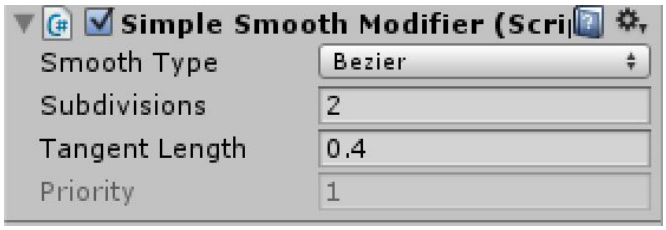
The game object has a script called "Astar Path". If your level will be bigger or smaller, just adjust the width and depth properties until the grid covers your whole level and, if needed, change the position of the grid. The grid consists of small nodes of a specific size. You could also set other node size values, but keeping the default values is recommended. The nodes are used to determine paths from one point to another point on the grid. The target is to find a path, even if blocking elements, i.e. obstacles, are placed in the level. The above image shows the areas where movement is possible in blue and blocked areas in red. The objects that represent obstacles need to be assigned a certain layer. In our case, this layer is simply called "Obstacles". The layers must be added to the "mask" property in the "Collision testing" menu entry of the "Astar Path" script in order to enable the recognition of obstacles by the A* algorithm.

Obstacles are the subjects placed in the bar, e.g. the furniture, beer barrels and similar things. Furthermore, the guests of the bar also represent obstacles that need to be bypassed by the player. At last, one can define invisible colliders which are also specified as "Obstacles" in order to clearly restrict the player's movements on the desired paths.



After that, place a “Player” prefab on the level and drag this object into the main camera’s “camera follow player” script. The relevant property is “Following Pos”.

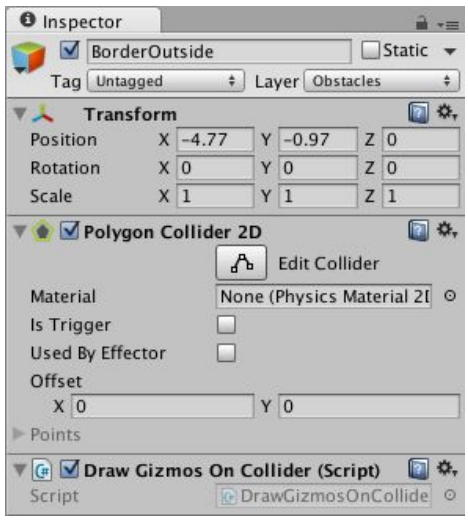
Two scripts need to be assigned to the player object in order to enable the pathfinding process. The first one is the “Seeker” script that is located in the A* project. This script enables to find a path between two points within the grid displayed in the image above. The second script is called “Simple Smooth Modifier”. It is used to smooth the calculated path which would otherwise lead exactly along the grid lines and thus result in sudden directional changes of the player object in the movement process. It is suggested to use the smooth type “Bezier” with parameter values of 2 for the subdivisions and 0.4 for the tangent length. The two script components are depicted below.



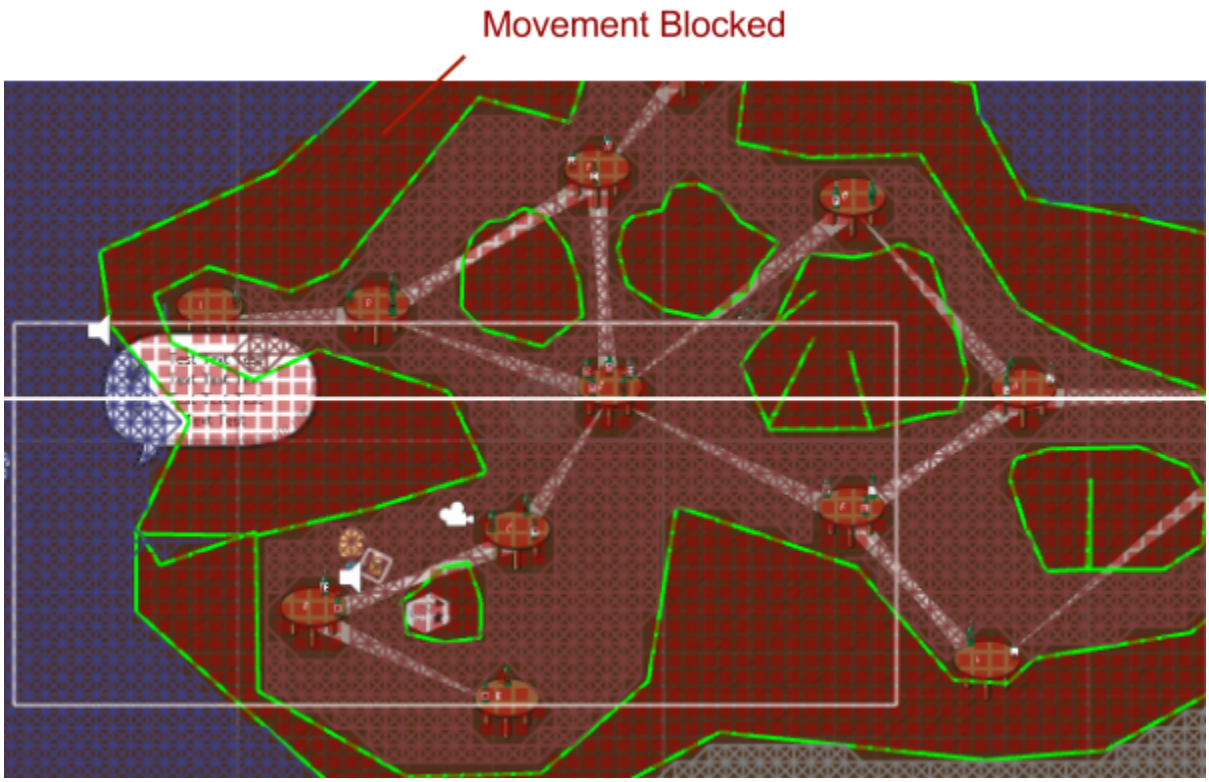
In the game, the movement of the player always happens between two game objects. The game objects require a 2D collider since the borders of those colliders are used to determine the endpoints of the movement process. This enables a natural movement behavior as the player object stops at the border of the game object, e.g. a table.

The actual movement logic is placed in a separate script called “Movement script”. This script resides on the player game object and uses the aforementioned scripts to enable the player’s movement.

5.1.2 Adding PathBorders



Delete any existing objects in the “*PathBorders*” object and create a new empty game object. Add a “*Polygon Collider 2D*” and a “*Draw Gizmos On Collider*”-Script to this object. Change the object’s layer to “Obstacles”. Then, edit the collider and model borders for the movement of the barkeeper. When everything is set up, click on the A* object to check if the area is blocked for pathfinding.



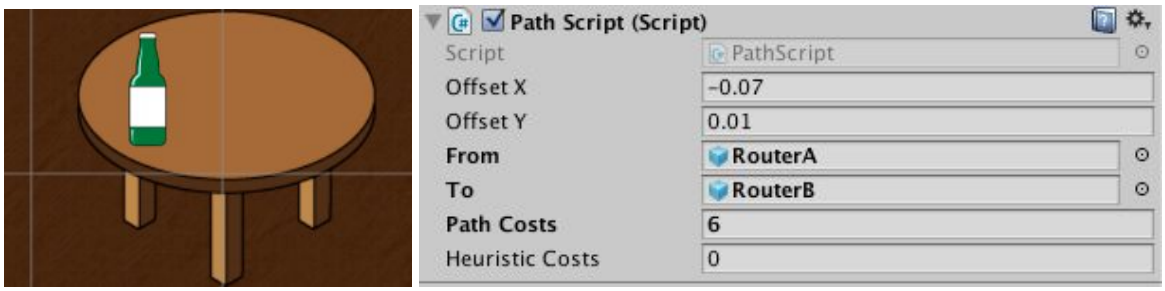
5.1.3 Adding Routers

After that, drag the “router” prefab in your router structure, and rename the objects accordingly. Alternatively, duplicate the existing prefab. Then edit the “*RouterScript*” attached to the router by assigning a “*Router Name*”. This name in the script should be a character between A and Z. The script automatically assigns a fitting index for the internal graph structure. This works only when using subsequent characters such as A,B,C,...

Repeat those steps until you have the amount of routers you want.

5.1.4 Adding Paths

After that, drag a “*uni-path*” object onto the table. Alternatively, just duplicate an existing path.



In the matching "*PathScript*", you then need to specify source in "*From*" and destination in "*To*", as well as path costs for uniform cost or dijkstra game modes. If you are using greedy game mode, you do not need to adapt the heuristic costs nor specify "*Path Costs*". After that, place another bottle on the destination router and switch source and destination to enable the user to determine which paths are connected. Repeat those steps for every edge in your graph.

5.1.5 Static Blocking Objects

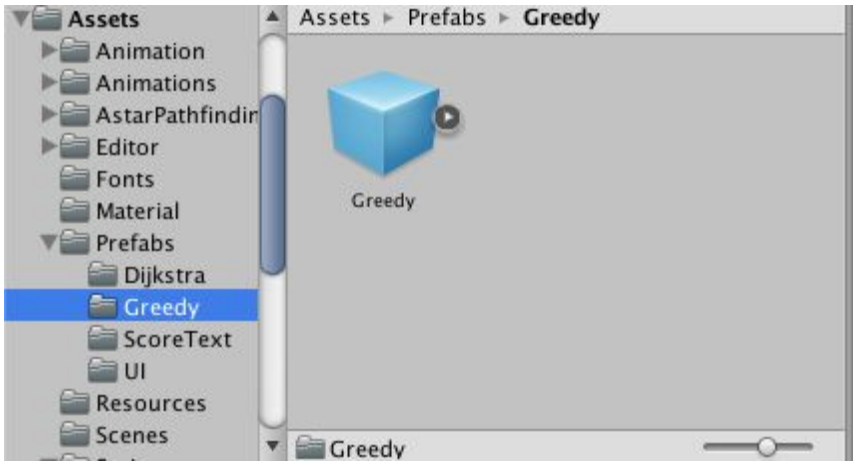
Optionally, you are now able to add blocking objects to the scene. Just drag those prefabs into the scene and their appearance will be specified randomly as soon as the game starts, in cases they support this behavior in form of randomization scripts. Group these objects in "*Static Blockers*".

5.1.6 Adding Dynamic Person Objects

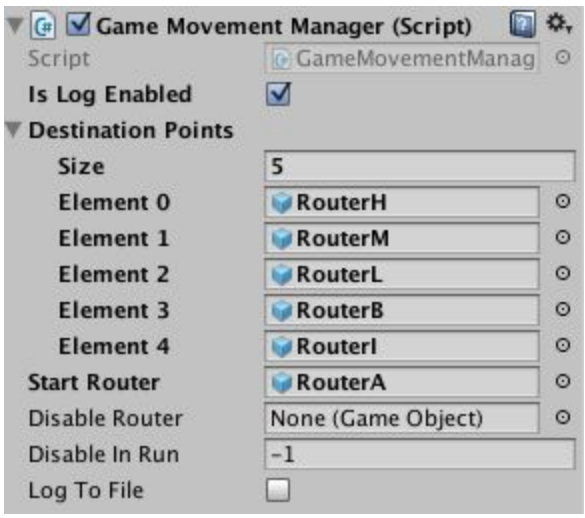
As specified in the first section, you can now add women or men objects by copying the corresponding objects in your hierarchy and placing them appropriately.
When not copying the level, set up a new empty object as a folder for women and men. Then, drag in one women prefab and place it in your level. After that, duplicate it as often as needed. When adding men objects, you have to add one "*PersonGroup*" and then add "*PersonNew*" as a child to the "*PersonGroup*". You can then duplicate the "*PersonNew*" object until you filled your level. The structure would look like this:
Women: <Folder object name>/Women...
Men: <Folder object name>/PersonGroup/PersonNew...

5.1.7 Defining the Game Mode

You are able to define the mode of the current game. When you followed the tutorial, this is not necessary as the appropriate objects are already in the game.



When you created a level from scratch, you can find the matching prefab under the “Prefabs” folder. When expanding the “Greedy” object in the hierarchy you will see a “GreedyMovementManager”. This structure applies to every game mode, it just starts with the name of the appropriate mode. When using non Dijkstra-Algorithm modes, you need to specify a list of target nodes and one starting node - the player then has to complete the whole goal list. For example, start is A, targets are [F,B,G,J]. Then, the player first needs to find a way from A to F, then from F to B and from G to J. This can be seen in the screenshot below:



It also allows you to disable one router for all runs after the specified run has been performed. This operation expects that a group named “RouterBlockers” exists. The game objects specified in this group will be disabled and hidden until the specified run is performed. After the specified run, the objects appear and the topology changes. The picture below shows such a group containing several women objects.

For more information refer to 2.6 Dynamic Events.

5.1.8 Logging

You can log the player's actions to a separate file in Unity's default persistent data path. Just go to the "MovementManager" and enable logging via a flag. The directory of the log file depends on the operating system.

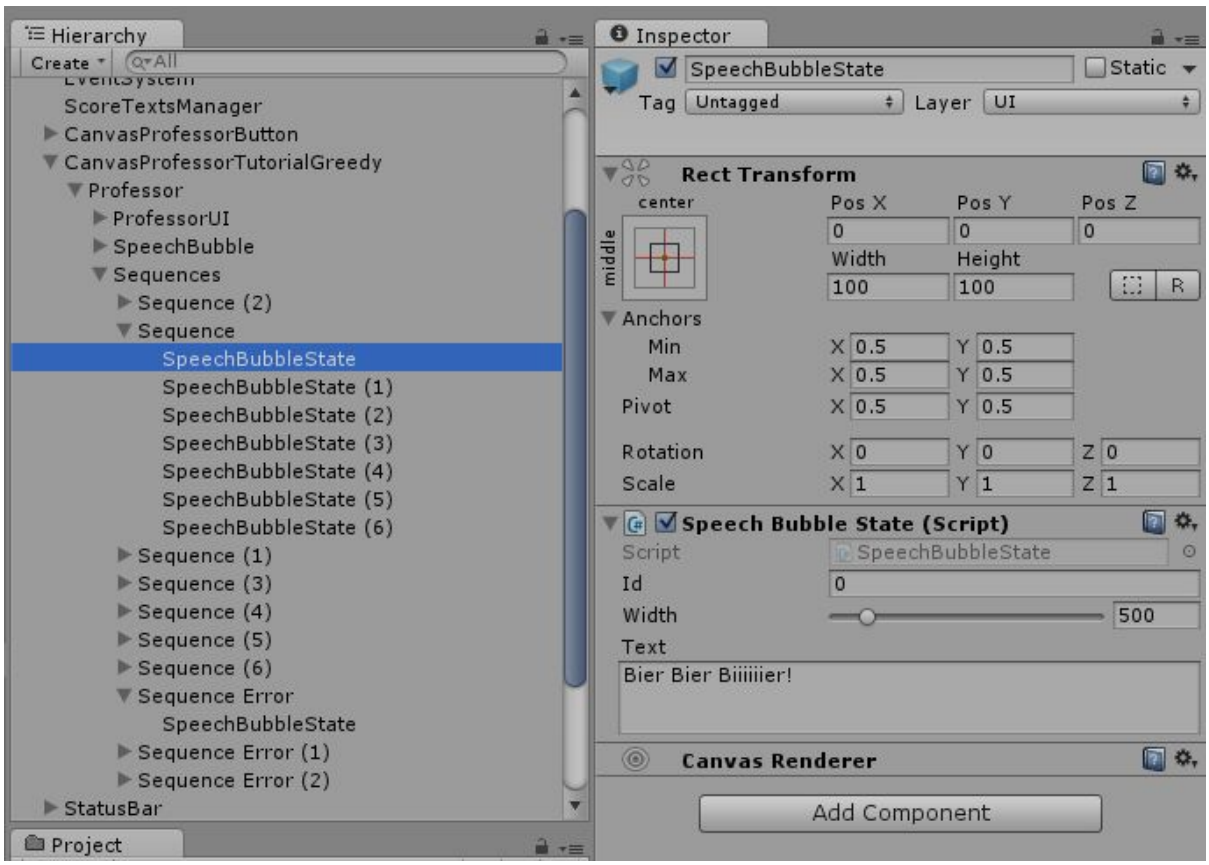
Mac OS:
/Users/<name>/Application Support/BeerRouting ULTD_/BeerRouting/SurveyLog
Android: /data/data/com.beerrouting/files/SurveyLog
Windows:
C:\Users\<name\AppData\LocalLow\BeerRouting ULTD_\BeerRouting\SurveyLog

5.2 Adding the Professor

5.2.1 Professor prefab

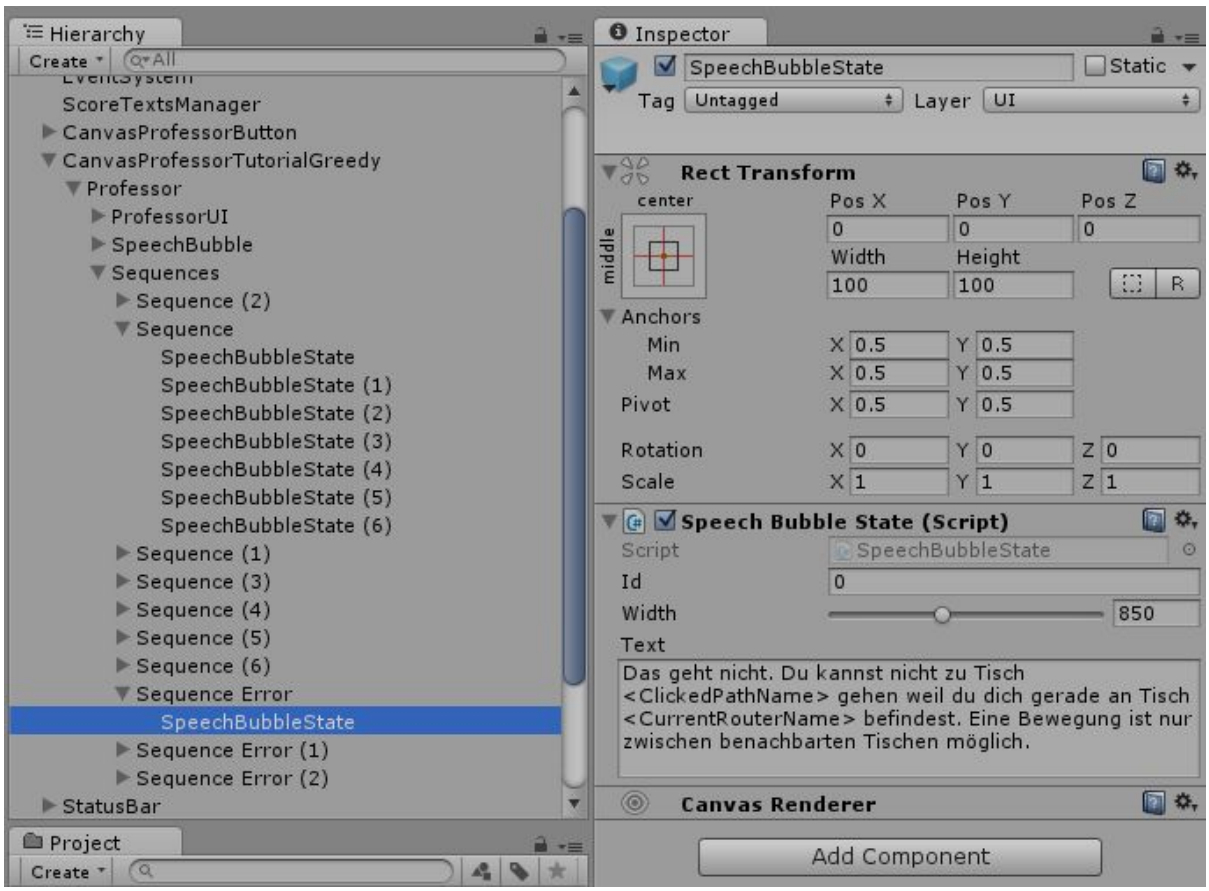
Utilize a prefab called *CanvasProfessor** for example *CanvasProfessorLevelGreedy* to add the professor to a level. The prefabs differ only in the speech bubble texts which are used for a specific game mode. Just use one of them and adapt it. You need to provide a camera for the professor canvas. Use the main camera of the scene.

5.2.2 Professor structure



The professor canvas holds a game object which consist of 3 parts.
First, the *ProfessorUI*. It contains the professor sprites and animation. Second, the *SpeechBubble*. The speech bubble contains text, navigation buttons and sometimes a star rating bar. Third, the speech bubble *Sequences*. It may contain multiple *Sequence* game object which may contain multiple *SpeechBubbleStates*.
A speech bubble state defines the text and the size of the speech bubble. A click on the next button shows the next speech bubble state. At the end of a sequence the professor hides. When the professor shows again, usually the first speech bubble state of the next sequence is shown. However, the level controller could define a different behaviour. The level controller also defines the professor's appearance, e.g. angry or with beer (see *LevelController*).

5.2.3 Adapt the professor



To modify the text of the speech bubble you need to change the content of the *Sequences* game object. Copy and paste or delete any *Sequence* game objects. The sequence id describes when the sequence if shown. The first sequence id must be 0. The second id must be 1. The ids need to increase gapless. Sequences for error messages should have a negative id. So the first error sequence should have the id -1. The id of the *SpeechBubbleState* defines its position within a sequence. The first speech bubble state must have the id 0. The ids need to increase gapless. The width defines how big the speech bubble is drawn. The more text, the greater the width.
You are able to use several variables like *<ClickedPathName>* which are replaced by the level controller.

5.3 Linking the components

5.3.1 Level controller

Each level must have a level controller or tutorial controller (see below). It accesses and connects a variety of different scripts to provide a proper game flow. It listens to professor and player events, reads the game state of the current routing algorithm, controls the professor, game input and more.
To control the professor, adapt the professor event method *OnStopDisappear()* which gets called when the professor disappears. This method usually just changes to the

next professor sequence. However, in special cases you could do something else. For example after a special sequence or even speech bubble state you could trigger a special professor behaviour. You can access and control the professors sequence and speech bubble state via the *ProfessorController*. Furthermore, this script allows you to change the professor's appearance, e.g. show the professor with a beer bottle. For each game mode (routing algorithm) there is a level controller script, e.g. LevelControllerGreedy. That script is used by all levels of the respective game mode. Therefore all theses levels have the same behaviour. If you need a special behaviour for a level, you need to provide another level controller script or add an exception to an existing one.

5.3.2 Tutorial controller

A tutorial controller is basically the same as a level controller. The main difference is, that a tutorial controller is used only within one level, the tutorial. Usually, there is more code in a tutorial controller than in a level controller since the tutorial uses the professor more extensive.

6 Implementation aspects

In the project planning phase we had to decide how to realize the planned game modes. The game modes comprise several algorithms for graph based problems such as the shortest path problem. The algorithms, that can be used in network technology for routing, should be taught to the players by having them carry out the single steps the algorithms would take successively. We thought about defining the solution of a specific level in advance and then evaluate every step of the player against this solution. This approach makes it unnecessary to execute the actual algorithm during runtime, it simply requires to perform the algorithm once in advance to determine the solution. In this case, this could be performed by hand. However, the approach has several drawbacks. Creating different levels becomes more difficult as it is required to determine and hard code the solution for each new level. Furthermore, some levels may have multiple solutions, e.g. multiple valid next hops when there are several paths with equal costs in a shortest path scenario. Hard coding all possible solutions quickly becomes cumbersome. Having these aspects in mind, we decided to take another approach. Our approach was to implement the actual algorithms in a way that enables us to execute them step by step and to evaluate for every action of the player whether the algorithm would perform an equal action next. Even if this approach requires more effort in the first place in order to implement the algorithms, it provides several advantages that should pay off in the end. Having an actual implementation of the algorithm, which can be fed with a graph instance and then permits a step by step execution, makes it easy to design new levels. No hard coded solutions are required. This enables us to easily add new levels or change existing ones without changing

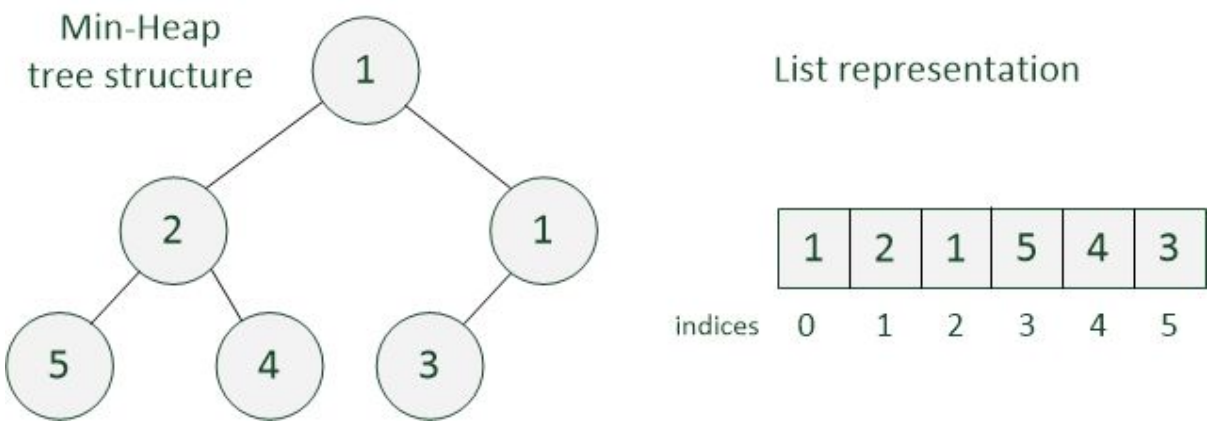
one line of code in the core implementation of our game. Furthermore, scenarios with multiple solutions due to paths with equal path costs are no problem. If there is a scenario where multiple hops are considered valid due to equal path costs, the algorithm will recognize the user's action as a valid hop regardless of which of those possible next hops the user has selected.

To realize our approach, we needed to provide actual implementations of the chosen algorithms. The implementation should enable the step by step execution and the evaluation of player's actions in order to determine them as valid or invalid. Specific data structures are used for many of the algorithms to provide efficient implementations. For this reason, we have implemented a priority queue data structure which is used in the implementation of the Dijkstra algorithm and the uniform cost search. The following paragraph describes the design of our priority queue and several other systems we constructed.

6.1 Priority Queue

A priority queue is a data structure that provides access to data elements depending on a priority value associated with each of the elements. In our case, the data elements are nodes of the graph, more precisely, each node is considered a router. The queue enables quick access to the element that is associated with the highest priority at that time. How the priority value is represented is implementation specific. In our case, we have chosen a numeric priority value. The priority value that is associated with an element of the queue, i.e. a router, represents the path costs of the path, from the start router to the specific router, which is considered the shortest at that time. A priority queue can be implemented using several concepts. We have decided to use a heap structure, i.e. a partially sorted tree. A heap structure is a tree structure where only one condition needs to be satisfied for each node. The condition depends on whether it is a „min heap“ or a „max heap“. In a „min heap“ structure, the value of a node needs to be always smaller or equal to those of its children. The node with the smallest value is the root node. In a „max heap“ it is the other way round. We have decided to use a binary „min heap“ structure where the value of the nodes in the heap is represented by the priority value. A lower priority value implies a higher priority, i.e. the router with the currently shortest path (from the start router to this router) has the highest priority at that time. The details regarding the heap structure are described in the following.
The figure shows an example of a binary „min heap“ structure. The elements are represented using circles which contain the associated priority value. The heap condition is satisfied. Every node has a smaller or an equal value compared to its children. Such a structure can be represented in code by a simple list. The root element is always the first element of the list. Further nodes and their relationships

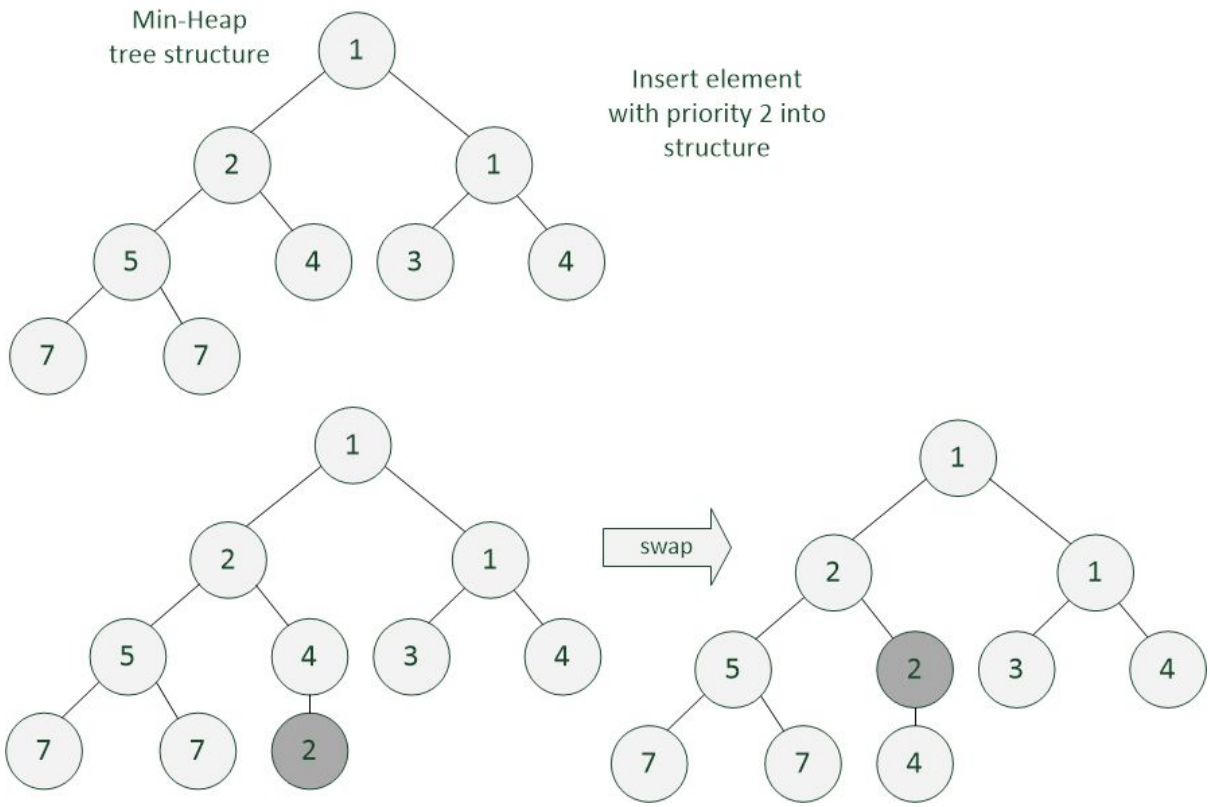
among each other depend on the indices of the elements in the list. For example, the children of a node with a known index can be easily determined by taking a look at the following indices: *2 * parentIndex + 1* and *2 * parentIndex + 2*, respectively.



The goal of a priority queue to provide a quick access to the element with the highest priority at a time can be easily achieved using this binary „min heap“ structure. The element with the highest priority at a time is always the root element of the heap. A lookup on this element thus requires only a constant time (runtime O(1)). However, the priority queue does not only require lookup functionality, but also other functionalities such as inserting or removing elements. More precisely, our priority queue offers functionality to add an element to the queue, to lookup the element with the highest priority at a time, to remove the element with the highest priority from the queue, to check whether an element is contained in the queue, and also to decrease the priority of an element during runtime. In the following paragraphs, it is shown how this functionality is realized using the binary „min heap“ structure.

6.1.1 Add an element to the queue

An element should be added to the priority queue in a way that keeps the ordering of the elements within the queue in a consistent state. The element has an associated priority value and needs to be inserted into the order of the already contained elements correctly. This can be achieved by temporarily inserting the element at the end of the list, i.e. in the leaf level of the heap as the rightmost element . After that, the element is evaluated against its parent node and the nodes are swapped if the heap condition is violated. This is done until the heap condition is satisfied. Due to the tree structure, this will require a maximum of O(log n) steps. The following example shows the insertion process. The element with a priority value of 2 is inserted into an existing priority queue. The element moves up in the tree until the heap condition is satisfied. After that, the ordering of the priority queue is in a consistent state.

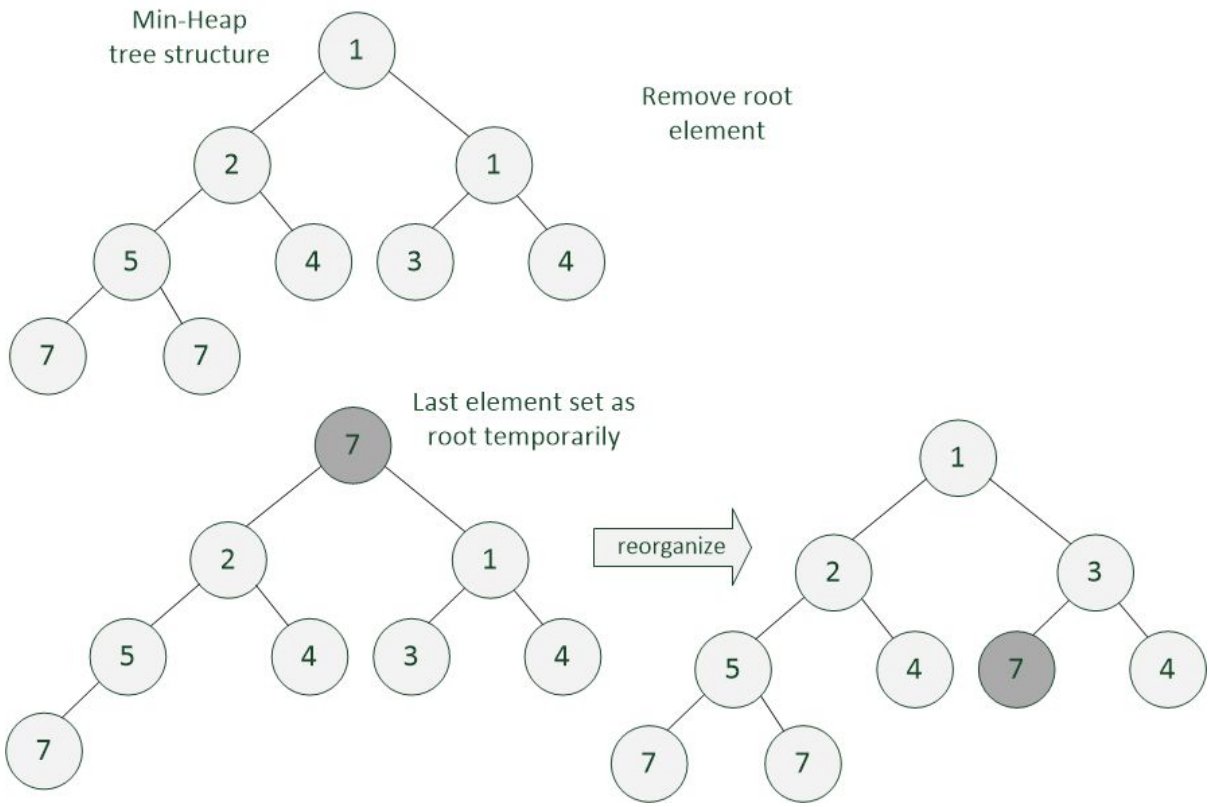


Remark:
A router element is represented by a C# script. Since the priority queue is implemented using generic types, the script needs to realize an interface called *IComparable*. This interface defines a method called *CompareTo* which is used to compare the priority values between two elements of the queue. Furthermore, an interface called *Priority* needs to be realized which provides direct access to the priority value of the router. The latter is required in order to be able to decrease a priority value during runtime in the priority queue.

6.1.2 Remove element with highest priority from the queue

The element with the highest priority is the root element of the heap structure. The element can thus be extracted easily. However, the heap condition needs to be retained, thus a reordering process is required after the root element has been removed. The last element of the list, i.e. the rightmost element on the leaf level is set as the root element temporarily. After that, it is evaluated against its children. It is first checked which of the child nodes has a higher priority. The child with the higher priority is evaluated against the parent node. If the heap condition is satisfied, the process is finished. If the heap condition is violated, the nodes are swapped. This is done until the ordering of the structure is in a consistent state. Due to the tree structure, this process also requires a maximum of O(log n) steps.

In the example, the root element with value 1 is removed. The last element, i.e. the element with value 7 in this case, is set as the new temporary root element. However, the consistency check against the child with the higher priority, which is the right child, fails and the elements are swapped. The same is done in the next step. After that, the heap condition is satisfied and the priority queue is in a consistent state again.



6.1.3 Decrease priority of an element

The problem in decreasing the priority of an element in the queue is to find out its index in the first place. For this reason, an index map is kept and maintained besides the actual queue. This index map enables to perform a quick lookup of which index belongs to an element at that time. After the index is determined, the priority of the element can be decreased. It is possible that decreasing the priority violates the heap condition. The affected element is thus evaluated against its parent node and the elements are swapped in case of a violation. This is continued until the heap condition is satisfied.

6.1.4 Check if element is contained

Using the mentioned index map, it is no problem to determine whether an element is contained in the queue at a time. A simple lookup query, which can be performed in constant time, is sufficient to return the answer to the requestor.

The mentioned mechanisms provide a robust and efficient implementation of our priority queue. The priority queue is an essential part of the implementation of two of our game modes, the Dijkstra game mode and the Uniform Cost game mode. More details about the implementation of the game modes can be found in later paragraphs.

6.2 Saving and Loading game state

Our game consists of several game modes and levels for each of these game modes. The levels should be performed subsequently, i.e. all levels except one are locked at the beginning. For each level, the player is provided with a score. If the achieved score is high enough, the next level is unlocked. To realize this, we required a mechanism to store the game state, i.e. the achieved scores for all played levels, in a persistent manner. This section describes how the saving and loading of the game state was realized in our implementation.

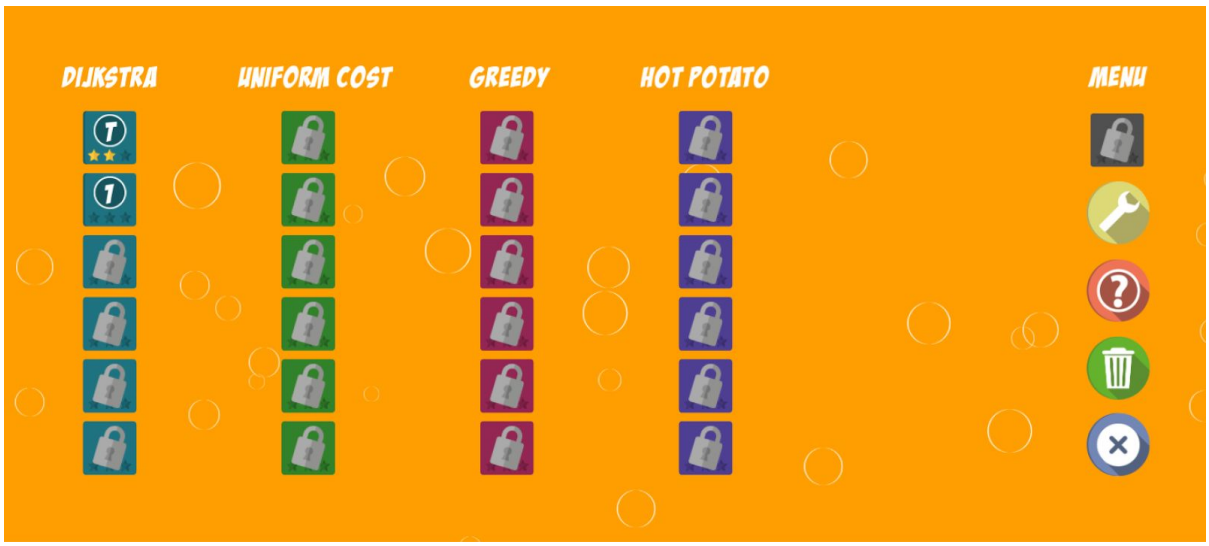
Before the actual implementation started, we had thought about the information that needs to be stored. We agreed on three attributes that must be stored persistently for each performed level: the id of the level, the achieved score and the corresponding numbers of stars representing this score. A player can gain a maximum of three stars per level depending on his achieved score. The amount of stars determines whether the next level should be unlocked and for this reason, it was decided to store this information together with the score. In our implementation, we use a class named *LevelState* to encapsulate the three attributes.

Two further questions had to be answered: “Where should the data be stored?” and “How should the data be stored?”. For the latter, we had intended a simple JSON document, but we noticed that players then would be able to directly modify the document. This could be used to unlock levels by simply adding entries to the JSON document. However, in our final solution we still rely on a JSON document, but we modified it in order to prevent manual changes. The following paragraph describes our approach.

The game state is managed by a script called *GameState*. The script contains a dictionary data structure mapping instances of the data class *LevelState* onto the

corresponding level ids. Functionality was added to enable access and modifications of the data sets for individual levels. The script furthermore contains methods to store the state persistently and to read the stored state. If the *store* method is called, the entries of the dictionary are parsed into the JSON format using the *JsonUtiltiy* class provided by Unity. To prevent modifications, the JSON document is encoded using a simple Base64 scheme and stored in a binary file. This should make modifications on the file itself impossible for unskilled users. The file itself is stored using the persistent data path provided by Unity which is determined based on the current operating system. This enables us to store the file at a valid location independently of the underlying operating system. In the *read* method, the mentioned file is read and its content is parsed and stored back into the dictionary.

When the player starts the game and the main menu is loaded, the *GameState* script is used to unlock levels based on the stored data. To do this, the individual levels are checked and set to unlocked if their predecessor level has already been played and the numbers of stars representing the achieved score is at least one. The picture below shows the state where the tutorial level of the dijkstra game mode has been played and the player has achieved a score that is worth two stars. The following level is thus unlocked.



Every time the player finishes a level, two actions are performed. First of all, it is checked whether the player has already played the same level before. If this applies, the score value for the level in the *GameState* instance is only updated in cases of a higher score compared to the currently stored one. Otherwise, no action is performed. However, if the player achieves a higher score or performs the level for the first time, the score and the number of stars are added to the dictionary of the *GameState* instance. After that, the whole dictionary is stored persistently.

6.3 Consecutive errors

While developing we also came up with the problem, that errors should not lead to resetting the level. It should always be possible for the player to return to the last viable game state. To do this, we used an enum that indicates whether the performed action was correct or not. This allowed us to detect errors and consecutive errors. We in turn were able to only deduct points when the player has made a mistake and not to penalize consecutive errors at all or to penalize them with less minus points. In both cases the player would get helpful feedback from the professor.

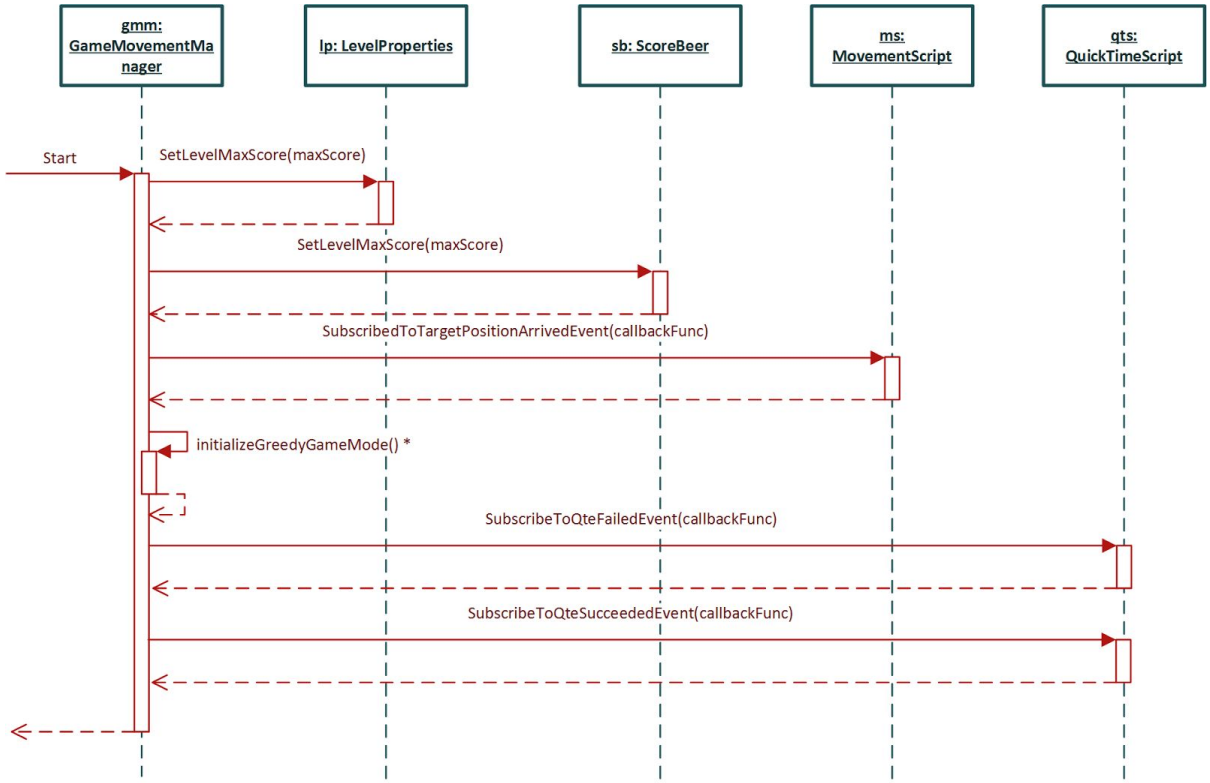
6.4 Generic Managers and a way to use them with the movement script

When we finished the greedy game mode, we also thought about splitting the logics into separate classes or interfaces to minimize clutter and enable reusing commonly used operations. All the common methods were then moved into the *GameManagerInterface*. This interface expects all implementing classes to provide the following operations:

- PerformHop, PerformWrongHop, PerformErrorRecoveryHop
- IsValidHop, Start
- Getters for player position, pathscripts, routerscripts and gamestatus.

The interface in turn is implemented by all game modes except for Dijkstra's algorithm.
Additionally we separated the whole movement of the player from the game logic. This allowed us to keep one movement script for all game modes. The *GameMovementManager* is called when user interaction happens and manages the interaction between the algorithm and the player. Below you can see the sequence

diagram that displays the interaction between those three elements.

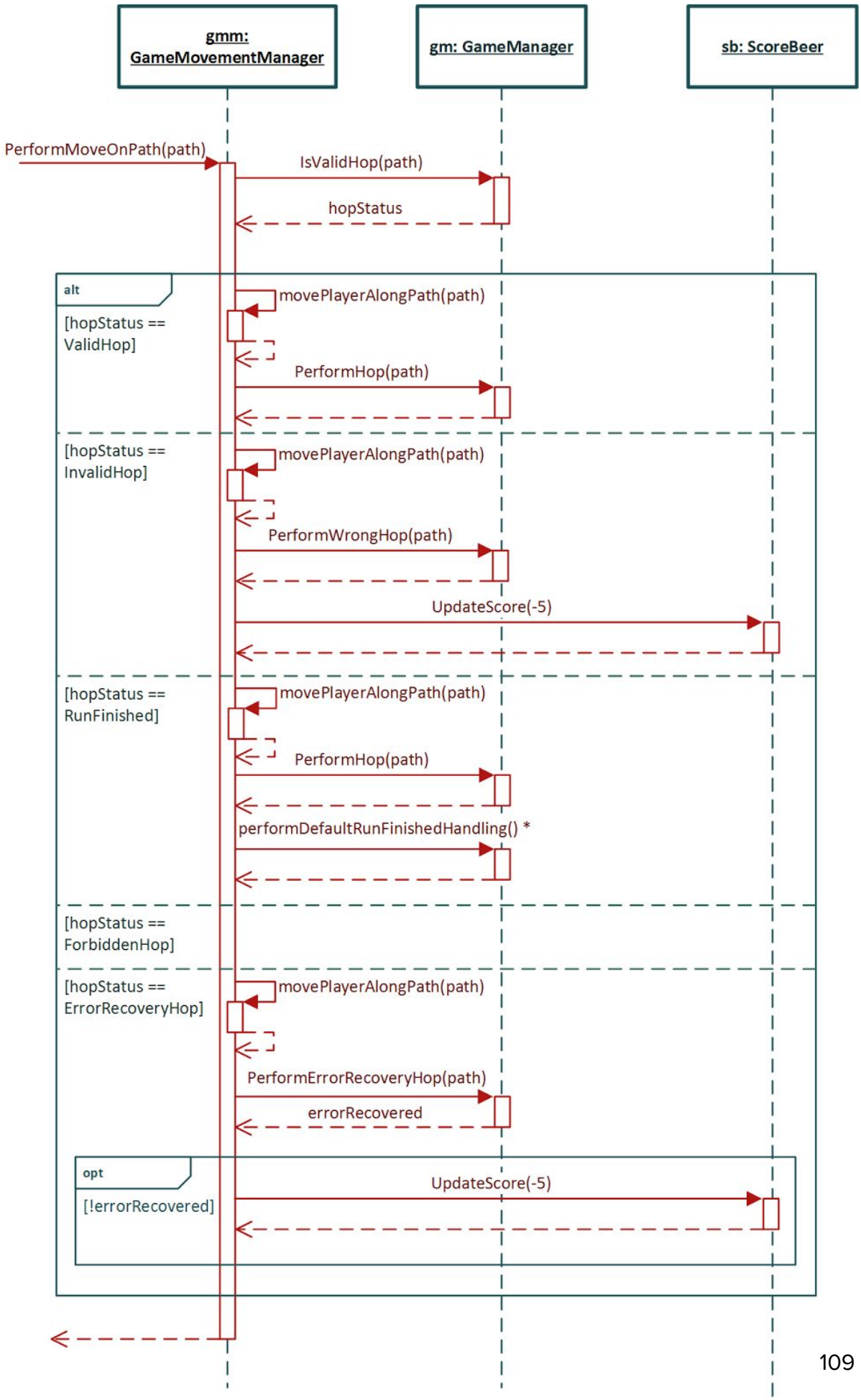


When the game is initialized, the *GameMovementManager* object calculates the maximum score depending on the current level and sets the maximum score of the game in both the *LevelProperties* and the *ScoreBeer* object. After that, it subscribes to the *TargetPositionArrived* Event from the *MovementScript*, that is used to determine if the movement process is completely done. The event fires a callback method when the player has arrived at the target position. The registration is followed by the initialization for the specific game modes. First, it will decide which game mode is used, then the initialization process begins. In the diagram, it is depicted for the Greedy game mode. After that the *GameMovementManager* subscribes to two callback functions, first the *QTEFailedEvent* and second the *QTESucceededEvent*. Both are used to determine whether the player failed or completed the Quick Time Event.

Below you can see the diagram for *PerformMoveOnPath*. If the player has clicked on a path, this method is called. It first evaluates the move of the player and checks the *GameStatus* that is returned for this move from the algorithm implementation. Then it executes different operations depending on the status.

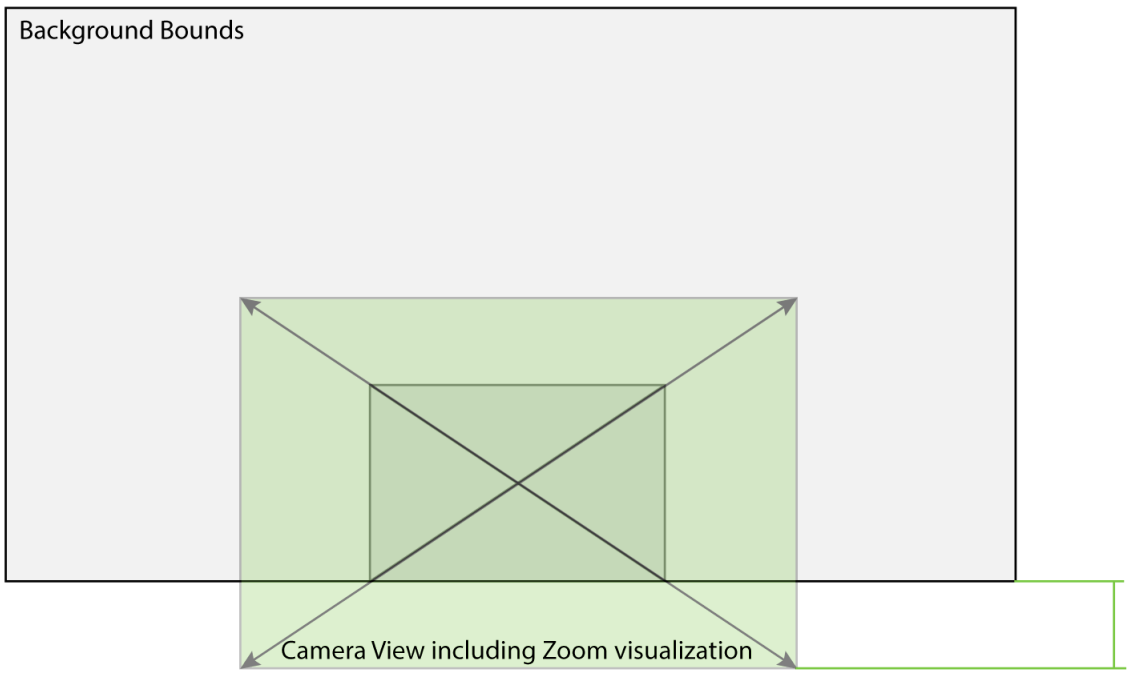
- **ValidHop**: Movement can proceed, perform the hop in our *GameManager's* graph structure.
- **InvalidHop**: Movement can proceed but perform the *PerformWrongHop* method that does not update the graph structure. Instead, the error recovery process is started. Additionally, remove 5 points from the score.

- **RunFinished**: Allow the physical movement and perform the hop in the graph. Additionally, perform the run finished handling. This depends on the specific game mode what is symbolized by the asterisk.
- **ForbiddenHop**: Do nothing here. This status is used by the professor's script to inform the player, that the interaction does not work that way.
- **ErrorRecoveryHop:** Moves the player graphically and asks the *GameManager* whether the error is recovered. When the error is not recovered, deduct 5 points from the score.


gmm:
GameMovementManager
gm: GameManager
sb: ScoreBeer
PerformMoveOnPath(path)
IsValidHop(path)
hopStatus
alt
[hopStatus ==
ValidHop]
movePlayerAlongPath(path)
PerformHop(path)
[hopStatus ==
InvalidHop]
movePlayerAlongPath(path)
PerformWrongHop(path)
UpdateScore(-5)
[hopStatus ==
RunFinished]
movePlayerAlongPath(path)
PerformHop(path)
performDefaultRunFinishedHandling() *
[hopStatus ==
ForbiddenHop]
[hopStatus ==
ErrorRecoveryHop]
movePlayerAlongPath(path)
PerformErrorRecoveryHop(path)
errorRecovered
opt
[!errorRecovered]
UpdateScore(-5)

6.5 Keeping the camera inside the bounds while allowing the user to zoom

While adding a free camera to the game, we discussed the need to prohibit the user from moving the camera too far away from the playing field. We added one GameObject “BackgroundBounds” to our scenes that is used as our limiter for the camera movement. Additionally, we needed the option to disable free camera movement by setting a flag. When the player is moving, we want the camera to follow the player’s movement.
When free moving was allowed, we just checked whether the camera’s bounds are outside of the maximum bounds. This worked fine while the user was not using the zoom, but when he zoomed, the option persisted. If one edge was outside of the bounds after zooming out, the movement of the camera was disabled altogether.



We adjusted the script to check the X and Y axis separately and check cases like the one depicted above. This now allowed the user to zoom out while maintaining the possibility to move the camera when outside of the predefined bounds. In the case depicted above, the player can move the camera back into the bounds, however, movement further away from the bounds is not possible.

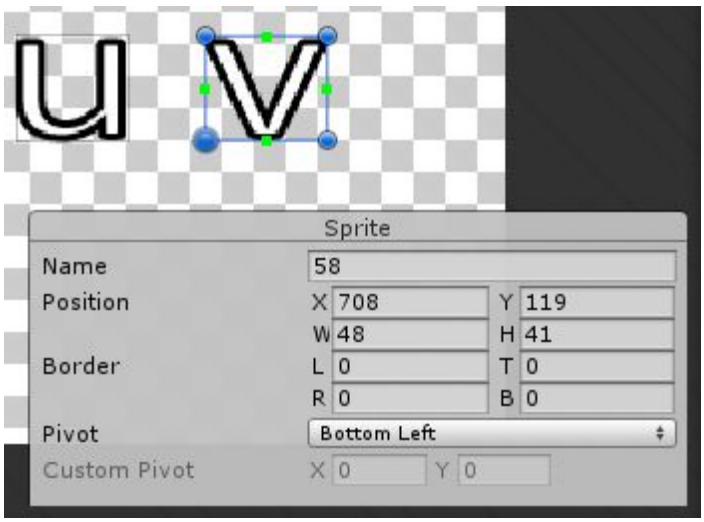
6.6 Move and zoom to a defined location or display an overview of the level

Additionally, we decided to add additional camera features to enhance our in-game tutorials for our players. The *CameraPan* script provides two methods for the public. One for moving to a location and zooming into it until we reach the defined camera height, the second one provides the means to zoom out from a specific point until the level borders are reached. We used those methods mostly for the tutorials to provide the player an overview or to point to specific things that are limited to certain levels. The zoom-in is mostly used to show specific sub-goals in levels, for example the specific autonomous system the player needs to go to. This allows us to be sure about the player's knowledge of his subgoals.

6.7 SpriteFont Renderer

While programming we encountered the problem of adding an outline to text in unity. As far as we saw, there was no easy solution that looked decent. Thus we started working on a Script that takes string input and displays the input via sprites. On the right side you can see one of our testing fonts we used to write the script.

When reusing the script, the first thing that is required is importing the picture containing the letters in the *Assets/Resources* Folder. After that, the sprite editor needs to be opened and the characters must be sliced. Be sure to set the pivot point to bottom left. This allows alignment of the characters in the script. The name of the character needs to be set so that numbers are first (0-9), then the comma, after that we expect capital letters before lowercase letters.



When slicing is done, just create an empty GameObject and attach the SpriteFontRender script.

This script uses a number of parameters to display a font. First, the resource name that is used in the *Assets/Resources* folder. This is used to load all individual sprites into a sprite array. Additionally, it is possible to set up the sorting layer name, the offset in the layer, the text that should be displayed, a means to scale all axis and a spacing parameter that determines spacing between the characters.

The Characters are determined by using their ASCII decimal representations. To get numbers from 0-9 to display we check whether this representation is between 48(0) and 57(9) (including start and end number). We then just subtract 48 and use the result to load the correct sprite (0-9). This is done for all characters, but each time we skipped unused symbols.

The GameObjects will change in real time when using the editor to manipulate values. This always allows previewing of texts when creating them in the editor. The complete package containing the used fonts can be reused by importing the unity package “SpriteFontRendererV3”.

7 Results and discussion

One of the main results after this project is a fully working Unity game that is customizable and allows the creation of new levels. It has several systems to keep players motivated. We used tonal feedback in addition to visual feedback in the form of cartoonish graphics and animation that indicate the player whether the action was recognized as good or bad. Additionally, we also added a system that requires the user to complete levels to unlock further levels in order to keep up the motivation.

Our game was presented as a demo at the Streiflicht event on the 20th of April, 2016. Participants were able to test our game and to provide feedback. We gained mostly positive feedback. Players told us that the game looks nice. It wraps up the teaching content in good manner and is still fun to play. Issues that were mentioned mostly affect the interaction concept of the game. However, as the change of the underlying interaction concept requires a lot of work, we have skipped to implement it so far due to time issues. Future work can thus comprise adding further interaction possibilities such as clicking on adjacent tables (routers) to select them as the next node.

Furthermore, the interaction concepts can be evaluated in a small study to get detailed information about the advantages and disadvantages of the concepts. We also noticed that most players skipped large parts of the introduction provided by the professor. We assumed this is mainly due to the amount of text in the tutorial levels. An idea is to provide the introductions using spoken text instead of the placeholder sounds we are using so far. This might be a further aspect to investigate.

A study can be performed to also determine the learning success of the players. It has been suggested that the game should be distributed to students who participate in a networking lecture and learn about routing algorithms. This would allow us to determine whether students can profit from our game and thus learn about the implemented algorithms. To get more detailed information, the game could be adjusted further to offer a mode with minimalistic graphical support, i.e. a less game-like mode. The underlying graph structure could be presented in a way similar to normal exercise sheets the students face in their networking course. This mode could be compared to the one using full graphical support and thus permits to investigate whether the game-like design actually provides benefits.

8 References

[1] M. Muratet, P. Torguet, J.-P. Jessel, and F. Viallet, “Towards a Serious Game to Help Students Learn Computer Programming,” *Int. J. Comput. Games Technol.*, vol. 2009, pp. 3:1–3:12, Jan. 2009.

[2] C. Kazimoglu, M. Kiernan, L. Bacon, and L. Mackinnon, “A Serious Game for Developing Computational Thinking and Learning Introductory Computer Programming,” *Procedia - Social and Behavioral Sciences*, vol. 47, pp. 1991–1999, 2012.

[3] M. Zyda, “From visual simulation to virtual reality to games,” *Computer*, vol. 38, no. 9, pp. 25–32, Sep. 2005.

[4]“Tomorrow Corporation : Human Resource Machine.”, Available https://tomorrowcorporation.com/humanresourcemachine

[5]“Zachtronics | TIS-100.”, Available: http://www.zachtronics.com/tis-100/

9 Acknowledgements

Some graphics and images were available from third parties. However, most of them were adapted for our needs.

9.1 Graphics

Designed by Freepik:

- Touch Icon: http://www.flaticon.com/free-icon/hand-finger-pressing-a-circular-ring-button_30895
- Various Birds included in our Bonus Level: http://www.freepik.com/free-vector/funny-cartoon-birds-collection_833180.htm
- Different Landscapes included in our Bonus Level: http://www.freepik.com/free-vector/cute-nature-designs_849634.htm
- Speech bubble http://www.freepik.com/free-vector/hand-drawn-bubbles-speech_764271.htm
- Barrel http://www.freepik.com/free-vector/mouse-vector-37_397702.htm
- Score Beer http://www.freepik.com/free-vector/beer-icons_767010.htm

- UI check button
  http://www.freepik.com/free-vector/check-marks_796828.htm
- Wooden signs
  http://www.freepik.com/free-vector/wooden-signs-collection_753569.htm
- Stars
  http://www.freepik.com/free-vector/hand-drawn-stars-icons_787340.htm

9.2 Sounds

Waves and seagulls:

- https://www.freesound.org/people/juskiddink/sounds/149488/

Cat Meow:

- https://www.freesound.org/people/timtube/sounds/60963/

Glas bottle

- http://www.freesound.org/people/milton./sounds/69133/
- http://www.freesound.org/people/jbates18/sounds/94788/
- http://www.freesound.org/people/FreqMan/sounds/42909/
- http://www.freesound.org/people/RSilveira_88/sounds/216303/

Pouring beer

- http://www.freesound.org/people/ShotgunPicker/sounds/169027/
- http://www.freesound.org/people/milton./sounds/69133/
- http://www.freesound.org/people/ShotgunPicker/sounds/169027/
- http://freesound.org/people/HDM2013/sounds/179439/
- http://freesound.org/people/producerdan/sounds/222763/
- https://www.freesound.org/people/jmayoff/sounds/253338/
- http://www.freesound.org/people/YleArkisto/sounds/322468/
- http://www.freesound.org/people/ShotgunPicker/sounds/169027/

Bar noise

- http://www.freesound.org/people/Islabonita/sounds/178525/

Jump sound

- http://www.freesound.org/people/qubodup/sounds/331381/

Stone in water

- http://www.freesound.org/people/RSilveira_88/sounds/216394/

Speech bubble popup

- http://www.freesound.org/people/qubodup/sounds/188797/

Button click

- http://www.freesound.org/people/potentjello/sounds/194071/

Success

- http://freesound.org/people/GabrielAraujo/sounds/242501/
- http://www.freesound.org/people/shinephoenixstormcrow/sounds/337049/

Professor voice

- https://www.freesound.org/people/unfa/sounds/165539/

9.3 Code

- A-Star Pathfinding to enable movement between routers in normal levels. http://arongranberg.com/astar/front