CSE3000 – Research Project – Final Report (2023)
Delft University of Technology
M. Skrodzki(Responsible Professor), R. Bidarra(Supervisor) and G. Smaragdakis(Examiner)

Shortest Path algorithms for the traversal of an Order-5 square tiling from within a confined space

R. Snellenberg[†][1]

[1]TU Delft

Abstract

It is possible to use a different representation of space in a Virtual Reality (VR) game, instead of using the euclidean representation we are used to. The reason why that is interesting is that it opens up the possibility of traversing infinitely far in the virtual space while being confined to a relatively limited space in the real world. But for this to be a useful use case, people will need the ability to traverse this new space in an intuitive or at least competent way. One way to help people with navigation in such a space would be to show the person what the fastest path from point A to point B is. This paper researches the possible ways to calculate this path so it can be used to help people while they are traversing this different type of space. Concluding that a trade-off needs to be made between the speed and accuracy of the result depending on the situation, and that no algorithm exists which is the best in both speed and accuracy. Meaning that the context determines what algorithm should be used.

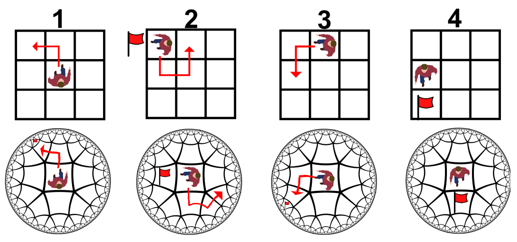
1. Introduction

The use of hyperbolic geometry in Virtual Reality (VR) applications brings the possibility for more realistic movement in VR applications. At the moment most VR applications that require the player to move around, will accomplish this by either having the player use a joystick or by allowing the player to teleport [FSW17]. The applications that mimic the movements of the player in a more realistic manner usually do this by having the player drag themselves forward. This is in most cases because the player does not have a large enough physical space to move around. There have already been several proposed solutions in which the player actually walks around, but they still require large amounts of space and/or specific hardware [DCC97, SPW*18, CCC*21].

Having a VR world that implements some form of hyperbolic connection introduces the possibility for users of VR applications to traverse the space by walking around (even when limited space is available) because of holonomy [Wee21], which is the property in a hyperbolic space that describes the phenomenon that translations/movement in a hyperbolic plane gets an added rotation. The effect is that players who walk a path in the real world, will often end up in a different spot in the virtual world. For instance a player who walks in a circle expects to end on the same point where they started, but in the virtual world they will not (see Figure 1 or Figure 7).

A downside of hyperbolic space is that for many people it can be counter-intuitive to traverse. But by giving the player the shortest path to the destination, they should be able to navigate this space. And thus the main research question this paper will aim to solve is: *What algorithm can compute the shortest path between the user and the objective with a runtime that isn't noticeable to the player?* The research question is divided into two sub-questions:

- What algorithm can compute the shortest path between the player and the objective the fastest?
- Are there ways to speed up the computation by making compromises to the resulting quality?



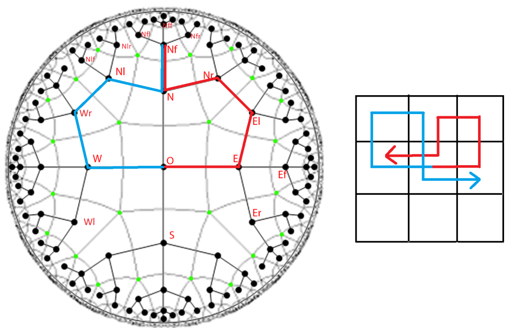
The player in this figure tries to reach the red flag, the red flag however is too far away for the player to reach it by simply walking there in the 3x3 grid. So the player will use the offset between the virtual and the real world they will acquire when traversing the space to shift the flag into the 3x3 grid as shown.

Figure 1: *The world is a combination of two grids*

†

The environment in which the shortest path needs to be computed is a combination of a 3x3 grid, which represents the play area, and a plane with the Order-5 square tiling. Getting the shortest path in either the plane or the grid would not be that hard to compute. It is the combination of the two that currently does not have an efficient method, which is the problem that this report aims to resolve.

Another complication is that there is not always a single optimal solution. Since there are many destinations that have at least 2 shortest paths. A simple example of this is the destination Nf as can be seen in figure 2. There are two shortest paths of equal length that reach Nf, the paths in this example are symmetric but this is not the case for all instances (or at least not as easily identifiable symmetric paths).



There are two different shortest paths to Nf from the center tile, the red path and the blue path. The left figure shows the path in the hyperbolic plane while the right figure shows the path in the 3x3 grid.

Figure 2: *Holonomy in the hyperbolic world*

2. General approach

This report solves the problem using 2 different approaches. By modelling the problem as a graph traversal problem and by using domain-specific rules.

When modelling the problem as a graph, inspiration can be taken from existing graph traversal algorithms. The used graph traversal algorithms will be further explained in section 4. The graph that is traversed by the algorithms is defined as follows. Each Node is defined by a unique combination of 4 parameters: the current location in the grid (CG), the rotation the user has in the grid (RG), the current location in the plane (CP) and finally the previous location in the plane (PP). Each edge is defined as a possible step that can be made from that position.

A node contains all the information to make sure each node is unique and neighbouring nodes can be computed. The CP and CG together point to the locations the node represents. The CG and RG are used to determine what steps can be made from that location and thus determine the edges of the node. And finally, CP and PP can be used to determine the rotation in the hyperbolic plane which is important when determining to what node an edge will connect.

It is necessary that each node has the capability to compute its neighbouring nodes since the graph is lazily generated. It is lazily generated because the hyperbolic plane is infinite and grows exponentially the further away it goes from the origin. So even just pre-generating significantly far away from the origin is infeasible because of its exponential nature.

Not modelling the problem as a graph allows quick predictions by using problem-specific rules. This is possible since both the grid and the hyperbolic plane don't contain any unpredictable problems. These rules create patterns of movement that can be used to make predictions. Unfortunately, there is no previous work to build on in this approach.

3. Background

When finding the shortest path (in a point-to-point problem) there exist solutions that will always guarantee the optimal path. But sacrificing that optimality guarantee for a speedup is also possible, and depending on the problem either of these could be the preferred solution. For that reason, both of them will be looked at in this section. This section will also explain the underlying data structure the game uses for the hyperbolic plane part of the problem

3.1. Underlying data structure for the hyperbolic tiling

Each location in the Order-5 square tiling is defined by the shortest path it takes to get there from the origin point (see figure 3). The origin point is the first point from which the algorithm starts keeping track of the traversed path. The shortest path is represented with the initial of one of the 4 cardinal directions as the first step. With further steps represented by the 3 directions left, forward and right, backtracking would just mean removing the last step.

The id of a location is found by adding a direction to a neighbouring location. If the new direction violates one of the rules described below, then a normalization operation is applied. This normalization operation will output the correct shortest path for that location (see Figure 3).

The normalization algorithm applies the following transformations:

$$\{x,R,R\} \rightarrow \{r(x),L\}$$

$$\{x,L,L\} \rightarrow \{\ell(x),R\}$$

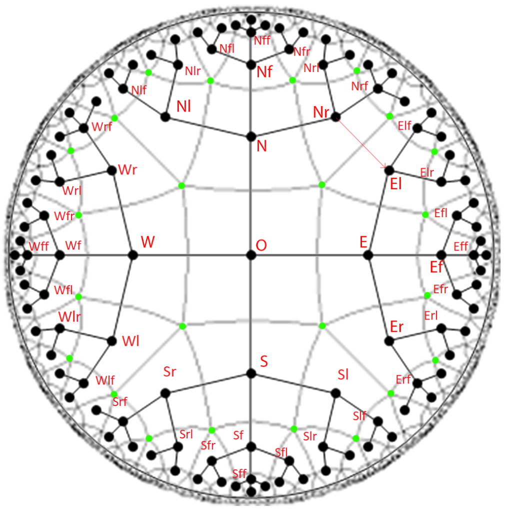
$$\{x,\{R,F\}_n,R,R\} \rightarrow \{r(x),L,\{F\}_n\}$$

$$\{x,L,\{F\}_n,L\} \rightarrow \{\ell(x),\{R,F\}_n,R\}$$

. R is a step right, L is a step left and F a step forward. $r(x)$ and $l(x)$ mean a rotation of the x step either right or left respectively (so $r(l)$ would become F. For more information or the normalization steps and conditions see [YBS*22].

3.2. Algorithms that provide provable optimal results

Depending on the type of graph that needs to be traversed, different algorithms are needed to guarantee optimal results. A simple breadth-first search can find the shortest path in $\mathcal{O}(n+m)$ time



Grey vertices are the edges of the tiles in the world, black vertices are the shortest paths to get to a room and define their location ids. Adjacent tiles that are not connected by a black edge require a normalization operation.

Figure 3: *The underlying data structure*

for unweighted graphs. While for weighted graphs (with only positive edges), the most notable algorithm is Dijkstra's algorithm (with Fibonacci heap) which can find the optimal shortest path in $\mathcal{O}(E + VlogV)$ [FT87]. These algorithms are the asymptotically fastest ones under the most general conditions. But if it is allowed to make more assumptions about how to calculate the shortest path further speedups can be made by using more specialized algorithms.

For example, by allowing the computer to do some precomputation, the precomputation that is done differs from implementation to implementation but it is often some variation of already precomputing the shortest paths between node pairs. Selecting a set of landmarks that guarantee optimal results is also possible [GH03] but too long to be practical for huge graphs (non-optimal landmark selection is way faster and it is thus further discussed in the next subsection).

Another possible way to speed up the algorithm (for some graphs) is by making it a bi-directional search, expanding the search area not only from the start node but also from the goal [PP09].

Other variations and possible combinations of the above suggestions are also possible as suggested in this paper [GKW05], which proposes an algorithm that combines bi-directional search, Dijkstra and precomputation. It also uses the fact that the problem domain models the real world, and often certain heuristics can be found in real-world problems [Mir20]. An often-used algorithm which incorporates these heuristics is A*. A* provides provable optimal results as long as the heuristic that is used is admissible [HNR68]. A Heuristic being admissible means that the value of the heuristic from a node *n* is always less or equal to the value of the optimal path from *n* to the goal.

3.3. Algorithms that provide estimations

With A* further speedup can often be achieved if more weight is put on the given heuristic. So the cost of a node will then be defined as: $f(n) = g(n) + w * h(n)$, where $g(n)$ is the actual cost to get to n, $h(n)$ the heuristic and w the weight that is applied. This does often violate the optimality constraint, though a bound on the suboptimality of the given result is also guaranteed [Pea84].

A* can further be modified to become Anytime A* [LGT03]. Anytime algorithms are algorithms that are given a time limit after which they will return an answer. They often do this by first finding a very rough estimation of the answer in a very small timeframe. And then use their remaining time to keep refining the result they have gotten.

precomputations can also be used here to make algorithms faster. For example by computing *landmarks* [PBCG09]. Landmarks are nodes that are picked using various methods and are used to create a precomputed network. This precomputed network only includes these landmarks and is thus drastically reduced in size making computations on them faster, and making it possible to compute the fastest paths between each node pair in this new graph. Now it is possible to get an approximate path by computing the path to the closest landmark from the starting point. And from the landmark that is closest to the goal, to the goal. Since all of the paths between those two landmarks are already precomputed they can simply be read from wherever the precomputed paths were stored.

Depending on what landmarks are selected it is possible to guarantee optimal results. But selecting landmarks in a way to guarantee optimality often takes too long. So some landmark selection algorithms make use of certain heuristics (different ones depending on the implementation) to drastically speed up this landmark selection process and make it a viable strategy for quickly computing a good estimation for the shortest path. [PBCG09]

4. Graph approach

This section will go over the different graph traversal methods that are used in the performance tests. Some of the methods discussed are already commonly used and researched algorithms and this section will thus not go too much in-depth into the specifics and will just give an overview of the idea behind the algorithms which are assumed to be implemented on an unweighted graph as this section talks about implementation details. But before we go into the details of the traversal algorithms, some prerequisite knowledge is discussed.

4.1. Hyperbolic plane operations

Each location in the hyperbolic plane is identified by the shortest path to get to that location. This shortest path can be seen as a vector of the hyperbolic space. Using these vectors it is possible to find out the relative position of location B from location A. This is done the same way one calculates the difference between two vectors x = B - A. Applying this operation can not be done by just adding or subtracting two numbers like in a Euclidean space. But one actually needs to walk along the entire path (starting from B and walking to A) since it is possible that some normalization operations need to be made, and these normalization operations can only normalize one illegal step at a time (at least in the used implementation, see section 3.1). To get -A, A will be flipped (the first step becomes the last and so on) and reversed (a right step becomes a left step).

Just using normal vectors (using numbers instead of paths) is not practical in a hyperbolic space. Even though ways exist to map hyperbolic coordinates to a vector (for example by using polar or Poincaré coordinates), they all have the problem of quickly running into floating-point errors due to the exponential nature of hyperbolic space, and are therefore not suitable for our use case.

4.2. Breadth first

Breadth-first traversal will always find the shortest path in the case of an unweighted graph, which the graph that models the problem is. So this algorithm will find the optimal path in the problem case. It is a slow algorithm but it is selected to use as a base case to compare the other methods against because of its simple and widespread nature.

The idea of a breadth-first traversal is that it always picks one of the closest (to the origin) unexplored nodes. It does this by putting all the children of the current node (the children save what node they came from) in a queue and pulling from that queue when selecting the next node. When it reaches the goal it walks back to the start by repeatedly going to its parent node. The reverse of the path it walks back to, will be the shortest possible path.

4.3. A* with scalar

A* is a variation of Dijkstra that incorporates heuristics. The heuristic that is used in the tested implementation is the length of the fastest path from Point A to B in only the hyperbolic plane. This length can be calculated by using the vector difference operation explained in section 4.1. This heuristic satisfies the admissible property (as the shortest path without constraints is always at least as fast as with constraints) and will thus guarantee an optimal path. [HNR68]

It is possible to break this optimality guarantee though in return for an algorithm that converges to a result quicker. By increasing the heuristic with a scalar that is bigger than one. This results in an algorithm that puts more focus on decreasing the heuristic (in this case the distance between the goal and the current node).

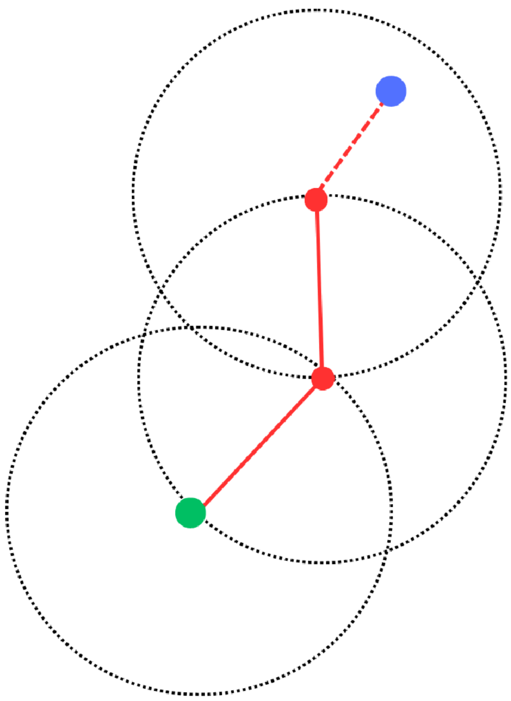
4.4. Anytime A*

Anytime algorithms are a subset of algorithms that can return an answer at anytime. Though the answer it returns might not always be a good answer. For the current use case (see here [YBS*22]) it is more important that the answer is returned below a certain threshold (when it takes long enough to be noticeable for the player) than to get the most accurate result. It doesn't matter if the result that is returned does not actually reach the goal yet as the player that needs the shortest path can not see further than 3 tiles away, as can be seen in Figure 1 in which the player can not see outside of the 3x3 grid and can barely see the flag in the minimap (the bottom figure) when the flag is 3-4 tiles away. So, the purpose of this algorithm is to quickly give the start of the shortest path.

The implementation that is tested in this report uses the A* with a scalar implementation, which is explained above, as its base. But now with an additional parameter that gives the algorithm a time limit. While searching for the shortest path, the algorithm keeps track of the traversed node that is closest to the goal (in the hyperbolic world) and saves that node. When the time limit has run out (or if the algorithm has found the optimal path), it returns the path to get to that closest node, though it leaves out the last step as initial tests have shown that the last few steps are still exploring the space and have not yet converged on the optimal path.

4.5. precomputed areas as nodes

The environment of the problem contains a lot of symmetries that this method tries to take advantage of. The most important one is that every spot on the tiling has the same tiling around it because the hyperbolic space is an Order-5 square tiling. Meaning that it is always possible to see any location on the hyperbolic plane as the origin spot. The core idea of this method is to take advantage of that fact by precomputing as large an area as possible around the origin and reusing that precomputation as often as possible (Figure 4 gives an idea of how that would look like).

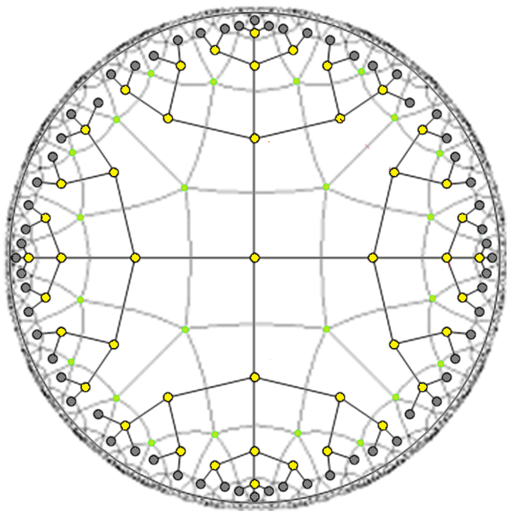


The green dot in this figure is the starting point, the blue dot represents the goal, the dotted circles represent the precomputed area around each node and the red line/nodes represent the walked path.

Figure 4: *A possible path using abstraction nodes*

This will be done by constructing a graph with new rules. Now every node is represented by an area centred around a point in the hyperbolic plane instead of a single point (See Figure 5). Each node now also keeps track of the path that was walked to get there instead of keeping track of its previous node (as finding out what child it was to the previous node requires extra computation). The other values that the node of the original graph contains are also present in this node. Since the position in the 3x3 grid and the rotation in both the grid and the plane still have an influence on the possible paths that can be walked. Despite of the changes, there are still as many nodes as before, representing the same positions.

The edges in this new graph are different from the original graph representation, as the neighbours of a node are now the nodes that lay on the edges of the precomputed area. Each of these edges is now also weighted by the number of steps it took to get to that location. This new graph will thus have many more edges per node but fewer nodes that need to be traversed to get to the goal. It should also be combined with one actual traversal algorithm as this method is more about restructuring the graph and does not inherently have a traversal algorithm linked to it.



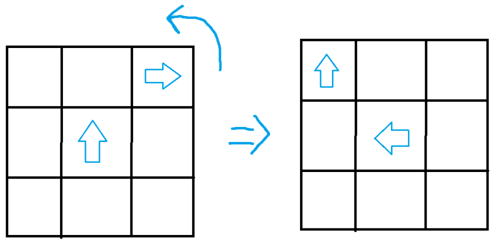
The new abstraction node represents the area of all yellow and grey colored nodes. If the goal is present in one of these nodes then that will be the only neighbour that is returned. If the goal node is not present, then it will return all of the grey nodes that are closer to the goal than the center node is to the goal.

Figure 5: *A small abstraction node of radius 4*

When a new node gets a call to return its neighbours the first thing it does is check if the goal location is present within the radius of the precomputed area. It does this by shifting the perspective as if this new node was the origin node which makes it possible to use the precomputed data. Which is done by first calculating the difference between the current location and the goal location (using the method of section 4.1). If the goal location is present in the precomputed radius around the new node it will only return the goal node. If the goal is not present in the area, it will return every location that lies on the edge of the precomputed area. But as a way to trim the neighbours that are returned, the method checks if a neighbour is actually closer to the goal than the current node and will only include it in the list of returned neighbours if that is the case (based on initial testing no detrimental effects on the returned solutions were found by including this if the radius of the precomputed area was larger than 3).

The amount of precomputation that needs to be done is large. Since the path can start on any of the 9 tiles in the grid and it can start in any of 4 rotations on both the grid and hyperbolic plane. So when the place in the hyperbolic plane is excluded, there are still a total of 144 starting positions. But this can be significantly reduced by taking advantage of the symmetries present in the problem. The rotations present in the hyperbolic plane can be dealt with by rotating the end goal since the Order-5 square hyperbolic plane is symmetric in each of its cardinal directions. The rotation on the grid can be dealt with by selecting the shortest path of a different tile (like is explained in figure 6). And finally, we can reduce the number of locations we need to calculate in the centre tile since it is symmetric for any rotation. So any location that differs only by their starting direction (North, East, South or West) can be shifted to any other starting location by changing the starting rotation.

At the moment there is also the problem that the resulting location of adding two hyperbolic vectors together can only be found by walking along the hyperbolic vectors. This is quite detrimental as this means that even with the answer provided by the precomputation that was done, there is still a need to traverse all the nodes in the path. This results in this implementation visiting more nodes instead of fewer as the same location can be travelled twice.



By rotating the entire grid, the starting position will keep the same possibilities for any future steps but shifts its rotation by changing its location in the grid.

Figure 6: *How to use the symmetry of the grid.*

5. Domain specific prediction

The shortest path can also be calculated without first mapping the problem to a graph, by taking advantage of some problem-specific knowledge.

Unfortunately, the implementation this paper will use will come with some drawbacks, since this method has not been extensively researched and refined. First, the implementation will always assume the player starts in the centre tile. And secondly, the implementation doesn't offer the optimal shortest path to the goal (the suspected reason for this will be discussed in Section 7). With these assumptions in mind the question rises: how does the program actually generate a path to the goal without traversing a graph?

There are certain paths that give an offset in the virtual world while not having an offset in the grid (see Figure 7). Meaning we can have an offset (in any direction) and still return to the position we started at in the grid. Since the end and start positions are the same, it is possible to link these paths together to traverse a path without keeping track of our current position. This comes with some problems.

Even though the player returns to the place they started at, the rotation in the grid and in the plane has changed. By using the same tricks that are explained in section 4.5, this can be solved. But this will lead to some backtracking, which can be resolved by removing the last step and starting from the second step in the predefined path while correcting it for the new rotation.

The path it tries to follow is the shortest path to get to the given goal without interference from the 3x3 grid. This path is the id of the location and thus can be easily acquired.

The last 3 steps of the path do not use the domain specific rules but an A* implementation since with the current constraints it will only finish if it reaches the goal at the center tile. It would create unnecessary extra steps to move the goal to the center tile if the domain specific algorithm also did the last few steps.

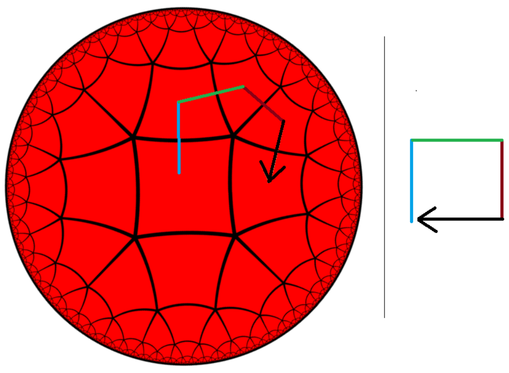


Figure 7: *The path has an offset in the hyperbolic plane but not in the grid*

6. Experimental setup and results

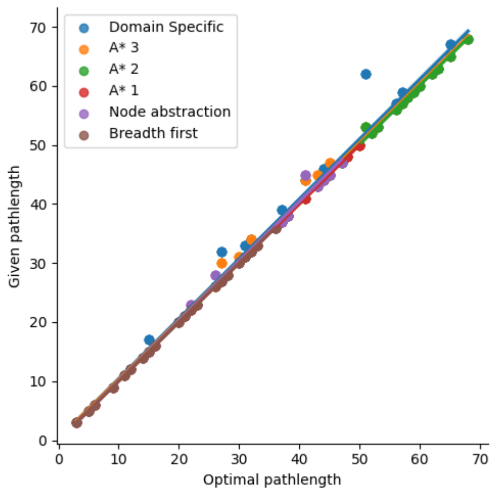
In order to demonstrate the effectiveness of each algorithm they will each be run on a selection of problem instances. This selection is a spread of problem instances, each with a different optimal path length. This has been done instead of a selection sorted by their radius away from the origin. Since even problem instances at the same radius from the origin can have differing optimal path lengths and the path lengths are what decide the complexity of the problem (for example Nfff and Nlrl are both a radius of 4 away but the optimal path of Nfff is 9 steps long and that of Nlrl is 8 steps). This selection has been made by using the A* algorithm (which has an optimality guarantee) to generate the optimal paths for a set of problem instances and selecting only instances with a unique length according to a relatively even spread.

Each algorithm has been run on each of the selected instances 5 times to take into account any natural deviation in the results. These results have been converted to a graph form as seen in Figure9 and Figure 8. These graphs should have confidence intervals but since the natural deviation is so small they can't be seen.

Some of the slower algorithms have not been run on the more complex problem instances since a limit of 60 seconds has been included into the testing set up. When an algorithm has a runtime of more than that limit it is excluded from the more complex instances. This has been done to prevent the analysis of results from taking too long. The results before that point will already show a clear trend in runtime performance.

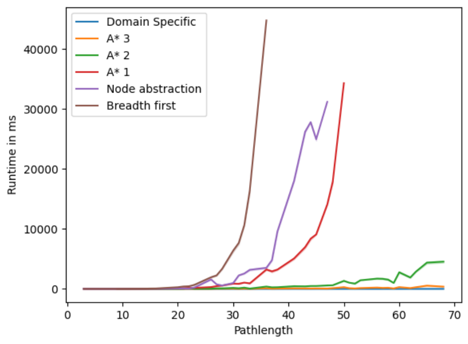
The results are valid as proof of the different performances of the algorithms as the variation between them is large and the natural deviation is quite small, requiring only a small test sample to get a valid result [CW19].

The anytime A* algorithm is tested differently since it tries to optimize different parameters. To test it, several instances of the algorithm will be run, each with a different time limit and scalar. If the resulting paths do not reach the goal (which most will not), then the optimal path will be calculated starting from where the anytime algorithm ended and the results will be combined. The difference between the length of the path using the anytime algorithm as a start and an optimal path will be calculated with a difference of zero meaning the anytime algorithm has correctly predicted the start of the optimal path. These algorithms will be run on 21 different problem instances (with a path length between 50 to 70) and the results will be aggregated (See Figure 10)



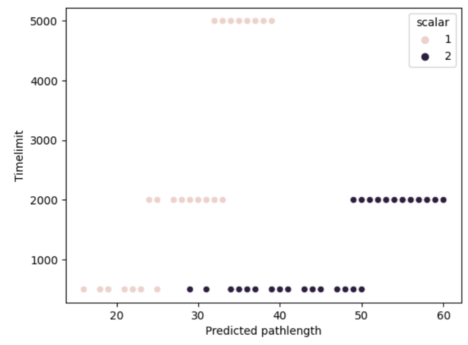
A scatter plot with regression lines. The x-axis displays the length of the optimal path and the y-axis the length of the path that the algorithm actually returned.

Figure 8: *Results of the predicted path length.*



A line graph. The x-axis displays the length of the optimal path and the y-axis the runtime in ms.

Figure 9: *Results of the algorithm runtimes.*



A scatter plot. The x-axis displays the length of the anytime algorithm's prediction and the y-axis is the time-limit that was given to the algorithm. The difference between the results is not displayed since the difference in all the results was equal to 0.

Figure 10: *Results of the anytime algorithm.*

There are some results that are not easily seen from the graphs(or simply not displayed) but are worth noting.

1. The runtime of the domain-specific prediction algorithm appears flat in the graph and this seems to hold even when the complexity of the problem is increased several fold.
2. As can already slightly be seen in Figure 3, the prediction of the domain-specific prediction algorithm becomes worse the further away the path gets from the origin. The other algorithms do not seem to have this, or have it to a lesser non-noticeable degree. This effect is not very noticeable at the tested path lengths but some of the tests not included in the graph have shown regular suboptimal path lengths at further away locations. These test however do not give a clear picture as they were comparing 2 suboptimal algorithms against each other, since the algorithms that guarantee optimal results are far to slow to find the shortest path on problem instances of path length 200+.
3. The anytime algorithm seems to always return the correct start of an optimal path, as the difference in the path length when starting from the starting point, or starting from the end of the path the algorithm returned, is always 0, at least according to the tested problem instances.

7. Discussion

7.1. Node abstraction method

The node abstraction seems to make a search algorithm both slower and make the results worse. The reason (or at least one of them) why the results seem to be worse, is that the shortest path algorithms that are proposed in this report do not uphold the triangle inequality. Meaning that if the shortest path from point A to point B goes through C then the shortest path to C is contained in the shortest path from A to B. The reason this is not the case for the proposed algorithms is that they are only finding the shortest path to one of the multiple endpoints. Since the endpoint can be in any place on the grid in any orientation to be a valid endpoint but to actually use this endpoint as an in-between stop, a certain orientation or endpoint might be needed. An example of this can be seen in Figure 2 where the shortest path to N is to just take a step up, but instead the paths take 4 steps to walk a circle to get to N so they have the correct orientation to walk toward Nf.

So if one wants to make use of precomputed shortest paths to get an optimal path, one would need to make many more precomputations, one for each specific possible endpoint (which would be 144 different points) and explore each possible one which would slow things down even more.

The reason (or at least one of them) why the method slows things down is that it visits nodes multiple times. Not nodes of the abstract graph but of the normal graph as those still needs to be traversed to find the new location of adding two paths together. As a method of adding two hyperbolic vectors together without walking along the entire path has not yet been found.

7.2. A* with scalar

It seems to be performing quite well. A* with scalar 1 is the fastest algorithm that was tested that can guarantee an optimal path. Though for paths that have a path length of 35+, the runtime will already be too large to be viable for all use cases.

A* with a scalar larger than 1 is quite fast and the suboptimality seems to be quite limited. It would thus be a good algorithm to guide a person with realtime updates in a virtual environment (like the one this paper was built upon [YBS*22])

7.3. Anytime A*

Anytime A* seems to be performing really well in predicting the start of an optimal path. It is known however that the A* with scalar 2 algorithm will sometimes return a suboptimal path even with no time limit. That the anytime algorithm that uses this as its base does always return the optimal path seems unlikely, so a larger sample size might be required to display more accurate results.

Its use cases will also be limited since it does not actually return a path that will reach the goal. One use case in which it would fit really well is the initially proposed case of helping a person navigate a hyperbolic space while confined in a limited space, as the user would only be able to perceive a limited radius of the world [YBS*22].

7.4. Domain specific prediction

This method is incredibly fast and will easily be able to find any path the player would need in an unnoticeable amount of time. The current downside is that the paths it returns are often not the optimal paths, this is especially noticeable in more distant paths.

We suspect the reason for that is the given path it walks. It currently takes the id of the goal (which is the shortest path without taking into account the 3x3 grid) and follows that path as closely as possible within the 3x3 constraints. But we have observed many optimal paths that deviate from that path and got better results. So it might be that it would be better to follow a different path. To determine what that path would be more research is needed.

8. Responsible research

All of the used methods and their implementations have been explained, furthermore, the experimental setup has clear guidelines that should make repeating the experiments that were done, possible. The experiments also do not require any specific or hard-to-acquire hardware that would make checking the validity of the experiments inaccessible to a majority. The absolute values would vary depending on the hardware but the ratio between the methods is the important part, not the absolute values.

As this research does not contain any user test no user confidentiality can be breached.

9. Conclusion and Future work

The answer to the question of what algorithm can compute the shortest path between the player and the objective with a runtime that isn't noticeable to the player, is that it depends on the circumstances. As there are two things to take into consideration: how far away the objective is and does the returned path need to be the shortest path or is a good estimation enough? Since if the objective is close, an A* with scalar 1 algorithm will give the shortest path quickly. But if an objective is more than 35+ steps away it is not possible to give the shortest path quickly enough to be unnoticeable. One would need to compromise and take an A* with scalar 2 or 3 algorithm which would give a good (but not a guaranteed optimal) path in a much smaller timeframe. And if the objective is very far away an anytime algorithm would do if not the entire path is required or, use the domain-specific rules to construct a path that will reach the objective very fast but will return an even longer path than an A* with a large scalar.

If further research will be done on this subject the topics with the most potential would be to make the path construction using domain-specific rules return an optimal path, as this could lead to an algorithm faster than any of the graph traversal algorithms could be. Or to find a way to add two paths together without needing to traverse all the steps between them, as a solution for that problem would also have a use in other topics that work with discrete hyperbolic spaces.

Acknowledgements

We extend our appreciation to Joris Rijsdijk for providing Figure 1 and To Scott Jochems 3 for providing Figure 3 of this paper.

References

[CCC*21] Cheng J.-H., Chen Y., Chang T.-Y., Lin H.-E., Wang P.-Y. C., Cheng L.-P.: Impossible staircase: Vertically real walking in an infinite virtual tower. In *2021 IEEE Virtual Reality and 3D User Interfaces (VR)* (2021), pp. 50–56. 1

[CW19] Campelo F., Wanner E.: Sample size calculations for the experimental comparison of multiple algorithms on multiple problem instances. 6

[DCC97] Darken R. P., Cockayne W. R., Carmein D.: The omni-directional treadmill: A locomotion device for virtual worlds. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 1997), UIST '97, Association for Computing Machinery, p. 213–221. 1

[FSW17] Frommel J., Sonntag S., Weber M.: Effects of controller-based locomotion on player experience in a virtual reality exploration game. pp. 1–6. 1

[FT87] Fredman M. L., Tarjan R. E.: Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 3 (jul 1987), 596–615. 3

[GH03] Goldberg A., Harrelson C.: Computing the shortest path: A* search meets graph theory. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (04 2003). 3

[GKW05] Goldberg A., Kaplan H., Werneck R.: Reach for a *: Efficient point-to-point shortest path algorithms. 3

[HNR68] Hart P., Nilsson N., Raphael B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (1968), 100–107. 3, 4

[LGT03] Likhachev M., Gordon G., Thrun S.: Ara*: Anytime a* with provable bounds on sub-optimality. In *Proceedings of (NeurIPS) Neural Information Processing Systems* (December 2003), pp. 767 – 774. 3

[Mir20] Mirjalili S.: Special issue on "real-world optimization problems and meta-heuristics". *Neural Computing and Applications 32* (05 2020). 3

[PBCG09] Potamias M., Bonchi F., Castillo C., Gionis A.: Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management* (New York, NY, USA, 2009), CIKM '09, Association for Computing Machinery, p. 867–876. 3

[Pea84] Pearl J.: *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984. 3

[PP09] Pijls W., Post H.: Yet another bidirectional algorithm for shortest paths. *Erasmus University Rotterdam, Econometric Institute, Econometric Institute Report* (01 2009). 3

[SPW*18] Sun Q., Patney A., Wei L.-Y., Shapira O., Lu J., Asente P., Zhu S., Mcguire M., Luebke D., Kaufman A.: Towards virtual reality infinite walking: Dynamic saccadic redirection. *ACM Trans. Graph. 37*, 4 (jul 2018). 1

[Wee21] Weeks J.: Body coherence in curved-space virtual reality games. *Computers & Graphics 97* (2021), 28–41. 1

[YBS*22] Yarar B., Bakker B., Snellenberg R., Slotboom R., Li W.: *"Holonomy": a non-Euclidean labyrinth game in virtual reality*. Tech. rep., TU Delft, 2022. 2, 4, 7, 8