Final Report **University of Bonn** September 29, 2016

Lab: Efficient Algorithms

Rademacher, Loka

1 Introduction

The goal of the lab *Efficient Algorithms and Selected Problems* was to engage with a given problem and state-of-the-art algorithms which solve them really well. The problem we are given was path finding on grid graphs and the main algorithm was jump point search with the pruning technique bounding boxes.

We wrote an implementation of this algorithm and some similar ones which are frequently used to compare them amongst themselves. The similar algorithms are different variants of jump point search and the well known A star algorithm. On top of implementing the algorithm we have added some tools to visualize the algorithms and to benchmark our algorithm variants. Our actual implementation is independent from the visualization so that the algorithm is also applicable outside of the visualization environment. The purpose of the visualization environment is to help to understand the algorithm.

The source code of the implementation will be available at `https://github.com/dhaunac/lab-jump-point-search` after the final presentation as all our other work relating to the lab.

2 Path finding on grid graphs

Our problem setting is to find the shortest path on a grid graph G. The drawing of a grid graph forms a regular tiling in the Euclidean space. Each tile which touches another one in the drawing is connected in the graph trough an edge. We refer to the missing tiles in the drawing as obstacles. The cost of each edge is determined only by the direction. An orthogonal edge has cost of 1 and a diagonal edge has cost of $\sqrt{2}$.

For that cost function on Euclidean grids there are a number of so called heuristic functions. The heuristic function estimates the cost of the total way between two vertices. The existence of such a function helps our algorithms to decide whether a walked path has led to a good or bad vertex. Formal speaking a heuristic function is a function $h : V \times V \to \mathbb{R}$. The constraint any heuristic function has to fulfill is the admissibility. A admissible heuristic is never overestimating the cost between two points, i.e. for all $a, b \in G$ it holds $h(a, b) \leq c(a, b)$. The other constraint is consistency which says that the heuristic cost between two points is not higher that the heuristic from a to a neighbor of b and the cost from that neighbor to b, i.e. $\forall a, b, c : h(a, b) \leq h(a, c) + c(c, b)$. A consistent heuristic is also admissible. If we would use a heuristic which is not admissible then all of our algorithms would not return a correct shortest path. A consistent heuristic improves our running time. All the heuristics we regarded are admissible.

Every problem instance has given two additional points. One of it is the start point and the other the goal point of that instance. This now yields a 4-tuple of grid graph G, start point s, goal point g and heuristic function h. Now the algorithm has to determine the shortest path from start to goal point [HG11].

3 Algorithms

The implementation of the following algorithms was the main aspect of the lab task. At first we give an outline of the algorithms so that the reader is familiar with those before discussing implementation issues.

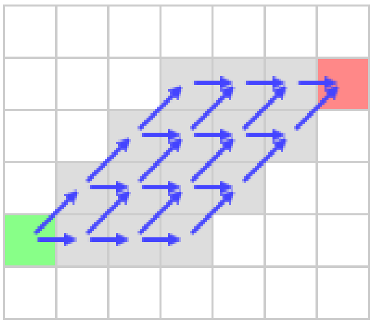
3.1 A star search $A^\star$

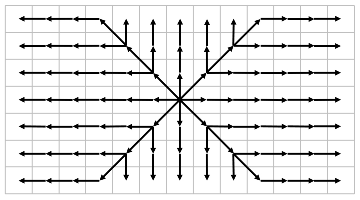
In this section we will discuss the $A^\star$ search algorithm [Wik]. The algorithm is basically a natural extension of the Dijkstra algorithm. It consist of an open list of next candidate vertices and a closed list of already processed vertices. The open list is usually some kind of priority queue and the closed list is a set.

The algorithm then gets a graph, a cost function and an admissible heuristic function as input. It starts with adding the start node into the open list and then processes a main loop. The main loop extracts the vertex v of the open list with minimum cost and expands all its adjacent vertices. The expansion step checks whether the vertices are not in the closed list and if so adds it to the open list with a cost value of the sum from the cost to v and the cost from v to the expanded point. After that add v to the closed list and repeat the main loop. The minimum cost is the sum of the cost value and the heuristic from that point to the goal node. So while progressing closer to the goal point the minimum cost might decrease.

3.2 Jump point search JPS

In 2011 the paper [HG11] introduced a new algorithm to find shortest paths. The main idea is to reduce the set of shortest path found by $A^\star$ 1a to only one single shortest path. That shortest path is the one which takes diagonal moves over orthogonal moves whenever possible. Figure 1b shows how to explore the map to find only this single shortest path by preferring diagonal moves over orthogonal ones.


(a) Different possibilities for the shortest path in $A^{\star}$.


(b) Natural order of exploration in JPS.

To calculate the shortest path efficiently we can make use of the following observation. As long there are no changes in direction on the searched shortest path we can skip those points and jump to the last vertex before the direction change. Thus it seems to be efficient to find all points where the direction may change and adds only this specific points to the open list and ignores the others. In the figures group below is shown how to proceed to find these so called jump points.

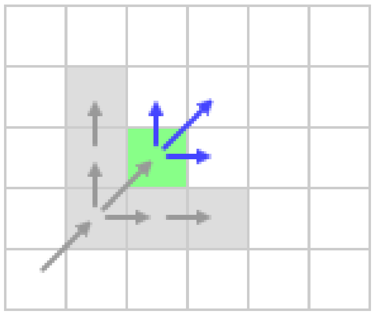
The figures 1a and 1b show how to proceed with diagonal movements which are preferred over orthogonal movements. First of all we explore the orthogonal directions which are aligned to the diagonal movement. Whenever the algorithm finds a jump poin during exploration the outgoing point becomes a jump point and is added to the open list. If there is no jump point found, we skip the current point, add it not to the open list and continue at the next point in diagonal direction. In case of an passed obstacle the point right behind the obstacle will not be reached by any of the orthogonal explorations. Therefore this field becomes a forced field and the current explored field becomes a jump point and is added to the open list. Exploring in orthogonal directions is quiet similar. We jump over all points in the required orthogonal direction until we found out that there is nothing or until we pass an obstacle. In this case there is a forced field behind this obstacle which is not found by any preferred diagonal movement, because obstacle hinders movement. This makes the current point become a jump point and its added to the open list.

The colored points in the pictures have a special meaning. The green point is the currently being explored point. The grey points are points which are not visited because they are reached faster by the predecessor of the current point. The purple points are forced fields of the current point and black points are obstacles.

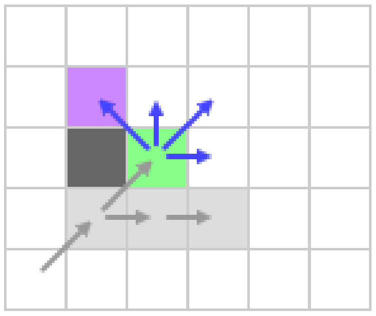
Whenever a point is taken from the open list to explore all directions to forced

Figure 1: JPS exploring rules [Wit].

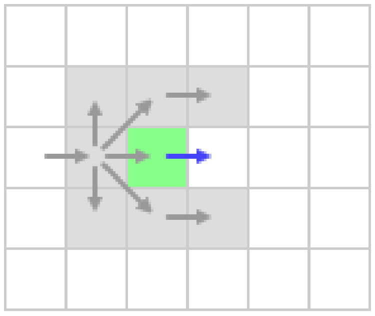
(a) diagonal moves.



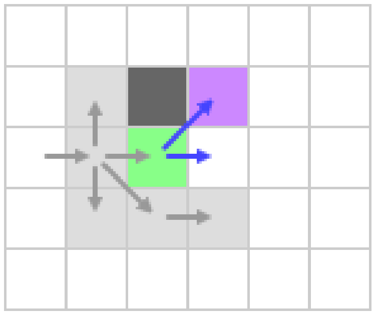
(b) diagonal moves with obstacle.



(c) orthogonal moves.



(d) orthogonal moves with obstacle.



fields, the orthogonal directions aligned to the diagonal direction and the current incoming direction are explored in this order.

3.3 Jump point search Improvements JPS^+

There is an improvement of JPS called JPS^+ [HG14]. Its a preprocessing technique to reduce the work of JPS during runtime. The preprocessing is done by calculating for every passable point and every outgoing direction the next jump point on the map and stores this information in a look-up table together with the cost for that jump. During the search at runtime the algorithm just takes in every explore step the stored value from the lookup table and adds the costs instead of exploring the map.

The goal point is not necessarily a jump point, because the goal point is not known while preprocessing. That has to be taken into account. To do so we add jump points at runtime whenever jumping past the goal point. Figure 2 is an example where we are at N jumping to S past goal point T. To avoid that we add J as the next jump point to the open list to find T.

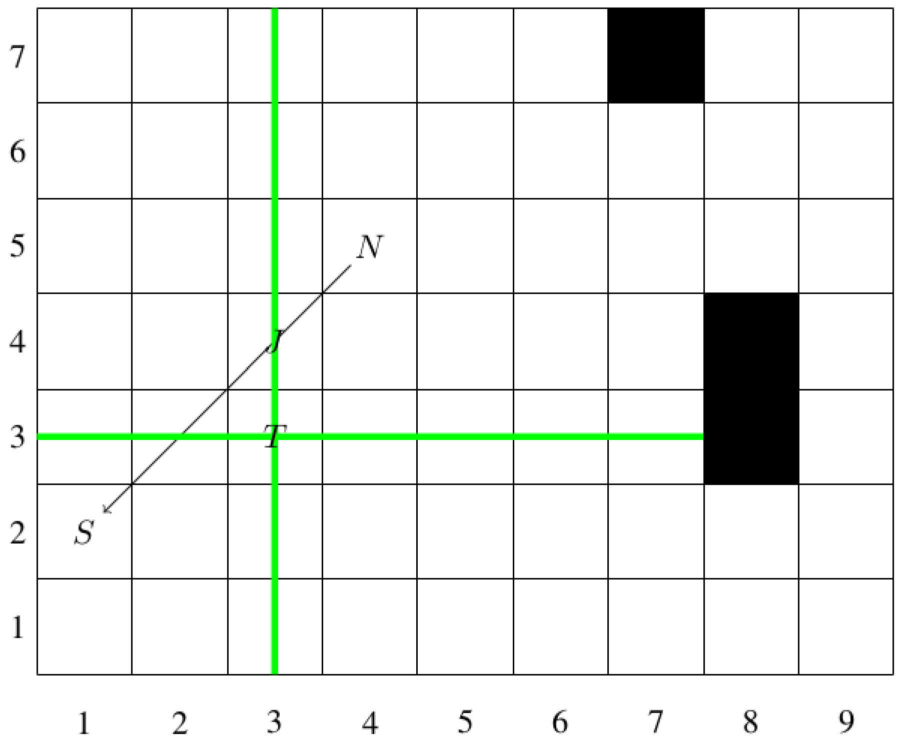


Figure 2: Goal point modification in JPS^{+}

3.4 Bounding boxes pruning BB

Bounding boxes is a pruning technique that can be applied to lot of different path finding algorithms [RS16]. The pruning requires some preprocessing which can be done offline, i.e. before the actual start and goal point are known.

For each pair of points and direction we have a dedicated bounding box. This box consists of all points which are can be reached optimally on a path from that given point. The box itself is the smallest geometric container, in this case a rectangle, to have every of those points inside. This implies that any point outside the box will be pruned since they will not help finding an optimal path.

The bounding depends on the underlying algorithms, in case of $A^{\star}$ and JPS it is sufficient to have minimum and maximum values for the X and Y coordinates. For $A^{\star}$ we can determine a predecessor p for the point v by inverting the direction. Then the bounding box of the point consists of all points which have a shortest path from p to those points using v as intermediate point. For JPS we can go even further and use the knowledge of our jump points and natural ordering. Here the pruning will be even more effective which can be observed in our visualization of the algorithm.

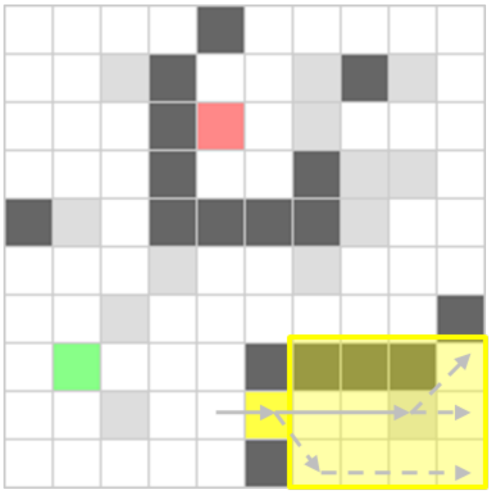


Figure 3: Visualization of bouding boxes.

4 Implementation

4.1 Application Core

The main goal and therefore the most important part of the lab work was to implement our algorithms. Our core offers an interface for external usage. One can use it as a library for any kind of application. We split up every choice of algorithm settings in a way everything can be combined as needed. The algorithm settings have the main point algorithmic structure, heuristic function and grid graph movement rules. The structure is the basis of everything, here you can choose between every algorithm described above. The algorithms are $A^\star$, JPS and JPS^+. Everyone can be additionally combined with bounding boxes pruning to speed up search. The heuristic functions can be chosen from one of the commonly used ones which are Manhattan distance, Euclidean distance or Grid distance. Grid distance is the shortest possible distance on any grid graph where no obstacle hinders the movement. Its also possible to choose no heuristic function which will for example make $A^\star$ behave like the Dijkstra algorithm. At last one has to choose the movement rule. The decides in which way an algorithm is allowed to move on the grid graph. One option is orthogonal movement where every diagonal edge is forbidden. The other options decide whether edge cutting is allowed. Edge cutting offers an option to use diagonal edges even though part of an obstacle is lying on the way.

4.2 Visualization and user interface

The visualization is the second substantial aspect of our work. To begin we will explain the basic working of the user interface. At first one can use the *File* menu to load any map. There are a number of different possibilities to create a random map satisfying a specific layout, for example perfect mazes or random rooms connected in a selected way. The next menu is *Edit* where the user can edit the map or change the algorithm settings. We offer $A^\star$, JPS, JPS^+ and each of them with additional BB pruning. Those algorithms are explained in section 3. On top of that one can modify each of those algorithms by choosing a heuristic and a moving rule. The heuristic modification leads into situations where different candidates of the open list will be explored. The change in the moving rule offers interesting observations when used with some kinds of jump point search and shows also the limits of jump point search. Then the user is supposed to pick locations for the start and goal point. Then the algorithm can be run after some prepossessing if necessary. We have separate steps for those actions, so one can easily modify the algorithm settings and see the changes after running the other algorithm settings.

The visualization will show the result of the algorithms. The red line between the chosen start and goal point indicates the shortest path. We have different colors for points which are in the open list or closed list. Every type of points can be shown or hidden by using the *View* menu.

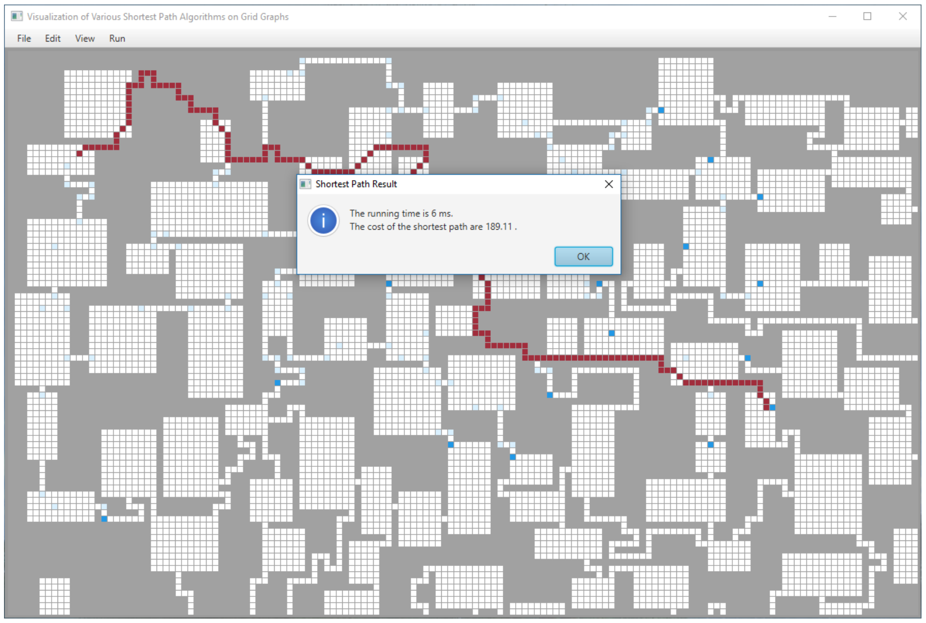
The Java implementation is written using JavaFX, the successor of Swing.

4.3 Benchmarking measurements

In addition to the visualization we wrote a benchmark application for terminal usage. This benchmark application could be used by other people who want to find out which algorithm settings is best for their real world application. We used it for a given set of benchmark maps and so called scenarios to find out how other our implementations are. The results of this benchmarking can be found in section 5. The usage of the terminal is simple. You can start it by using following command

```
# java -classpath $PATH_TO_JAR$/LabApplication.jar
  terminalapplication.Main $DIR$ $ALGO$
```

In that command we have three parameters where the name should be self-explaining. The accepted names of algorithms are *astar*, *astarbb*, *jps*, *jpsbb*, *jpsplus*, *jpsplusbb*. The directory has some requirements to be fulfilled. The first is that a directory with that name is a sub directory in both *maps* and *scenarios* directory. In the sub directory in maps there has to be maps with .map suffix and format encoding as in MovingAI [Stu12]. The same holds for


Visualization of Various Shortest Path Algorithms on Grid Graphs
File
Edit
View
Run
Shortest Path Result
OK

The running time is 6 ms.
The cost of the shortest path are 189.11 .

Figure 4: Screenshot of the user interface.

the sub directory in scenarios, where each file has to have the .scen suffix and appropriate encoding.

Each map will we processed along with the corresponding scenario file. To benchmark a single map it has to be in its own folder if that is necessary.

5 Results

This section presents the experiment results on the benchmark data from MovingAI lab.[Stu12] The experiments were executed by our benchmark application described in section 4.3. The benchmarking data are divided into different map types which are artificial maps like mazes, rooms, random maps or real maps from existing computer games. For each type there exist between 40 and 160 of map instances with 5000 to 15000 pairs of start and goal points. A pair of start and goal point for measurement is referred to as scenario.

Every map measurement that was executed by our application did match the same results as the original benchmarks. Therefore our algorithm implementations seem to be are correct and bug free holds. The plotting of our measurement data is built up into two steps. For all map scenarios the average value for the category (e.g. runtime) is calculated. The second step was to determine the whisker boxes of the resulting average values. This was done to avoid a weighting of maps, because different maps have a different number of scenarios.

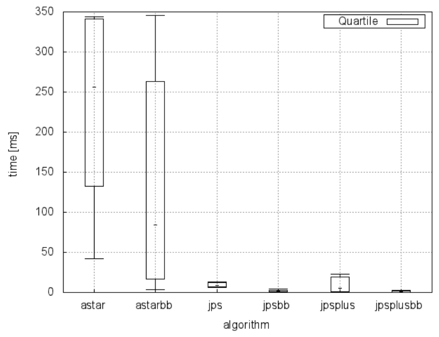
The first group of plots compares the runtime for searching a shortest path of different map types with different algorithms. $A^\star$ is performing very bad in every scenario when compared to any jump point search algorithm. In the figures 5a, 5b and 5c is shown that JPS^+ does not perform better than JPS and BB have a significant influence on the runtime on artificial map types. In contrast to the artificial maps we have the plots of real maps in figures5d, 5e and 5f. Here the influence of JPS^+ dominates the influence of BB. This seems logical since real maps include big areas of passable fields which have to be explored while executing JPS while exploring these big areas are done in preprocessing in the JPS^+ setting. In consequence JPS^+ makes just one big step while passing these areas while JPS explores them completely in runtime. Moreover figure 5d shows that bounding boxes can have a bad influence on the runtime, especially observable for on $A^\star$ and on $A^\star + BB$. Bounding boxes require additional time for each step to decide whether an expansion node can be ignored. If the pruning can be applied too seldom, then the cost for the bounding boxes check ups might be more expensive than the gain of exploring less points.

In figures 6a and 6b we compare different sized mazes where the size is the width of a floor in the maze. The result plot supports the observation made in the previous plots. The bigger the floors are, the more the results come close the real world maps. $A^\star$ performs the worse the bigger the floor size is. Meanwhile JPS^+ performs even better during a growing size. With growing floor size (resulting in more connected passable space) the influence of the different exploring algorithms dominates the effect of bounding boxes.

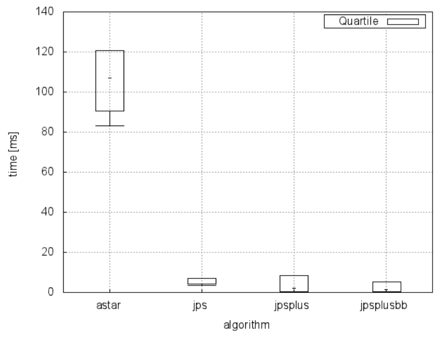
Figures 7a, 7b and 7c show that the preprocessing time for bouding boxes in independent of the underlying algorithm. This comes from the fact that the most part of the computation is the same in every case. JPS^+ is incredible fast and is completed in milliseconds even though bounding boxes may need several hours. Figure 7d shows that the preprocessing effort for bounding boxes in artificial maps is significant bigger than in real maps. That is because artificial maps have much more passable fields and calculation requires to explore the whole map for every passable field. The total effort therefore is $\Omega(m^2)$ where m is the amout of passable fields on the map.

Figure 5: Runtime to find the shortest path.

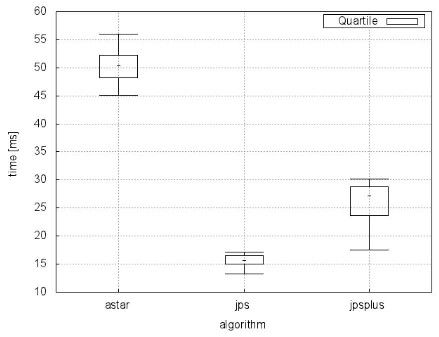
(a) Runtime in mazes.



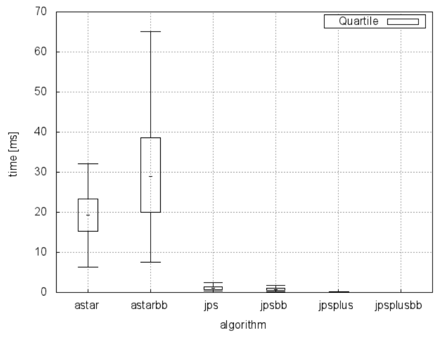
(b) Runtime in rooms.



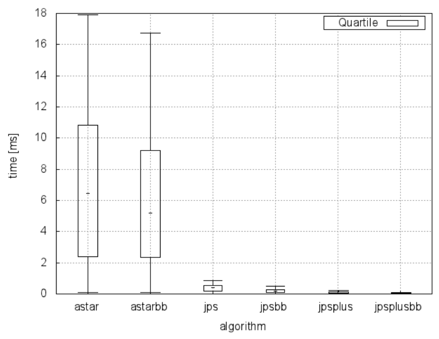
(c) Runtime on random maps.



(d) Runtime on Baldurs Gate maps.



(e) Runtime on Dragon Age maps.



(f) Runtime on StarCraft maps.

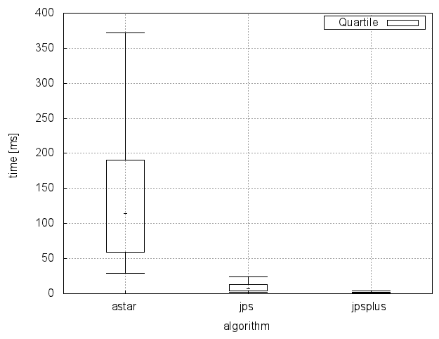
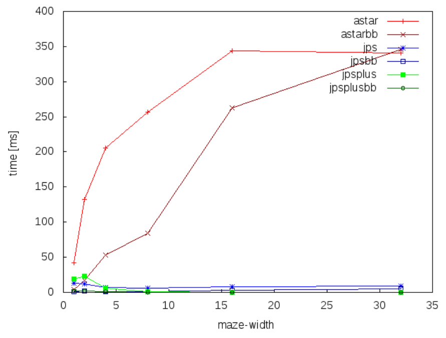


Figure 6: Runtime to find the shortest path in different mazes.

(a) Runtime in different sized mazes.



(b) Runtime in different sized mazes with logscale.

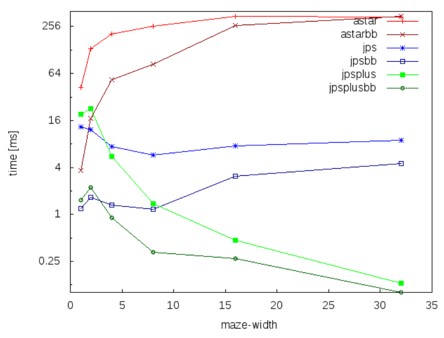
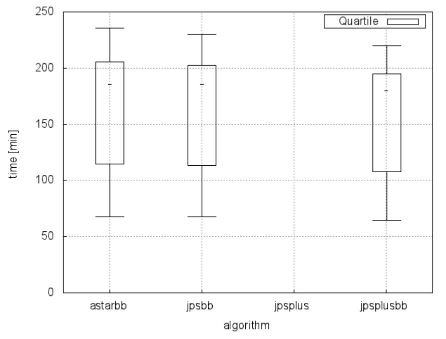
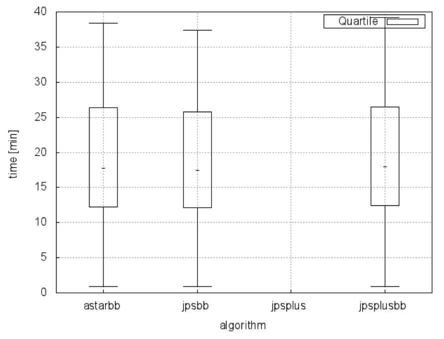


Figure 7: Preprocessing time to find the shortest path.

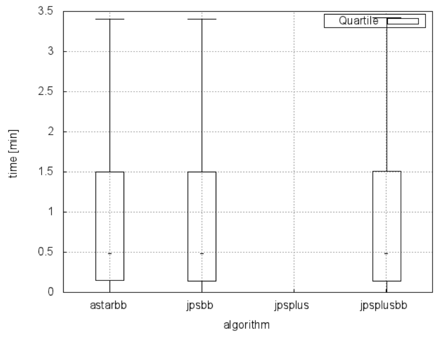
(a) Preprocessing time in mazes.



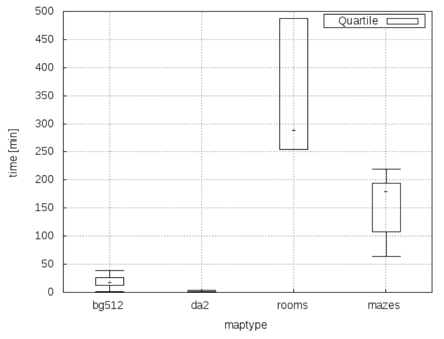
(b) Preprocessing time on Baldurs Gate maps.



(c) Preprocessing time on Dragon Age maps.



(d) Preprocessing time for JPS+ with BB on different map types.



References

[HG11] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011.* AAAI Press, 2011.

[HG14] Daniel Damir Harabor and Alban Grastien. Improving jump point search. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014.* AAAI, 2014.

[RS16] Steve Rabin and Nathan R. Sturtevant. Combining bounding boxes and JPS to prune grid pathfinding. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 746–752. AAAI Press, 2016.

[Stu12] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[Wik] Wikipedia. A-star search algorithm. `https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=734434637`.

[Wit] Nathan Witmer. Jump point search explained. `http://zerowidth.com/2013/05/05/jump-point-search-explained.html`.