# Two-Oracle Optimal Path Planning on Grid Maps

**Matteo Salvetti**
Università degli Studi
di Brescia, Italy

**Adi Botea**
IBM Research
Ireland

**Alfonso E. Gerevini**
Università degli Studi
di Brescia, Italy

**Daniel Harabor**
Monash University
Australia

**Alessandro Saetti**
Università degli Studi
di Brescia, Italy

**Abstract**

Path planning on grid maps has progressed significantly in recent years, partly due to the Grid-based Path Planning Competition GPPC. In this work we present an optimal approach which combines features from two modern path planning systems, SRC and JPS+, both of which were among the strongest entrants at the 2014 edition of the competition. Given a current state $s$ and a target state $t$, SRC is used as an oracle to provide an optimal move from $s$ towards $t$. Once a direction is available we invoke a second JPS-based oracle to tell us for how many steps that move can be repeated, with no need to query the oracles between these steps. Experiments on a range of grid maps demonstrate a strong improvement from our combined approach. Against SRC, which remains an optimal solver with state-of-the-art speed, the performance improvement of our new system ranges from comparable to more than one order of magnitude faster.

## 1 Introduction

Path planning on grid maps is an important AI problem, with applications in games and robotics. The problem has received a significant attention in AI research. In recent years, part of the progress is due to the Grid-Based Path Planning Competition, GPPC.* In this work we combine two successful approaches from the 2014 edition of the GPPC to develop a new pathfinding system that improves the best available speed results for optimal path planning on grid maps.

The first approach which we consider is SRC (Strasser, Harabor, and Botea 2014), a type of compressed path database (CPD) that relies on stored all-pairs shortest path data. Given any two nodes $s$ and $t$, this method can be queried to retrieve an optimal move from $s$ towards $t$. We call the method that retrieves such an optimal move an *SRC oracle*. SRC repeatedly invokes this oracle to compute an optimal path, or a prefix of any desired length.

The second approach which we consider is JPS+ (Harabor and Grastien 2014), a preprocessing based algorithm that exploits symmetries on a gridmap. When expanding a node JPS+ identifies a set of directions $\{\vec{v_1}, \ldots \vec{v_k}\}$, each of which leads to a successor called a *jump point*. It also tells the distance (in number of steps) from the current position to each jump point. We call this method a *JPS oracle*.

Our new method, called Topping (Two-Oracle Path PlannING), combines the two oracles as follows: the SRC oracle identifies a move to perform in the current position; the JPS oracle tells for how many steps the move can be repeated, with no need to query the oracles between these steps.

The main advantage of Topping is that one call to the JPS oracle is significantly faster than one call to the SRC oracle. To retrieve one optimal move towards the target at hand, SRC performs a binary search in a compressed string of symbols, representing optimal moves from the current position towards various potential targets. The JPS oracle answers a query with a constant-time lookup. Thus, at steps where the oracles are invoked, the overhead of the JPS oracle is small in comparison to the SRC oracle. At steps where no oracle call is needed, the savings are significant.

In Topping, the preprocessing time is comparable to the SRC preprocessing time. The memory requirements are increased however, due to two reasons. Firstly, we store information for two oracles instead of one. Secondly, to ensure the correctness of the approach, we need to synchronise the two oracles so that they break ties among multiple optimal moves in a similar fashion. As a side effect, this reduces the compression power of SRC. Standalone SRC can break ties arbitrarily and in a way that favours improved compression. However, JPS+ breaks ties by favouring paths where diagonal moves appear as early as possible. Imposing a similar strategy to SRC may rule out (non-diagonal) optimal moves that would achieve a better CPD compression. See an additional discussion in Section 4.

Until now, SRC has represented the state of the art in terms of speed for computing grid-optimal shortest paths. For instance, in GPPC-2014 (Sturtevant et al. 2015), SRC was found to compute optimal paths significantly faster than all other systems at the competition. In addition, SRC provides individual optimal moves in the order of 100 nanoseconds, or even faster (Strasser, Harabor, and Botea 2014). As other existing systems require to complete a search before telling the first optimal move, CPD-based systems, such as SRC, are orders of magnitude faster in providing a prefix of an optimal path. Given such a strong timing performance, achieving a better speed is challenging.

In experiments on a range of gridmaps from Sturtevant's



*http://movingai.com/GPPC/

repository (Sturtevant 2012) we report a significant speed improvement. The performance of Topping ranges from comparable to SRC to more than one order of magnitude faster. On average, Topping computes full optimal paths 3.84 times times faster than SRC. Moreover, when computing the first 20 moves of an optimal path, Topping is faster than SRC by a factor of 6.41 on average.

## 2 Related work

We focus our attention on well known and state-of-the-art works that improve the efficiency of grid-optimal path planning search. The literature in this area can be divided into two broad categories: works that are purely online, and works that employ offline preprocessing and auxiliary data to achieve greater speedup. Performance is often measured with respect to A* (Hart, Nilsson, and Raphael 1968).

For online grid path planning, one popular and very strong starting point is the symmetry-breaking approach known as Jump Point Search (Harabor and Grastien 2014) (JPS). Since JPS is a key ingredient for our work we describe it in some detail in Section 3.1. Recent variations of this method, which may further improve performance, are described in (Sturtevant and Rabin 2016). Other related works include studies into alternative search strategies, such as the bi-directional algorithm NBS (Chen et al. 2017) — which is sometimes preferable to A* on grids — and studies into more efficient implementations of the OPEN list, such as those evaluated in (Larkin, Sen, and Tarjan 2014).

For grid path planning with offline preprocessing the speed state of the art belongs to methods that exploit all-pair-shortest-path (APSP) data. Compressed Path Databases (CPDs) are a family of such methods and the variant known as SRC (Strasser, Harabor, and Botea 2014) is a key ingredient for our work. We describe these approaches in Section 3.2. In (Rabin and Sturtevant 2016) APSP precomputation is used to develop a method known as JPS+BB. This work is in some ways similar to our own and has more modest memory requirements. Its speed performance for full path computation is comparable to SRC, a method which in this work we significantly improve upon. A further advantage of our work vs JPS+BB is that we are able to compute a first optimal move, or a prefix of the optimal path, much faster, without solving the entire problem first.

A broader cross-section of recent grid-based path planning techniques from across the literature — including optimal and sub-optimal, some online and others preprocessing-based — appears in the results from the 2014 Grid-based Path Planning Competition (Sturtevant et al. 2015).

## 3 Background

A *gridmap* is a two-dimensional environment with $n \times m$ square cells or *tiles*. Each tile is marked as traversable or non-traversable and has up to 8 adjacent neighbours. Navigating across a gridmap involves transitioning from one tile to the next by repeatedly applying a *move* operator $\vec{v}$. There are eight possible moves corresponding to each of the four cardinal directions (N, S, E, W) and each of the four ordinal directions (NW, NE, SE, SW). Each cardinal (or straight) move has a cost of 1 while each ordinal (or diagonal) move has a cost of $\sqrt{2}$. A move $\vec{v}$ is *valid* when it is used to transition between two traversable tiles. In this work we also consider a number of common rules which further constrain the validity of a move. These are illustrated in Figure 1 and often appear in computer games and the literature.

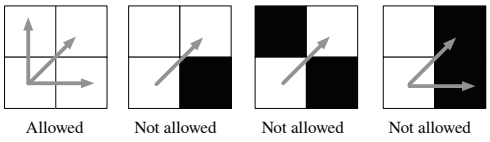


Figure 1: Examples of moves that are allowed and disallowed.

A *path* is a sequence of valid moves $\pi = \langle \vec{v_0}, \ldots, \vec{v_k} \rangle$ used to transition from a start tile $s$ to a target tile $t$. The cost of a path $c(\pi)$ is equal to the sum total of all its move costs. When $c(\pi)$ is minimum, among all paths between $s$ and $t$, that path is said to be (grid) *optimal*.

### 3.1 Jump Point Search

Jump Point Search (JPS) (Harabor and Grastien 2014) is the combination of A* (Hart, Nilsson, and Raphael 1968) with an online pruning technique that allows for fast and optimal pathfinding on a grid. JPS works by eliminating *path symmetry*; i.e., equivalences between sets of paths that are very similar and often found together on a grid. Consider, for example, a sequence of moves such as correspond to the path $\langle$ E, E, NE, E $\rangle$ which has a cost $\simeq 4.4142$. Obstacles notwithstanding, it is easy to see that there are several permutations of this path, all of which are equivalent w.r.t. the start and target location and all of which have the same cost. These permutations are obtained by taking the moves in a different order, such as: $\langle$E, E, E, NE$\rangle$, $\langle$NE, E, E, E$\rangle$, $\langle$E, NE, E, E$\rangle$

To avoid enumerating all such paths JPS applies a series of simple and recursive pruning rules during node expansion. The intuition is as follows: when generating a successor which can be reached along several different but symmetric paths, JPS prefers the path where diagonal moves appear sooner. All other equivalent paths can be safely pruned. Stated differently, this strategy imposes over the set of all grid moves a canonical partial order known as *diagonal-first*.

Diagonal-first pruning can dramatically reduce the branching factor of a node, often leaving just a single successor. By recursing in such cases JPS can avoid (i.e. "jump over") many non-branching nodes that would otherwise need to be explicitly expanded. The recursion stops when one of two conditions are met: (1) an obstacle is encountered, in which case the recursion is said to have failed and no successor is returned, or (2) the recursion reaches a so-called *jump point*: a node with multiple successors, all diagonal first, and none of which can be further pruned. By "jumping" from one node to the next JPS can improve the performance of A* search, in grid pathfinding scenarios, by an order of magnitude and sometimes more.

**JPS+** It is possible to further improve the performance of Jump Point Search using offline pre-processing. One such approach involves computing, for every node in the grid, the number of steps to the next jump point in each of the eight available move directions.

JPS exploits this information during online search to traverse the map more quickly: for each valid move $\vec{v_i}$ at the current node $x$, simply repeat $\vec{v_i}$ exactly $k$ times, where $k$ is the number of steps to the next jump point (or to an obstacle, if no jump point successor exists). Such pre-processing is very fast, typically requiring just a few seconds even for grids with millions of nodes. The memory overhead introduced by this procedure is at most eight integers per grid node. In our implementation an integer is 16-bits wide which allows a maximum jump distance of 65535 steps. Smaller integers can reduce the total memory size, possibly at the expense of more time per online search.

### 3.2 Compressed Path Databases

A Compressed Path Database (CPD) is a pre-processing based speedup technique which uses all-pairs path data to find optimal shortest paths without the need for state-space search. The idea is simple: given a grid with $m$ traversable nodes we compute and store a $m \times m$ matrix $\mathbf{M}$ where each entry $\mathbf{M}[i, j] = \vec{v_k}$ indicates the first move on the optimal path from node $i$ to node $j$. To extract a shortest path using $\mathbf{M}$ one need only recursively look up and apply the optimal move $v_k$, stopping only when $i = j$.

The matrix $\mathbf{M}$ is created offline. Populating each row requires a Dijkstra search, making for a total of $m$ Dijkstra searches. However, since every row is independent from all the rest the time requirements can be easily improved using parallel computation. To improve the space requirements of $\mathbf{M}$ — which in a naive encoding would require $O(m^2)$ records — a variety of row-based compression schemes have been suggested. Common to all is the following idea: group together nodes which are located in close proximity on the grid and which share the same first move. There are many possible ways to group nodes including quad-trees (Sankaranarayanan, Alborzi, and Samet 2005), rectangle decompositions (Botea and Harabor 2013) and run-length encoding (Strasser, Harabor, and Botea 2014). As a general rule bigger groups yield better compression.

We use SRC (Strasser, Harabor, and Botea 2014), a state-of-the-art CPD method which was also the fastest entry in GPPC-2014. On data such as GPPC gridmaps, SRC can extract a single move within the order of 100 nanoseconds or less. Its compression ratio is often in the range of 300 to 400, compared to an uncompressed table.

## 4 Our Approach

Algorithm 1 shows our approach in pseudocode. At line 3, the SRC oracle returns an optimal move $\vec{v}$ from a current state $s$ towards a target state $t$. The second oracle, based on JPS+, tells how many times the move $\vec{v}$ can be applied before querying the SRC oracle again (line 4). At line 5, a corresponding number of copies of the move are appended to the solution. The current state becomes the result of performing these moves (line 6), and the process repeats.

**Algorithm 1:** Method `Topping`

```
   input : init state s, target state t
   output: optimal path from s to t
1  π ← ∅
2  while s ≠ t do
3  |  v⃗ ← getMoveSRC(s, t)
4  |  c ← getNrStepsJPS(s, t, v⃗)
5  |  append c copies of v⃗ to path π
6  |  s ← makeMoves(s, v⃗, c)
7  return π
```

| Path length | CPU time A* | SRC | JPS+ | Topping | Speedup w.r.t. A* | SRC | JPS+ |
|---|---|---|---|---|---|---|---|
| $[0, 150]$ | 252 | 4.7 | 8.2 | **1.6** | 157 | 2.9 | 5.1 |
| $(150, 300]$ | 1948 | 9.7 | 41.5 | **2.8** | 676 | 3.3 | 14.4 |
| $(300, 500]$ | 5029 | 14.9 | 99.8 | **4.3** | 1160 | 3.4 | 23.0 |
| $(500, 750]$ | 9420 | 22.6 | 184.0 | **6.2** | 1502 | 3.6 | 29.4 |
| $(750, 1200]$ | 17024 | 35.2 | 437.0 | **6.3** | 2664 | 5.5 | 63.4 |
| $\geq 1200$ | 23039 | 63.1 | 527.0 | **14.9** | 1546 | 4.2 | 35.4 |

Table 1: Average CPU time (micro-seconds) required by A*, SRC, JPS+, and Topping to compute a full path, and average speedup factor of Topping w.r.t. A*, SRC, JPS+.

Algorithm 1 is obtained from the way that SRC computes paths with a slight but effective modification. Specifically, if we remove line 4, and use a value of $c = 1$, we fall back on the path extraction technique utilized in CPD-based approaches, such as Copa (Botea 2012) and SRC.

Our approach is a simple and effective combination of two oracles with complementary strengths, available in two modern pathfinding systems, SRC and JPS+. The combination is easy to implement. It requires integrating one call to a JPS+ method into the SRC path computation loop.

In addition, SRC and JPS+ need to break ties among optimal moves in a similar fashion. As said earlier, JPS+ prefers to make diagonal moves first. We have modified the Dijkstra search in the CPD preprocessing, to give priority to diagonal first moves. In other words, when two or more optimal moves are available from a current node $s$ towards a target $t$, and at least one is diagonal, the CPD returns a diagonal move. Intuitively, this tie-breaking synchronization is needed to stay within the framework of Lemma 1 below. The lemma is further needed to prove Theorem 1. The proofs are left for a longer report.

**Lemma 1.** *Consider a jump point $p$ and a move $\vec{v}$ available in $p$. Consider further that $p$ and $\vec{v}$ belong to an optimal path that is not pruned by the JPS+ rules. Then the entire segment from $p$ to the next jump point in the direction of $\vec{v}$ belongs to an optimal path.*

**Theorem 1.** *Topping computes optimal paths.*

## 5 Experimental Results

In this section, we present the results of an experimental analysis conducted with the aim of evaluating the effectiveness of using the Topping method.

| Path length | SRC | Topping | Ratio |
|---|---|---|---|
| $[0, 150]$ | 62.11 | **11.27** | 5.51 |
| $(150, 300]$ | 192.70 | **32.42** | 5.94 |
| $(300, 500]$ | 340.70 | **52.03** | 6.54 |
| $(500, 750]$ | 517.10 | **73.70** | 7.02 |
| $(750, 1200]$ | 830.90 | **120.30** | 6.91 |
| $\geq 1200$ | 1344.70 | **210.30** | 6.69 |

Table 2: Average number of binary searches performed by SRC and Topping in the CPD, and ratio of these average numbers for different path lengths.

The test data consist of 8-connected gridmaps ranging from about 538 to 137,375 nodes (Sturtevant 2012). We used 54 real game maps from games *Dragon Age: Origins*, and *Baldurs Gate II*. Each map comes with a set of shortest-path queries. Overall, the number of queries is 82,850. Maps are undirected. Straight edges have weight 1 and diagonal edges have weight $\sqrt{2}$.

The experiments were conducted using an Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz CPU running Ubuntu 16.04. The SRC code is taken from the 2014 GPPC repository.[†] The JPS+ code is taken from the public repository of the original author.[‡] All algorithms were compiled using g++ 5.4.0 with -O3.[§]

Table 1 compares for different path lengths the average time used by A*, SRC, JPS+, and Topping to compute the full path. In each case Topping is the fastest. Table 1 also shows the speedup of Topping as compared to the other approaches. For every considered path length but lengths longer than 1200, the performance gaps between Topping and A* and between Topping and JPS+ increase consistently with the path length increase. Topping is up to more than three orders of magnitude faster than A* and up to 63 times faster than JPS+. For paths longer than 1200, the performance gap is still largely in favor of Topping.

When comparing SRC and Topping, Table 1 shows that, for every considered path length, the performance gap is similar: Topping is from about 3 to 5 times faster than SRC. As mentioned earlier, in the approaches using SRC the CPD queries are performed by a binary search in a compressed string of symbols stored in the CPD. The reason why Topping is faster than SRC is a combination of related factors: querying the JPS oracle is much faster than SRC performing a binary search; and moves can often be applied repeatedly, with no need to query the oracles in between.

Table 2 shows the average number of binary searches performed by SRC and Topping in the CPD. The results indicate that Topping performs many fewer binary searches. Interestingly, for every considered path length the ratio between the average number of binary searches performed by SRC and Topping is about the same. This is the reason why the performance gap between SRC and Topping is relatively stable across different path lengths (i.e., the second last column in Table 1).

[†]https://code.google.com/archive/p/gppc-2014/

[‡]https://bitbucket.org/dharabor/pathfinding

[§]For an easier reproducibility of the results in the paper, our code and experimental data will be available on request.

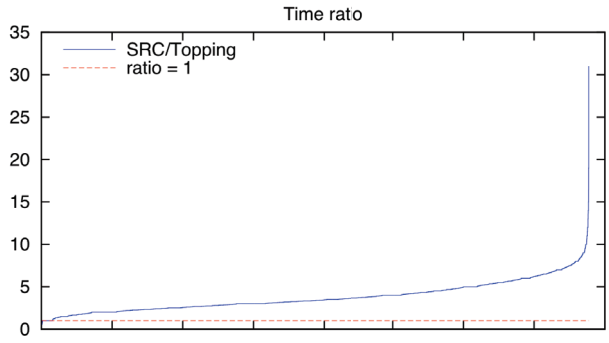


Figure 2: Performance gap between SRC and Topping. Queries are ordered so that the curve in monotonic.

| Systems | Mem [MB] | CPU time [$\mu$sec] Full path | 20 mov. | Speedup of Topping Full path | 20 mov. |
|---|---|---|---|---|---|
| SRC | 10.40 | 25.10 | 3.95 | 3.84 | 6.41 |
| JPS+ | **3.99** | 216.00 | 216.00 | 33.00 | 355.00 |
| Topping | 23.00 | **6.54** | **0.61** | – | – |

Table 3: Average memory, average CPU time to compute the full path, average time to compute the first 20 moves of SRC, JPS, and Topping, and average speedup factor of Topping w.r.t. SRC, JPS+ for all maps and queries.

Figure 2 shows the ratio between the time required by SRC and Topping to answer every query. Ratio values above the dashed line represent a speedup for Topping. It is interesting to observe that often Topping is more than five times faster than as SRC, and sometimes it is even more than one order of magnitude faster.

Table 3 shows a comparison in terms of CPU time and used memory for all maps and queries of our benchmark. Overall, Topping is the fastest path planning system in terms of time required to compute both the full path and the first 20 moves. However, Topping requires more memory than the other approaches. The memory used by Topping is about two times the memory used by SRC. As mentioned in Sections 1 and 4, there are two reasons for this: Topping needs to store the preprocessing information of JPS+, and the memory used in Topping to store the CPD increases.

## 6 Conclusion

We have presented an approach that significantly improves state-of-the-art speed results in optimal path planning on grid maps. Our technique is a combination of two oracles readily available in two modern path planning systems, SRC and JPS+. The need to synchronize the two oracles, in terms of their tie-breaking scheme across optimal moves, results in an increase in the size of a compressed path database.

In future work, we plan to further reduce the size of CPDs, and to further speed up the computation of optimal paths and path prefixes. A generalization to neighbourhoods larger than 8 is another interesting direction.

## References


Botea, A., and Harabor, D. 2013. Path planning with compressed all-pairs shortest paths data. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy*, 288–292.

Botea, A. 2012. Fast, optimal pathfinding with compressed path databases. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada*, 204–205.

Chen, J.; Holte, R. C.; Zilles, S.; and Sturtevant, N. R. 2017. Front-to-end bidirectional heuristic search with near-optimal node expansions. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia*, 489–495.

Harabor, D. D., and Grastien, A. 2014. Improving jump point search. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA*.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics* SSC-4(2):100–107.

Larkin, D. H.; Sen, S.; and Tarjan, R. E. 2014. A back-to-basics empirical study of priority queues. In *Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA*, 61–72.

Rabin, S., and Sturtevant, N. R. 2016. Combining bounding boxes and JPS to prune grid pathfinding. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, Arizona, USA*, 746–752.

Sankaranarayanan, J.; Alborzi, H.; and Samet, H. 2005. Efficient query processing on spatial networks. In *Proceedings of the Thirteenth ACM International Workshop on Geographic Information Systems, ACM-GIS 2005, Bremen, Germany, Proceedings*, 200–209.

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast first-move queries through run-length encoding. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic*.

Sturtevant, N. R., and Rabin, S. 2016. Canonical orderings on grids. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA*, 683–689.

Sturtevant, N. R.; Traish, J. M.; Tulip, J. R.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The grid-based path planning competition: 2014 entries and results. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, Ein Gedi, the Dead Sea, Israel*, 241–251.

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