Two-Oracle Path Planning Consolidated by Heuristic-Rich Information

Anonymous submission

Abstract

Grid-based path planning is a classic problem in AI, widely applied in robotics, computer games, and scheduling. Two-oracle path planning (Topping) is a state-of-the-art fast path-finding method for grid maps. Topping iteratively utilizes SRC and JPS oracles to determine the first moves and number of steps, respectively. This enables faster search than SRC, yet incurs high storage and search costs due to inadequate compression. In this paper, we aim to leverage heuristic information as much as possible to enhance the compression performance of Topping and to further improve the search efficiency (i.e., the first-move decision cost). Ultimately, this also improves Topping's overall search performance. Experiments on five benchmarks (478 maps in total) show that our methods can reduce the first-move decision cost by an average of about 60% (maximum 71%) and achieve a maximum speedup of 48% in runtime. Remarkably, they also have gains in compression performance and reduce storage costs.

Introduction

Path planning is a classic problem in artificial intelligence, widely used in real-world scenes such as video games, robotics, navigation software, drone swarm coordination, and so on (Antsfeld et al. 2012; Cui and Shi 2011; Delling et al. 2017; Freund and Hoyer 1986; Hönig et al. 2018; Shen 2023; Sturtevant 2012b). In recent years, grid-based path planning has received significant attention, driven by the Grid-based Path Planning Competition (GPPC) (Sturtevant et al. 2015), resulting in the emergence of a number of excellent methods. One notable achievement inspired by the GPPC 2014 is the Two-Oracle Path Planning (Topping) (Salvetti et al. 2018).

Topping is a state-of-the-art method in search speed. It combines two oracles, SRC oracle and JPS oracle, using Single Row Compression (SRC) (Strasser, Harabor, and Botea 2014) and Jump Point Search + (JPS+) (Harabor and Grastien 2014) as the core respectively, to find the optimal path on grid maps. Both SRC and JPS+ are winners of GPPC 2014. Topping iterates over the following two steps when searching for the optimal path:

1. The SRC oracle determines the optimal first move from the current node to the target in compressed first-move arrays.
2. Then, the JPS oracle calculates the number of steps to the next jump point based on the first move provided by step 1.

Topping achieves ultra-fast speeds. However, to collaborate effectively with the JPS oracle which breaks ties by *diagonal-first*, the SRC oracle prioritizes diagonal moves over compression-friendly moves when deciding the first move, resulting in limited compression which in turn leads to larger CPD size and higher storage cost.

In the next section, we provide an overview of related work. Then, we provide background on Topping and the key technologies utilized. After that, we provide a detailed description of the ToppingH, ToppingPW, and ToppingRPW along with description diagrams and algorithm pseudo codes. We present our experiment results and analysis, finally concluding with future work.

Related Work

Compressed Path Databases (CPDs) (Botea 2011; Botea and Harabor 2013) is a state-of-the-art in static grid path finding, enabling rapid extraction of the optimal path and first move without state-space search. Their main drawback is the high construction cost, requiring all-pairs precomputation.

Therefore, Single Row Compression (SRC) (Strasser, Harabor, and Botea 2014), an advanced CPDs method and the winner of the GPPC 2014, uses Run-Length Encoding (RLE) to compress each row of the first-move array, allowing fast retrieval from source s to target t.

Jump Point Search (JPS) (Harabor and Grastien 2011) is an online symmetry-breaking method that prefers the path with the earliest diagonal move among equivalent paths, pruning others efficiently. In grid path finding, it speeds up A* by over an order of magnitude. JPS+ (Harabor and Grastien 2014) an improved version with offline preprocessing and another GPPC 2014 winner, pre-identifies all jump points to accelerate search.

Two-Oracle Path Planning (Topping) (Salvetti et al. 2018) combines SRC and JPS+ through SRC and JPS oracles. It is faster than SRC and among the quickest methods for grid-based shortest-path computation, but incurs large storage and first-move decision costs due to limited compression in the SRC oracle. Notice that each call to the SRC oracle is much more costly than a call to the JPS oracle, making it worthwhile to reduce their frequency.

Hu et al. optimizes the JPS oracle (Hu et al. 2021, 2019) to reduce storage cost. They increase the number of calls to the JPS oracle at an additional cost and only store the first-move data for nodes related to the jump point. Their methods, TOPS and Topping+ (Topping+ is related to Topping) greatly reduce storage via selective data retention. However, this approach sacrifices the completeness of information, increasing search overhead that scales with map size.

Our work reduces first-move decision costs while retaining full CPDs by optimizing the SRC oracle and enhancing heuristic usage. We mainly work with several methods: heuristic redundant symbols (Chiari et al. 2019), proximity wildcards (Chiari et al. 2019), and rectangular proximity wildcards (RPW) (Chen, Zhang, and Zhang 2024). Additionally, inspired by end point search (EPS) (Shen et al. 2020), we choose to use the Euclidean distance to determine the heuristic move.

Background

Gridmap. A gridmap (Figure 1 (a)) is a two-dimensional environment consisting of $n \times m$ cells or nodes, where each node is either completely traversable or completely obstacle (in black). In this work, we follow the common rules. As shown in Figure 1 (b), W is the only feasible direction.

Compressed Path Databases (CPDs). A CPD is a data structure that stores the compressed first-move array T. CPDs are a group of techniques that use all-pair path data to find the optimal first move in T without state-space search.

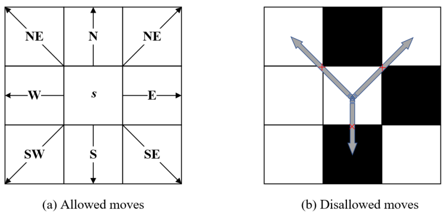

Figure 1: Examples of allowed moves on the 8-connected grid map.

Run Length Encoding (RLE). RLE is a matrix compression technique that computes an id for each node based on the input order in the graph and compresses a string of symbols by more compact substrings, called runs. To work with the JPS oracle, the SRC oracle of Topping adopts C-RLE (Strasser, Botea, and Harabor 2015), which breaks ties by prefer *diagonal-first* moves.

Topping. The Topping is shown in Algorithm 1, where the SRC oracle decides the first move $\vec{v}$ in line 3, and the JPS oracle calculates the steps c in line 4.

Heuristic Redundant Symbol. The heuristic redundant symbol h marks the nodes (shown in bold on the gridmap) in the first-move array T whose first move is equal to the heuristic move. In this work, we use the Euclidean distance and heuristic function $F_e(s,t)$ to determine the heuristic move, where n is a neighbor of s, $w(s,n)$ is the real cost from s to n, and $f_e(s,t)$ is the heuristic distance function used to estimate the distance from n to t. E is a set of feasible edges, where $(s,n) \in E$ means that the path from s to t is feasible.

Algorithm 1: Topping(s,t)

Input: init start s, target state t
Output: optimal path from s to t

```
1: $\pi \leftarrow \emptyset$
2: while $s \neq t$ do
3:     $\vec{v} \leftarrow getMoveSRC(s,t)$
4:     $c \leftarrow getNrSteps(s,t,\vec{v})$
5:     Append $c$ copies of $\vec{v}$ to path $\pi$
6:     $s \leftarrow makeMoves(s,\vec{v},c)$
7: end while
8: return $\pi$
```

$$f_e(s,t) = \sqrt{(s.x - t.x)^2 + (s.y - t.y)^2} \quad (1)$$

$$F_e(s,t) = \underset{(s,n)\in E}{\arg\min}\{\omega(s,n) + f_e(n,t)\} \quad (2)$$

Proximity Wildcards. Proximity wildcards define the largest square proximity area centered on each source node, including obstacles, blanks, and traversable nodes marked with h. The traversable nodes in the proximity area of s can be reached directly by the heuristic move.

Rectangular Proximity Wildcards. Rectangular Proximity Wildcards defines the largest rectangle proximity area for each source grid. Furthermore, the traversable nodes within the rectangular proximity area must be marked with a heuristic redundant symbol.

Topping consolidated by heuristics

Topping (Algorithm 1) uses the SRC oracle to determine the first move $\vec{v}$ from the current node s to the target t via binary search on compressed first-move arrays (CPDs), while the JPS oracle retrieves the number of steps c that $\vec{v}$ can execute without invoking SRC. Optimal paths are extracted by repeatedly calling these two oracles. Our contributions are based on the following observations:

1. Limited compression. In Topping, the SRC oracle uses C-RLE with a diagonal-first tie-break, which limits compression of first-move arrays and generates larger CPDs with inefficient compression.
2. Inefficient first-move decision. Larger CPDs make the SRC oracle perform more binary searches (i.e. first-move decision cost) per first-move decision, increasing search costs.

To address these issues, we enhance Topping with three CPD compression methods—heuristic redundant symbols, proximity wildcards, and rectangular proximity wildcards—producing more concise CPDs and improving search performance. We also extend the proximity area to oracle interactions, calling it the interest area, where applied heuristic information groups further boost search efficiency (Definition 1).

Definition 1. A *heuristic information group* H_{ig} is a set of heuristic move data calculated by the heuristic function and defined by the same *interest area* S_{ia}. $H_{ig}(s)$ is the heuristic

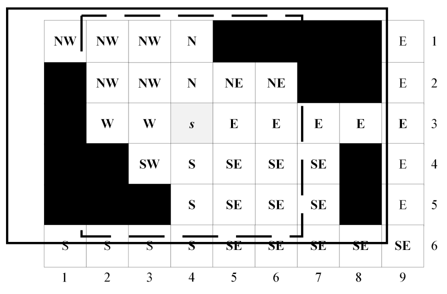

Figure 2: An example of the first move. The bold nodes are marked with a heuristic redundant symbol, the dashed square is the interest area for ToppingPW, and the solid rectangle is ToppingRPW's interest area.

information group defined by the $S_{ia}(s)$ centered on source node s. Each heuristic move in $H_{ig}(s)$ uniquely corresponds to a traversable node in $S_{ia}(s)$.

In other words, for any traversable node n in the interest area $S_{ia}(s)$, the first move from s to n will be directly returned by the heuristic move corresponding to n in $H_{ig}(s)$. We also propose the following three methods:

1. ToppingH (Two-Oracle Path Planning with Heuristic Redundant Symbol). Extends Topping with the heuristic redundant symbol h to replace qualified nodes (where the first move equals the heuristic move) in the first-move array.
2. ToppingPW (Two-Oracle Path Planning with Proximity Wildcards). Builds on ToppingH by using proximity wildcards and the largest square interest area to define heuristic information groups.
3. ToppingRPW (Two-Oracle Path Planning with Rectangular Proximity Wildcards). Uses rectangular proximity wildcards and the largest rectangle interest area, defining larger heuristic information groups to further boost search performance, especially in complex terrains.

Note that the side lengths of the interest areas in ToppingPW and ToppingRPW are stored together with the compressed first-move arrays as auxiliary data in CPDs. That is, for both ToppingPW and ToppingRPW, the size of CPDs is the sum of the size of first-move array compression results and auxiliary data.

ToppingH

ToppingH uses the heuristic redundant symbol to mark nodes where the first move matches the heuristic move (calculated by Euclidean distance). In the offline preprocessing stage, consecutive first moves marked with h can be compressed into a single h.

Figure 2 provides an example: Topping compresses it to $1NW$; $4N$; $9E$; $11NW$; $13N$; $14NE$; $18E$; $20W$; $23E$; $30SW$; $31S$; $32SE$; $36E$; $40S$; $41SE$; $45E$; $46S$; $50SE$, a total of 18 RLE runs. The cells in bold are marked with h by ToppingH. After compression with ToppingH, the first-move array becomes $1h$; $9E$; $11h$; $18E$; $20h$; $36E$; $40h$; $45E$; $46S$; $49h$, a total of 10 RLE runs. The compression result of ToppingH is more concise than the original result, which indicates greater memory saving and enhanced compression efficiency (a reduction of 8 RLE runs).

ToppingPW

ToppingPW defines the largest square centered on the current source node s as the proximity are and interest area. Essentially, ToppingH can be seen as ToppingPW with a side length of 1, while ToppingPW enlarges the interest area by several to hundreds of times. During offline preprocessing, ToppingPW uses proximity wildcards within this area to enhance compression.

The interest area also accelerates online search by reducing calls to the SRC oracle. As shown in Figure 2, the dotted square represents the proximity area (is the same as interest area during online search), and all traversable nodes inside are marked with h. The resulting compression is $1h$; $9E$; $18E$; $25h$; $36E$; $43h$; $45E$; $46S$; $49h$, a total of 9 RLE runs.

Algorithm 2 presents the pseudo-code of ToppingPW. During online search, before each SRC oracle call, ToppingPW checks whether the target node lies within the interest area. If so, the heuristic move is returned directly, eliminating the SRC call and speeding up the search. In other words, the interest area along with the heuristic information group can sometimes alternative the call to SRC oracle, which can accelerate the search.

Algorithm 2: ToppingPW(s, t)

```
[h] **Input**: start node $s$, target node $t$
**Output**: optimal path $\pi$
 1: $\pi \leftarrow \emptyset$
 2: $d \leftarrow pd(s)$
 3: **while** $s \neq t$ **do**
 4:     **if** $t$ in $GetInterestArea(s)$ **then**
 5:         $\vec{v} \leftarrow F_e(s,t)$
 6:     **else**
 7:         $\vec{v} \leftarrow getMoveSRC(s,t)$
 8:         **if** $\vec{v} = h$ **then**
 9:             $\vec{v} \leftarrow F_e(s,t)$
10:         **end if**
11:     **end if**
12:     $c \leftarrow getNrSteps(s,t,\vec{v})$
13:     Append $\vec{v}$ for $c$ steps to $\pi$
14:     $s \leftarrow makeMoves(s,\vec{v},c)$
15: **end while**
16: **return** $\pi$
```

ToppingRPW

ToppingRPW uses a rectangular interest area centered on the current node s to leverage more heuristic information for improved compression and search. In Figure 2, the solid rectangle shows the rectangular proximity area. The compression result is $9E$; $18E$; $27h$; $36E$; $45E$; $46S$; $49h$, totaling 7 RLE runs, the most concise among Topping, ToppingH, and ToppingPW.

The pseudo-code is similar to Algorithm 2, replacing line 4 with $GetRecInterestArea()$. Larger interest areas

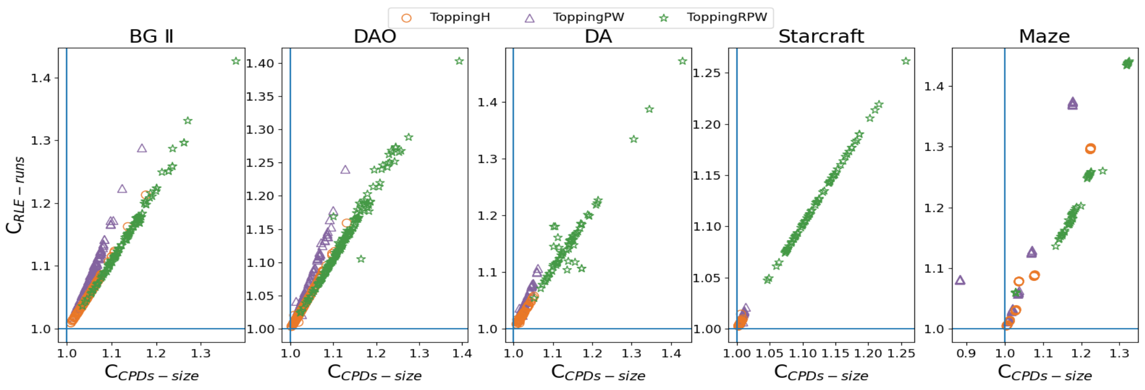

Figure 3: Distribution of compression factors. Line x=1 is the baseline of CPDs size, and the line y=1 is the baseline of RLE runs, i.e. the values are the same as Topping. Game benchmark: *BGII*, *DAO*, *DA*, *Starcraft*. Artificial benchmark: *Maze*. The definition of C_{metric_i} is shown in Equation 3

and richer heuristic information groups help minimize first-move decision costs and SRC oracle calls.

Experiments

We conduct experiments on 478 maps from five benchmarks provided by (Sturtevant 2012a), 4 game benchmarks: *Baldurs Gate II* (*BGII*), *Dragon Age: Origins* (*DAO*), *Dragon Age II* (*DA*), *Starcraft*, and an artificial benchmark *Maze*, with the number of nodes ranging from 100 to more than 7.5×10^5. 70% of the maps in *BGII* have nodes fewer than 1000, 74% of the maps in *DAO* and 94% of the maps in *DA* have nodes between 1,000 and 5×10^4, all maps in *Maze* have nodes between 1×10^5 and 3×10^5, and nearly 40% of the maps in *Starcraft* have more than 3×10^5 nodes. We use Topping as the experimental baseline, and compare it with SRC in compression and search.

Our main experimental metrics are: CPD size, RLE runs, binary search (the first-move decision cost) and runtime.

Since the map sizes can vary by tens to thousands of times in the same benchmark, resulting in huge difference in metric values. Thus, we normalize the metrics with a factor C_{metric_i} and define it as Equation 3.

$$C_{metric_i} = \frac{value_{map_j}(Topping_{metric_i})}{value_{map_j}(X_{metric_i})} \tag{3}$$

Where $metric_i$ represents the i-th metric and X can be assigned to ToppingH, ToppingPW, and ToppingRPW. For all metrics, the larger C_{metric_i} is, the better the methods are.

All algorithms are implemented in C++, and the experiments are conducted on Ubuntu 20.04.3 LTS, with processor AMD® Ryzen 9 5900*12core processer*24 and 31.4GiB RAM. All benchmarks can be found at https://movingai.com/benchmarks/grids.html.

Offline Preprocessing

As shown in Figure 3, compared with Topping, our methods have higher $C_{RLE-runs}$ on 100% of the maps, and RPW performs the best among them, reducing RLE runs by 12% on average and 32% at most (map *lt_gamlenshouse_n* with 1.9×10^4 nodes or so). Fewer RLE runs lead to more efficient compression and smaller first-move array sizes. A more concise first-move array compression result will effectively promote savings of search overhead.

From Figure 3, we can also find that ToppingH and ToppingRPW have $C_{CPDs-size} > 1$ on all maps, and ToppingPW has $C_{CPDs-size} > 1$ on all game maps and 90% of *Maze*. $C_{CPDs-size} > 1$ indicates that our methods typically have smaller storage costs. Among them, ToppingRPW performs the best, reducing the size of CPDs by up to 31% (map *lt_gamlenshouse_n*). We notice that ToppingPW produces larger sizes (≈13%↑) on 10% maps of *Maze*. The reason is that both ToppingPW and ToppingRPW need to store additional auxiliary data (length of the interest area, no more than 5% of the size of CPDs). Although ToppingPW has better compression capability and efficiency than Topping, its CPDs size would be larger on some *Maze* maps due to the auxiliary data. ToppingRPW, with larger interest areas and richer heuristic information groups, has much better compression performance than ToppingPW.

Online Search

Our methods achieve optimal path extraction with better search efficiency, which is mainly attributed to the significant reduction of the first-move decision cost. Figure 4 shows distribution of search factors for all maps. The $C_{binary-search}$ of our methods are always obviously greater than 1. In other words, they effectively reduce the first-move decision costs, particularly on larger maps such as those in *Starcraft* ($\min(C_{binary-search}) > 2.1$), and *Maze* ($\min(C_{binary-search}) > 2.2$). In addition, we list the first-move decision costs for several maps in Table 1. The three methods we propose can reduce the first-move decision cost by 59%-60% on average. Among them, ToppingPW performs better on small maps (with fewer than 1000 nodes), while ToppingRPW performs the best on large maps, achieving a reduction of the first-move decision cost by up to 71%. This is because our methods utilize richer heuristic information, leading to more concise first-move arrays, thereby effectively reducing the number of binary searches required

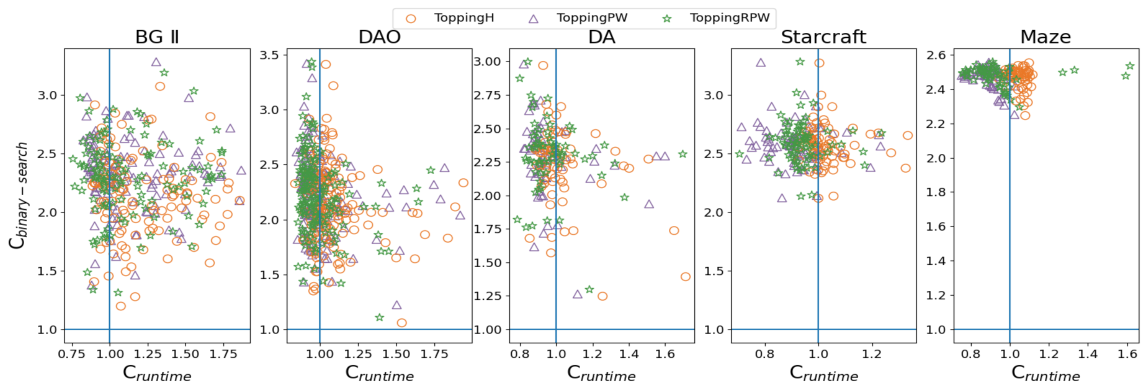

Figure 4: Distribution of search factors. Line x=1 is the baseline of runtime, and the line y=1 is the baseline of binary search. The definition of C_{metric_i} is shown in Equation 3.

	Map	Node	Metric	Topping	ToppingH	ToppingPW	ToppingRPW
	AR0203SR	2037	Binary searches	1683	548 (67.44% ↓)	**513** (**69.52%** ↓)	528 (68.63% ↓)
BGII			Runtime (s)	6.00 10^{-4}	4.50 10^{-4}(25.00% ↓)	4.59 10^{-4}(23.50% ↓)	**4.41** 10^{-4}(**26.50%** ↓)
120 maps	AR0330SR	811	Binary searches	634	302(52.37% ↓)	**269**(**57.57%** ↓)	278(56.15% ↓)
			Runtime (s)	3.46 10^{-4}	1.86 10^{-4}(46.24% ↓)	**1.85** 10^{-4}(**46.53%** ↓)	2.06 10^{-4}(40.46% ↓)
	orz800d	59689	Binary searches	270180	79206 (70.68% ↓)	78992 (70.76% ↓)	**78650** (**70.89%** ↓)
DAO			Runtime (s)	8.78 10^{-2}	**8.46** 10^{-2}(**3.71%** ↓)	9.62 10^{-2}(9.52% ↑)	9.29 10^{-2}(5.83% ↑)
156 maps	den207d	874	Binary searches	816	446 (45.34% ↓)	**400** (**50.98%** ↓)	427 (47.67% ↓)
			Runtime (s)	4.78 10^{-4}	2.60 10^{-4}(45.61% ↓)	**2.49** 10^{-4}(**47.91%** ↓)	5.01 10^{-4}(4.81% ↑)
	w_blightlands	14945	Binary searches	218448	73567 (66.32% ↓)	73300 (66.45% ↓)	**72887** (**66.63%** ↓)
DA			Runtime (s)	**4.15** 10^{-2}	4.48 10^{-2}(7.90% ↑)	5.06 10^{-2}(21.98% ↑)	4.93 10^{-2}(18.74% ↑)
67 maps	lt_house	864	Binary searches	39	28 (28.21% ↓)	**17** (**56.41%** ↓)	**17** (**56.41%** ↓)
			Runtime (s)	2.40 10^{-5}	**1.40** 10^{-5}(**41.67%** ↓)	1.50 10^{-5}(37.50% ↓)	2.00 10^{-5}(16.67% ↓)
	CatwalkAlley	225934	Binary searches	461337	141110 (69.41% ↓)	140858 (69.47% ↓)	**140651** (**69.51%** ↓)
Starcraft			Runtime (s)	1.13 10^{-1}	**1.13** 10^{-1}(**0.36%** ↓)	1.44 10^{-1}(27.47% ↑)	1.21 10^{-1}(7.34% ↑)
75 maps	Backwoods	208734	Binary searches	248396	96722 (61.06% ↓)	96969 (60.96% ↓)	**96406** (**61.19%** ↓)
			Runtime (s)	9.44 10^{-2}	8.31 10^{-2}(11.93% ↓)	**7.66** 10^{-2}(**18.86%** ↓)	8.70 10^{-2}(7.89% ↓)
	maze512-8-7	232926	Binary searches	2850273	1116540 (60.83% ↓)	1115170 (60.87% ↓)	**1113940** (**60.92%** ↓)
Maze			Runtime (s)	9.02 10^{-1}	**8.21** 10^{-1}(**8.97%** ↓)	1.01 10^{-1}(12.00% ↑)	9.88 10^{-1}(9.57% ↑)
60 maps	maze512-16-2	246136	Binary searches	650348	258537 (60.25% ↓)	258067 (60.32% ↓)	**256632** (**60.54%** ↓)
			Runtime (s)	2.73 10^{-1}	2.74 10^{-1}(0.49% ↑)	2.91 10^{-1}(6.57% ↑)	**1.69** 10^{-1}(**38.09%** ↓)

Table 1: The search metrics of several maps.

by the SRC oracle to determine the first move, along with the improvement of search efficiency. ToppingRPW benefits from the largest interest areas and the richest heuristic information groups, giving it a distinct advantage on large maps and complex terrains.

The horizontal axis of Figure 4 shows runtime. Combining it with Table 1, it can be found that ToppingPW searches faster on small maps such as *AR0330SR* and *den207d*. Among all the methods, ToppingPW exhibits the best runtime performance on the *den207d* and reduces the runtime by approximately 48% on this map. Although ToppingRPW does not perform well in terms of runtime on small maps, as the map size increases, it becomes more prominent. For example, on the *maze-512-16-2*, ToppingRPW reduces the runtime by about 38%. The interest area along with heuristic information group can sometimes replace the call to the SRC oracle, which helps to speed up the search. Therefore, ToppingPW and ToppingRPW can gain runtime benefits by effectively cutting the SRC oracle calls with the help of rich heuristic information. Nevertheless, they may take more time in the cases where Topping calls the SRC oracle less often.

Discussion

Topping+ (Hu et al. 2021), another two-oracle method, reduces memory usage by storing only a subset of CPDs, which also shortens preprocessing time. However, it leads to information loss and causes increased search overhead, particularly on larger maps (17.5% ↑ in *Maze* and 22% ↑ in *Starcraft* for average path query times). Even with wildcards (C-RLE-JPW) to improve CPD compression, additional runtime costs are incurred, resulting in query times 46% and 73% higher than Topping in *Maze* and *Starcraft*, respectively. In contrast, our method retains the complete CPD, achieving significantly higher search efficiency than both Topping and Topping+, using less memory, and requiring slightly more preprocessing time than Topping. Overall, although reducing offline preprocessing is appealing, improvements in search efficiency are far more important for practical applications.

Conclusion

Our work extends the heuristic redundant symbol, proximity wildcards and rectangular proximity wildcards to Topping. Additionally, we adopt interest area along with rich heuristic information groups to replace the SRC oracle calls in the online search, and propose heuristic Topping methods: ToppingH, ToppingPW, and ToppingRPW. Experimental results on five benchmarks demonstrate that our methods effectively cut down the first-move decision cost and improve search efficiency, and have gains in compression. Furthermore, methods with richer heuristic information perform better in large maps and complex terrain.

Among the recent work, "bounded suboptimal reverse CPDs" (Zhao et al. 2020) provides a new type of CPD. We plan to extend the reverse compression to Topping and our methods to further accelerate the extraction of the optimal path. In addition, "LLMs can help planning" (Kambhampati et al. 2024) is an interesting direction to explore and verify.

References

Antsfeld, L.; Harabor, D.; Kilby, P.; and Walsh, T. 2012. Transit routing on video game maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 8, 2–7.

Botea, A. 2011. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 7, 122–127.

Botea, A.; and Harabor, D. 2013. Path planning with compressed all-pairs shortest paths data. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23, 293–297.

Chen, X.; Zhang, Y.; and Zhang, Y. 2024. More Flexible Proximity Wildcards Path Planning with Compressed Path Databases. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, 77–85.

Chiari, M.; Zhao, S.; Botea, A.; Gerevini, A. E.; Harabor, D.; Saetti, A.; Salvetti, M.; and Stuckey, P. J. 2019. Cutting the size of compressed path databases with wildcards and redundant symbols. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 106–113.

Cui, X.; and Shi, H. 2011. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1): 125–130.

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2017. Customizable route planning in road networks. *Transportation Science*, 51(2): 566–591.

Freund, E.; and Hoyer, H. 1986. Pathfinding in multi-robot systems: Solution and applications. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, 103–111.

Harabor, D.; and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI conference on artificial intelligence*, volume 25, 1114–1119.

Harabor, D.; and Grastien, A. 2014. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 128–135.

Hu, Y.; Harabor, D.; Qin, L.; and Yin, Q. 2021. Regarding goal bounding and jump point search. *Journal of Artificial Intelligence Research*, 70: 631–681.

Hu, Y.; Harabor, D.; Qin, L.; Yin, Q.; and Hu, C. 2019. Improving the combination of JPS and geometric containers. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 209–213.

Hönig, W.; Preiss, J. A.; Kumar, T. K. S.; Sukhatme, G. S.; and Ayanian, N. 2018. Trajectory Planning for Quadrotor Swarms. *IEEE Transactions on Robotics*, 34(4): 856–869.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L. P.; and B Murthy, A. 2024. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, 22895–22907. PMLR.

Salvetti, M.; Botea, A.; Gerevini, A.; Harabor, D.; and Saetti, A. 2018. Two-oracle optimal path planning on grid maps. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, 227–231.

Salvetti, M.; Botea, A.; Saetti, A.; and Gerevini, A. E. 2017. Compressed path databases with ordered wildcard substitutions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 27, 250–258.

Shen, B. 2023. *Advances in Pathfinding Algorithms for Games, Route Planning Software, and Automated Warehouses*. Ph.D. thesis, Monash University.

Shen, B.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2020. Euclidean pathfinding with compressed path databases. In *International Joint Conference on Artificial Intelligence-Pacific Rim International Conference on Artificial Intelligence 2020*, 4229–4235.

Strasser, B.; Botea, A.; and Harabor, D. 2015. Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research*, 54: 593–629.

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast first-move queries through run-length encoding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 5, 157–165.

Sturtevant, N. 2012a. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2): 144 – 148.

Sturtevant, N.; Traish, J.; Tulip, J.; 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 International Symposium on Combinatorial Search*, volume 6, 241–250.

Sturtevant, N. R. 2012b. Moving Path Planning Forward. In Kallmann, M.; and Bekris, K., eds., *Motion in Games*, 1–6. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-34710-8.

Zhao, S. 2022. *Improving Pruning and Compression Techniques in Path Planning*. Ph.D. thesis, Monash University.

Zhao, S.; Chiari, M.; Botea, A.; Gerevini, A. E.; Harabor, D.; Saetti, A.; and Stuckey, P. J. 2020. Bounded suboptimal path planning with compressed path databases. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 333–341.