ISSN: 1803-3814 (Printed), 2571-3701 (Online)
https://doi.org/10.13164/mendel.2022.2.097

MENDEL Soft Computing Journal

Explanation and Speedup Comparison of Advanced Path-planning Algorithms Presented on Two-dimensional Grid

Petr Soustek✉, **Radomil Matousek**✉, **Jiri Dvorak**, **Lenka Manakova**
Institute of Automation and Computer Science, Brno University of Technology, Czech Republic
soustek@fme.vutbr.cz✉, rmatousek@vutbr.cz✉, dvorak@fme.vutbr.cz

Abstract
Path planning or network route planning problems are an important issue in AI, robotics, or computer games. Appropriate implementation and knowledge of advanced and classical path-planning algorithms can be important for both autonomous navigation systems and computer games. In this paper, we compare advanced path planning algorithms implemented on a two-dimensional grid. Advanced path planning algorithms, including pseudocode, are introduced. The experiments were performed in the Python environment, thus with a significant performance margin over C++ or Rust implementations. The main focus is on the speedup of the algorithms compared to a baseline method, which was chosen to be the well-known Dijkstra's algorithm. All experiments correspond to trajectories on a two-dimensional grid, with variously defined constraints. The motion from each node corresponds to a Moore neighborhood, i.e., it is possible in eight directions. In this paper, three well-known path planning algorithms are described and compared: the Dijkstra, A and A* /w Bounding Box. And two advanced methods are included, namely Jump Point Search (JPS), incorporated with the Bounding Box variant (JPS+BB), and Simple Subgoal (SS). These advanced methods clearly show their advantage in the context of the speed up of solution time.*

Keywords: *Path planning, Route planning, JPS algorithm, Subgoal algorithm, A^* algorithm, Dijkstra's algorithm.*

Received: 18 July 2022
Accepted: 16 November 2022
Online: 20 December 2022
Published: 20 December 2022

1 Introduction

Path planning methods are often encountered in robotics [10, 8, 12], video games [33], and many other artificial intelligence applications [22, 5]. Their vast utilization and popularity are widely recognized [1, 23]. The classical path planning methods include the *Dijkstra's algorithm* [9], the A^* algorithm [18] or the *Iterative deepening* A^* algorithm (IDA^*) [21].

Furthermore, A^* is still one of the most commonly used methods due to its simplicity and speed. However, new advanced methods have emerged over the last few years. These are based, for example, on the reduction of the nodes to be explored using symmetry [13, 14], hierarchical abstraction [4, 31], exact heuristics [2, 7, 32], or by preprocessing maps using, for example, subgoal techniques or other methods [34, 3, 6, 29, 19].

In this paper, we review two such advanced methods and make a comparison against the classical methods on the lattice graph – these are the *Jump Point Search* (JPS) and the *Simple Subgoal* (SS) algorithms. Let us state that the grid undirected graph represents a popular form of the environment on which the search for the shortest path between a start and a goal point is performed. All nodes of this graph have a so-called Moore's neighborhood, i.e., each node has eight neighboring nodes. In such an environment we often encounter a high degree of symmetry [13]. This is a property where in a lattice graph some paths, or parts of paths with the same start and end point, have the same length and differ only in some states, and in the extreme case they may differ in all states. Symmetry can thus cause the search algorithm to traverse unnecessarily many equivalent paths. Symmetry in this case is the property that in a lattice graph, some paths (or parts of paths) with the same start and end points (i.e. nodes) have the same length and differ only in part of the states. Of course, in the extreme case, all paths may be different.

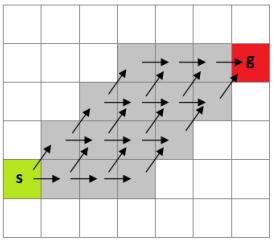


Figure 1: An example of shortest path symmetry.

The case of symmetry, i.e., equivalent shortest paths, is shown in Fig. 1. This symmetry may cause the search algorithm to traverse unnecessarily many equivalent paths. The JPS algorithm [16, 27] reduces this symmetry by pruning the graph according to certain rules until it finds some *interesting* states, which it then expands further.

2 Description of the Algorithms

In shortest path search problems, Dijkstra's algorithm [9] is undoubtedly the basic and essentially reference method. The second one is the modern algorithm A^* [18]. Both algorithms return an optimal path (Gelperin 1977), and can be considered as special forms of dynamic programming (Bellman 1957). Both methods are very well known and do not need to be described at length. Just briefly on the A^* algorithm, let us mention that it uses an evaluation function to search the space, based on which it selects the state to expand, which leads to a faster search. The evaluation function is given by (1)

$$f(i) = g(i) + h(i). \tag{1}$$

In this relation, the function $g(i)$ represents an estimate of the distance between the initial state and the current state i. The value of this function for each state j that is a successor of state i is obtained from the calculation

$$g(j) = g(i) + c(i, j), \tag{2}$$

where $c(i, j)$ is the distance between state i and j. The function $h(i)$ is a heuristic function that represents an estimate of the distance between the current state i and the final state j using a problem-specific heuristic. It is also possible to switch the direction of the search in A^*, so that planning is performed from the goal state towards the start state. In this case we can call it "backward" A^*.

Algorithm 1 A^* algorithm

```
1:  open ⟵ start
2:  closed ⟵ ∅
3:  g(start) = 0
4:  while open¬∅ do
5:      u ← Extract − Min(open)
6:      if u == t break then
7:      end if
8:      for all edge e = (u, v) do
9:          distance ← g[u] + weight[e]
10:         if v ∉ open then
11:             g[v] ← distance
12:             h[v] ← Distance − To − Target(v, t)
13:             f[y] ← g[v] + h[v]
14:             open ← v
15:         else
16:             if distance < g[v] then
17:                 g[v] ← distance
18:                 h[v] ← Distance − To − Target(v, t)
19:                 f[y] ← g[v] + h[v]
20:             end if
21:         end if
22:     end for
23:     closed ← u
24: end while
```

2.1 JPS Algorithm

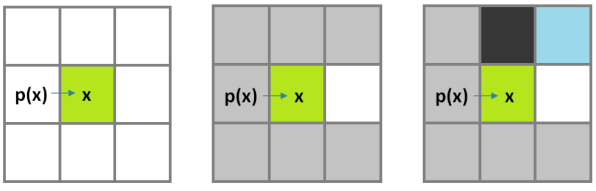
The Jump Point Search (JPS) algorithm [14, 16], is a relatively new method published by Daniel Harabor and Alban Grastier in 2011 in their paper *Online graph pruning for pathfinding on grid maps* [14]. They further extended the JPS method in their paper *The JPS Pathfinding System* [16].

There is also a variant named JPS+, in which the entire environment is preprocessed before the actual search [28]. Another variant is the Jump Point Graph (JPG), combining JPS and the Subgoal graph [17]. For completeness, let us mention the JPS-3D variant [26], which is a formulation of JPS in 3D space. The JPS algorithm reduces the mentioned symmetry by pruning the graph according to certain rules until it finds some interesting states, then expanding these states further.

JPS is based on the A^* algorithm. We still use the heuristic function and the OPEN and CLOSE lists. The JPS algorithm differs only in that it uses the node expansion operator. This operator is based on two types of rules. These are *pruning rules* and *jump point rules*. In the case of pruning rules, we understand pruning to mean ignoring certain nodes whose expansion would lead to nothing and it would be useless to add them to the OPEN list. Consider a node x, its predecessor p, and a successor y of node p that is different from x. Then as path $\pi = \langle n_0, n_1, ...n_k \rangle$ we understand a path starting at n_0 and ending at n_k. In general, to prune (ignore) a node n that is a neighbor of node x, one of the following conditions must be satisfied according to [16]:

1. There is a path $\pi' = \langle p, y, n \rangle$ or $\pi' = \langle p, n \rangle$, which is shorter than the path $\pi = \langle p, x, n \rangle$.
2. There is a path $\pi' = \langle p, y, n \rangle$ or $\pi' = \langle p, n \rangle$, which has the same length as $\pi = \langle p, x, n \rangle$, but π' has diagonal motion earlier than π.

Pruning occurs in two directions, namely in the straight and diagonal direction. For the sake of clarity, we will explain each direction separately. In the straight direction, we prune (ignore) a neighboring node n of a node x if we reach this node from a predecessor p by a shorter or the same path as if we reached this node from node p by a path passing through x. Fig. 2 shows an example. In the case of Fig. 2a the grid contains no obstacle, i.e. all nodes can be traversed.



(a) (b) (c)

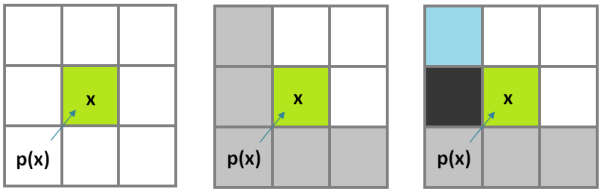
Figure 2: Pruning rules in the straight direction. Where (a) is default state for direct pruning, (b) shows principle of direct pruning, and (c) shows direct pruning with an obstacle area.

Figure 2b shows the principle of direct pruning. Here, the predecessor of the node x is the node $p(x)$. The cost of a node-to-node transition in a straight cut is 1 and in a diagonal cut is $\sqrt{2}$. All nodes that are represented by grey boxes are pruned. Thus, all nodes are pruned except the node to the right of x. Such a neighbor node of node x is called *natural neighbor* of node x according to [14]. Now we change the situation a little by placing an obstacle over node x, see Fig. 2c. In this case, the node represented by the blue cell could not be cut. Such a node is called *forced* according to [14], see the following definition 1.

Definition 1 (*forced node*)**.** Let n be a neighbor node of node x. We call n a *forced node* if:

1. Node n is not a natural neighbor of node x.
2. The path $\pi = \langle p, x, n \rangle$ is shorter than the path $\pi' = \langle p, ..., n \rangle$, which does not contain x.

This definition is very important because it links the graph pruning rule with the jump point rule, which will be introduced later. For diagonal pruning, the principle is similar. We prune an adjacent node n of node x if we reach this node from a predecessor p by a strictly shorter path, as if we had reached this node from node p by a path leading through x. Fig. 3 shows the default state of this case.



(a) (b) (c)

Figure 3: Pruning rules in the diagonal direction. Where (a) is default state for direct pruning, (b) shows principle of diagonal pruning, and (c) shows diagonal pruning with an obstacle area.

Here the predecessor of node x is node $p(x)$ and the cost of moving from node to node in the straight direction is again 1 and in the diagonal direction $\sqrt{2}$. Figure 3b shows the pruning principle. All nodes represented by grey boxes are pruned. The remaining nodes (white cells) are among the natural neighbors of node x.

A different situation occurs again when there is an obstacle in the space. This case is illustrated in the following Fig. 3c. The black cell represents the obstacle, all grey nodes are pruned and the node represented by the blue cell is by definition (1) a forced neighbour.

Unlike A^*, JPS expands only selected nodes. Such nodes are called *jump points* (Jump points) and are selected according to the jump point determination rule. Jump points because if such a point (node) exists in our search space, we *jump* to it and expand it. The following definition (2) taken from [14] tells how we determine these nodes.

Definition 2 (*jump point*)**.** A node y is a jump point from a node x in the direction $\overrightarrow{d}$ if y minimizes k such that $y = x + k\overrightarrow{d}$ and one of the following conditions holds:

1. The node y is the target node.
2. Node y has at least one forced neighbor.
3. Let $\overrightarrow{d}$ is vector of a diagonal motion and there is a node $z = y + k_i\overrightarrow{d_i}$ that lies k_i ($k_i \in \mathbf{N}$) steps in the direction of $\overrightarrow{d_i} \in \left\{\overrightarrow{d_1}, \overrightarrow{d_2}\right\}$ such that z is the jump point from y according to condition 1 or condition 2 of this definition, and where the directions $\overrightarrow{d_1}, \overrightarrow{d_2}$, are the straight directions that make an angle of 45° with the direction $d.$, are the straight directions that make an angle of 45° with the direction d.

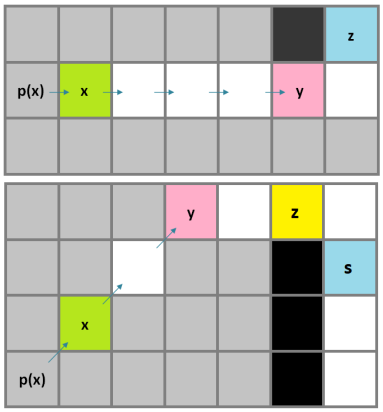


Figure 4: Origin of the jump point. Origin of the jump point (pink cell) in straight motion (above), origin of the jump point (pink cell) in diagonal motion (bottom).

Let us now look at definition 2. It says that a jump point is a node that has at least one forced neighbor. In the case of direct motion, the origin of the jump point is shown in Fig. 4.

In this example, the node x whose ancestor is the node $p(x)$ is expanded. The grey nodes above and below the x node are the nodes we cut (ignore). We can see that there is no interesting node, so we move one node further. We look again above and below the node, again nothing interesting, so we move on until we hit the y node. Above it there is an obstacle that causes a forced neighbor, z, and so node y is our jump point. Node y is added to the OPEN list, is labeled as a successor of node x, and is assigned the value $g(y) = g(x) + c(x, y)$, as in the A^* algorithm. In this way, we have jumped from node x to node y, without expanding any other node on the path from x to y. For diagonal motion, the principle is the same.

Now let's look at condition 3 from the previous definition 2. Such a case is described in Fig. 4. We start at node x and move progressively to node y. Node z is two steps (shifts) away from node y and has a forced neighbor (cell s in blue). Which means that node z

is a jump point with predecessor y and y is a jump point with predecessor x. Both nodes are added to the OPEN list. All grey nodes are pruned (ignored).

The actual flow of the JPS algorithm works similarly to the A^* algorithm, except that it uses the *Identify Successors* function to find the set of successors with the recursive *jump* function, see [14]. For the *Identify Successors* process, we work with the set of neighbors of the current node x (*neighbours*) that have not been pruned using pruning rules. Using the *for* loop (lines 3 to 5 in the pseudocode, see below), we try to add to the OPEN list a node (jump point) that is as far away from node x as possible and also in the same direction as the direction of transition from node x to any neighbouring node n (e.g. If node n is to the right of x, we look for a jump point to the right of x, etc.) If we find such a node, we add it to the OPEN list (the set *successors*), otherwise we add nothing to the OPEN list. The process continues until the set of neighbours (*neighbours*) is empty.

Algorithm 2 Identify Successors [14]

1: $successors(x) \leftarrow \emptyset$
2: $neighbours(x) \leftarrow prune(x, neighbours(x))$
3:
4: **for all** $n \in neighbours(x)$ **do**
5: $\quad n \leftarrow jump(x, direction(x, n), s, t)$
6: $\quad$ add n to $successors(x)$
7: **end for**
8: **return** $successors(x)$

Regarding how to classify such nodes (whether they are our jump points), the *jump* function is used. This requires, in addition to the start, end, and current node, the direction of $\overrightarrow{d}$. In fact, the function traverses node by node in this direction from x and determines if it is a jump point in correspondence with the definition of 2.

Algorithm 3 Function *jump* [14]

1: $n \leftarrow step(x, \overrightarrow{d})$
2: **if** n is an obstacle or is outside the grid **then**
3: $\quad$ **return** *null*
4: **end if**
5: **if** $n = t$ **then**
6: $\quad$ **return** n
7: **end if**
8: **if** $\exists n' \in neighbours(n)$ s.t. n' is forced **then**
9: $\quad$ **return** n
10: **end if**
11: **if** $\overrightarrow{d}$ is diagonal **then**
12: $\quad$ **for all** $i \in \{1, 2\}$ **do**
13: $\quad\quad$ **if** $jump(n, \overrightarrow{d}_i, s, t)$ is not *null* **then**
14: $\quad\quad\quad$ **return** n
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: **end if**
18: **return** $jump(n, \overrightarrow{d}, s, t)$

The following Fig. 5 shows a comparison of the algorithms with respect to the contents of the OPEN list, showing the difference between the JPS algorithm and the A^* algorithm.

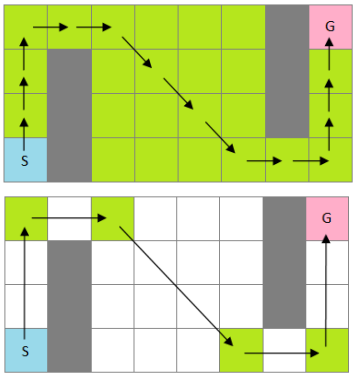


Figure 5: Comparison of A^* (above) and JPS (bottom) algorithms according to cells added to the OPEN list

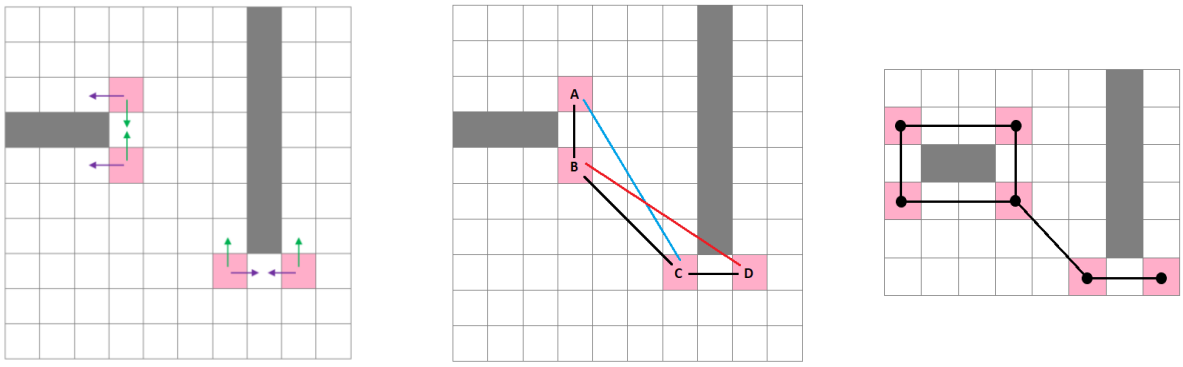
2.2 Algoritmus Subgoal

Tansel Uras, Sven Koenig and Carlos Hernández presented the Subgoal algorithm in their 2013 paper *Subgoal graphs for optimal pathfinding in eight-neighbor grids* [34]. This algorithm is based on preprocessing the search environment. This environment is represented by a grid with possible motion in eight directions, similar to the visibility graph [24] but with an exception for diagonal motion. Diagonal motion is only possible if adjacent cells in the straight direction are not blocked. The authors introduced two algorithms, first *Simple Subgoal Algorithm*, which we discuss in this semi-review paper, and then *Bipartite Subgoal Graph*. To describe the Subgoal algorithm, we first need to define the following terms. Illustrative images are based on [33].

Definition 3 (*subgoal*)**.** By *subgoal* in a two-dimensional square lattice, we mean an unblocked cell s if there are two orthogonal straight directions c_1 and c_2 such that motion in the direction $s + c_1 + c_2$ is blocked and motion in the directions $s + c_1$ and $s + c_2$ is not blocked.

For the following definition 4, it is useful to explain the difference between a *trajectory* and a *path*. Trajectories between two cells s and s' is a sequence of movements in a grid that gets, for example, a robot from state s to state s' when all obstacles are removed. A path is a trajectory that gets the robot from state s to state s' in the current grid (i.e., in the presence of obstacles).

Definition 4 (*reachability*)**.** Two cells s and s' are *reachable* if a path of length $h(s, s')$ exists between them. Two reachable cells s and s' are *safely reachable* if all shortest trajectories between them are also paths. Two safely reachable cells s and s' are *directly reachable* if none of the shortest paths between them contains the subgoal $s'' \notin \{s, s'\}$.



(a) Location of subgoals in the grid (b) Subgoals and their interconnections (c) Simple subgoal graph

Figure 6: Representation of the basic stages of the Subgoal algorithm.

Figure 6b shows the subgoals and their interconnections. A, B, C and D are subgoals. $A-B$, $B-C$ and $C-D$ are *directly reachable*. $A-C$ are *safely reachable*, but they are not *directly reachable* because the shortest path between them is through the subgoal B. $B-D$ are *reachable*, but are not *safely reachable* because the shortest path between them is blocked.

Definition 5 (*simple subgoal graph*)**.** *Simple subgoal graph* $G_S = (V_S, E_S)$ is undirected see figure 6c, where V_S is the set of subgoals and E_S is the set of edges connecting directly reachable subgoals to each other. The edge lengths are the distances of the subgoals in terms of a given metric defined on a two-dimensional square lattice (Moore's neighborhood).

Finding directly reachable subgoals is an important phase of the algorithm both for constructing a simple subgoal graph and for connecting the start and goal positions to this graph. The algorithm 4, finds all directly reachable states from the selected state s and returns all directly reachable subgoals. How the algorithm works is shown using the example in Fig. 7.

The algorithm works with *clearance* values. This value of the cell s in the direction d is obtained using the function $Clearance(s, d)$. This is the maximum number of movements the robot can make from the s state through the d direction without hitting a subgoal or being stopped by an obstacle. For example, the default *clearance* value for state s from the example in Fig. 7, is 6, because cell $M6$ is blocked by an obstacle. The northern *clearance* value is 2 because $F3$ is a subgoal. These *clearance* values can be computed on the fly or can be computed in advance.

The algorithm for identifying directly reachable subgoals works in two phases:

A - *The first phase.* The first phase identifies all directly reachable states from state s that can only be reached in one direction. This is achieved by tracking the *clearance* values of state s in all possible eight directions of movement from this state to see if there are any directly reachable subgoals in these directions. For example, the northern *clearance* value of state s is 2 and the algorithm checks not only for movement two states up, but also for $2+1$ to see if state $F3$ is a subgoal. As can be seen in Fig. 7, the result of the first phase of the algorithm is that the states $C3$ and $F3$ are directly reachable states from the state s.

B - *The second Phase.* The second phase identifies all directly reachable states from state s that can be reached by a combination of movements in the cardinal direction c and the diagonal direction d. There are 8 possible combinations of cardinal and diagonal motions (shown by the solid arrow in Fig. 7). Each of these combinations identifies the corresponding region. The algorithm discovers each region row (column) by row (column), these are the dashed arrows in the Fig. 7. For each state that is directly reachable from a state s moving in the d direction, the exploration is performed in the cardinal c direction. It starts with the row (column) closest to state s and continues until all rows (columns) are explored.

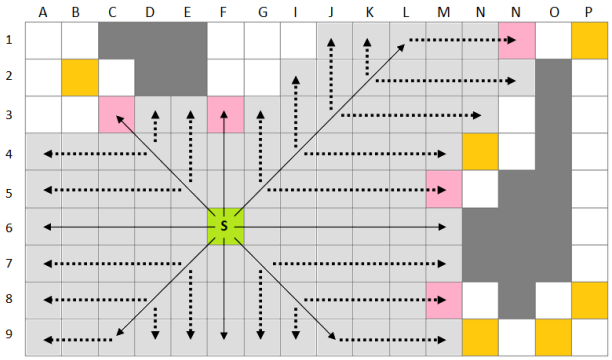


Figure 7: The principle of searching for directly achievable sub-goals. Subgoals that are directly reachable from the s state are marked in pink, subgoals that are not directly reachable from the s state are marked in orange. The solid arrows look for directly reachable states from state s that can be found by moving in only one direction, and the dashed arrows look for directly reachable subgoals from state s that can be found by combining two directions.

Algorithm 4 Search for directly reachable subgoals [34]

```
1: function CLEARANCE(cell s, direction d)
2:     i := 0;
3:     while true do
4:         if movement not possible from s + id to s + id + d then
5:             return i;
6:         end if
7:         i := i + 1;
8:         if IsSubGoal(s + di) then
9:             return i;
10:        end if
11:    end while
12: end function
13: function GETDIRECTHREACHABLE(cell s)
14:    S := ∅;
15:    for all directions d do
16:        if IsSubGoal(s + Clearance(s, d) × d) then
17:            S := S ∪ {s + Clearance(s, d) × d};
18:        end if
19:    end for
20:    for all diagonal directions d do
21:        for all cardinal directions c associated with d do
22:            max ⟵Clearance(s, c);
23:            diag ⟵Clearance(s, d);
24:            if IsSubGoal(s + max × c) then
25:                max := max − 1;
26:            end if
27:            if IsSubGoal(s + diag × d) then
28:                diag := diag − 1;
29:            end if
30:            for i = 1...diag do
31:                j :=Clearance(s + id, c);
32:                if j ≤ max and IsSubGoal(s + id + jc) then
33:                    S := S ∪ {s + id + jc};
34:                    j := j − 1;
35:                end if
36:                if j < max then
37:                    max := j;
38:                end if
39:            end for
40:        end for
41:    end for
42:    return S;
43: end function
```

Now three rules are applied. These rules tell how far to explore each of the rows (columns) [33].

R1: The first rule says that the exploration of a row (column) ends when the subgoal is reached or just before it hits an obstacle.

R2: The second rule says that the explored row (column) cannot be longer than the previous explored row (column).

R3: The third rule extends the second rule and says that the explored row (column) cannot be longer than the previous row (column) and must be one state smaller if the previous row (column) ended in a subgoal. For example, if we look at the northeast region from state s in the image 7, the first row that is explored is row 5, for which the algorithm encounters the subgoal $M5$ after 5 moves. Since the exploration of this row ends at the subgoal, in the following row 4, only $5 - 1 = 4$ moves can be made and the algorithm stops before the subgoal $N4$, which is not directly reachable from the state s.

Finally, let us now consider the algorithm 5, which creates a simple subgoal graph, where the function *GetDirectHReachable(s)* returns the set of subgoals that are directly reachable from cell s.

A Simple Subgoal (SS) algorithm is based on the creation of a subgoal graph, see def. 5. The algorithm

Algorithm 5 A simple subgoal graph construction [34]

```
1: procedure ConstructSubgoalGraph( )
2:     V_S := E_S := ∅;
3:     for all unblocked cells s do
4:         for all perpendicular cardinal directions c_1
   and c_2 do
5:             if s + c_1 + c_2 is blocked then
6:                 if s + c_1 and s + c_2 are unblocked
   then
7:                     V_s := V_s ∪ {s};
8:                 end if
9:             end if
10:        end for
11:    end for
12:    for all s ∈ V_S do
13:        S ← GetDirectHReachable(s);
14:        for all s' ∈ S do
15:            E_s := E_s ∪ {(s, s')}
16:        end for
17:    end for
18:    G_s := (V_S, E_S)
19: end procedure
```

places subgoals in the corners of obstacles and then creates edges between the directly reachable subgoals. To find the shortest path between the start and goal cells, the start and goal must first be connected to their respective, directly reachable subgoals.

Then the algorithm find the shortest path (called high-level path) between the start and the goal on this modified subgoal graph.

Then the algorithm follows to find the shortest path (called low-level path) between the start and the goal by finding the shortest paths on the grid between consecutive subgoals on the path and then connecting them.

For a better understanding, the Simple SubGoal algorithm for finding the shortest path can be described using the example in Fig. 8.

In the first step of the algorithm, it is checked whether the start position S and the goal position G are directly reachable, see Fig. 8a, where the pink cells A, B, C, D are subgoals, the green cell S is the starting position and the red cell G is the goal position. If they are, the algorithm returns a direct path between them.

In the second step, the algorithm connects the start and finish positions with their corresponding directly reachable subgoals, see Fig. 8b. This step produces a simple subgoal graph, which is used in the third step to find the so-called high-level shortest path from the start position to the goal position via the subgoal graph, see Fig. 8c.

In the last fourth step, the algorithm finds the low-level shortest path by following the subgoals from the high-level shortest path, see Fig. 8d.

Algorithm 6 Searching a simple subgoal graph [34]

```
1: procedure ConnectToGraph(cell s)
2:     if s ∉ V_S then
3:         V_s := V_s ∪ {s};
4:         S ← GetDirectHReachable(s);
5:         for all s' ∈ S do
6:             E_s := E_s ∪ {(s, s')};
7:         end for
8:     end if
9: end procedure
10: function FindAbstractPath(cells s, s')
11:     ConnectToGraph(s);
12:     ConnectToGraph(s');
13:     Π ←find a shortest path from s to s'
   over the modified graph;
14:     restore original graph;
15:     return Π;
16: end function
17: function FindPath(cells s, s')
18:     π ←TryDirectPath(s, s');
19:     if π ≠ nopath then
20:         returnπ;
21:     end if
22:     Π ← FindAbstractPath(s, s');
23:     if Π = nopath then
24:         return nopath;
25:     end if
26:     π := emptypath;
27:     for all segments(s_i, s_{i+1}) in Π,
   in increasing order of i do
28:         π := append(π, FindHReachablePath(s_i, s_{i+1}));
29:     end for
30:     return π;
31: end function
```

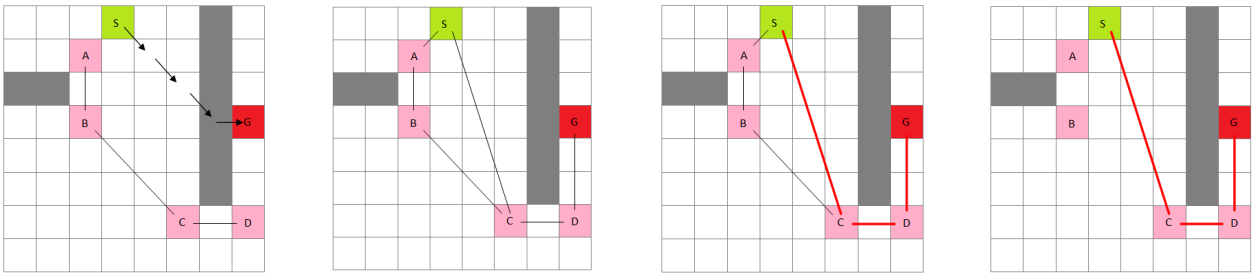
3 Experiments and Results

Within the experiments, tests were performed both on the test corpus itself (self-generated maps) and on maps from the MovingAI [30] corpus. Thus, the implemented algorithms and their results were compared.

Note that the MovingAI benchmark/corpus is one of the most used benchmarks for path planning on the grid. Other benchmarks include a brand new benchmark based on the Iron Harvest maps presented in [15].

The methods compared were as follows:

- *Dijkstra* algorithm
- A^* algorithm
- *JPS* algorithm (Jump Point Search)
- A^* algorithm with Bounding Box combination ($A^* + BB$)
- *JPS* algorithm with Bounding Box combination ($JPS + BB$)
- *SS* algorithm (Simple Subgoal)



(a) The first step (b) The second step (c) The third step (d) The fourth step

Figure 8: The steps of the Simple Subgoal algorithm.

For the experiments with these algorithms, a simulation environment was created using the *Python* [35, 25] programming language. The user interface was created using the *Pygame* library and the *NetworkX* [11, 20] library was used for graph construction. The node-to-node transition cost in the vertical and horizontal directions was set to 1 and the node-to-node transition cost in the diagonal direction was set to $\sqrt{2}$. The metric thus corresponded to the planar unit Euclidean metric (also referred to as the octile metric in this case).

The monitored values were the number of expanded nodes, shortest path length and time. The resulting values of the achieved times are the medians of the hundred searches performed. These resulting times are the basis for the presented speed up comparisons.

It is also important to extend the selected methods with the so-called bounding box technique, which is based on a simple idea. First, a bounding box is computed around the start and destination nodes. These nodes are then treated as obstacles. The algorithm then search for the shortest path within this box and if it does not find a path inside the box, it expands the box. The initial size of the bounding box, as well as the number of nodes it expands by, should be chosen appropriately with respect to the shape of the map. Since expanding the bounding box laterally by one or more nodes slows down the algorithm.

Experiment 1 (Tab. 1) – A map containing simple obstacles was chosen as the first map, as shown in Fig. 9. The results from finding the shortest path on this type of map are in favor of the Simple Subgoal and JPS algorithms. In the case of Simple Subgoal, the optimal path was not found, it is longer than the other methods due to the limitations of this algorithm. The median preprocessing time for this method was 45 ms.

Experiment 2 (Tab. 2) – Another class of maps in the test was the maze map with obstacle density 45 %, as shown in Fig. 10. The results of the second experiment show that in terms of time and in case of JPS also in number of expanded nodes, the SS and JPS algorithms performed the best. Preprocessing for the SS algorithm took 39 ms. This value ranked the SS algorithm first in speed again. Note that the optimal route was not guaranteed by the principle of the algorithm and in this case was not found.

Table 1: Experiment 1, simple map

Method	Path length	Expand nodes	Speed-Up
Dijkstra	36.38	506	1
A*	36.38	264	1.75
A*+BB	36.38	253	1.45
JPS	36.38	39	2.66
JPS+BB	36.38	18	2.23
SS	38.14	69	4.21
Dijkstra 16.39 ms			

Table 2: Experiment 2, maze, 45% obstacle

Method	Path length	Expand nodes	Speed-Up
Dijkstra	30.31	360	1
A*	30.31	68	2.01
A*+BB	30.31	67	1.68
JPS	30.31	20	2.20
JPS+BB	30.31	20	1.88
SS	34.41	103	2.94
Dijkstra 8.72 ms			

Experiment 3 (Tab. 3) – This set of experiments was performed on two randomly generated maps of 50x50 and 100x100 nodes. The results are again in the spirit of the previous experiments. The JPS and SS algorithms are the fastest, even in terms of the number of expanded nodes. In the case of the 50x50 map, JPS augmented with a bounding box performed the best. Preprocessing for Simple Subgoal took 280 ms for the 50x50 map and 330 ms for the 100x100 map. The Simple Subgoal algorithm was the fastest but by a very small margin, not optimal.

Table 3: Experiment 3, map size $n \times n$

Method, for $n = 50$	Path length	Expand nodes	Speed-Up
Dijkstra	41.14	2124	1
A*	41.14	221	2.67
A*+BB	41.14	221	2.43
JPS	41.14	83	3.07
JPS+BB	41.14	83	3.36
SS	42.31	740	3.06
Dijkstra 55.73 ms			
Method, for $n = 100$	**Path length**	**Expand nodes**	**Speed-Up**
Dijkstra	169.30	10646	1
A*	169.30	7633	1.14
A*+BB	169.30	7633	1.10
JPS	169.30	10	3.23
JPS+BB	169.30	10	2.21
SS	169.88	7	9.28
Dijkstra 234.45 ms			

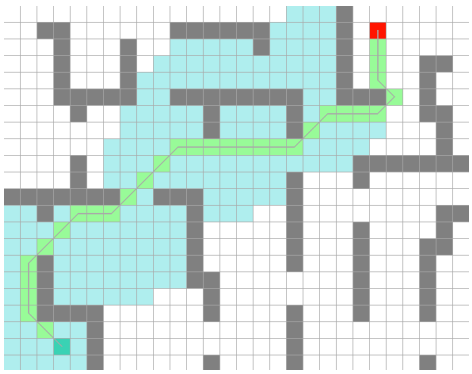


Figure 9: Experiment 1 – test map with simple obstacles. The optimal solution obtained by the A^* algorithm is shown.

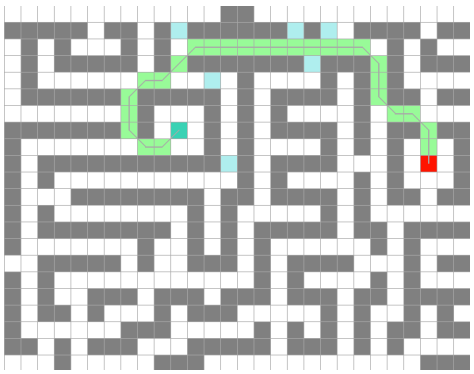


Figure 10: Experiment 2 – test map as a maze with 45% obstacles. The solution achieved by the JPS algorithm is shown.

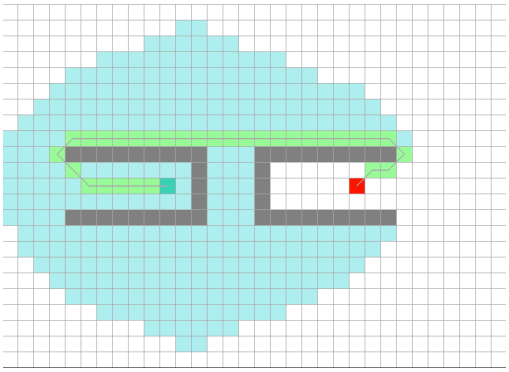


Figure 11: Experiment 4 – test map with deceptive problem and solution using A^*. The turquoise area represents the search space (expanded nodes).

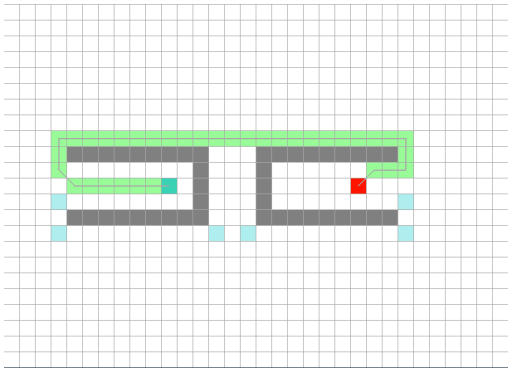


Figure 12: Experiment 4 – deceptive problem and a solution using the SS algorithm. The turquoise area represents the searched space (expanded nodes).

Experiment 4 (Tab. 4) – This experiment was performed on a map containing two identical obstacles. It is a simple deception problem of a known type. Of the results obtained, the Simple Subgoal and JPS algorithms performed best again. Both in terms of number of expanded nodes and computation time. The speed up of the Simple Subgoal algorithm was considerable in this case.

Table 4: Experiment 4, simple deception problem

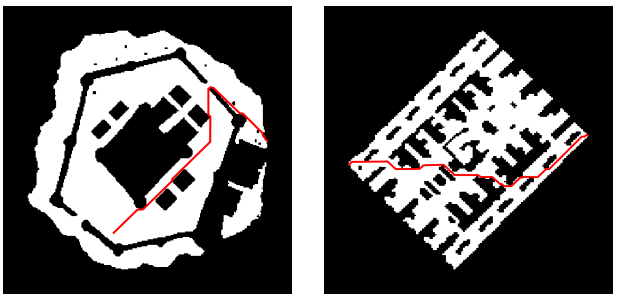
Method	Path length	Expand nodes	Speed-Up
Dijkstra	34.49	698	1
A*	34.49	336	1.34
A*+BB	34.49	320	1.11
JPS	34.49	12	2.06
JPS+BB	34.49	13	1.43
SS	36.83	12	7.58
Dijkstra 18.13 ms			

Experiment 5 (Tab. 5) – This class of experiments was conducted on larger maps from the MovingAI [30] test corpus. Specifically, the maps were AR0011SR and AR0013SR from the Baldur's Gate 2 game. Both of these maps are 512 x 512 cells in size. As can be seen from the results, methods using the bounding box are significantly slower than methods without this extension. The reason for this slowdown is due to the design of the bounding box itself and its extension. The map preprocessing time for the Simple Subgoal algorithm was 19 758 ms for the AR0011SR map and 7 945 ms for the AR0013SR map.

Table 5: Experiment 5, MovingAI corpus, 512 × 512

Method, for AR0011SR	Path length	Expand nodes	Speed-Up
Dijkstra	553.95	115706	1
A*	553.95	30442	1.72
A*+BB	553.95	30442	0.62
JPS	553.95	62	3.04
JPS+BB	553.95	62	1.15
SS	556.29	1133	9.43
Dijkstra 2 923 ms			
Method, for AR0013SR	**Path length**	**Expand nodes**	**Speed-Up**
Dijkstra	525.24	70791	1
A*	525.24	56261	1.04
A*+BB	525.24	55484	0.33
JPS	525.24	147	2.93
JPS+BB	525.24	147	0.60
SS	525.83	1133	7.91
Dijkstra 1 923 ms			

For reference, the experiments were run on a computer with an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz and 16GB of RAM.



(a) (b)

Figure 13: Games gridworld domains as benchmarks, Moving AI Lab [30], where (a) is AR0011SR map, (b) is AR0013SR map.

4 Conclusion

The aim of this paper was to present and evaluate two very interesting and relatively new algorithms for path planning on the chosen test problems. All tasks were implemented on an orthogonal two-dimensional mesh. This lattice with a defined Moore's neighborhood for each point is a good approximation for mobile object motion planning tasks and path planning tasks, respectively. The advanced JPS and Simple Subgoal algorithms streamline the search process by reducing the number of state space nodes explored. To compare them, representatives of classical path planning methods have also been implemented, these are the A^* algorithm and the Dijkstra algorithm. Five experiments were performed to compare the implemented algorithms. In these experiments, the length of the path found, the number of expanded nodes, and the time required to find the shortest path were evaluated. In addition, a simple search space reduction method was also implemented for the experiments, in the form of a bounding box for the JPS and A^* methods. The benefit of adding a bounding box depends on its setting and the form of the map. Comparative experiments have shown that the use of the JPS and Simple Subgoal methods leads to better shortest path search results with respect to the number of expanded nodes and also to lower time consumption. For the Simple Subgoal algorithm, it is important to note that it solves the problem in an engineering-optimal manner, but in best time. The JPS method is the second best in terms of speed, with a significant bonus of optimal solution. The very frequent and optimal A^* method has also proven its excellent applicability despite its age. All of the above methods except SS are optimal for the path length found. Given the presented detail and illustrative description of the JPS and SS algorithms, the authors hope to benefit from the good understanding of other authors and the creation of new implementations of these advanced path planning algorithms. The experiments were performed on implementations of the algorithms according to the authors' sources. The dependence of algorithms' speedup according to the implementation of the data structure is obvious.

Acknowledgement: This work was supported by IGA of BUT: FME-S-20- 6538 "Industry 4.0 and AI methods", and FEKT/FSI-J-22-7968 "Optimization of the cloud resources for the cyber security games.".

References

[1] Abd Algfoor, Z., Sunar, M. S., and Kolivand, H. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology 2015* (2015).

[2] Björnsson, Y., and Halldórsson, K. Improved heuristics for optimal path-finding on game maps. *AIIDE 6* (2006), 9–14.

[3] Botea, A. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2011), vol. 6.

[4] Botea, A., Müller, M., and Schaeffer, J. Near optimal hierarchical path-finding. *J. Game Dev. 1*, 1 (2004), 1–30.

[5] Brewka, G. Artificial intelligence—a modern approach by stuart russell and peter norvig, prentice hall. series in artificial intelligence, englewood cliffs, nj. *The Knowledge Engineering Review 11*, 1 (1996), 78–79.

[6] Bulitko, V., Björnsson, Y., and Lawrence, R. Case-based subgoaling in real-time heuristic search for video game pathfinding. *Journal of Artificial Intelligence Research 39* (2010), 269–300.

[7] Cazenave, T. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *2006 IEEE symposium on computational intelligence and games* (2006), IEEE, pp. 27–33.

[8] Costa, M. M., and Silva, M. F. A survey on path planning algorithms for mobile robots. In *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)* (2019), IEEE, pp. 1–7.

[9] Dijkstra, E. W., et al. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[10] Galceran, E., and Carreras, M. A survey on coverage path planning for robotics. *Robotics and Autonomous systems 61*, 12 (2013), 1258–1276.

[11] Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using networkx. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[12] Hamed, O., and Hamlich, M. Hybrid formation control for multi-robot hunters based on multi-agent deep deterministic policy gradient. *MENDEL 27*, 2 (Dec. 2021), 23–29.

[13] Harabor, D., and Botea, A. Breaking path symmetries on 4-connected grid maps. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference* (2010).

[14] Harabor, D., and Grastien, A. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2011), vol. 25.

[15] Harabor, D., Hechenberger, R., and Jahn, T. Benchmarks for pathfinding search: Iron harvest. In *Proceedings of the International Symposium on Combinatorial Search* (2022), vol. 15, pp. 218–222.

[16] Harabor, D. D., and Grastien, A. The jps pathfinding system. In *SOCS* (2012).

[17] Harabor, D. D., Uras, T., Stuckey, P. J., and Koenig, S. Regarding jump point search and subgoal graphs. In *IJCAI* (2019), pp. 1241–1248.

[18] Hart, P. E., Nilsson, N. J., and 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.

[19] Hernández, C., and Baier, J. A. Fast subgoaling for pathfinding via real-time search. In *Twenty-First International Conference on Automated Planning and Scheduling* (2011).

[20] Klobušníková, Z. Mobile robot path planning. Master thesis (in czech), Brno University of Technology, Brno, 2018.

[21] Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence 27*, 1 (1985), 97–109.

[22] LaValle, S. M. *Planning algorithms*. Cambridge university press, 2006.

[23] Lee, J.-Y., and Yu, W. A coarse-to-fine approach for fast path finding for mobile robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2009), IEEE, pp. 5414–5419.

[24] Lozano-Pérez, T., and Wesley, M. A. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM 22*, 10 (1979), 560–570.

[25] Maňáková, L. Advanced methods of mobile robot path planning. Master thesis (in czech), Brno University of Technology, Brno, 2020.

[26] Nobes, T. K., Harabor, D., Wybrow, M., and Walsh, S. D. The jps pathfinding system in 3d. In *Proceedings of the International Symposium on Combinatorial Search* (2022), vol. 15, pp. 145–152.

[27] Pochter, N., Zohar, A., Rosenschein, J. S., and Felner, A. Search space reduction using swamp hierarchies. In *Twenty-Fourth AAAI Conference on Artificial Intelligence* (2010).

[28] Rabin, S., and Silva, F. Jps+: An extreme a* speed optimization for static uniform cost grids. In *Game AI Pro 360*. CRC Press, 2019, pp. 95–108.

[29] Sankaranarayanan, J., Alborzi, H., and Samet, H. Efficient query processing on spatial networks. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems* (2005), pp. 200–209.

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

[31] Sturtevant, N., and Buro, M. Partial pathfinding using map abstraction and refinement. In *AAAI* (2005), vol. 5, pp. 1392–1397.

[32] Sturtevant, N. R., Felner, A., Barrer, M., Schaeffer, J., and Burch, N. Memory-based heuristics for explicit state spaces. In *Twenty-First International Joint Conference on Artificial Intelligence* (2009).

[33] Uras, T., and Koenig, S. Subgoal graphs for fast optimal pathfinding. In *Game AI Pro 360*. CRC Press, 2019, pp. 109–124.

[34] Uras, T., Koenig, S., and Hernández, C. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Twenty-Third International Conference on Automated Planning and Scheduling* (2013).

[35] VanRossum, G., and Drake, F. L. *The python language reference*. Python Software Foundation Amsterdam, Netherlands, 2010.