14

JPS+

An Extreme A Speed Optimization for Static Uniform Cost Grids*

Steve Rabin and Fernando Silva

14.1 Introduction

Jump point search (JPS) is a recently devised optimal pathfinding algorithm that can speed up searches on uniform cost grid maps by up to an order of magnitude over traditional A* [Harabor 12]. However, by statically analyzing a map and burning in directions to walls and jump points, it is possible to dramatically speed up searches even further, up to *two orders of magnitude* over traditional A*. To illustrate the difference in speed on a particular 40 × 40 map, A* found an optimal solution in 180.05 ns, JPS in 15.04 ns, and JPS+ in 1.55 ns. In this example, JPS+ was 116x faster than traditional A*, while remaining perfectly optimal.

Both JPS and JPS+ use a state-space pruning strategy that only works for grid search spaces where the cost of traversal is uniform with regard to distance. This chapter will explain in detail how JPS+ works and the exact specifics on how to implement it. JPS+ was first unveiled on June 2014 at the International Conference on Automated Planning and Scheduling (ICAPS); however, one of the authors of this chapter (Steve Rabin) independently invented the improved algorithm for storing directions to walls and

jump points a month before Harabor's initial publication. For consistency, the terms in this chapter will be from the original ICAPS paper [Harabor 14].

The code from this chapter, along with a full source demo, can be found on the book's website (http://www.gameaipro.com).

14.2 Pruning Strategy

JPS gets its tremendous speed from pruning the search space at runtime. This exploit exists because open areas in grids can be visited multiple times through equivalent paths. Consider a uniform cost grid space of 2×5 that contains no walls, as in Figure 14.1. Now consider how many optimal paths exist from the bottom left to the top right of this search space. Figure 14.1 shows that there are four identical cost paths. In many problems, traditional A* will redundantly check the same nodes along these paths to verify that a shorter path cannot be found. This is wasted work; with the ideas behind JPS+, we can avoid it.

JPS has a clever method for systematically choosing a single route through these equivalent paths. Figure 14.2 shows how JPS searches from the start node, visiting each node only once. That is, JPS will only consider the successors of a state in the direction of the arrows; all other successors are ignored. Of the total JPS speedup over A*, this pruning strategy accounts for roughly 50% of the speed improvement.

To understand this strategy in more detail, consider the node east of the center node in Figure 14.2. This is a node visited straight (nondiagonally) from the parent. Nodes visited straight from their parent only have successors that continue on in that same direction. In this case, the node east of the center node only considers the node to the east as a possible neighbor (pruning from considering the other seven directions).

Now consider the node northeast of the center node in Figure 14.2. This is a node visited diagonally from the parent. Nodes visited diagonally only consider three neighbors in the diagonal direction. The node northeast of the center node only considers nodes to the north, northeast, and east as possible neighbors (pruning from consideration the northwest, west, southwest, south, and southeast nodes).

JPS has another trick where it only puts relevant nodes, called *jump points*, on the open list. Since the open list is a bottleneck in traditional A*, a dramatic reduction in nodes on

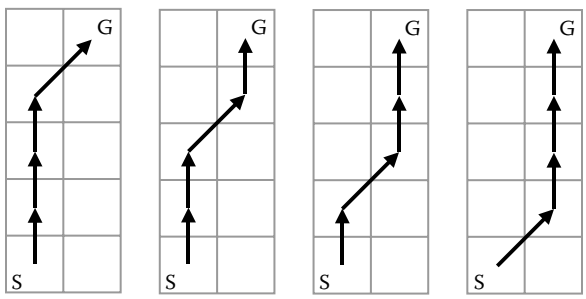


Figure 14.1

Four equivalent and optimal paths exist from the start node to the goal node.

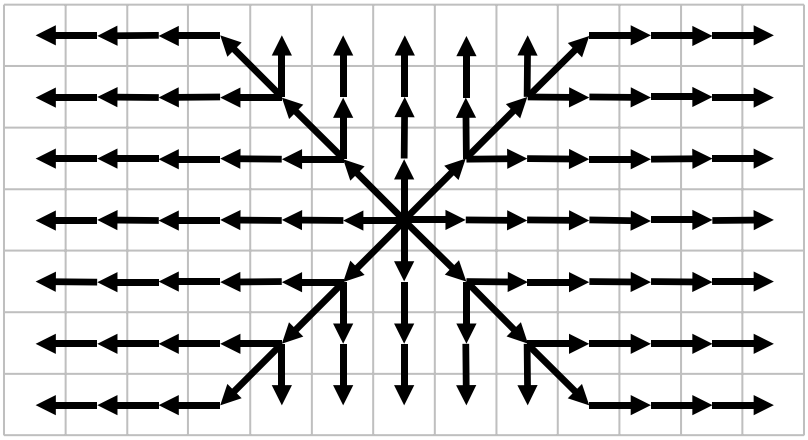
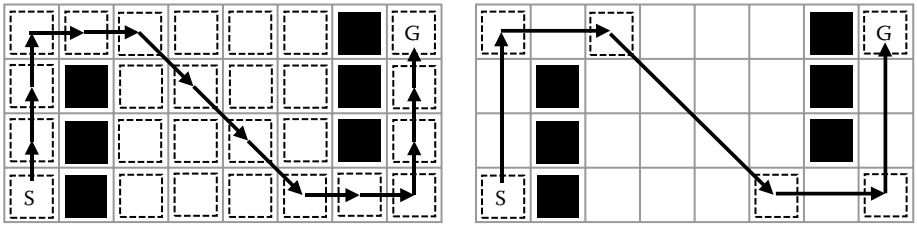


Figure 14.2

Systematic pruning rules keep nodes from being visited multiple times.



(a) Traditional A*: nodes placed on the open list

(b) JPS: jump point nodes placed on the open list

Figure 14.3

Nodes put on the open list in (a) traditional A* versus (b) JPS.

the open list results in a tremendous runtime savings. This second trick accounts for the other 50% speedup over traditional A*. Figure 14.3 illustrates the difference in the number of nodes that get placed on the open list.

Assuming that these two general ideas are understood, the rest of this chapter will provide a detailed explanation of how to correctly and efficiently implement them.

14.3 Forced Neighbors

There are certain cases where the pruning strategy from Figure 14.2 fails to visit a node due to walls in the search space. To address this, JPS introduced the concept of *forced neighbors*. Figure 14.4 shows the eight forced neighbor cases. Forced neighbors only occur when traveling in a straight or *cardinal* direction (north, south, west, or east). Forced neighbors are a signal that the normal pruning strategy will fail and that the current node must consider additional nodes.

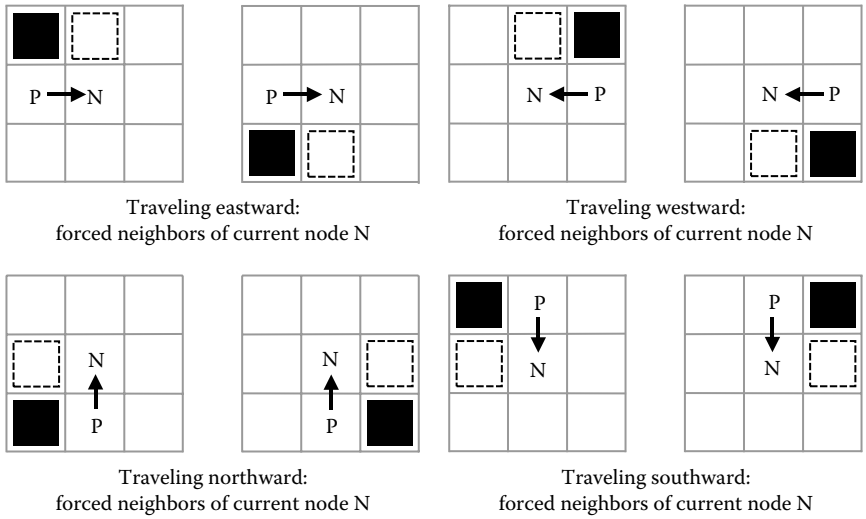


Figure 14.4

The eight forced neighbor cases. Forced neighbor nodes are outlined with a dashed line. Node P is the parent node and node N is the current node being explored. Note that forced neighbors are dependent on the direction of travel from the parent node.

14.4 Jump Points

Jump points are another concept introduced by JPS. Jump points are the intermediate points on the map that are necessary to travel through for at least one optimal path. JPS+ introduces four flavors of jump points: *primary*, *straight*, *diagonal*, and *target*.

14.4.1 Primary Jump Points

Primary jump points are easy to identify because they have a forced neighbor. In Figure 14.4, the primary jump points are the current node, N, for each situation shown. In Figure 14.5, we introduce a new map that shows the primary jump points, given the travel direction from the parent node.

Note that a node is only a primary jump point when traveling to the node in the direction indicated. So the same node can both be a jump point and not a jump point depending on the direction of travel during the search. Given that jump points only occur when traveling in a cardinal direction, there are four possible jump point flags for each node.

14.4.2 Straight Jump Points

Once all primary jump points have been identified, the straight jump points can be found. The straight jump points are nodes where traveling in a cardinal direction will eventually run into a primary jump point for that direction of travel (before running into a wall), as shown in Figure 14.6.

Note that a node can be both a primary and straight jump point for each direction of travel. Think of straight jump points as directions on how to get to the next primary jump point for that direction of travel.

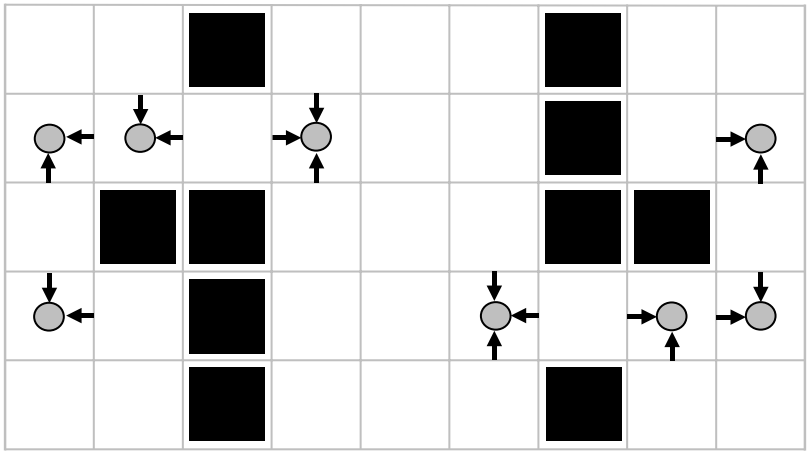


Figure 14.5

Primary jump points due to forced neighbors. Nodes marked with a circle are the jump points. The arrow direction indicates the direction of travel from the parent node for that node to be a jump point in that direction of travel.

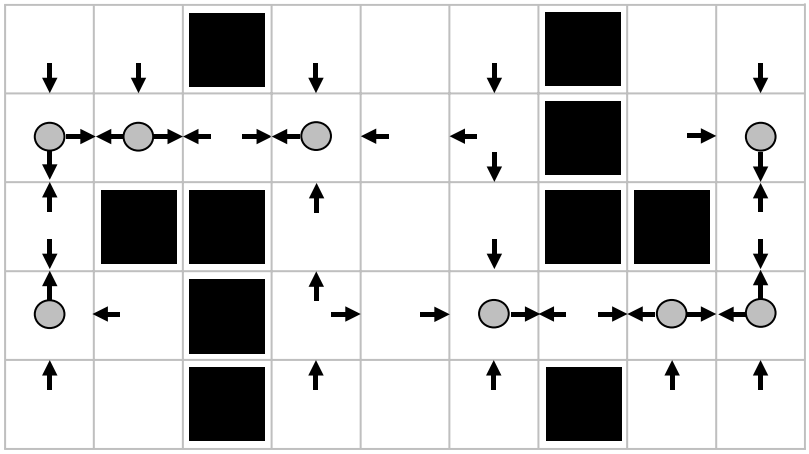


Figure 14.6

Straight jump points point to primary jump points. Primary jump points are marked with a circle. Straight jump points are any node with an arrow in it.

Figure 14.7 is a more detailed version of Figure 14.6 where the arrows have been replaced with distances. The distances indicate how many nodes away is the next primary jump point for that direction of travel. However, there is a very tricky thing going on in Figure 14.7 that wasn't apparent in Figure 14.6. The distance values aren't the distance to just any primary jump point, but only to primary jump points *in that direction of travel*. For example, the node on the very bottom left has a distance value of 3 in the up direction.

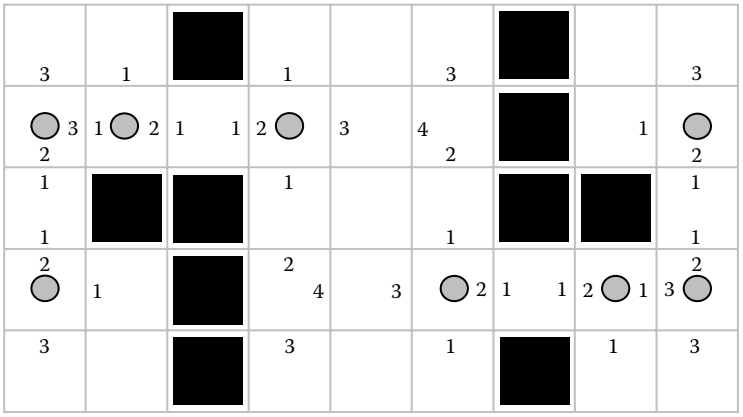


Figure 14.7

Straight jump points are marked with a distance indicating how many nodes away the next primary jump point is for that direction of travel.

It looks as if it should be a 1, but the primary jump point in the node above it is not a primary jump point *for that direction of travel* (double check Figure 14.5 to confirm). The actual primary jump point for traveling up is 3 nodes above.

14.4.3 Diagonal Jump Points

After straight jump points and their respective distances to primary jump points have been identified, we need to identify diagonal jump points. Diagonal jump points are any node in which a diagonal direction of travel will reach either a primary jump point or a straight jump point *that is traveling in a direction related to the diagonal direction* (before hitting a wall). For example, a node with a diagonal direction moving northeast is only a diagonal jump point if it runs into a primary or straight jump point traveling either north or east. This is consistent with the pruning strategy introduced in Figure 14.2.

As we did with straight jump points, we need to fill diagonal jump points with distances to the other jump points. Figure 14.8 shows our map with the diagonal jump points filled in.

14.5 Wall Distances

The information built up in Figure 14.8 is very close to the finished preprocessed map, but it needs wall distance information to be complete. In an effort to minimize the memory required for each node, we will represent wall distances as zero or negative numbers. Any node direction that is not marked with a straight or diagonal distance will be given a distance to the wall in that direction. Figure 14.9 shows the completed map where every node has distances for all eight directions. Additionally, we have deleted the primary jump point markers since they were only used to build up the distance map and are not needed at runtime.

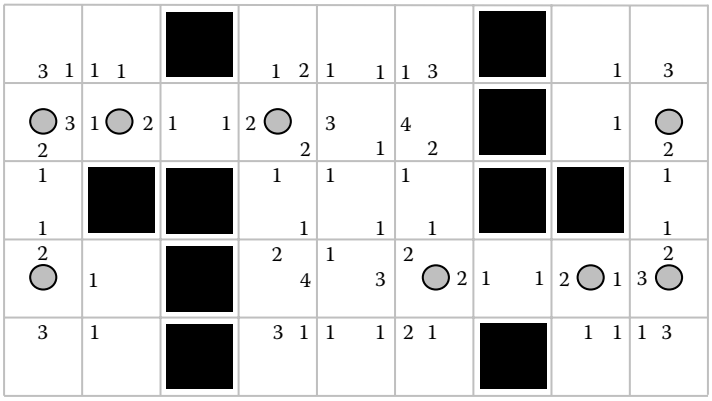


Figure 14.8

Diagonal jump points filled in with distances to primary and straight jump points.

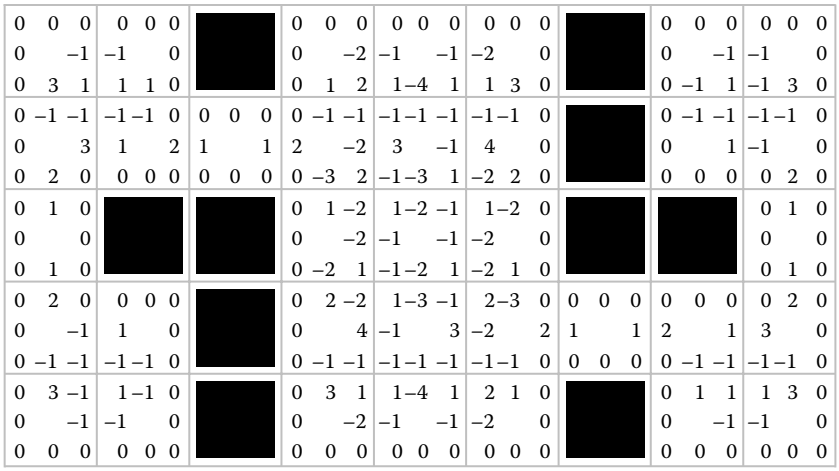


Figure 14.9

Wall distances added to all unmarked directions. Walls are either zero or negative and represent the distance to a wall (discard the negative sign).

14.6 Map Preprocess Implementation

As a practical matter, the complete precomputed map in Figure 14.9 can be created with the following strategy:

1. Identify all primary jump points by setting a directional flag in each node.
2. Mark with distance all westward straight jump points and westward walls by sweeping the map left to right.
3. Mark with distance all eastward straight jump points and eastward walls by sweeping the map right to left.

4. Mark with distance all northward straight jump points and northward walls by sweeping the map up to down.
5. Mark with distance all southward straight jump points and southward walls by sweeping the map down to up.
6. Mark with distance all southwest/southeast diagonal jump points and southwest/southeast walls by sweeping the map down to up.
7. Mark with distance all northwest/northeast diagonal jump points and northwest/northeast walls by sweeping the map up to down.

Listing 14.1 provides an example of the sweep to compute westward numbers in step 2. Similar code must be written for the other three sweep directions in steps 3–5. Listing 14.2 provides an example of the sweep to compute southwest numbers as part of step 6. Similar code must be written to compute southeast, northeast, and northwest numbers.

Listing 14.1. Left to right sweep to mark all westward straight jump points and all westward walls.

```
for (int r = 0; r < mapHeight; ++r)
{
    int count = -1;
    bool jumpPointLastSeen = false;

    for (int c = 0; c < mapWidth; ++c)
    {
        if (m_terrain[r][c] == TILE_WALL)
        {
            count = -1;
            jumpPointLastSeen = false;
            distance[r][c][West] = 0;
            continue;
        }

        count++;

        if (jumpPointLastSeen)
        {
            distance[r][c][West] = count;
        }
        else //Wall last seen
        {
            distance[r][c][West] = -count;
        }

        if (jumpPoints[r][c][West])
        {
            count = 0;
            jumpPointLastSeen = true;
        }
    }
}
```

Listing 14.2. Down to up sweep to mark all southwest diagonal jump points and all southwest diagonal walls.

```
for (int r = 0; r < mapHeight; ++r)
{
    for (int c = 0; c < mapWidth; ++c)
    {
        if (!IsWall(r, c))
        {
            if (r == 0 || c == 0 || IsWall(r-1, c) ||
                IsWall(r, c-1) || IsWall(r-1, c-1))
            {
                //Wall one away
                distance[r][c][Southwest] = 0;
            }
            else if (!IsWall(r-1, c) && !IsWall(r, c-1) &&
                (distance[r-1][c-1][South] > 0 ||
                 distance[r-1][c-1][West] > 0))
            {
                //Straight jump point one away
                distance[r][c][Southwest] = 1;
            }
            else
            {
                //Increment from last
                int jumpDistance =
                    distance[r-1][c-1][Southwest];

                if (jumpDistance > 0)
                {
                    distance[r][c][Southwest] =
                        1 + jumpDistance;
                }
                else
                {
                    distance[r][c][Southwest] =
                        -1 + jumpDistance;
                }
            }
        }
    }
}
```

14.7 Runtime Implementation

The genius of the preprocessed map is that it contains many of the decisions required for the search, thus making the runtime code much simpler and faster than traditional JPS. For example, the recursive step from JPS is completely eliminated and only jump points are ever examined.

The pseudocode in Listing 14.3 is the complete JPS+ runtime algorithm. However, there are several aspects that need clarification. The first is the `ValidDirLookUpTable`. This table is used to only consider directions in the spirit of Figure 14.1. For example, if traveling in a diagonal direction such as northeast, the directions east, northeast, and north

Listing 14.3. Complete runtime pseudocode for JPS+.

```
ValidDirLookUpTable
    Traveling South: West, Southwest, South, Southeast, East
    Traveling Southeast: South, Southeast, East
    Traveling East: South, Southeast, East, Northeast, North
    Traveling Northeast: East, Northeast, North
    Traveling North: East, Northeast, North, Northwest, West
    Traveling Northwest: North, Northwest, West
    Traveling West: North, Northwest, West, Southwest, South
    Traveling Southwest: West, Southwest, South

while (!OpenList.IsEmpty())
{
    Node* curNode = OpenList.Pop();
    Node* parentNode = curNode->parent;

    if (curNode == goalNode)
        return PathFound;

    foreach (direction in ValidDirLookUpTable
             given parentNode)
    {
        Node* newSuccessor = NULL;
        float givenCost;

        if (direction is cardinal &&
            goal is in exact direction &&
            DiffNodes(curNode, goalNode) <=
            abs(curNode->distances[direction]))
        {
            //Goal is closer than wall distance or
            //closer than or equal to jump point distance
            newSuccessor = goalNode;
            givenCost = curNode->givenCost +
                        DiffNodes(curNode, goalNode);
        }
        else if (direction is diagonal &&
                 goal is in general direction &&
                 (DiffNodesRow(curNode, goalNode) <=
                 abs(curNode->distances[direction]) ||
                 (DiffNodesCol(curNode, goalNode) <=
                 abs(curNode->distances[direction]))))
        {
            //Goal is closer or equal in either row or
            //column than wall or jump point distance

            //Create a target jump point
            int minDiff = min(RowDiff(curNode, goalNode),
                              ColDiff(curNode, goalNode));
            newSuccessor =
                GetNode (curNode, minDiff, direction);
            givenCost = curNode->givenCost +
                (SQRT2 * minDiff);
        }
```

```
        else if (curNode->distances[direction] > 0)
        {
            //Jump point in this direction
            newSuccessor = GetNode(curNode, direction);
            givenCost = DiffNodes(curNode, newSuccessor);
            if (diagonal direction) {givenCost *= SQRT2;}
            givenCost += curNode->givenCost;
        }

        //Traditional A* from this point
        if (newSuccessor != NULL)
        {
            if (newSuccessor not on OpenList or ClosedList)
            {
                newSuccessor->parent = curNode;
                newSuccessor->givenCost = givenCost;
                newSuccessor->finalCost = givenCost +
                    CalculateHeuristic(newSuccessor, goalNode);
                OpenList.Push(newSuccessor);
            }
            else if(givenCost < newSuccessor->givenCost)
            {
                newSuccessor->parent = curNode;
                newSuccessor->givenCost = givenCost;
                newSuccessor->finalCost = givenCost +
                    CalculateHeuristic(newSuccessor, goalNode);
                OpenList.Update(newSuccessor);
            }
        }
    }
}

return NoPathExists;
```

must be considered. If traveling in a cardinal direction, continued movement in the cardinal direction plus any possible forced neighbor directions plus the diagonal between the forced neighbor and the original cardinal direction must be considered. For example, if traveling east, the directions east plus the possible forced neighbors of north and south plus the diagonals northeast and southeast must be considered. The actual distances in each node will further clarify if these are worth exploring, but this is a first-pass pruning. Figure 14.10 shows an example computed path where you can see the directions considered at each node based on the `ValidDirLookUpTable`.

Consider another important clarification regarding the middle conditional in the for loop to create a target jump point (Listing 14.3). The need for this is subtle and nonobvious. When approaching the goal node, we might need a *target* jump point (our fourth type of jump point) between the current node and the goal node in order to find a path that is grid aligned to the goal. This arises only when traveling diagonally, the direction of travel is toward the goal node, and the distance to the goal node in row distance or column distance is less than or equal to the current node's diagonal distance to a wall or jump point. The newly synthesized target jump point will be constructed by taking the minimum of the row distance and the column distance to the goal node and continuing travel diagonally by that amount. If the goal node is directly diagonal in the direction of travel, this new target

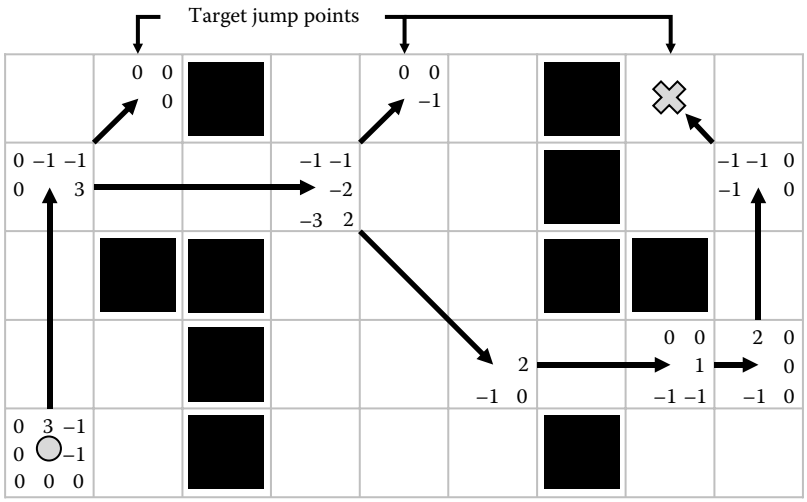


Figure 14.10

Runtime search from bottom left node to top right goal node. Only node distances used in the search are included in the figure. Note that three target jump points are created along the top row based on potential intermediate routes to the goal.

jump point will be equal to the goal node. Otherwise, this target jump point is a possible intermediate jump point between the current node and the goal node. Figure 14.10 shows an example computed path where three target jump points are created at runtime.

To appreciate the speed of JPS+, consider the search in Figure 14.10. Only 10 nodes are even considered during the entire search with 7 of them being absolutely necessary for an optimal path. Contrast this with traditional JPS that would look at every node on the map during its recursive step. Additionally, since all wall information is stored in the grid cells themselves as distances, this data structure is extremely cache friendly since neighboring rows or columns aren't constantly checked for the existence of walls. The cumulative result is that JPS+ does very little runtime processing and considers an extremely limited number of nodes. This accounts for the tremendous improvement over traditional JPS.

In order to achieve good results with JPS+, it is advised to use a heapsort algorithm for the open list, preallocate all node memory along with a dirty bit for cheap initialization, and use an octile heuristic [Rabin 13]. If you have between 1 and 10 MB of extra memory available, you can achieve a ~10x speed increase by using a bucket-sorted priority queue for the open list. Create a bucket for every 0.1 cost and then use an unsorted preallocated array for each bucket. If you use a LIFO strategy in each bucket, the final path won't be more than 0.1 worse than optimal and it will be blazing fast. For A*, this data structure would require 10 to 100 MB, but JPS+ puts far fewer nodes on the open list, which makes this optimization much more practical.

14.8 Conclusion

JPS+ takes a great deal of the runtime computation from JPS and stores it directly in the map. As a result, the algorithm is up to an order of magnitude faster than traditional JPS and two orders of magnitude faster than a highly optimized A*. The trade-off for this

speed is that the map is static (walls can't be added or removed), the map must be preprocessed with eight numbers stored per grid cell, and the map is a uniform cost grid.

The degree of speed gains is directly proportional to the openness of the map. If the map contains large open areas, JPS+ is extremely fast and will achieve up to two orders of magnitude speed improvement over A*. If the map is intensely mazelike consisting primarily of jagged diagonal walls, then JPS+ is more conservatively around 20% faster than traditional JPS and about 2.5x faster than a highly optimized A* solution.

References

[Harabor 12] Harabor, D. and Grastien, A. 2012. The JPS pathfinding system. In *Proceedings of the Fifth Symposium on Combinatorial Search* (*SoCS*), Niagara Falls, Ontario, Canada. Available online at: http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-socs12.pdf (accessed September 10, 2014).

[Harabor 14] Harabor, D. and Grastien, A. 2014. Improving jump point search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, Portsmouth, NH. Video available at: https://www.youtube.com/watch?v=NmM4pv8uQwI (accessed September 10, 2014).

[Rabin 13] Rabin, S. and Sturtevant, N. 2013. Pathfinding architecture optimizations. In *Game AI Pro: Collected Wisdom of Game AI Professionals*. A K Peters/CRC Press, Boca Raton, FL.