Artificial Intelligence 4 Games

Jump Point Search

2020

Jakub Kowalski, University of Wrocław

JPS+

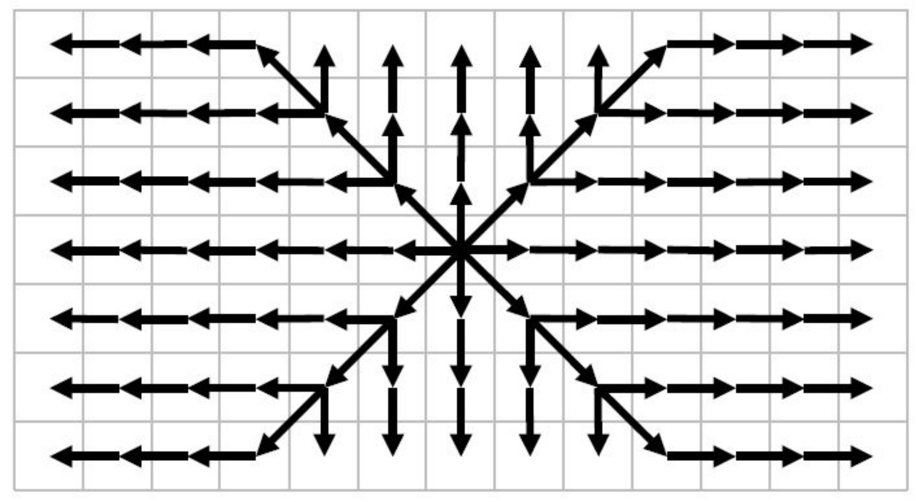
Literature

★ S. Rabin, F. Silva. An Extreme A* Speed Optimization for Static Uniform Cost Grids. Game AI Pro 2: Collected Wisdom of Game AI Professionals, pp. 131-143, 2015.

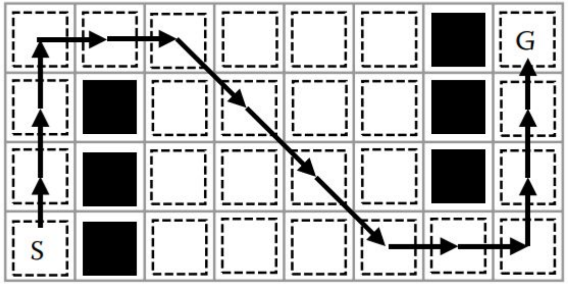
JPS tricks

Single route for equivalent optimal paths

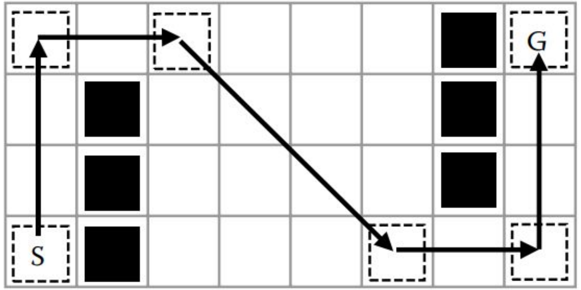
Pruning rules keep nodes from being visited multiple times (canonical ordering).



Reducing open list size by introducing jump points



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

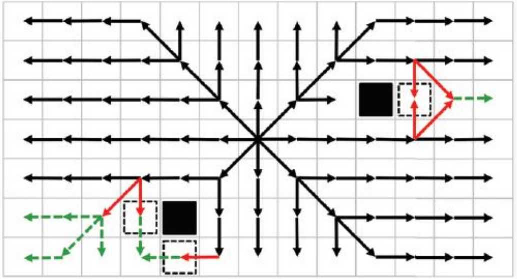


JPS:
(b) jump point nodes placed on the open list

JPS+

Map Preprocessing

1. Computing Forced Neighbors



2. Computing Jump Points
    2.1. Primary (due to forced neighbors)
    2.2. Straight (pointing primary)
    2.3. Diagonal (pointing primary and straight)
3. Computing Wall Distances

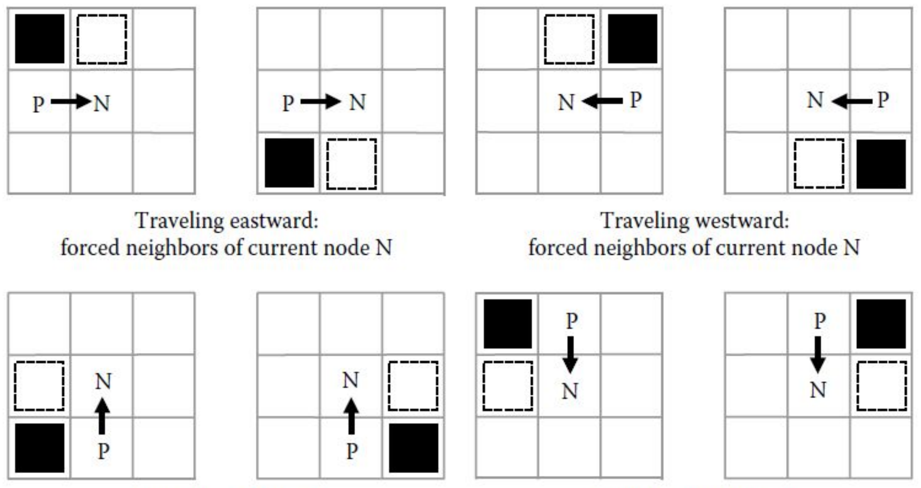
Runtime

- ➔ Based on A*
- ➔ First it checks for target in cardinal direction of travel and forces jump points in diagonal direction when “near” the target
- ➔ Then checks jump points in directions corresponding to the one we came from
- ➔ And adds them as successor nodes
- ➔ Managing nodes and open/closed lists as in standard A*

Primary Jump Points

Computing Forced Neighbors

We note the primary jump points (if they occur before hitting the wall) **in a cardinal direction of travel**.



Traveling eastward: forced neighbors of current node N

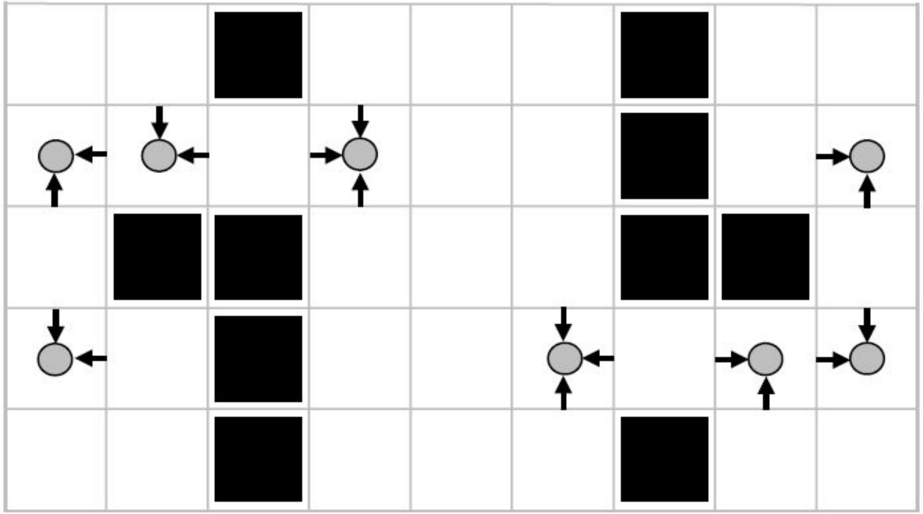
Traveling westward: forced neighbors of current node N

Traveling northward: forced neighbors of current node N

Traveling southward: forced neighbors of current node N

Placement

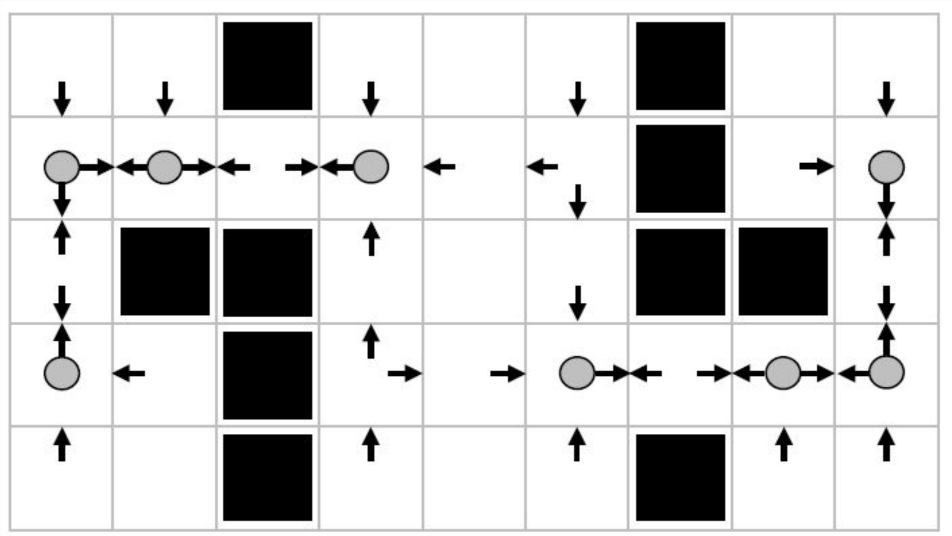
Placed for the given *(node, direction)* pair if the node has a forced neighbor for that direction.



Straight Jump Points

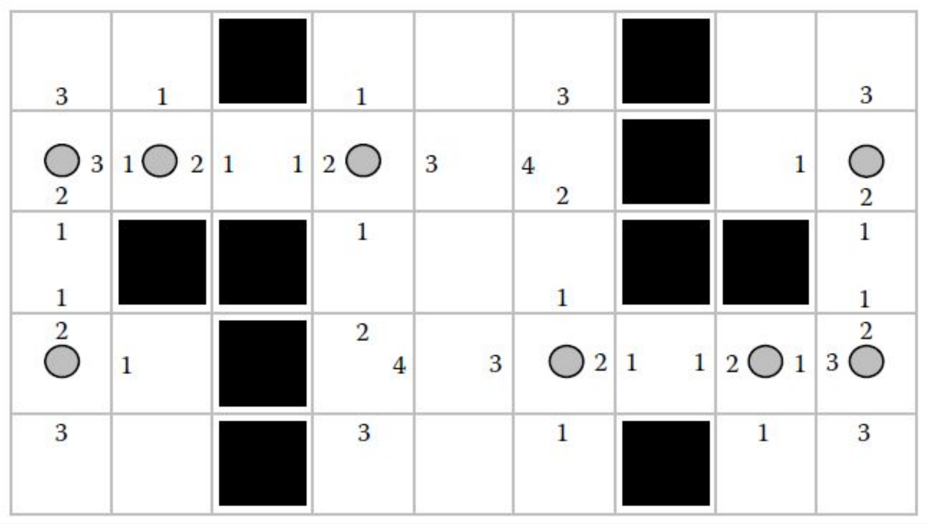
Placement

We note the primary jump points (if they occur before hitting the wall) **in a cardinal direction of travel**.



Distances

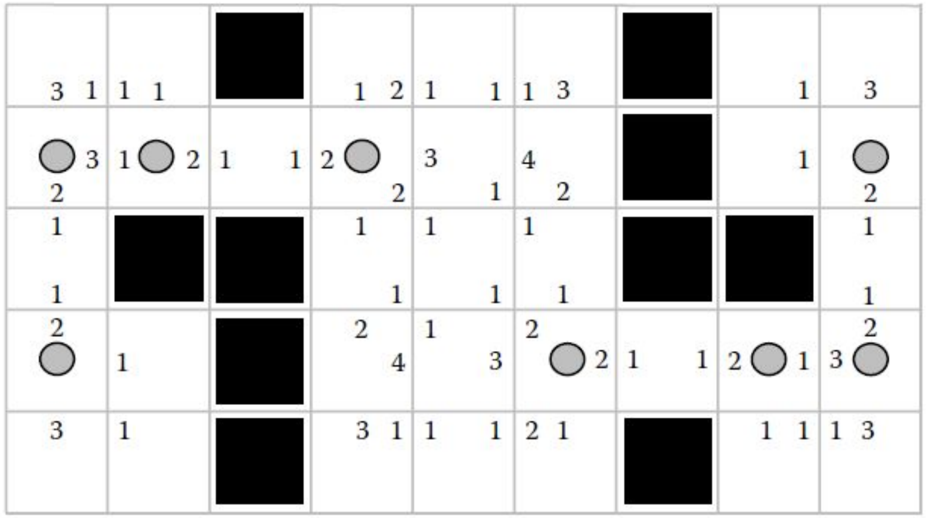
We replace arrows with the distances to primary jump points **complying with the direction of travel** (see the leftmost bottom node).



Diagonal Jump Points

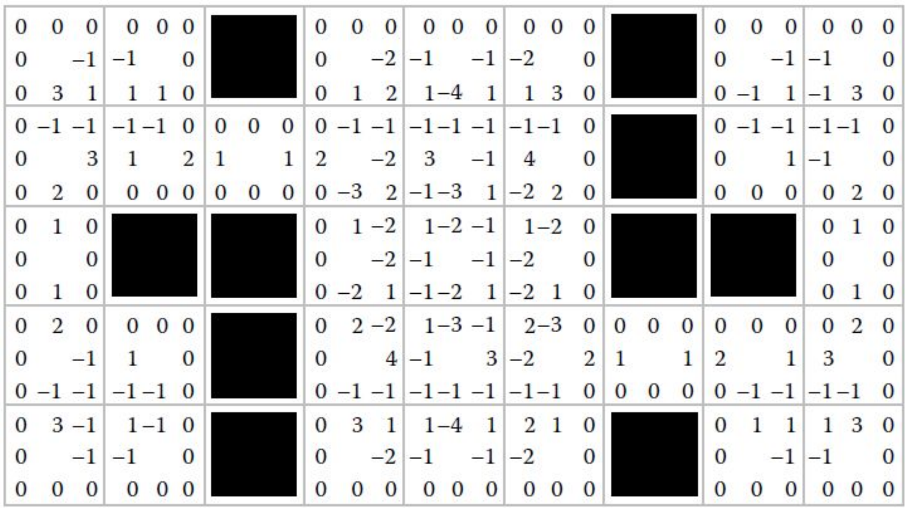
Placement

We note the distances in the diagonal direction of travel to straight or primary jump point in a **related** direction.



Distances to the walls

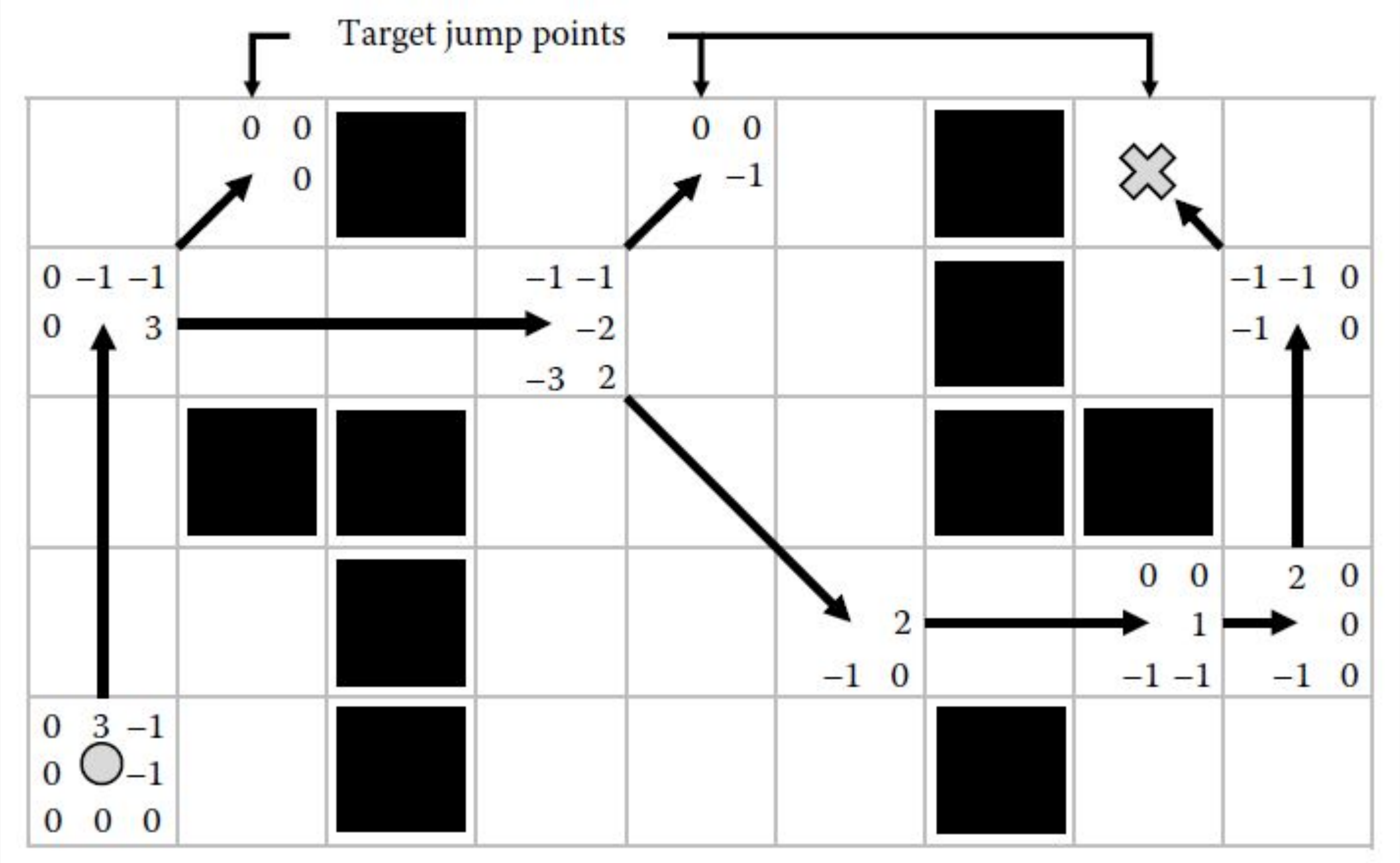
Distances to walls added for every direction not hitting a jump point. To save the space they are encoded as the negative numbers.



Runtime Pseudocode

```
ValidDirLookUpTable
    Traveling South: West, Southwest, South, Southeast, East
    Traveling Southeast: South, Southeast, East
    Traveling East: . . .
while (!OpenList.IsEmpty())
{
    Node* curNode = OpenList.Pop(); ClosedList.add(curNode);
    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 &&  minDiff>0 &&
                 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;
```

Running Example



Summary

Requirements

- Uniform-cost grids
- Only static obstacles
- Diagonal movement; zero-width yes/no
- Preprocessing with eight numbers per cell

Implementation

- Octile distance for heuristic
- Heapsort for the priority queue
- Alternatively bucket sort (e.g. for 0.1 cost) to major speed up but slightly suboptimal paths

Speedups

- Example 40×40 times:
  - A*: 180.05 ns
  - JPS: 15.04 ns
  - JPS+ 1.55 ns
- Efficiency gain is proportional to the openness of the map
- With large open areas JPS+ may achieve two orders of magnitude speedup over A*
- For maze-like maps it is still ~2.5× faster

Original JPS

Literature

★ D. Harabor, A. Grastien: Online graph pruning for pathfinding on grid maps. AAAI, pp. 1114-1119, 2011.

★ D. Harabor, A. Grastien: The JPS Pathfinding System. SOCS, 2012

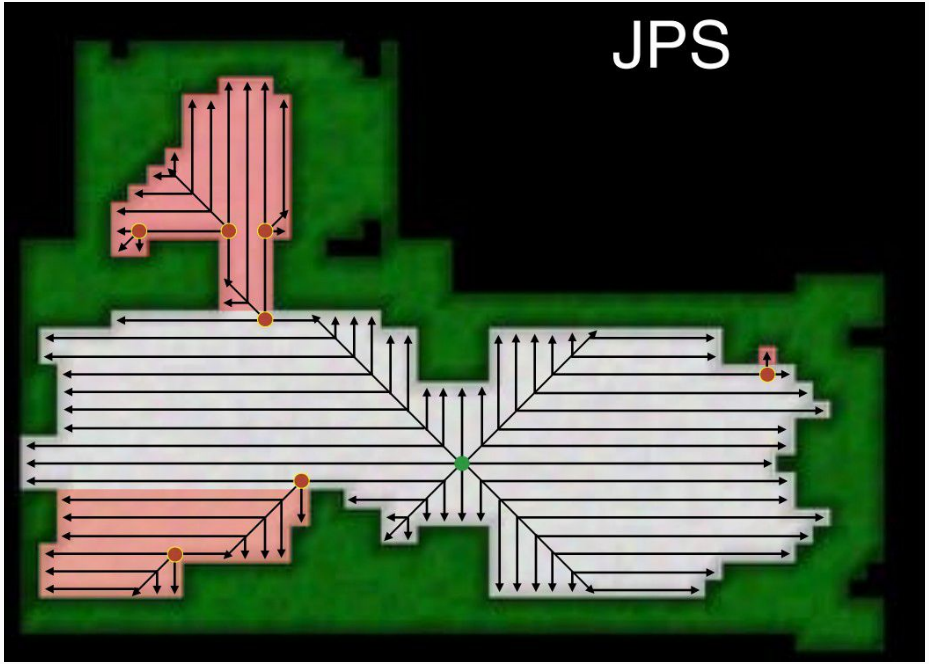
★ D. Harabor, A. Grastien: Improving Jump Point Search. ICAPS, pp. 128-135, 2014.

“Online” JPS

Original JPS does not have the preprocessing phase. Following the canonical ordering and discovering jump points is done during a runtime search.

Thus, its main advantage over A* is far less nodes placed in the open list (we still place there only jump points and goal).

Its advantage over JPS+ is that we can work on dynamic maps. Also, no additional memory is required.



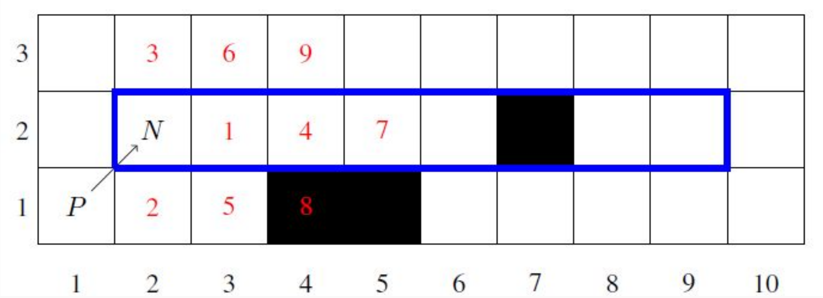
Block-based Symmetry Breaking

Low-level search optimization

- ➔ We can encode grid as a matrix of bits (0-empty, 1-blocked)
- ➔ Copy 90° rotated map to handle vertical travel

Specific bitwise operations

- ➔ Little-endian (lowest bit leftmost) for left to right travel; big-endian operations otherwise
- ➔ Shifting to get current node
- ➔ Detecting dead-ends
- ➔ Detecting forced neighbours
- ➔ Detecting the target node



$$B_{\uparrow} = [0,0,0,0,0,0,0,0]$$
$$B_{N} = [0,0,0,0,0,1,0,0]$$
$$B_{\downarrow} = [0,0,1,1,0,0,0,0]$$

Improved Pruning Rules

- ➔ JPS distinguishes between jump points that have at least one forced neighbour and those that have none
- ➔ The latter are just the intermediate jump points used to change direction.
- ➔ They can be safely pruned by immediately replacing them by their (at most three) successors.

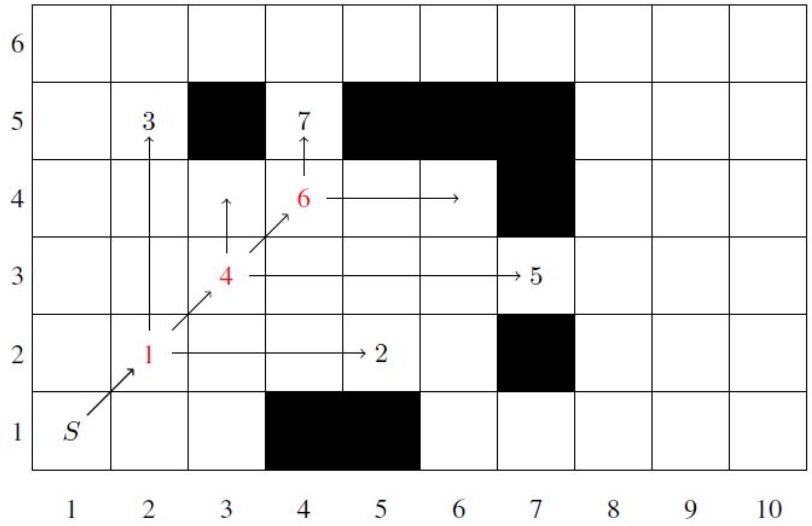


Figure 4: We prune all intermediate jump points (here nodes 1, 4 and 6) and instead generate their immediate successors (nodes 2, 3, 5 and 7) as children of the node from where initiated the jump (i.e., S).

Summary

Speedups

	A*		JPS	
	M.Time	G.Time	M.Time	G.Time
D. Age: Origins	58%	42%	14%	86%
D. Age 2	58%	42%	14%	86%
StarCraft	61%	39%	11%	89%

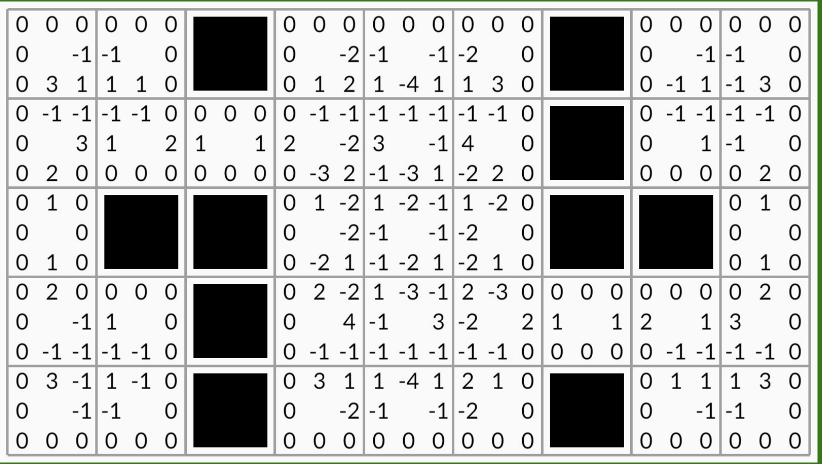
Table 1: A comparative breakdown of total search time on three realistic video game benchmarks. M.Time is the time spent manipulating nodes on open or closed. G.Time is the time spent generating successors (i.e. scanning the grid).

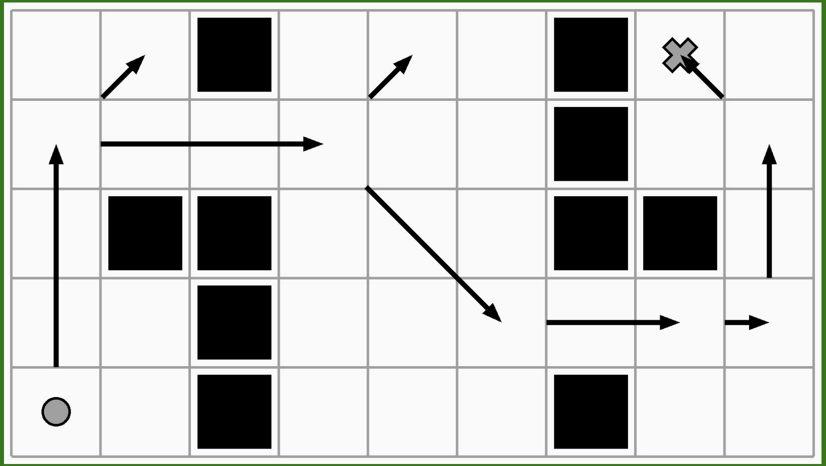
	StarCraft		Dragon Age: Origins		Dragon Age 2	
	Time (μs)	Branches	Time (μs)	Branches	Time (μs)	Branches
JPS 2011	19.89	3.76	6.36	3.31	4.54	3.25
JPS (B)	1.85	3.76	0.93	3.31	0.85	3.25
JPS (B+P)	7.10	22.72	1.96	8.11	1.54	6.62
JPS+	0.38	3.76	0.21	3.31	0.20	3.25
JPS+ (P)	1.56	22.72	0.52	8.11	0.46	6.62

Summary

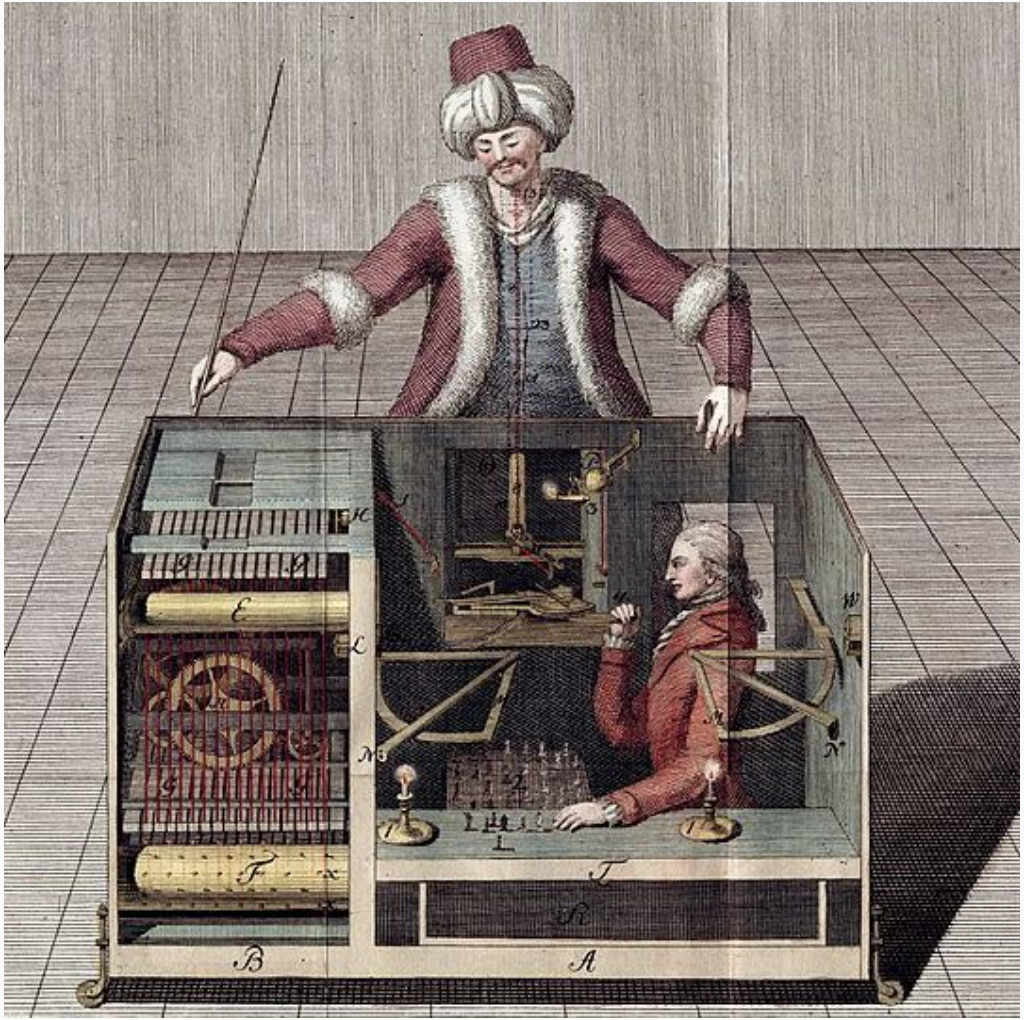
★ JPS is blazingly fast A* improvement for uniform cost grids

★ JPS+ is even faster but require maps to be static





Thanks!



Bonus reference quiz