Report First Project IAJ

João Vitór Sebastião Carvalho Tiago Antunes

2023-09-21

1 Introduction

The goal of this project was to create different levels of path finding algorithms, and compare their performance.
We compared 4 differente algorithms: Basic A*(unordered list for open set, unordered list for closed set), Basic A* but using tiebreaking (unordered list for open set, unordered list for closed set), NodeArray A* (NodeArray for open and closed set) and NodeArray A* with Goal Bounding.

2 Basic A*

2.1 Algorithm

The A* is a search algorithm that uses a heuristic to find the best path between 2 nodes. Even though it's a basic algorithm, it shows relatively good performance when compared to other algorithms like djikstra. Despite being a basic algorithm, it's performance can be improved by using better data structures, or using other optimizations like we're going to show in the next sections.

2.2 Data

Table 1: Basic A* performance (Path 1)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	10181.85
GetBestAndRemove	1904	44.09
AddToOpen	1954	1.82
SearchInOpen	18564	269.64
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	1904	1.45
SearchInClosed	18460	9710.25
RemoveFromClosed	0	0

Table 2: Basic A* performance (Path 2)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	21432.88
GetBestAndRemove	2890	114.64
AddToOpen	3020	2.54
SearchInOpen	28291	759.53
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	2890	1.76
SearchInClosed	27990	20319.63
RemoveFromClosed	0	0

Table 3: Basic A* grid information (Path 1)

TotalPNodes	MaxOpenNodes	Fill
1904	77	Very Large

Table 4: Basic A* grid information (Path 2)

TotalPNodes	MaxOpenNodes	Fill
2889	133	Very Large

3 Basic A* with tiebraking

3.1 Algorithm

This algorithm is basically the previous one, but we use tiebraking to break ties between nodes with the same f value. When 2 nodes have the same f value, we use order the nodes by smallest h value. This way we first pick the node that is closer to the goal.

3.2 Data

Table 5: Basic A* with tiebraking performance (Path 1)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	10029.02
GetBestAndRemove	1904	112.83
AddToOpen	1954	1.56
SearchInOpen	18564	260.07
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	1904	1.27
SearchInClosed	18460	9524.91
RemoveFromClosed	0	0

Table 6: Basic A* with tiebraking performance (Path 2)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	21970.11
GetBestAndRemove	2890	282.13
AddToOpen	3021	1.94
SearchInOpen	28291	768.2
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	2890	1.61
SearchInClosed	27990	20761.34
RemoveFromClosed	0	0

Table 7: Basic A* with tiebraking grid information (Path 1)

TotalPNodes	MaxOpenNodes	Fill
1904	77	Very Large

Table 8: Basic A* with tiebraking* grid information (Path 2)

TotalPNodes	MaxOpenNodes	Fill
2889	133	Very Large

3.3 Comparison

Looking at the data from this and the previous algorithm, that there's a little optimization on the data of path 1 even though there's more spent time getting the best node from the open set, due to more comparisons. In path 2 we still see an increase on the time for GetBestAndRemove, but surprisingly there's more time spent this may be due to bad measurement of data, since this contradicts what is expected.

4 NodeArray A*

4.1 Algorithm

NodeArray A* is an A* implementation that uses a NodeArray to store the nodes. We use this array as our open and closed set, and we change the status property of the nodes when we add them to the open or closed set. This way, we can search for nodes in the open and closed set in constant time.

4.2 Data

4.3 Comparison

NodeArray A* is faster than the previous algorithms, due to the fact that we can search for nodes in the open and closed set in constant time, as we can see by the reduction of the SearchInOpen and SearchInClosed time, even though it spends more time on the AddToOpen due to the use of a PriorityHeap.

Table 9: NodeArray A* performance (Path 1)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	5.48
GetBestAndRemove	200	1.96
AddToOpen	232	1.13
SearchInOpen	1044	0.04
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	100	0.01
SearchInClosed	1009	0.04
RemoveFromClosed	0	0

Table 10: NodeArray A* performance (Path 2)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	159.11
GetBestAndRemove	2890	38.36
AddToOpen	3019	9.31
SearchInOpen	28247	1.84
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	2885	0.58
SearchInClosed	27932	1.73
RemoveFromClosed	0	0

Table 11: NodeArray A* grid information (Path 1)

TotalPNodes	MaxOpenNodes	Fill
1904	77	Very Large

Table 12: NodeArray A* grid information (Path 2)

TotalPNodes	MaxOpenNodes	Fill
2884	135	Very Large

5 NodeArray A* with Goal Bounding

5.1 Algorithm

By using precomputation of the grid, we can make bounding boxes for each node and improve the NodeArray A* algorithm. We do this by using djikstra to calculate fastest path from each node to all other nodes. This way, we know which direction we should choose when trying to go to a specific node. This optimization causes, sometimes, a heavy increase on the starting time, due to the precomputation, but it improves the runtime of the algorithm by a lot.

5.2 Data

Table 13: NodeArray A* with Goal Bounding performance (Path 1)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	10.62
GetBestAndRemove	200	0.43
AddToOpen	216	0.34
SearchInOpen	235	0
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	100	0.01
SearchInClosed	126	0
RemoveFromClosed	0	0

Table 14: NodeArray A* with Goal Bounding performance (Path 2)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	17.12
GetBestAndRemove	158	0.40
AddToOpen	165	0.35
SearchInOpen	388	0.01
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	158	0.03
SearchInClosed	282	0.01
RemoveFromClosed	0	0

Table 15: NodeArray A* with Goal Bounding grid information (Path 1)

TotalPNodes	MaxOpenNodes	Fill
228	9	Very Small

Table 16: NodeArray A* with Goal Bounding grid information (Path 2)

TotalPNodes	MaxOpenNodes	Fill
157	8	Very Small

5.3 Comparison

Comparing this data with the previous ones, we can see that this is by far the best optimization in terms of runtime. This is due to the use of bounding boxes, that shorten the amounts of nodes we process, and thus the amount of calls to add, remove and search in the open and closed set.

6 Bonus Level - Dead-End Heuristic

6.1 Algorithm

For the Bonus Level, we implemented the A* algorithm with the Dead-End heuristic. This heuristic is calculated by using a precomputation of the grid, where we create clusters for each room. This clusters are created using a floodfill in the beginning of the precomputation. At runtime, we calculate the possible paths in the room graph and update the heurisitic.

6.2 Data

Table 17: NodeArray A* with Goal Bounding performance (Path 2)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	10.62
GetBestAndRemove	200	0.43
AddToOpen	216	0.34
SearchInOpen	235	0
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	100	0.01
SearchInClosed	126	0
RemoveFromClosed	0	0

Table 18: NodeArray A* with Goal Bounding performance (Path 3)

Method	Calls	Execution Time (ms)
A*Pathfinding.Search	1	17.12
GetBestAndRemove	158	0.40
AddToOpen	165	0.35
SearchInOpen	388	0.01
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	158	0.03
SearchInClosed	282	0.01
RemoveFromClosed	0	0

Table 19: NodeArray A* with Goal Bounding grid information (Path 2)

TotalPNodes	MaxOpenNodes	Fill
228	9	Large

Table 20: NodeArray A* with Goal Bounding grid information (Path 3)

TotalPNodes	MaxOpenNodes	Fill
157	8	Large

6.3 Comparison

7 Conclusions

Analysing all algorithms we can acess that A* by itself is already a good algorithm, but it's optimizations make it must faster, without compromising finding the best path.

Also, we can notice that adding pre-processing to the algorithm can improve it's runtime by a lot, even though it takes some time to do it. We can see this in the use of precomputation on the Goal Bounding algorithm and on the calculation of the Dead-End heurisitic.