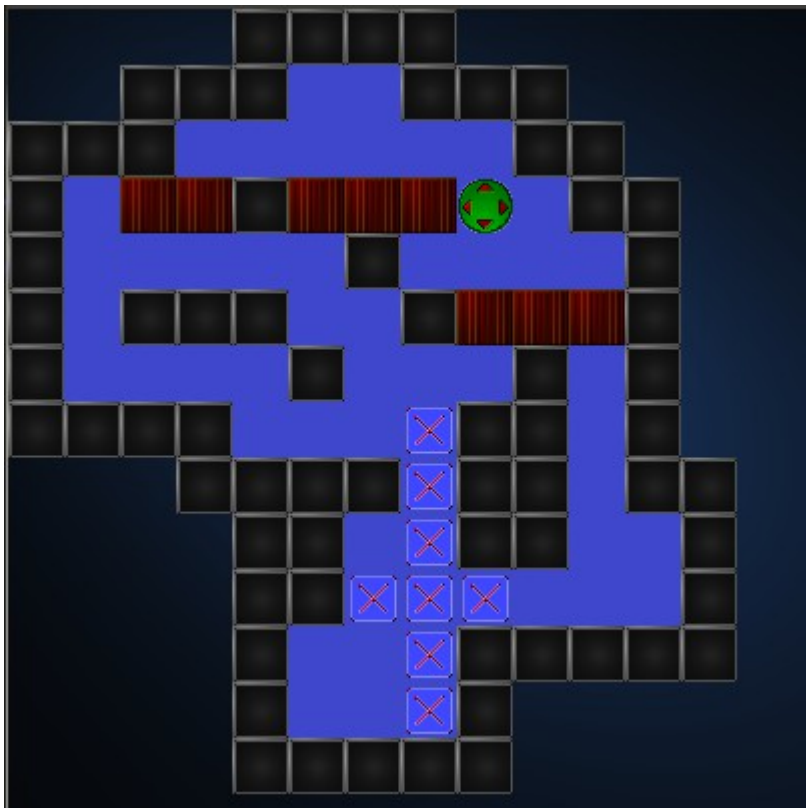
Sokoban

AI – Final Project

Prof. Jeff Rosenschein
Mr. Yoad Lewenberg

By: Jamal Mashal



Contents

1.The Game Description....3
1.1 Overview....3
1.2 Rules....3
1.3 Scientific research....3
1.4 Challenges in the game....3
2.Algorithms....4
2.1 Represent the game as search problem....4
2.2 Overview....4
2.3 Search with A*....5
2.4 A* the first version....6
2.5 A* the second version....7
2.5.1 sub-A*....7
2.5.2 main-A*....7
3. Heuristics....8
3.1 Zero Heuristic....8
3.2 Manhattan Distance: Boxes, X's....8
3.3 Manhattan Distance: Player, Boxes....8
3.4 Complex Heuristic....8
3.5 Heuristic that I didn't implement....8
4. Comparison: A* vs A*....8
4.1 Zero Heuristic....9
4.2 Manhattan Distance: Boxes, X's....10
4.3 Manhattan Distance: Player, Boxes....11
4.4 Complex Heuristic....12
5. Conclusion....13
6. Appendix....14

1.The Game Description

1.1 Overview

Sokoban (warehouse keeper) is a type of transport puzzle, in which the player pushes boxes or crates around in a warehouse, trying to get them to storage locations. The puzzle is usually implemented as a video game.

Sokoban was created in 1981 by Hiroyuki Imabayashi, and published in 1982 by Thinking Rabbit, a software house based in Takarazuka, Japan.

1.2 Rules

The game is played on a square grid, where each square is a floor or a wall. Some floor squares are marked as storage locations, and some of them have boxes.

The player is confined to the grid, and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The puzzle is solved when all boxes are at storage locations.

1.3 Scientific research

Sokoban can be studied using the theory of computational complexity. The problem of solving Sokoban puzzles was proven to be NP-hard. Further work showed that it was significantly more difficult than NP problems; it is PSPACE-complete. This is also interesting for artificial intelligence researchers, because solving Sokoban can be compared to designing a robot which moves boxes in a warehouse.

Sokoban is difficult not only due to its branching factor (which is comparable to chess), but also its enormous search tree depth; some levels require more than 1000 "pushes". Skilled human players rely mostly on heuristics; they are usually able to quickly discard futile or redundant lines of play, and recognize patterns and subgoals, drastically cutting down on the amount of search.

Some Sokoban puzzles can be solved automatically by using a single-agent search algorithm, such as IDA*, enhanced by several techniques which make use of domain-specific knowledge. The more complex Sokoban levels are, however, out of reach even for the best automated solvers.

1.4 Challenges in the game

- **The size of states domain**: the player and the boxes could be in any legal floor in the grid, thus the domain's size grows exponentially.
- **Number of steps in solution**: the optimal solution for a level, may be tens or even hundreds of steps.
- **Deadlock situations**: in some cases there would be box or boxes which hasn't any possible series of action to bring it to it's target. Identifying these cases may be difficult and leads to many unnecessary operations to identify the situation.

2.Algorithms

2.1 Represent the game as search problem

Formally a search problems defined as <**S**tates, **I**nitial state, **G**oal states, **A**ctions, **T**ransition Function> in our case:

- **State**: represented as the places of boxes and player over grid, each of them can be in legal floor square.
- **Initial state**: a state where the boxes and the player in their initial positions.
- **Goal state**: a state where all boxes are at storage locations.
- **Actions**: all possible actions that can be applied over state, in this game to move (top, down, left, right) or subset of these actions (not moving into wall).
- **Solution**: series of actions that leads the initial state to goal state.
- **Solvable initial state**: there is series of actions for each box to move it to storage location and there enough storage locations for all boxes.
- **Transition Function**: the cost of each actions is 1, the player make one step each time in any direction.

2.2 Overview

In this project I tried to solve the Sokoban game using A* and 2A* algorithms. Then compare the results for each of them using different heuristics. And here explanation of the formal expressions I used:

- $C: V \times V \rightarrow R, C(v_1, v_2) =$ The cost to move from v_1 to v_2
- $g(n) = C(root, v) =$ The real cost to reach the state v from the initial state
- $h: V \rightarrow R, h(v) =$ the heuristic value for state v. Which is the predicted cost to reach the goal state form the state v.
- $f: V \rightarrow R, f(v) = g(v) + h(v)$

*2.3 Search with A**

The Algorithm A* is BFS search algorithm that uses f to choose the next high priority state to check from the unvisited nodes. Here it's pseudo-code:

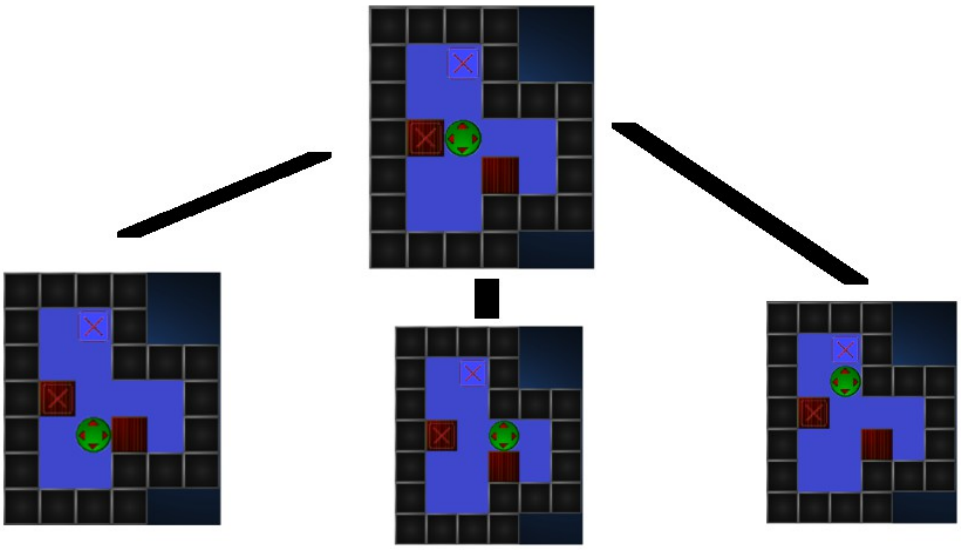
```
Astar(root) → solution


Astar(node):
    visited = set()
    toCheck = PriorityQueue()
    toCheck.push(node, 0)
    while toCheck not empty:
        currentNode = toChech.pop()
        if currentNode in visited:
            continue
        if currentNode is goal state:
            return solution
        visited.add(currentNode)
        for each node n in currentNode.expand():
            toCheck.push(n, f(n))
```

I implement the algorithm as graph search, that doesn't explore visited states to avoid loops because in this game it's possible to reach the same state with different ways. In graph search, A* is optimal if the heuristic is consistent and then it's also admissible.

The disadvantage of A* is the amount of memory it requires. In the search time it has to store in priority queue all the states it would explore, and this point is more critical in our case because as mentioned before in section (1.4) the states domain's size grows exponentially, and this would slow the speed of the search and may make it impossible if we have small memory. Thus I tried to implement a second version of the algorithm to overcome this problem, by dividing the problem to two sub-problems. The main difference is in the expanding step as we'll see later. First I'll present the original version of the algorithm, and then the second version followed by a comparison between both of them.

2.4 A the first version*

In this version of the algorithm, the original one, nothing special. The algorithm expands each state to get the successors which are the new states caused by the movement of the player.

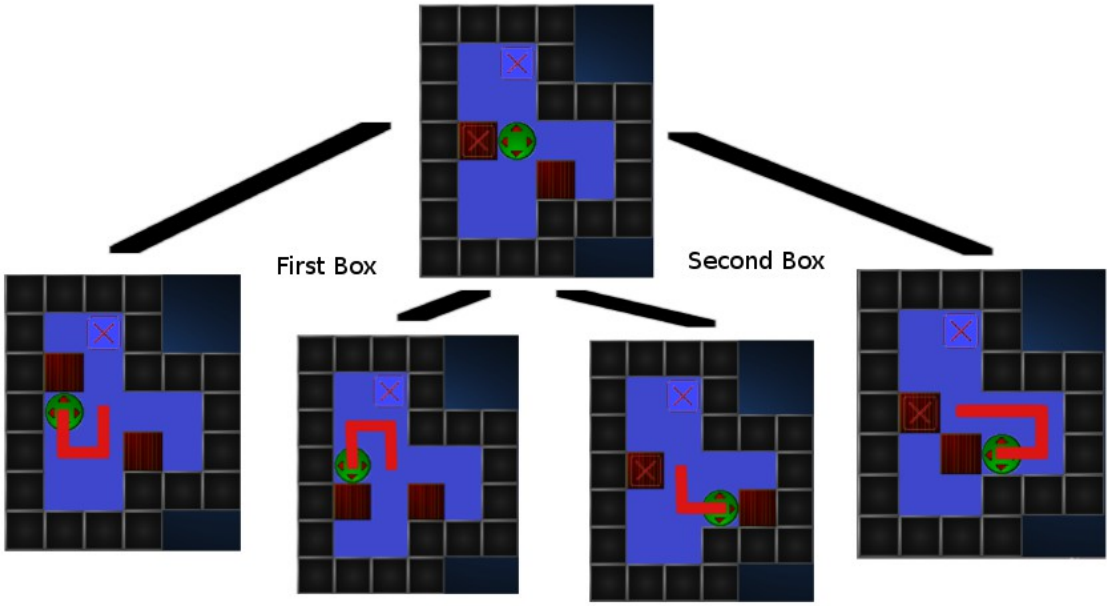


In this case the algorithm stores too many states, one state for each move the player do, so to check the movement of one box we have to visit and store all the states (moves) between the initial state and the new state after moving the box.

In other words, the search domain here is all possible ways to place the player and the boxes on the grid (no counting the same state more than one time if we can get there in different ways), so the domain size is: $\binom{N}{K} * N - K$ where N in the floor squares and K is the boxes number.

2.5 A the second version*

In this version of the algorithm, I divided the problem to two sub-problems. In the main algorithm instead of search over every possible move the player could do, it searches just over the possible ways that the player could move the boxes with. In this case I skip all irrelevant player's moves on the grid. In the sub-problem which is **getLegalActions(Current State)**, returns for each state it's successors. In other words it returns all the **minimum** series of actions for each box to move it one square in all possible directions (each box has maximum 4 different series like this to move it from it's current square) . Now instead of storing all different states caused by the player movements along the path, the sub-problem store temporarily these state to get the series of actions to move each box.



2.5.1 sub-A*

This sub-algorithm gets the current player position and the boxes positions from the main algorithm. And search for the minimal series of actions (player moves) to move a box in a certain direction, this algorithm uses Manhattan distance as heuristic. And he domain here is all possible move the player could do without moving any box .The image above shows the result of this part of the algorithm.

2.5.2 main-A*

This is the main algorithm which call the sub-A* for each state to get the successors and and explore them according to he given heuristic, which will be discussed next. Here the domain is all possible ways to move the boxes form there current position in all possible directions. The sub-A* run 4 times to get the results above, then the main-A* run over them to look for goal state.

3. Heuristics

3.1 Zero Heuristic

This heuristic always return 0, actually running A* with this heuristic is equivalent to BFS. But in case of my edited A* it's not, it's like running BFS just over the possible moves of the boxes, not the player.

3.2 Manhattan Distance: Boxes, X's

A basic heuristic which returns the minimal sum of distances between boxes and storage locations. This heuristic is consistent. It's also a lower bound to the numbers of moves to solve the problem.

3.3 Manhattan Distance: Player, Box

A basic heuristic which returns the minimal distance between player and any box. This heuristic is also consistent.

3.4 Complex Heuristic

This heuristic extend the the second heuristic (3.2), it tries to check more complex situations, particularly the states which would not lead to goal state and tries to give them very low priority.

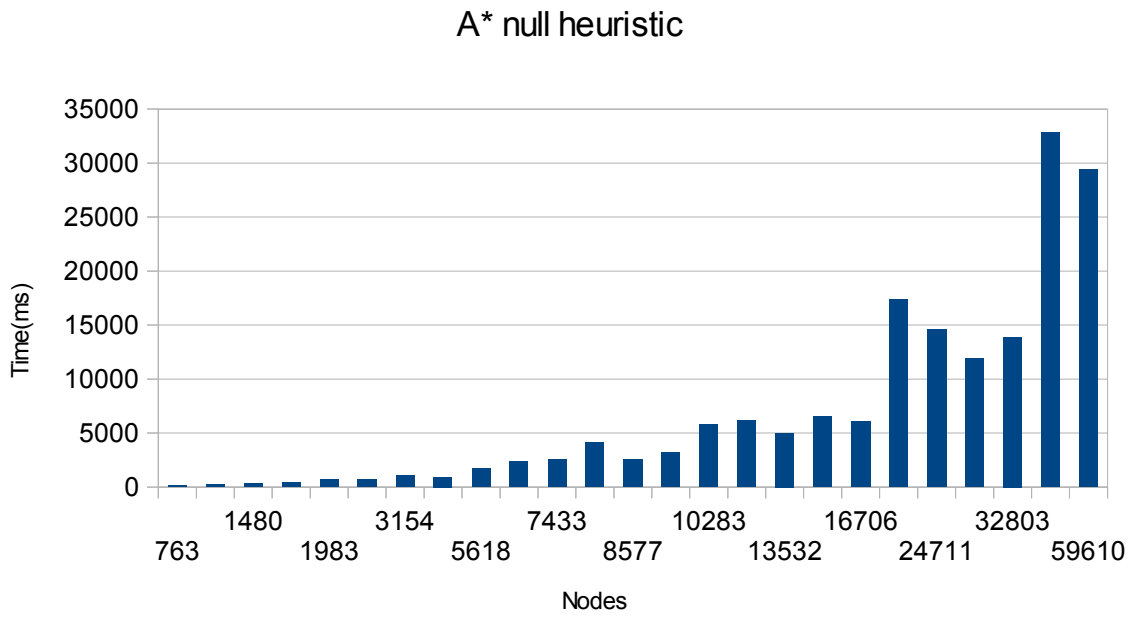
3.5 Heuristic that I didn't implement

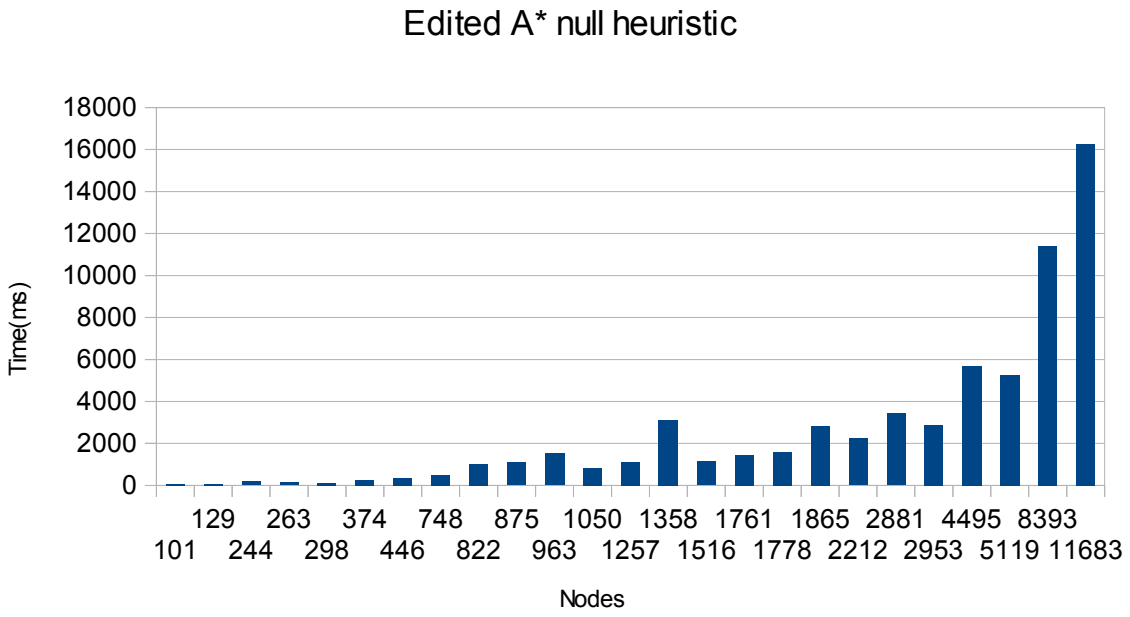
A heuristic smeller to the second heuristic but computing the real distances taking in accounts the wall squares in the way, this heuristic is more informative but also more expensive, because it's take $O(K^2 * A)$ where k is the boxes number and A is the complexity of the algorithm to find the minimal distance. It's trade off between information and time.

4. Comparison: A* vs A*

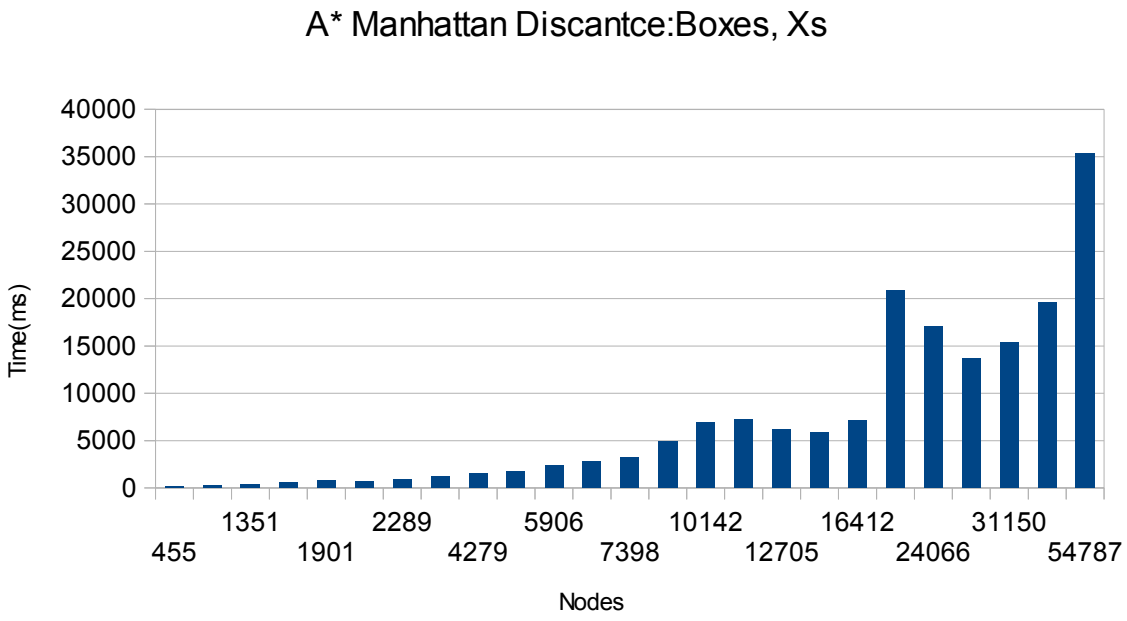
The original game has 11 different levels sets each has between 50-160 level with different sizes and difficulty, I chose to run the algorithms with the 4 heuristics over the first 25 level of the “Microban II” set. It was enough to get the conclusion. So it's $25 * 2 * 4 = 200$ running times to compare the algorithms and heuristics. As a result of the consistency of the heuristics all of them solves the levels with the same number of steps which is the optimal solution, so the main factors here were the number of explored nodes and the running time.

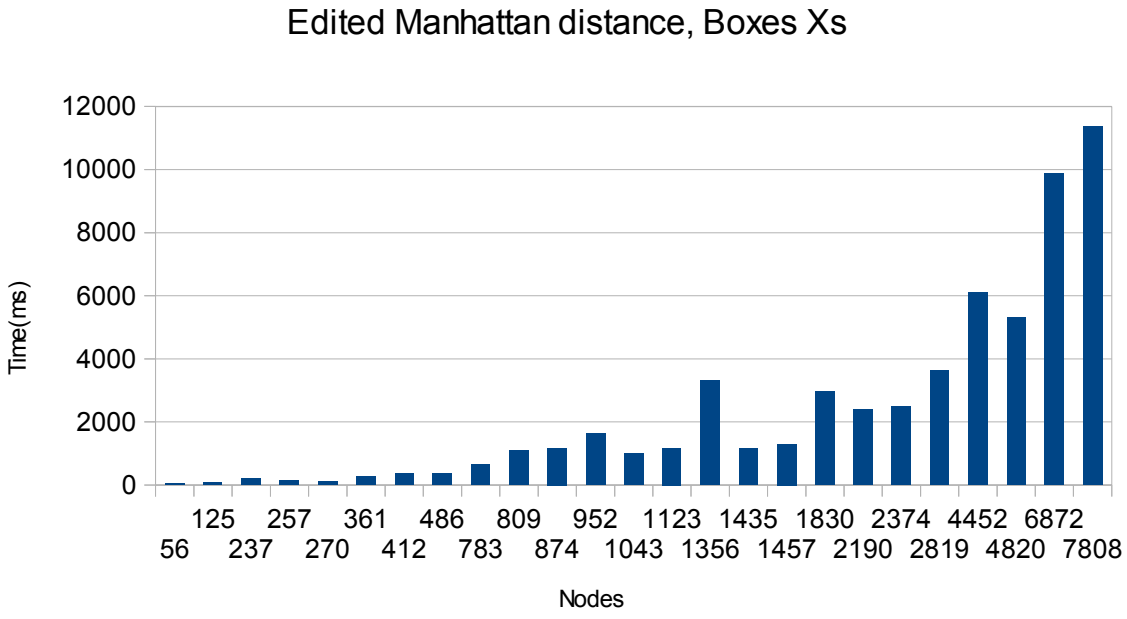
4.1 Zero Heuristic



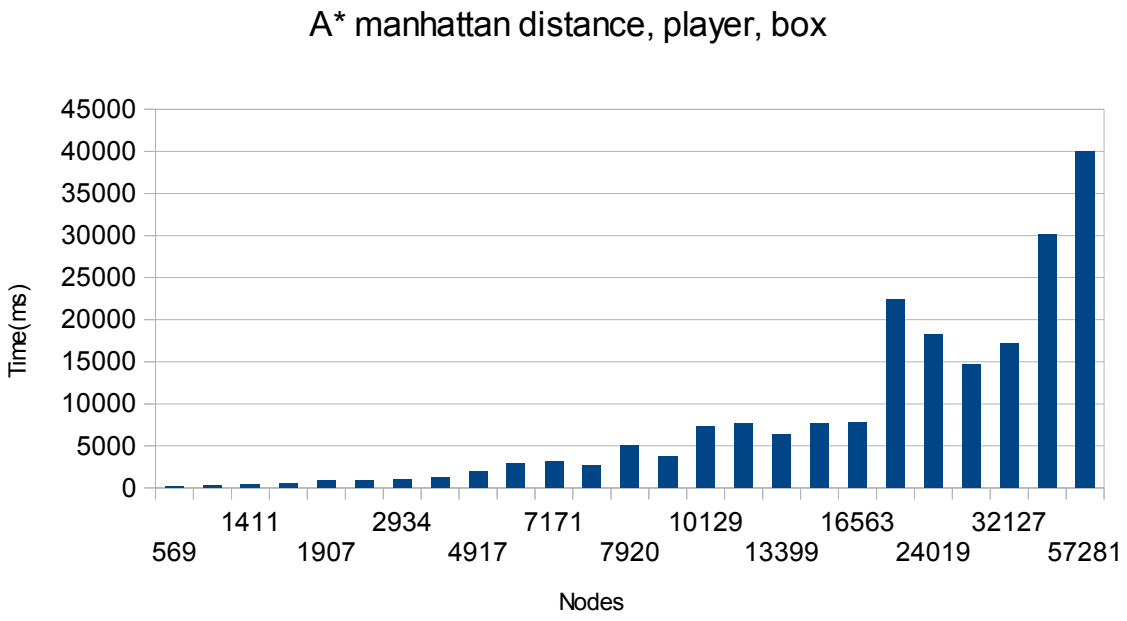


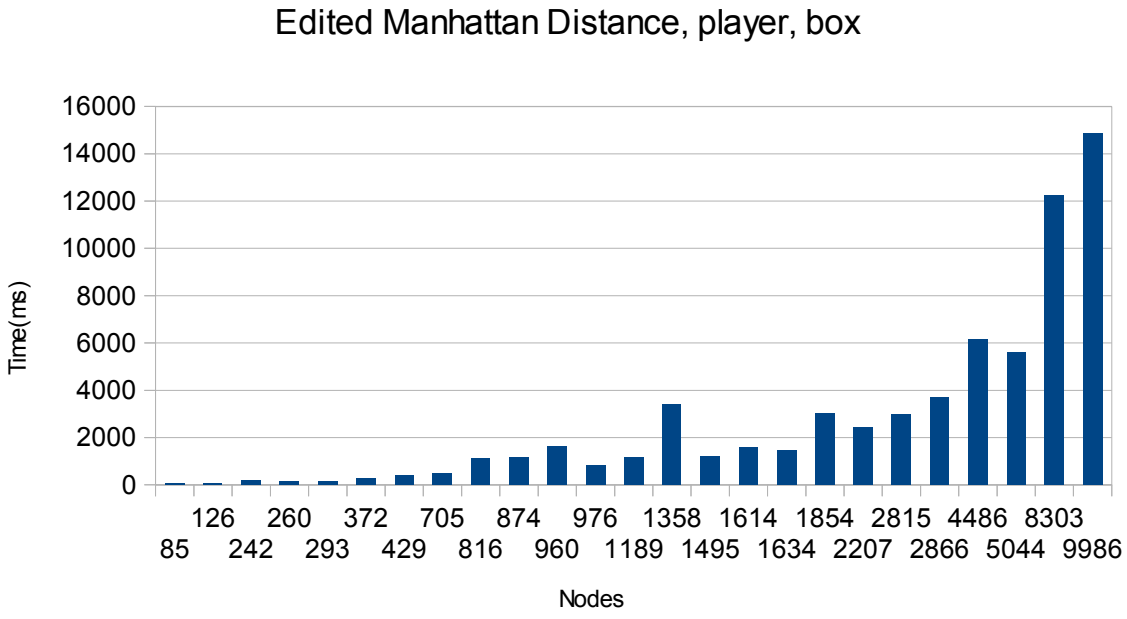
4.2 Manhattan Distance: Boxes, X's



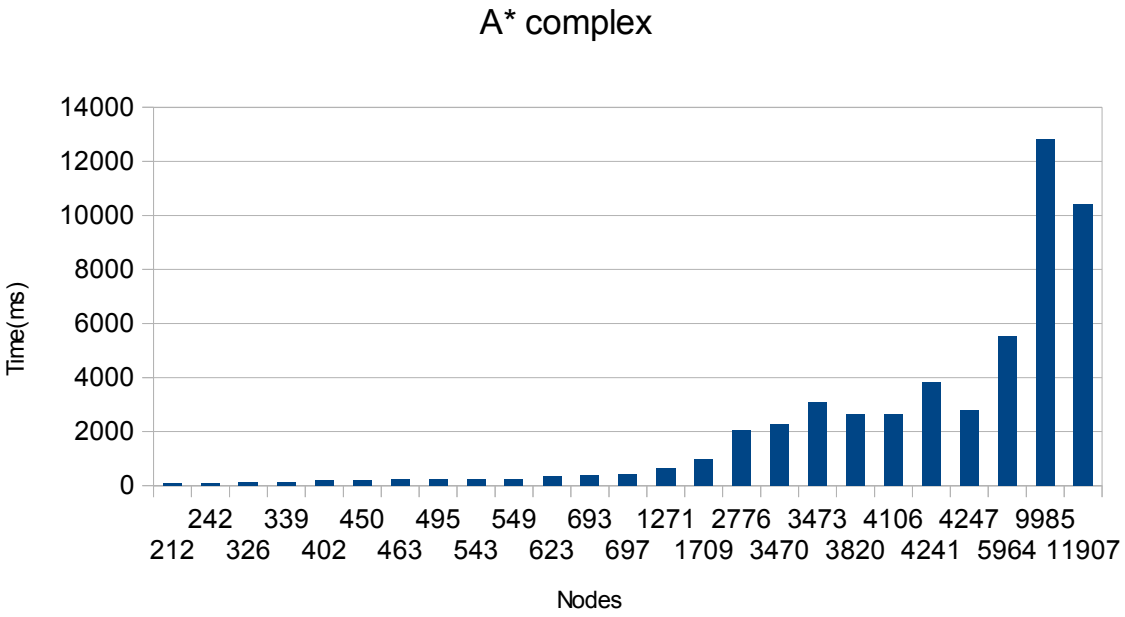


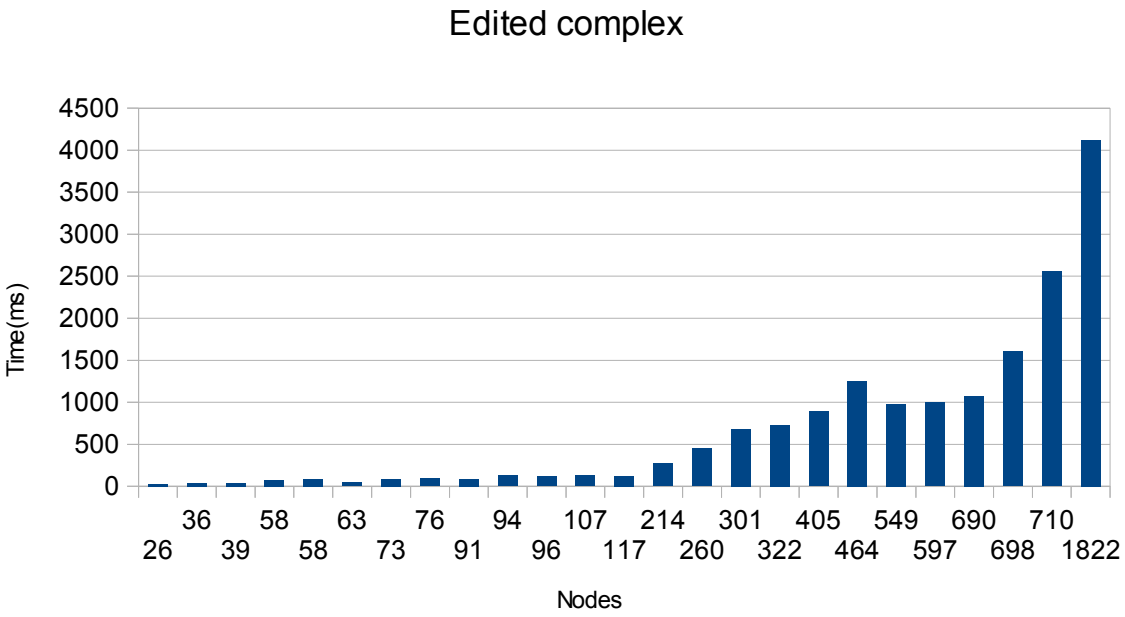
4.3 Manhattan Distance: Player, Boxes





4.4 Complex Heuristic





5. Conclusion

As the result shows, the the edited A* is faster (less time) and consume less memory (less nodes), as mentioned before the new version of the algorithm consume less memory because of the dividing to the problem to two sub-problems where in each iteration it hold sub-tree of the original problem to compute the minimal ways to move the boxes and then drop that sub-tree, thus it holds less nodes along the path.

By comparing the heuristics it's obvious that the complex heuristic gives the best results in both algorithms, because it skips a very very big part of the tree search that can lead to goal state.

6. Appendix

Null Heuristic:

Original: python gui.py -mod=ai -algo=astar -heu=nullHeu

Edited: python gui.py -mod=ai -algo=eastar -heu=nullHeu

Manhattan Distance Boxes, Xs

Original: python gui.py -mod=ai -algo=astar -heu=mdbx

Edited: python gui.py -mod=ai -algo=eastar -heu=mdbx

Manhattan Distance Player, Box

Original: python gui.py -mod=ai -algo=astar -heu=mdpb

Edited: python gui.py -mod=ai -algo=eastar -heu=mdpb

Complex

Original: python gui.py -mod=ai -algo=astar -heu=cmplxbx

Edited: python gui.py -mod=ai -algo=eastar -heu=cmplxbx