Intelligent Search - Motion Planning in a Warehouse: Report

Authors: Jamie Martin, Tolga Pasin

Introduction

Today, the use of intelligent agents is becoming more popular as the world increasingly moves towards replacing mundane tasks with fully automated systems. This report discusses our solution to a simulation in which an agent organises boxes within a digital representation of a warehouse. The forthcoming sections will discuss the state representations, heuristics, performance and limitations of said program.

State Representations

Our Sokoban Solver program uses a search algorithm to generate a list of directions based on the puzzle it is provided. There are three ways this can be computed which can be invoked from the following functions: ‘solve_sokoban_elem’, ‘solve_sokoban_macro’ and ‘solve_weighted_sokoban_elem’.

Each function passes in an object of the class ‘SokobanPuzzle’ into the ‘astar_graph_search’ algorithm from the supplied ‘search.py’ file. The best path from the agent to the goal is returned. The heuristic is shared between the functions as it is defined inside the ‘SokobanPuzzle’ class.

The puzzle is defined as an object of the class ‘Warehouse’ from ‘sokoban.py’ containing the properties; ‘boxes’, ‘targets’, ‘walls’, ‘worker’,’ ncols’ and ‘nrows’. ‘worker’ is a tuple displaying the x and y coordinate of the agent. The number of columns and rows are represented as the integers ‘ncols’ and ‘nrows’ respectively. While the rest of the properties are lists of x and y coordinates of the boxes, walls and targets within the warehouse.

Throughout the program it is necessary to convert the warehouse object to a two-dimensional matrix of chars and from a matrix to a string. This is achieved with the auxiliary functions; ‘warehouse_to_matrix’ and ‘matrix_to_string’. The matrix representation is used in order to check the value of a specific cell within the warehouse. Thus, ‘matrix_to_string’ is necessary to return it to a format that can be used elsewhere. These functions are used as we have found that it is more efficient to work with primitive data types than class objects.

Within our program we have created a subclass that inherits the class ‘Problem’ from ‘search.py’ called ‘PathProblem’. This is used in the ‘can_go_there’ method to search for paths from the agent to its goal. ‘PathProblem’ takes the parameters ‘self’, ‘warehouse’ and ‘goal’ on initialisation where ‘warehouse’ is an object of the ‘sokoban.Warehouse’ class and ‘goal’ is a tuple marking the x and y coordinate of the agent’s goal location.

Within ‘PathProblem’ the variable ‘state’ is mentioned various times. This is simply a tuple that holds an x and y coordinate of the agents position within the warehouse which is initialised to ‘self.initial’.

‘SokobanPuzzle’ is also a subclass of ‘search.Problem’ so that it can be passed into the ‘astar_graph_search’ function. On initialisation it takes ‘self’, an object of the class ‘sokoban.Warehouse’ and the optional boolean arguments ‘macro’, ‘allow_taboo_push’ and ‘push_costs’.

The variable ‘state’ is also used within this class, but refers to a tuple where the first element is ‘warehouse.worker’ and the second is a frozenset of ‘box’ and ‘cost’ for each box in the warehouse. The collection used needed to be hashable for the explored hashset used in the A* algorithm, to uniquely track each box. Frozenset was chosen as lists and sets are mutable and therefore unhashable. The frozenset is converted to a set when it needs to be worked with and then refrozen when returned to the A* algorithm.

This class is used by ‘search.py’ to find all allowable actions for the agent, check if the goal state has been accomplished, find the heuristic, find the cost of a given path and find the result of an action. ‘macro’ and ‘allow_taboo_push’ are set to ‘False’ and ‘push_costs’ is set to ‘None’ by default so that the necessary code is only computed when they have meaningful values. They can easily be set during initialisation, when necessary.

Heuristics

The heuristic employed within our program is the Manhattan Distance. This is calculated by summing the absolute horizontal and vertical lengths from the agent to the goal. Formulated as,

$$h(i,\ e)\ = \left| e_x\ -\ i_x \right| + \left| e_y\ -\ i_y \right|$$

where i is the x and y coordinates of the agent’s initial state and e is the coordinates of the goal state (Munoz, Bouchereau, Vargas, and Enriquez, 2009).

The Manhattan Distance was chosen because of the nature of the agent’s movement. Since, the agent is confined to equally spaced two dimensional squares, it can only move up, down, left or right. The Manhattan Distance is often used to represent the distance accrued by navigating through city blocks or to show the distance moved by a rook in chess. Therefore, it is more appropriate to use opposed to something like Euclidean Distance which is calculated as a straight line from the object to the goal (Café Scientifique, 2016).

In our program the heuristic is made up of the distance from the agent to the box and the distance from each box to it’s target square, which is multiplied by the push cost if it exists. For each box the heuristic from the agent is calculated and appended to a set. The same was done for each box to target square to find the total distance between all boxes and targets, while considering the relevant push costs. This included each permutation of box to target distances. The minimum values from each set are added together to get the final value. The minimum values ensure that the A* algorithm gets an accurate estimation of the shortest path which enables it to favour paths that move the agent closer to boxes and push boxes closer to targets while considering the cost of each push. When the push cost is not supplied, it is set to one so that it has no effect on the total distance.

Performance

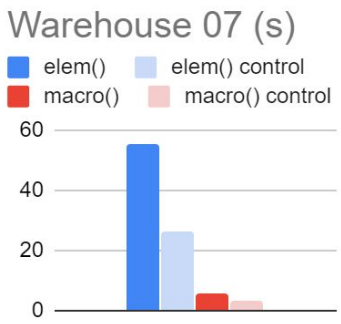

Figure 1

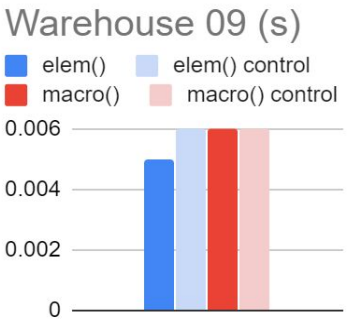

Figure 2

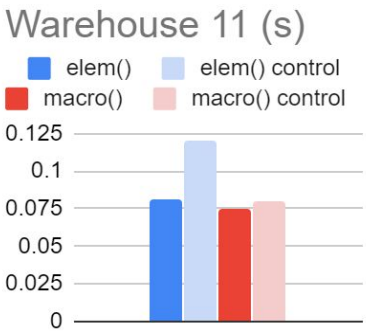

Figure 3

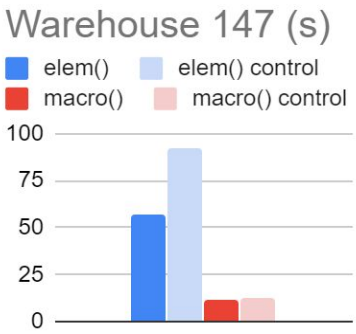

Figure 4

Figures 1 - 4 show the time taken for elementary and macro functions to find solutions. Control refers to tests done with Frederic Maire's implementation of the program (which were uploaded to Blackboard) while the others were tested on a Intel i7-8565U CPU. Times for the weighted function were not included as they were nearly identical to the elementary times.

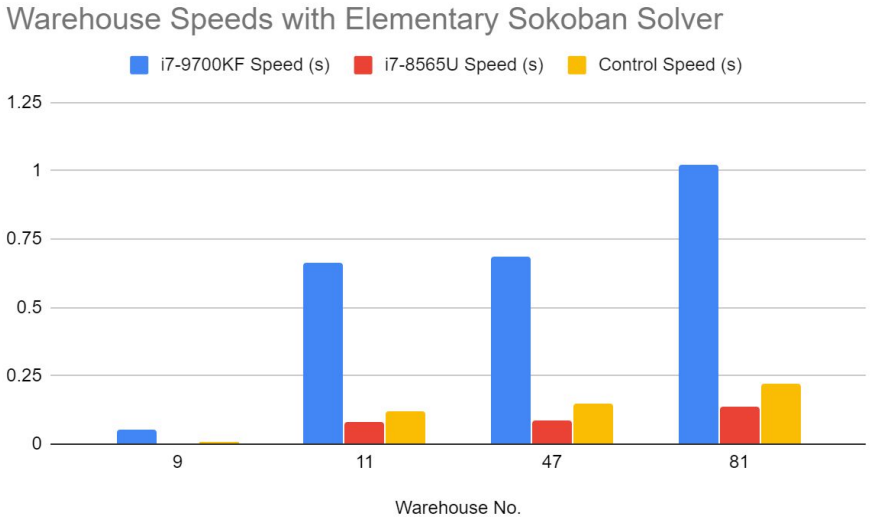

Figure 5

Figure 5 shows the difference in time to compute solutions to an array of warehouses between various CPUs.

Limitations

It was found that the performance of our program depends largely on the specifications of the computer it is run on. The speed of finding a solution can fluctuate based on the clock speed and the dependency on multi-core processing of the CPU.

While testing, a large difference in time was found when the program was run on different machines (seen in figure 5). Initially, this was unexpected because the slower CPU (i7-9700KF) is higher-performing in most other scenarios.

However, it is hypothesized that the downfall of the i7-9700KF is its reliance on the use of multiple cores to achieve a higher clock speed (as it has eight and the i7-8565U CPU has four) (Intel, n.d.). This is a problem because the C implementation of Python has a global interpreter lock (GIL) that blocks multiple threads from running at the same time. The interpreter can cause a function call by two threads to take double the amount of time of a single thread being called twice (Wouters, 2017). Which supports the results that were found.

A solution to this problem would be implementing the multiprocessing Python library. This library gets around the issues of the GIL by using multiple interpreters to run the processes in parallel (Quark Gluon, n.d.). However, this is outside the scope of the given assignment.

Conclusion

In summary, we have created a program that can solve the Sokoban warehouse puzzle. It can find a solution using the elementary, macro and weighted functions within similar timeframes to the times supplied as ‘control’ readings (where some are faster and some are slower). It uses the Manhattan Distance as its heuristic to estimate the ‘jagged-line’ distance from the agent to the goal. Though, due to the limitations of Python’s interpreter the program will show worse results when run on a computer that relies on multi-core processing to achieve its maximum speed.

Bibliography

- Munoz, D., Bouchereau, F., Vargas, C. and Enriquez, R. (2009). Heuristic Approaches to the Position Location Problem. Position Location Techniques and Applications. Academic Press. https://www.sciencedirect.com/topics/computer-science/manhattan-distance
- Café Scientifique. (2016). Distances in Classification. http://www.ieee.ma/uaesb/pdf/distances-in-classification.pdf
- Intel. (n.d.). Intel® Core™ i7-9700KF Processor. https://ark.intel.com/content/www/us/en/ark/products/190885/intel-core-i7-9700kf-processor-12m-cache-up-to-4-90-ghz.html
- Intel. (n.d.). Intel® Core™ i7-8565U Processor. https://ark.intel.com/content/www/us/en/ark/products/149091/intel-core-i7-8565u-processor-8m-cache-up-to-4-60-ghz.html
- Wouters, T. (2017). Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock
- Quark Gluon. (n.d.). Parallelising Python with Threading and Multiprocessing. https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing/