Sokoban Assignment

Assessment 1 of CAB320

Group Members

- Zeyu Xia (n11398299)
- Tian-Ching Lan (n11262141)

Design

In our process of designing the search algorithm and its implementation, we first came up with a question that how to effectively represent the game itself and the state which is changing all the time. After having a manually simulation of the game, we found that everything except the worker and boxes are static. So, that becomes our idea of designing the state. We also considered how to represent the moves in an effective way. It is intuitive that, the worker can be the agent, and then the worker has four moves: left, right, up, and down. However, this representation has some drawbacks: the worker may perform useless actions, such as move forward and then move back, or move as a loop, which consumes the cost, but is not toward the solution. Though the search algorithm themselves can drop some of the duplicated states, the search space still contains a lot of these wasted actions.

Later, after carefully analyzing, we found that the agent representation can be changed from the worker to the state of boxes. We found that, an action sequence of worker can always be converted to the movement sequences of boxes. For example, after several steps, the worker pushes one of the boxes forward. Then, it can be represented just by the movement of the box. Then, the movement of the worker can be calculated as using the shortest path algorithm towards the downside of the box, As the beginning state (the worker is somewhere in the warehouse) and the final state(the worker is at the box position, and the box is pushed one cell forward) are the same, and the path for a worker towards the final state is the shortest, we can prove that this representation is equal to the worker's sequence, and cost of this representation is minimal. Also, the solution is in the search space, as all the effective actions exist only when the boxes are pushed. By using the box representation, the search depth is significantly lower than the worker presentation, resulting in shorter running times and lower complexity. However, when we need to validate actions, the box representation becomes annoying. So, we kept both representations in our code. The box representation is call *SokobanPuzzle* for grading purposes. And the worker representation is called *SokobanPuzzleWorker*.

After figuring out the representation, we wrote some auxiliary functions. We found the data structure of Python a little bit tricky, especially the immutable tuple and mutable array. As the assignment statement in Python is only to pass the reference of some variable, and if the data type is array, then weird things happen when the array is modified inside of a sub-function, its original value will also change. To solve this problem, we import the copy library to do shallow copy of the array. It is worth noting that as the tuple is immutable, we can use normal assign statement to pass the value of it. And we don't have to use the very slow deep copy method.

We also spent some time checking the implementation of the *Node* class. We found that it contains the *state* but does not contain the *Warehouse* or *Problem*. In this way, when debugging, we can only build our own function to calculate the heuristic value of each node.

Our initial plan was to build a BFS implementation first to see if our codes are correct, and if our design can produce the result. However, after successfully developed the BFS algorithm, we found that the result is wrong for *warehouse_8a*. However, we can put boxes into correct locations with a higher cost. After some research, we found that although BFS can search the search space completely and find a solution, it is not optimal when the weights are introduced. In this case, Dijkstra, or A* is required to get the optimal solution.

And during the process of design our algorithm, we tried a lot of algorithms, and found that for each, we need to balance time efficiency and correctness well in most of the cases. For example, we want to estimate the h-value of boxes using the Dijkstra algorithm. Although we can have precise estimations, it takes a lot of time, and sometime when a box is blocked, it generates infinity, which is difficult to use.

State

We designed to classes for the problem: The *SokobanPuzzle* class extended from the *Problem* class is used to store static information such as the walls, the shape of the warehouse, taboo cells, and the weights of boxes. The *State* is used to store dynamic information in this problem, which is the position of the worker, and the location of each box. The static things, stored in the *SokobanPuzzle* class should be the problem instance, which the dynamic things, stored in the *State* class should go into the problem instance, as it will be copied every time the algorithm is discovery new nodes in the search space. This is an important way to save memory, and in our experiment, out of memory do exist when there's too many potential states. It is quite easy to distinguish what is static and what is dynamic, just trace all the variables to see what is changing.

For usage, most operations in the Problem class are driven by State. For example, to evaluate valid moves, we need both the wall positions and worker position, which are stored in the State object. Additionally, the Heuristics used to evaluate the priority of each case are based on State.

Since a State only represents the current situation, there is no difference between the State of the box-driven model and the State of the worker-driven model.

Heuristics

Heuristics refer to functions that estimate the cost of the cheapest path from a given node to the goal node. When the estimated cost is less or equal to the actual cost ($0 \leq h(n) \leq h^{\star}(n)$), we call it an admissible heuristic, and we achieved optimality. Also, our heuristic achieved consistency by ensuring $h(n) - h(m) \leq cost(n\ to\ m)$.

In our code, we implemented several combinations of heuristic algorithms. Finally, we choose to use box with a *nearest assign* algorithm, and a *Manhattan* distance algorithm. In our design, we first use a method to **assign each box to a target**, using what we called the "assign algorithm"; then, we calculate the distance of the box to its corresponding target, using **distance algorithm**. At last, we multiple $(1 + weight)$ (in case the box does not have a weight) to the calculated distance, sum all of them, and divide it by the number of boxes. As a comparison, we also added *uniform_cost_search* function, which always return 0, making $f(n) = g(n)$.

In our bag of **assign algorithm**, the *nearest* algorithm calculates the distance of a box to its closest target. In this algorithm, it is possible that multiple boxes are assigned to single target. This may cause some problem, but still, the heuristic is admissible, and it is very fast to calculate. Another assign algorithm is *permutation*. In this algorithm, we calculate every possible combination of the boxes and targets. Then, we calculate the cost of each combination, and choose the one with smallest total cost. This ensures that each target is only assign to one box, however, it takes a lot of time to calculate all the values. And it takes a lot of memory, which in our cases, takes all 16GB of memory and results of an out-of-memory kill. The last assign algorithm is Hungarian, which is an effective combinatorial optimization algorithm that solves the problem in polynomial time. In this method, we calculate the distance of each box and each target, generating a *cost_matrix*, then, the assignment is generated from the *cost_matrix*.

In our bag of distance algorithms, we have two: *Manhattan* distance and *Dijkstra* distance. The Manhattan distance is like Euclidean distance, but in our case, as the worker and boxes are only able to move horizontally and vertically, it is more realistic than Euclidean distance. Also, it is easier to calculate, and it generates integer. The Dijkstra distance is the distance that considering the walls and other boxes. It uses BFS to calculate the actual path of the box to its assign target. If the worker can push the box all the way, it is the most accurate estimate of the distance. However, as it will calculate the whole map every time it is called, it is extremely slow and memory consuming.

Our Dijkstra algorithm have a complexity of $O(n^2)$. However, in other heuristic algorithms in our A*, the complexity is only $O(n \log n)$ because of the *PriorityQueue*. There's a dilemma in designing the f function: if the weight of the heuristic evaluation h is too low relative to the path costs g, the algorithm may behave more like a greedy search, which becomes extremely slow. In other way, if the h is much higher than the path cost g, it will not always return the lowest cost solution.

"uniform" means that there are no heuristics involved to sort the order in the frontier list, which is equivalent to a Dijkstra algorithm. In this heuristic, the nodes will be explored in order of the cost. So, the lowest cost solution will always be found as the first solution. The disadvantage is that there is no pruning of any nodes, which leads to massively inefficient nodes being iterated, lowering the performance, especially in high-difficulty problems.

"Heuristics based on the nearest Manhattan distance" is a simple but effective approach to measure the possible cost for boxes to reach targets. Specifically, there are four steps: 1. Find Manhattan distances between a box and nearest the target. 2. Multiply the distance by the box weight. 3. Loop the above processes through all the boxes. The advantage of this heuristic is that it provides a general idea of what move makes the box closer to the targets, so the operation that explores the node that pushes boxes away would be prevented, increasing performance. The disadvantage is that it will tend to push the boxes to the same position of the a targets and then start Dijkstra after one arrives, which lowers its effectiveness in the last steps.

Figure 1 Our modified taboo test

To improve the weakness of the above algorithm, "heuristics based on the Manhattan distance of each paired target and box" was implemented. Compared to simply find the nearest target, in this algorithm, we use the Hungarian algorithm to find the lowest cost from each combination of targets and boxes. The advantage of this heuristic is that it is proven to be admissible since the Manhattan distance is the smallest step of the boxes. Besides, by assigning a box to its best-fit target, the case where the box goes to the middle of two targets will be prevented. Still, there is a weakness in this algorithm in that the Manhattan distance is not the real distance, so in this heuristic, it might encourage boxes to be pushed towards targets is correct in lowest cost solution.'

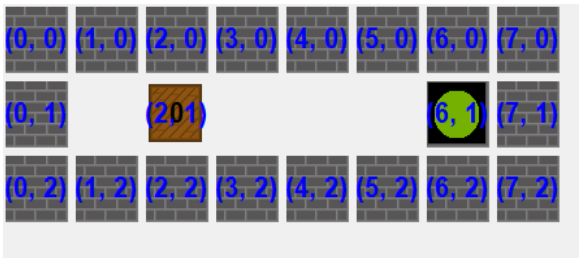

Figure 2 The corridor taboo test case

The "Dijkstra heuristics" is a version that uses Dijkstra's algorithm to calculate the distance between boxes and targets instead of the Manhattan distance. The structure is like the above two heuristics, combining the features of both the nearest and paired heuristics, but with more accurate cost predictions. With a better evaluation of the costs from the current state to the goal, the frontier list is sorted more accurately. However, the process of calculating the Dijkstra distance is far more complex than the Manhattan distance, so the calculation process will be much more time-consuming than the above two heuristics.

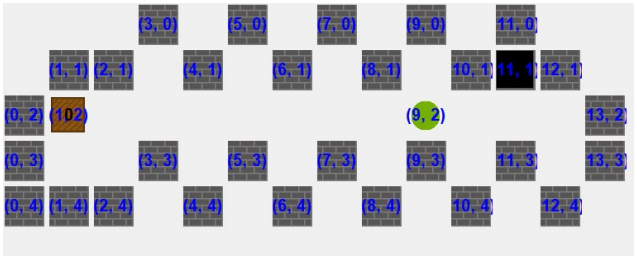

Figure 3 Another extreme taboo test case

Test

To validate the code, we modified the provided *sanity_check.py* to implement our own test cases automatically.

We created four novel test cases for taboo cell tests. Our first customed test case modifies the warehouse_01.txt to add a wall. In this way, we can detect if all conditions are satisfied when we are replacing the empty cells to a taboo cell. We found a bug that our original code will replace the wall into a taboo cell. In another test case, we created an extreme test case that the cells are surrounded by walls, but in an interlaced way. Our code is capable to detect and mark all the taboo cells in this test case. The third test cases we made is a straight corridor that have a target on one corner, and nothing on the other corner. This test case is a tricky one. We need the left corner to be a taboo cell, while the right corner, as well as the cells between shouldn't. And the last test case is like the previous one, but in a 2D square.

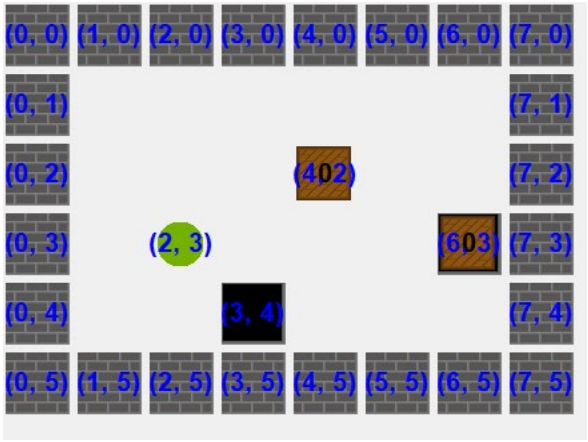

Figure 4 The square test case

To test the check_elem_action_seq() function, we made added a few test cases, that check if the sequence is legal or not. We didn't do a lot of modification to this test function.

We imported a lot of test cases from the Warehouse dataset when testing solve_weighted_sokoban() function. We also created a script to compare the cost and final state between our output and the expected output. If both answer is Impossible, then we can prove that our code is correct. If there's weight in the input warehouse, and following the given action sequences, we can find a different path with the same cost to put all boxes into their corresponding final location, then we can say our code is correct. When there's no weight, we just check if our code can put all boxes into the target location. During the testing, we found that there can be multiple paths in this case, and the only difference of the final state is the worker's position.

We also performed step-by-step debugging using Spyder during the development process. This helps us to verify the correctness individual functions.

Performance and Limitations

We are unable to design a simple yet universal heuristic function for this task. All the variant we designed are faster in some cases but gives wrong answer or significantly slower in other cases. We also tried to use Hungarian algorithm to assign boxes in the beginning, but it still takes a significant of time. At last, we choose to use box with a nearest assign algorithm, and a Manhattan distance algorithm. Our test passes 47 cases and achieved a good performance in case 7.

In general, all the heuristics have similar performance, but some of them are significantly lower. Especially, in worker-driven models, the performances are much lower than the box driven models while both return correct answers. Besides, while Dijkstra algorithm has higher accuracy compared to Manhattan distance, the resources it consumes slow down its performance by about 20%. In addition, the nearest cost search heuristics is also quicker than the paired cost search heuristics by 15%.

The overall performance is shown in the table below.

Worker/box	Search algorithm	Assign algorithm	Distance algorithm	Passed/Failed in 60 secs	Case 127	Case 7
Worker	BFS	N/A	N/A	35/73	Failed	Failed
Box	BFS	N/A	N/A	53/55	6.0625s	Failed
Worker	Uniform cost search	N/A	N/A	33/73	Failed	Failed
Worker	A*	Nearest	Manhattan	33/73	Failed	Failed
Box	Uniform cost search	N/A	N/A	47/61	9.0091s	211.6369s
Box	A*	Nearest	Manhattan	47/61	2.2946s	58.0195s
Box	A*	Permutation	Manhattan	45/63	8.2485s	247.5377s
Box	A*	Hungarian	Manhattan	47/61	9.6587s	250.4198s
Box	A*	Hungarian	Dijkstra	45/63	38.1777s	303.0625s