Sokoban

universidade de aveiro

Licenciatura em Engenharia Informática

UC 40846 - Inteligência Artificial

Eduardo Santos - 93107
Pedro Bastos - 93150

Search Algorithm

We chose to implement the **A* path-finder algorithm** because is one of the best techniques.

This algorithm can be **optimal** if used in with the **right heuristic function**, and compared to other algorithms, **A*** is proven to expand the **minimal** number of paths when using the same heuristic.

Although the **A*** uses cost as the sum of the node cost with the estimated heuristic, we opted to use only the heuristic value as the key to sort the open nodes because it was faster. Therefore it is very similar to the **CBFS algorithm**.

This algorithm is used to calculate the pushes. To find the path from the keeper to the box of each push, we use a **Breadth First algorithm**.

Besides this, we also use a **backtrack** using a dictionary that saves the states already visited. With this, we significantly **reduced** the search time because only the unvisited nodes are expanded.

Heuristic

We decided to implement the **greedy** heuristic to sort the nodes in the queue. It is a fast heuristic to calculate and reduces the search time by quite a bit.

Instead of calculating the manhattan distance in each node, we **precalculate** the distances from each position to each goal right in the beginning of the level, using the goal pull metric. This metric consists in pulling a box of a goal to each position of the map. The distances are all saved, for each goal, to every position. That way we can later sort in each node heuristic by the **precalculated** distances, making it a **lot faster**.

The heuristic is a **crucial** factor to make the search **faster and more efficient**.

Deadlocks

- **Simple deadlocks**

  Because simple deadlock squares of a level never change during the gameplay, they can be **precalculated** at the beginning of the level. We just have to check if a box, in a certain position, can be pulled onto a goal. If not, it is a simple deadlock square. This verification results in a **huge performance improvement** and allowed us to get through levels without getting into a deadlock state. In brief, this algorithm identifies the squares from which a box can never be pushed onto a goal.

- **Freeze deadlocks**

  This type of deadlocks have this name because one or more boxes are **frozen** on specific squares, meaning that they can never be pushed away from those squares for the rest of the level, entering in a deadlock state.

  To detect this type of deadlock we need to check whether a box **can be** pushed or **not**. And before making a move, we have to check if after the move the boxes are frozen or not.

Results

With the implementations above mentioned, we managed to get to **level 118** with 3487 pushes and around **31400 steps**.

Levels **68**, **106** and **117** are the ones that took the most time solving. These levels were the main obstacles we had to overcome. We managed to get through **68** with the **backtrack** and **106** with the **precalculated distances metric**.

In terms of time, implementing the **heuristic**, the **precalculated distances** and the **backtrack** were the **major improvements**.

Conclusion

During the elaboration of this project, we enjoyed the friendly competition, because the other groups drove us to work harder in order to surpass other teams.

We discussed some issues with another team, consisted by:

- Francisca Barros - 93102
- Lucas Sousa - 93019
- Margarida Martins - 93169

Despite getting till level 118, we know our code isn't optimal. Has a future work, we could try another type of heuristic implementations, optimize the search of the keeper's path and reduce the number of expanded nodes on each depth. We could also implement an algorithm to deal with corral deadlocks and improve our backtracking solution.

References:

- http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks
- https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf