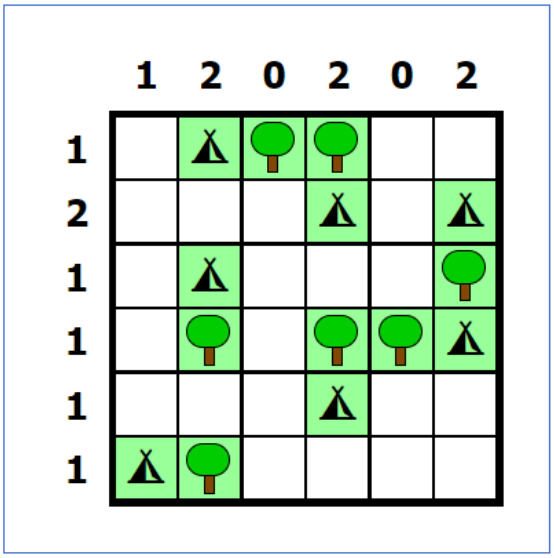
Trees And Tents

By Harsh Doshi,
Mtr. number 22203459.

Problem Description:



Trees and Tents is a logical, grid-based puzzle game with a few constraints.
The rules of the game are:

1. Pair each tree with a tent adjacent horizontally or vertically.
    a. This should be a 1 to 1 relation.
2. Tents should never be horizontally, vertically or diagonally adjacent to each other.
3. The clues outside the grid indicate the number of tents on that row/column.

[Image and rules taken from: Tents Puzzle]

We can approach the problem in 3 ways:

- Graph Search
- SAT and SMT
- Reinforcement Learning

Graph Search

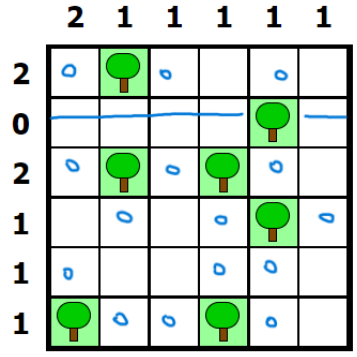
This problem is a good fit for graph search since we can think of this puzzle in terms of a start state, successors and a final goal state.

The start state contains no tents, the next states/successors contain tents placed in possible locations and the end state is when we have placed all the tents adjacent to trees without breaking any constraints.

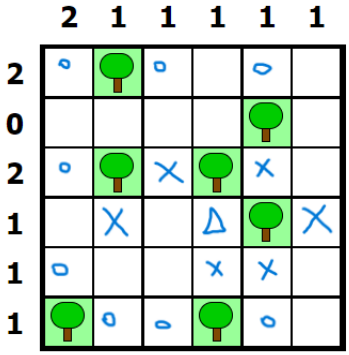
The algorithm takes a single tree and finds the legal positions for it, places tents in those and extrapolates from thereon.

Note:
Legal positions - Positions on the board where tents can be placed
Correct positions - A subset of legal positions, which contains the solution to the puzzle.

Placing a tent on the grid reduces the amount of legal positions available, this ensures that our graph doesn't end up running forever.



Pic 1: All legal positions
Total Positions Available: 15



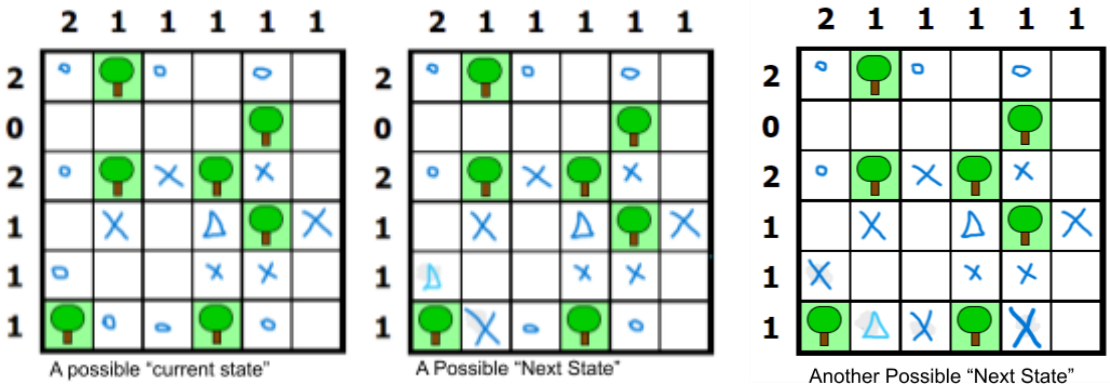
Pic 2: Eliminated positions after one placement
Total Positions Available Now: 8

A more concrete description of the solution:

1) The first step would be to declare a board of side N,
    a) Then we define the tree positions,
    b) The column and row restrictions.
    c) We create a list of tuples, where each tuple consists of tree positions and their corresponding tent positions.
        i) Initially, all of these tuples would have "None" as the tent position because we have not assigned any locations to our tents yet. This **list of tuples is our current state.**

2) Then we come up with a function called **check_positions** that checks whether a possible tent position is legal or not by checking that:
    a) There is no tent or tree already existing at that location
    b) The placement does not violate the row or column constraint (we keep track of how many tents have already been placed)
    c) The tent is not vertically, horizontally or diagonally adjacent to another tent,

3) Our **next_states** function goes to the first tree with an unassigned tent and finds out the legal tent positions for it using the **check_positions** function.
Then it creates **new states** by placing tents in those legal locations and returns them as a list of the next moves for our current state. We do this recursively.



4) We keep expanding these next states, If we run out of legal positions before all tents have been placed, this is a wrong solution and we backtrack from it.

5) Our **is_goal** function simply checks if all trees have a tent assigned to them, this is because we ensure that none of the constraints are broken during tent placement. Thus, if we have been able to place all tents, then we are at the solution.

6) We use the **DFS** algorithm to use these functions because we want to take a position and expand from it, we want to consider all possible locations for each tent while knowing about the existing placements. Importantly, there is **no shortest path** for this problem as the order of placing the tents does not matter to the final solution.

7) We can further optimize this approach by using **A*** search and implementing some **heuristics** such as:
    a) If a certain row/column has only X legal positions, and the number of tents in that row must also be X, then those are the correct positions and we place tents on them first.
    b) Trees that have only 1 valid tent position should be filled first.
We use these heuristics at every stage to reduce the number of possibilities and sort the next states such that we arrive at the solution quicker.

Overall, using graphs is a very intuitive approach to solving this puzzle, although it can consume lots of resources because we are considering every possibility.

The complexity of a DFS algorithm is calculated as O(V + E) where V is the number of vertices and E is the number of edges.

For a puzzle with N tents, we have ‘N’ vertices for our graph and for each vertex, we have, at max, 4 possible locations.

Total Edges = $_{Tree_1}4 * _{Tree_2}4 * \ldots * _{Tree_N}4$, which is 4^N edges in total.

Thus our worst-case complexity would be: $O(N + 4^N)$ which is $O(4^N)$.

Our complexity would be much less in an actual puzzle because most trees do not have 4 legal locations for tents, and because the total number of legal positions decreases with every placement.

Reinforcement Learning

While graphs are a **definite** way to solve the puzzle, they require the machine to go through all possibilities before arriving at the answer.
Reinforcement learning has the potential to solve puzzles without trying every combination.

The **environment or state** in this case would simply be the board, the new state being the board with the tent placed on it.

The **actions** are simply the next available legal locations after each tent placement.

Both of these can be carried over from the graph approach and modified, for example, we want to consider all of the tent placements we can make after we make a move instead of just the placements for the next tree.

Our biggest challenge is coming up with a policy that rewards/punishes the tent placements. We want to develop **relational reasoning** so that our bot can draw **logical** conclusions about the relationship between the column and row restrictions, and the tree locations.
[A similar approach was used here to solve Sudoku: Solving Sudoku with Neural Networks]

For this, we can use **Q-learning combined with Convolutional Neural Networks** because determining the optimal action for a certain state involves:

- Understanding relationships between not just the immediate tree we are considering, but also the overall placement of all the trees on the board,
- The column and row restrictions,
- The tent placement restrictions due to the already existing tents,
- The impact of placing a tent at a particular location.

The **Q-value** function would consider all of the above variables to predict the value of each possible move and its impact. We determine the **reward or punishment** as a function of the number of tents that the bot was able to place in a certain attempt. A high number of tent placements yields a greater value whereas a low number yields a lower value.

Our **long-term policy** is to be able to place all tents correctly, and we have the following short-term policies:

- Prioritize placing tents on the trees at the edges, as there are fewer valid moves around the edge.
- Target trees adjacent to columns/rows where no tent can be placed, as we will usually land on the right spot in those cases.
- In the case of 2 or more trees with a single spot in between them, generally, a tent can be placed in that spot.

We train our model using the vast amount of puzzles available on the internet. While this approach is the most data-intensive as it requires huge amounts of data as well as processing

power to learn, once completed, it has the chance to be the fastest method because it won't go through all of the possibilities.

SAT and SMT

This is a more hands-off approach to solving this puzzle because we simply declare the constraints of the puzzle and rely on the machine to solve it for us.

We can use a 2D list to form the start state and access the members of the list.

The constraints we shall use are:

1. We declare the **board size and tree locations**. This rules out the positions outside of the board, and the positions that match the tree since we cannot place a tent on a tree.
    a. Additionally, we don't want to puzzle to move the location of the trees to match the tents, so the tree locations are fixed.

2. Then we declare the **column and the row restrictions** so that the machine does not place more tents in a row or column than necessary.
    a. We sum up the tents in each row and column and check if it is equal to the constraint, and if it is then we block that row/column for further tents.

3. We make the constraint that a tent **cannot be placed adjacent** (vertically, horizontally and diagonally) to another tent.

4. We make a constraint that there cannot be a situation where there is a **tree with no tent** horizontally or vertically adjacent to it.
    a. Because we want to ensure that every tree at least has 1 tent adjacent to it.
    b. The reason we want *at least* one tent, not *at most, is because* a tent can be placed between 2 trees, but it can only be assigned to one.

        In this case, the tent position is valid, but the tent is adjacent to 2 trees, and the tree that gets assigned a new tent, would then have 2 tents adjacent to it, instead of 1.

5. Consequently, another constraint is that there cannot be a case where a **tent is not adjacent to any tree**.
    a. The exception above works in the opposite direction here. A tent placed in between 2 trees technically satisfies the criteria that all tents must have a tree adjacent to it, unfortunately, it means we have not assigned a unique tent to each tree.
    b. Thus we make the rule that it cannot be the case that a tent exists on the board without a tree adjacent to it. And since tree positions are fixed, the tent's position is changed to ensure every tree has a tent.

SAT-SMT provides a good layer of **abstraction**, the downside is that we don’t have a huge amount of control over the way the problem is solved so it is harder to optimize.