Tents and trees puzzle solver in Elixir

Short description of the puzzle

Given a map containing trees, you're supposed to place tents on some of the remaining fields such that each tree has a tent connected to it and each tent is connected to exactly one tree. In other words, there's a bijection between the trees and tents. Besides, you have to make sure that no tents touch each other horizontally, vertically or diagonally and that you place the expected number of tents in each row and column. Compared to the puzzles available on various websites, this particular puzzle is an enumeration problem, not an existance problem. This means that you have to find all possible solutions because there can be multiple valid tent placements which satisfy all of the puzzle constraints mentioned above. The satrak/1 function does exactly this: it receives a puzzle description and returns all solutions or an empty list if the puzzle can't be solved.

Planning of the algorithm

The most obvious approach to find all solutions of a puzzle is a simple brute-force algorithm. We generate all possible combinations of tent placements and check if they solve the puzzle or not. If a combination is valid, we put it in the result list, otherwise, we drop it. This method is guaranteed to find all solutions but it can be tremendously time consuming. Given a puzzle with 10 trees, we would have to create all $4^{10} = 1048576$ tent combinations and check their validity using Khf3.check_sol/2. My check_sol implementation takes 0.7s on average to check a direction list so altogether, it would take $4^{10} * 0.7s = 734003.2s \sim 8.5$ days to find all solutions. Obviously, this is way too slow so I had to come up with a faster and more efficient algorithm.

One of the reasons the brute-force approach is too slow is that it checks even those tent placements that can't possibly satisfy all puzzle constraints. For example, if we have 10 trees and the first one is on the 1-1 field, the method checks the [:n, …] and [:w, …] placements as well, even though the first tent would be placed outside the map. This means that we do $2 * 4^9 = 524288$ unnecessary checks because these placements clearly violate a puzzle constraint.

To avoid this, my goal was to eliminate the clearly invalid tent placements in the first place and then check the combinations of the other ones to find the solutions. This is achieved by using backtracking.

Algorithmic implementation

The main function of the solver is **satrak/1** which returns the list of all solutions with the help of other helper functions.

First of all, **init_tree_tents_map/1** initializes a map where the keys are the trees in the puzzle. Each key is associated with a list that contains the possible tent fields that can be connected to the given tree. Initially, all values of the map are empty lists.

After we initialized the tree-tents map, we replace the empty lists with actual content in **fill_map_with_possible_tent_placements/3**. To do this, we iterate through each tree in the key list of the map and call **get_tents_for_tree/4** for each tree. This function calls **get_tree_neighbor/5** with all 4 directions to check if the neighbor in the given direction is a valid candidate for a tent field. If the neighbor field is outside the map or has a tree on it, we can't place a tent there. In this case, the function returns nil and the field isn't included in the list of possible tents. This way, we don't have to check these clearly invalid placements later which is an improvement right away. Since the solution needs directions but the functions used later require actual coordinates, we store the tent placements in a {direction, {i, j}} format.

After creating the tree-tents map, we create another 2 maps by calling **create_tent_count_map/1** with tents_count_rows and tents_count_cols. These maps contain information about the number of expected tents in each row and column. We could've used the tents_count_rows and tents_count_cols lists as well but accessing elements of a map is quicker than accessing elements of a list based on the Elixir documentation.

After every necessary map is created, we start finding the solutions by calling **find_solutions/3**. This function simply calls **find_solutions/4** where the extra parameter is an accumulator where we collect the tents to build the solutions. Initially, we don't have any picked tents yet so tent_placements_so_far is an empty list.

The solutions are created using backtracking. In each iteration, we take the first tree of the tree-tent map and for each possible tent field of the tree, we check if picking that tent is a valid choice. So basically, we create a state space tree where a picked tent is a node and multiple possible tents associated with a tree create new branches in the tree.

To check if picking a tent is valid or not, we call **violates_row_constraint?/3**, **violates_col_constraint?/3** and **touches_other_tent?/2**. If picking the field would cause to exceed the tent limit of its row or column, we know it's not a valid choice so we skip it. We have to watch out for negative tent limits as well. If a row or column expects a negative number of tents, we can pick the field because no row or column constraint will be violated. Two tents touch each other if the difference between their rows and columns are both less than or equal to 1. To find out if picking a tent causes a touch error, we iterate through each previously picked tent and check this condition. If any of the three puzzle constraints is violated, we know it can't be a part of a solution so we terminate its branch. Instead, we go back to the node one level up and traverse its remaining branches. Otherwise, we pick the tent and remove its tree from the tree-tents map since we've found a tent for it already. After a while, the map will be empty which indicates that we found a possible solution. Now, we have to check if it's actually a solution by calling **solution_satisfies_row_constraints?/2** and **solution_satisfies_col_constraints?/2**. These functions check if the current solution matches each row tent constraint and column tent constraint exactly. If any row or column contains less tents than expected, it's not a valid solution so we drop it. Otherwise, we save the solution and continue backtracking.

After the entire state space tree is traversed, the solutions can be found in the solutions list of lists. We have to apply Enum.flat_map/2 on it to eliminate multiple levels of nesting.

This list of lists returned by find_solutions/3 contains the coordinates of the tents as well so we have to apply Enum.map/2 to get the directions only. This way, we get the list of all solutions containing the direction of the tents.

Example application of the algorithm

Let's see how the algorithm works by solving the puzzle provided in t01dcol.txt

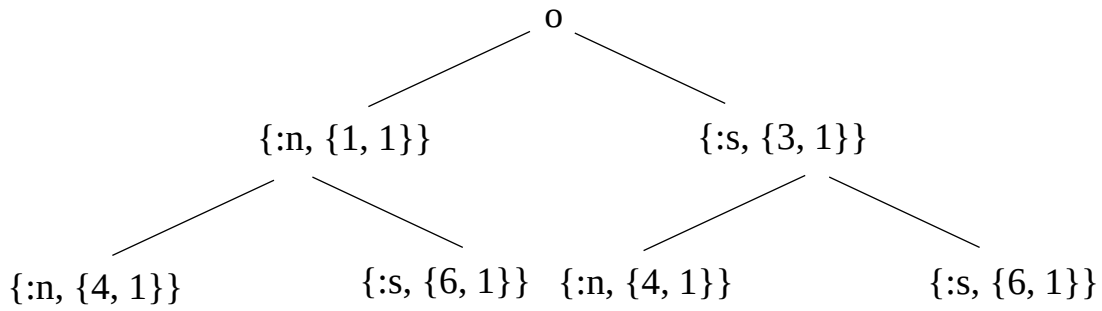
First, we create tree_tents_map:

```
%{
    {2, 1} => [ {:n, {1, 1}}, {:s, {3, 1}} ],
    {5, 1} => [ {:n, {4, 1}}, {:s, {6, 1}} ]
  }
```

Then, we create row_tent_count_map and col_tent_count_map:

```
%{ 1 => -1,                    %{ 1 => 2 }
   2 => 0,
   3 => -2,
   4 => -2,
   5 => 0,
   6 => -1
  }
```

The state search tree created by find_solutions/3 looks like this:



The {:s, {3, 1}} - {:n, {4, 1}} path contains an invalid solution because the two fields touch each other.

The other 3 paths contain valid solutions so this list is returned by find_solutions/3:

[[{:n, {1, 1}}, {:n, {4, 1}}], [{:n, {1, 1}}, {:s, {6, 1}}], [{:s, {3, 1}}, {:s, {6, 1}}]]

As we can see, the field coordinates are included in the list so we have to apply Enum.map/2. After mapping, we get the list containing the directions of the tents only: [[:n, :n], [:n, :s], [:s, :s]]

Test results

After running the tests, 9 of 10 tests gave the expected results and the remaining one was aborted because it exceeded the execution time limit. Obviously, this algorithm can be improved by doing more checks before choosing the next tent and aborting branches earlier to avoid traversing invalid subtrees unnecessarily. But of course, it's a lot faster compared to the brute-force approach. One of the tests contained more than 30 trees and the solution list was found in ~1.8s. With the brute-force algorithm, solving this puzzle would've taken a huge amount of time so we definitely achieved some improvement by using the backtracking algorithm.