Group 24: Confirming Tents and Trees

Cooper Harasyn (15cjh7)

Felix Miao (15fm23)

Gregory Secord (19gdjs)

Yuqi Yang (18yy66)

Course Modelling Project

CMPE 204 - Logic for Computing Science

December 9th, 2020

# Project Formulation

Tents and Trees is a logic game that involves placing "tents" within a square board containing a random assortment of "trees". The number of tents in each row or column is specified, and each tent must be vertically or horizontally adjacent to a tree. Each tent must also be paired to exactly one tree (and vice versa), but multiple tents can be touching one tree. Tents cannot touch vertically, horizontally, or diagonally (i.e. there can only be one tent within any 2x2 square).

A link to an online version of the Tents and Trees game can be found here: <https://tinyurl.com/tentsandtrees>

Our project aims to assess if a 5x5 Tents and Trees board has been solved.

The images below illustrate a puzzle before and after solving.

A picture containing graphical user interface

Description automatically generatedGraphical user interface, application

Description automatically generated

# Summary

We succeeded in creating a Tents and Trees propositional logic theory that can properly evaluate whether a Tents and Trees board is solved, and also extended our model to generate a solution for a given board as well as generate new Tents and Trees boards.

We began by determining how to represent the Tents and Trees board as propositions in our problem. Once we had determined our set of primary propositions and how they correspond to the board state, we began determining the constraints that we would need to represent the rules of the game. With an idea of what propositions and constraints were required, we began implementing them in Python. We hit a roadblock in that we did not initially know how to model the pairing of the tents and trees, but with the help of the TA feedback, we determined that we could model a three-dimensional array of propositions representing the location of a tree in the grid as well as the direction of the tent it is paired with. Using this, we were able to finish the basic Python implementation and verify a 5x5 Tents and Trees board.

In addition to simple verification of a hard-coded board, the Python program was extended greatly to support graphical display, the reading and writing of JSON board description files, and more. An example of the JSON and graphical output have been provided with example.json and example.html. Please run python run.py --help to see all features available. Regression testing features were also added, which can be exercised by running pytest regression-test.py.

# Propositions

In our problem, there are two main types of propositions which we will consider.

* “Primary” propositions – correspond directly to puzzle state
  + ti,j: This is true if there is a tree at the location (i, j)
  + xi,j: This is true if there is a tent at the location (i, j)
  + ri;n: This is true when the row i should have n tents in the row
  + cj;n: This is true when the column j should have n tents in the column
* “Auxiliary” propositions – describing puzzle state in less direct terms
  + Ad\_i,j: This is true when there is an association (tent/tree pair) between the tree at (i, j) and the tent in the direction d, where d=0 indicates the cell above, d=1 indicates the cell below, d=2 indicates the cell to the left, and d=3 indicates the cell to the right

# Constraints

We have divided the constraints into two sections: board-specific constraints which specify a certain board, and general constraints, which define the game itself.

## Board-specific Constraints

The board-state constraints were broken into three different Python functions: build\_board\_hint\_constraints, build\_board\_tree\_constraints, and build\_board\_tent\_constraints.

* Each row or column must contain one of 0, 1, 2, or 3 tents (“hint” constraints)
  + Maximum of three tents per row/column since tents can’t be adjacent on the 5x5 grid
  + Ex. For our particular board, row 4 has one tent, so there is the constraint (¬r4;0 ∧ r4;1 ∧ ¬r4;2 ∧ ¬r4;3)
* The board grid is modeled with the equation:  
  (¬t1,1 ∧ ¬x1,1) ∧ (¬t1,2 ∧ ¬x1,2) ∧ … ∧ (t1,5 ∧ ¬x1,5) ∧ (¬t2,1 ∧ x2,1) ∧ …   
  [similar for each grid square of the puzzle]
  + This defines what is in each cell of the board
  + This is broken into two parts: “tree” and “tent” constraints
    - It is represented as two constraints: ¬t1,1 ∧ ¬t1,2 ∧ … and ¬x1,1 ∧ ¬x1,2 ∧ …
    - The two different “tree” and “tent” constraint functions add these two constraints separately

Depending on the mode selected, some of these constraints may be omitted. For example, when using the model in the solver mode, we skip constraining the tent variables based on the board state as we want the solver to determine them, and when using the board generation mode, we constrain only the tree positions but not the row or column hints or the tent positions, which allows the solver to come up with a valid board configuration and output the row and column hints that allow the user to solve that pattern of trees. See the model exploration section for more information.

## General Constraints

* The row and column hints must be set properly
  + For each row, exactly one of ri;n is true, and likewise for columns
* Each row or column must have the correct number of tents
  + ri;n is true if and only if exactly n of xi,j (in the i-th row) are true
* A tent cannot occupy the same spot as a tree, i.e. there is either a tree, tent, or nothing in each square
  + Each square (i,j) is defined by: (¬xi,j ∧ ¬ti,j) ∨ (xi,j ∧ ¬ti,j) ∨ (¬xi,j ∧ ti,j)
* A tent must not be adjacent to another tent
  + Formerly, this was coded incorrectly – from the description that any given 2x2 square can only contain one tent, we coded something non-equivalent to optimize it:
    - xi,j → (¬ xi+1,j ∧ ¬xi,j+1 ∧ ¬xi+1,j+1)
    - If a cell contains a tent, then the cells to the right, below, and to the bottom right cannot contain a tent
  + This is a big optimization, since instead of saying “there are no tents, or there is a tent in the top-left and no other tents, or there is a tent in the top-right…” and so on for every 2x2 square in the board (this would have a lot of overlapping checks), we can make each cell be on the left-hand side of the implication exactly once, which optimizes the model
  + However, this overlooked the top-right to bottom-left direction pairing, meaning that it falsely verified boards having trees adjacent in this direction
  + This was fixed by adding a check for the cell in the bottom-left and blocking that possibility
    - xi,j → (¬ xi+1,j ∧ ¬xi,j+1 ∧ ¬xi+1,j+1 ∧ ¬xi+1,j-1)
  + This error has also been added to the regression-testing suite as “unsolvable/diagonal-left.json”
    - This puzzle was generated with the generation feature, and is where we noticed this diagonal issue
* A tent must be adjacent to a tree
  + If X,Y,Z,W are the elements of the tree grid for all directions horizontally or vertically adjacent to a point (i,j), then:
    - xi,j → (X ∨ Y ∨ Z ∨ W)
  + Note that it is not required for there to be only one tree adjacent to a tent; multiple trees can be adjacent to a tent and multiple tents can be adjacent to one tree, but…
* … A tree must be paired with exactly one tent and vice-versa
  + This is a set of constraints imposed on the A\* propositions.
  + If ti,j is true, then exactly one of Ad\_i,j (sweeping over d) must be true.
    - If there is a tree, then it must be associated with exactly one tent.
    - If we let X,Y,Z,W be A0\_i,j, A1\_i,j, A2\_i,j, A3\_i,j, then we have:
      * ti,j → exactly\_one([X,Y,Z,W])
      * where exactly\_one is a counting function that ensures exactly one of its inputs is true
  + If xi,j is true, then exactly one of the A entries pointing at (i,j) must be true.
    - If there is a tent, then it must be associated with exactly one tree.
    - If we let X,Y,Z,W be A0\_i+1,j, A1\_i-1,j, A2\_i,j+1, A3\_i,j-1, then we have:
      * xi,j → exactly\_one([X,Y,Z,W])
      * where exactly\_one is a counting function that ensures exactly one of its inputs is true
    - The reason for the +1 and -1 in the locations in the A matrix is because we need to get the location of the tree which, when pointing in a direction from that location, points to this tent.
  + If Ad\_i,j is true for any d, then ti,j is true.
    - If there is an association from a tree to a tent starting at this position, then there must be a tree at this position.
    - Ad\_i,j → ti,j
  + If Ad\_i,j is true for any d, then the space that direction d points to must be a tent.
    - If there is an association from a tree to a tent starting at this position, then if we travel in the direction d, there must be a tent there.
    - A0\_i,j → xi-1,j
    - A1\_i,j → xi+1,j
    - A2\_i,j → xi,j-1
    - A3\_i,j → xi,j+1

# Model Exploration

There are three different modes in the program provided, representing the three different model exploration paths taken. These were constructed by varying the constraints placed on the theory. For this purpose, the board-state constraints were broken into three different Python functions: build\_board\_hint\_constraints, build\_board\_tree\_constraints, and build\_board\_tent\_constraints.

There is the verification mode, which takes in a puzzle description and fully constrains the problem by setting up the row/column hint constraints, the tree position constraints, and the tent position constraints (all 3 constraint functions). It then runs the solver to check if a solution exists or not. If a solution exists, then the puzzle is solved correctly, but if a solution does not exist, then the puzzle has not been solved.

There is the solving mode, which takes in a puzzle description and sets up the row/column hint and tree position constraints but omits the tent position constraints (skips build\_board\_tent\_constraints). It then runs the solver to fill in the tent placements. This allows us to leverage our existing model to solve boards in addition to just verifying their pre-existing solution.

Finally, there is the board generation mode. This works by randomly generating a grid of trees and constraining only the tree positions (using only build\_board\_tree\_constraints). It then runs the solver and takes the first solution, generating a new board if there is no solution. It will also create a copy of the solution, and then remove the tree positions from the copy which creates the “unsolved” copy. This ensures that the board we create is always properly solvable, since it was a solved board that we removed the solution from. It will output both the unsolved and solved boards.

One feature that really assists in exploring the model is the graphical output feature, which can be used by specifying the --output-html option on the command-line (try --help for more information on command-line switches). This graphical output feature renders the puzzle solutions into an HTML page, where in addition to the visible board state, the association between tents and trees is shown. This was developed during debugging of the association constraints, and greatly helped in diagnosing the issues with this feature.

The board generation mode was an extremely useful addition. It allowed us to debug potential issues with the model very easily, since any illegal moves that were allowed by the model could easily be caught by human checking. This is how the “diagonal bottom-left” issue was found, which is documented in the Constraints section of this report. Once that issue with the model was fixed, twenty boards were generated and manually checked to ensure they were valid, and no further issues with the model were found.

# First-Order Extension

Our Tents and Trees problem extends well to first-order logic. To extend it to first-order logic, we could define the following functions over the set of positions in the grid:

* T(p): Tree function. It is T if the grid at position p contains a tree, and F otherwise.
* X(p): Tent function. It is T if the grid at position p contains a tent, and F otherwise.
* P(p1,p2): Partial adjacency function. This function is commutative. It is T if positions p1 and p2 are adjacent horizontally, or vertically, but not diagonally. It is F otherwise.
* F(p1,p2): Full adjacency function. This function is commutative. It is T if positions p1 and p2 are adjacent horizontally, vertically, or diagonally. It is F otherwise.
* A(t,x): Association function. This function is commutative. It is T if the tent at t is associated (paired) with the tree/tent at x, and F otherwise.
* E(p1,p2): Equality function. It is T if p1 is the same position as p2, and F otherwise.

This would allow us to represent our constraints more effectively. Instead of having one constraint per cell in the grid, we could do as follows: (note that two-letter variables are used as shorthand for subscript, meaning pt -> p­t)

* A tent cannot occupy the same spot as a tree
  + ∀p. ¬(T(p) ∧ X(p))
* A tent associated with a tree must be partially adjacent to it
  + ∀pt. (T(pt) → ∀px. (X(px) ∧ A(pt,px) → P(pt,px)))
* Two tents cannot be adjacent
  + ∀px1. (X(px1) → (∀px2. (F(px1,px2) → ¬X(px2)))))
* A tree must be associated (paired) with a tent
  + ∀pt. (T(pt) → ∃px. (X(px) ∧ A(pt,px)))
* A position cannot be associated with multiple other positions
  + ∀pt. (∀px1. (A(pt,px1) → (∀px2. (A(pt,px2) → E(px1,px2)))))
  + If any pt is associated with a px1, then any px2 which is also associated with pt must be equal to px1
  + … and likewise for all associations between a tent and two trees (px, pt1, pt2)
* If there is an association, it must be from tree to tent
  + ∀pt. (∀px. (A(pt,px) → (T(pt) ∧ X(px))))

# Jape Proofs

One interesting thing that we were able to prove relates to a specific board state. We showed that when you have a board with an arrangement of trees up against the edge as seen in Figure 1, that the top-left square (1,1) must contain a tent.

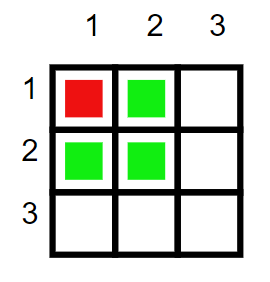


Figure - Tree arrangement for Jape proof. Green squares denote trees, red squares denote tents.

Before discussing further details of this proof, please note that in Jape, the propositions with the form TeXY represent the existence of a tent in position (X,Y) of the board. Ex. Te11 means there is a tent at (1,1) in the board.

Here is the complete theorem that was proved:

Te11 ∨ Te13, Te11 ∨ Te31, Te23 ∨ Te32, Te13 → ¬Te23, Te31 → ¬Te32 ⊢ Te11

This proof was completed by greatly simplifying the constraints from the full theory: all premises in this proof are derived from either the requirement of tents to be in certain locations, or the requirement that two tents cannot be adjacent. For example, from the tree at (1,2), we know that there must be a tent at (1,1) or (1,3), so we can add the premise Te11 ∨ Te13. Additionally, we know that if there is a tent at (1,3) then there cannot be a tent at (2,3), so we can add the premise Te13 → ¬Te23. By building up a set of these premises, we were able to prove unconditionally that a tent must go at (1,1) in all cases.

Unfortunately, we were only able to complete one Jape proof for this project.

# Incomplete Items

In the feedback from the project proposal, it was suggested to use an adder network when counting the number of tents in a given row or column. This number checks the validity of the tent positions (e.g. cannot be adjacent), as well as the number of counted tents to the corresponding hint (e.g. if the row hint is 2, then we should be counting exactly 2 tents in that row). Unfortunately, the team was not able to resolve the logical inconsistencies resulting from the formulas (see the table below). The tutorial session from November 13th was used as a starting point, but the specific implementation was not immediately apparent.

Table - A table showing the team's attempt at implementing an adder network to the project. Note the combinations in red font are invalid, and the team was unable to resolve these inconsistencies. 1 = tent, 0 = blank or tree.

|  |  |  |
| --- | --- | --- |
| **Index Value** | **Formulas** | **Possible Combinations from Formulas** |
| Index 0 (Square #1) | count\_by\_s0\_is\_0 = ¬x[i][0] | 0 |
| count\_by\_s0\_is\_1 = x[i][0] | 1 |
| count\_by\_s0\_is\_2 = False | - |
| count\_by\_s0\_is\_3 = False | - |
| Index 1 (Square #2) | count\_by\_s1\_is\_0 = count\_by\_s0\_is\_0 & ¬x[i][1] | 00 |
| count\_by\_s1\_is\_1 = count\_by\_s0\_is\_1 & ¬x[i][1] | count\_by\_s0\_is\_0 & x[i][1] | 10 | 01 |
| count\_by\_s1\_is\_2 = False | - |
| count\_by\_s1\_is\_3 = False | - |
| Index 2 (Square #3) | count\_by\_s2\_is\_0 = count\_by\_s1\_is\_0 & ¬x[i][2] | 000 |
| count\_by\_s2\_is\_1 = count\_by\_s1\_is\_1 & ¬x[i][2] | count\_by\_s1\_is\_0 & x[i][2] | 100 | 010 | 001 |
| count\_by\_s2\_is\_2 = count\_by\_s1\_is\_1 & x[i][2] | 101 | 011 |
| count\_by\_s2\_is\_3 = False | - |
| Index 3 (Square #4) | count\_by\_s3\_is\_0 = count\_by\_s2\_is\_0 & ¬x[i][3] | 0000 |
| count\_by\_s3\_is\_1 = count\_by\_s2\_is\_1 & ¬x[i][3] | count\_by\_s2\_is\_0 & x[i][3] | 1000 | 0100 | 0010 | 0001 |
| count\_by\_s3\_is\_2 = count\_by\_s2\_is\_2 & ¬x[i][3] | count\_by\_s2\_is\_1 & x[i][3] | 1010 | 0110 | 1001 | 0101 | 0011 |
| count\_by\_s3\_is\_3 = False | - |
| Index 4 (Square #5) | count\_by\_s4\_is\_0 = count\_by\_s3\_is\_0 & ¬x[i][4] | 00000 |
| count\_by\_s4\_is\_1 = count\_by\_s3\_is\_1 & ¬x[i][4] | count\_by\_s3\_is\_0 & x[i][4] | 10000 | 01000 | 00100 | 00010 | 00001 |
| count\_by\_s4\_is\_2 = count\_by\_s3\_is\_2 & ¬x[i][4] | count\_by\_s3\_is\_1 & x[i][4] | 10100 | 01100 | 10010 | 01010 | 00110 | 10001 | 01001 | 00101 | 00011 |
| count\_by\_s4\_is\_3 = count\_by\_s3\_is\_2 & x[i][4] | 10101 | 01101 | 10011 | 01011 | 00111 |

The above table shows the team’s attempt at implementing an adder network to count the number of tents in a given row or column. Many of the combinations from the formulas did not follow some of the other project constraints (i.e. tents cannot be adjacent), and thus, the combinations in red font are invalid. Because of this, an exhaustive list of combinations was used in the code.