Algorithms in Logic – Final Project – Summary

Elad Kapuza
Adam Goldbraikh

Puzzle Problem

Classic Game:

- Given a triangle board.
- Start with any one hole empty.
- As you jump the pegs remove them from the board as in "Damka" game.
- The goal: Remain with only one peg anywhere in the board.

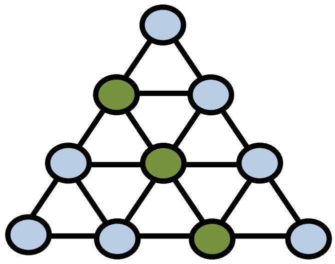
Advanced Game:

- Given initial board and final board.
- Rules of movements remain the same.
- The goal: Reach to the final board.

Logic Modeling

Sets definitions

- $A = \{(i, j, k):\ i, j, k\ are\ aligned\ neighbors\}$
  Aligned neighbors, for example, the green vertices:



- $U_{init} = \{v:\ v\ contains\ checker\ in\ the\ initial\ state\}$
- $U_{final} = \{v:\ v\ contains\ checker\ in\ the\ final\ state\}$
- $T = |U_{init}| - |U_{final}|$
  T - Number of steps from initial state to final state

Variables definitions

- $X_{i,j,k,p}$ - True if in phase p, vertices i, j, k are aligned neighbors, and k can be reached from I via j.
- $Y_{i,p}$ - True if vertex i has checker in phase p.

Constraints –Boolean Logic

(1) **Initial state:** In the 1st phase (phase 1) we require all vertices in U_{init} to have checker.

$$\varphi_{init} = \bigwedge_{i \in U_{init}} Y_{i,1} \wedge \bigwedge_{i \notin U_{init}} \neg Y_{i,1}$$

(2) **One step:** In each phase p, we perform only one step.

$$\varphi_{one\_step} = \bigwedge_{p=1}^{T} \left[\bigvee_{(i,j,k) \in A} \left(X_{i,j,k,p} \wedge \bigwedge_{\substack{(r,s,t) \in A \\ (r,s,t) \neq (i,j,k)}} \neg X_{r,s,t,p} \right) \right]$$

(3) **Legal step:** In each phase p there is at least one legal step

$$\varphi_{legal} = \bigwedge_{p=1}^{T}\left[\bigvee_{(i,j,k)\in A}\left(Y_{i,p} \wedge Y_{j,p} \wedge \neg Y_{k,p} \wedge X_{i,j,k,p}\right)\right]$$

(4) **Advancement**: In each phase p:
If there is a legal step from i to k via j, then in the next state, vertex i and j are empty and vertex k has checker.
All other vertices remain the same as in the previous state.

$$\varphi_{steps} = \bigwedge_{p=1}^{T}\left[\bigvee_{(i,j,k)\in A}\begin{pmatrix} X_{i,j,k,p} \wedge \neg Y_{i,p+1} \wedge \neg Y_{j,p+1} \wedge Y_{k,p+1} \wedge \\ \bigwedge_{a\neq i,j,k} Y_{a,p} \leftrightarrow Y_{a,p+1} \end{pmatrix}\right]$$

(5) **Final state:** There is only one vertex among the n ones that has checker, all other vertices are empty, and we are in the last phase (T+1).

$$\varphi_{final} = \bigvee_{i=1}^{n}\left(Y_{i,T+1} \wedge \bigwedge_{j\neq i} \neg Y_{j,T+1}\right)$$

Final formula: We would like to check the satisfiability of the formula:

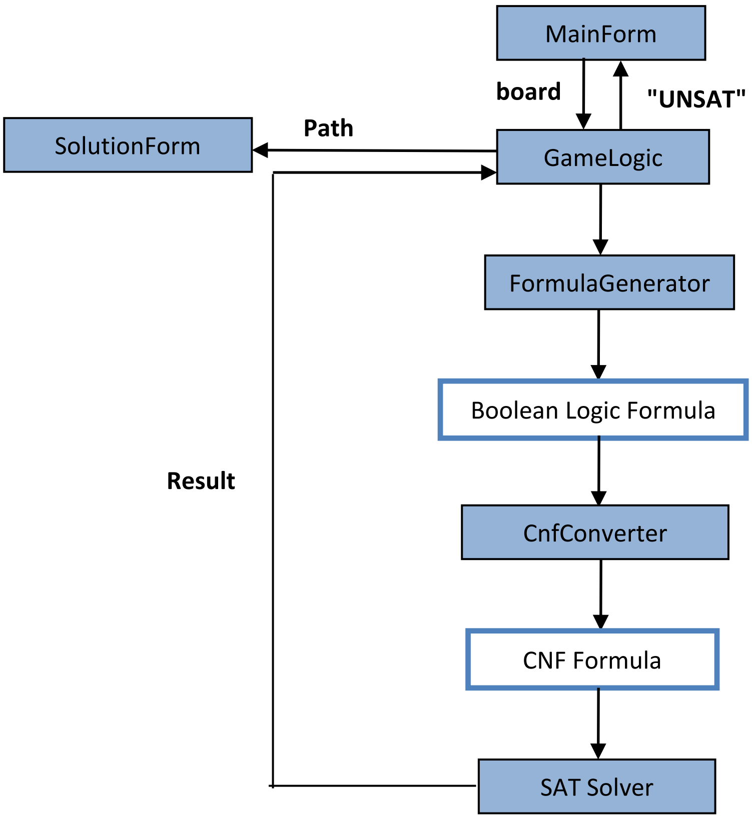
$$\psi = \varphi_{init} \wedge \varphi_{one\_step} \wedge \varphi_{legal} \wedge \varphi_{steps} \wedge \varphi_{final}$$

Advanced Game Small Modification:

- We allow the player to set both the initial board and the final board.
- All first 4 formulas remain the same.
- We change only the 5th formula:

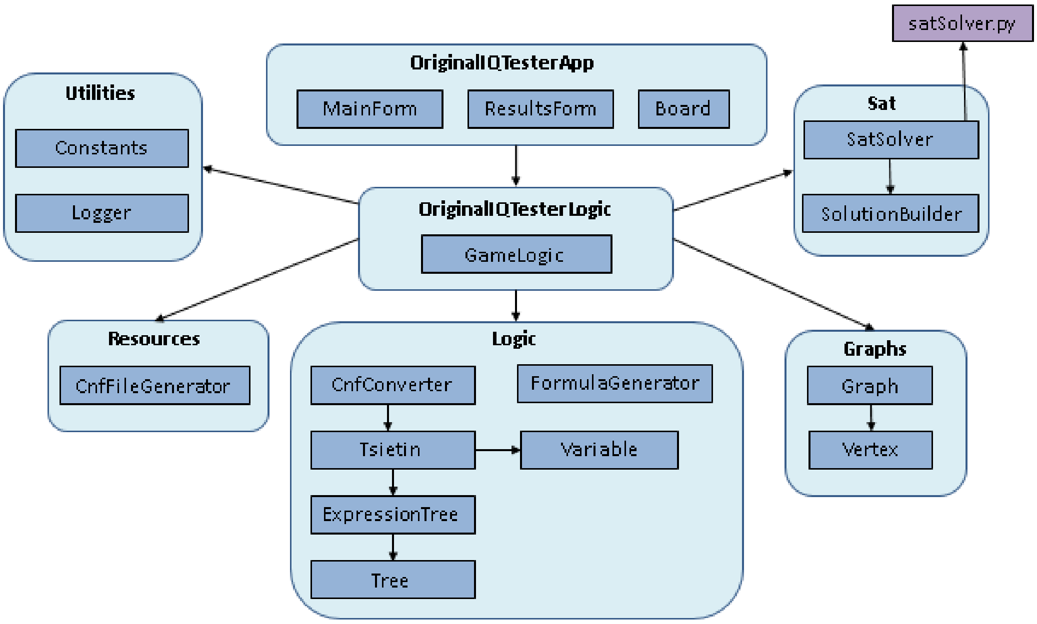
$$\varphi'_{final} = \bigwedge_{i\in U_{final}} Y_{i,T+1} \wedge \bigwedge_{i\notin U_{final}} \neg Y_{i,T+1}$$

Algorithm – Flow Diagram



1. The user enters initial board and final board (if mode="advanced") in the main screen.
2. The game logic object gets the board as input, derives from it the graph, variables and so on. Then it uses the Formula generator to generate Boolean logic formulas according the 5 constraints demonstrated earlier.
3. The CNF converter takes the formulas and converts them to CNF format using Tsietin algorithm.
4. The SAT solver takes the CNF formula, solves the whole CNF Boolean formula, and return the result back to the game logic object.
5. If result="UNSAT" a message is retrieved to the user accordingly.
6. Else: the solution form shows up and gets the solution path.

Implementation - Blocks Diagram



OriginalIQTesterApp – C# WinForms Application

- **MainForm** – main screen contains input objects – mode (classic or advanced), initial and final boards.
- **ResultsForm** (**SolutionForm**) – contains solution and steps towards the goal board.
- **Board** – Graphic object represents the board

OriginalIQTesterLogic

- **GameLogic** – Top level object the manages all the logic components of the application.

Logic

- **Tree** – General binary tree holding string nodes
- **ExpressionTree** – Uses Tree to implement logic expressions as trees.
- **Tseitin** – implements "Tseitin algorithm" to convert general Boolean formulas to CNF formulas using expression trees.
- **Variable** – represents a variable (regular or artificial in Tseitin algo).
- **CnfConverter** – uses Tseitin to convert general Boolean formulas to CNF.
- **FormulaGenerator** – Generate the game constraints as Boolean formulas.

Graphs

- **Graph** – represents the board as graph
- **Vertex** – represent vertex in the board.

Sat

- **SatSolver** – Runs python script "SatSolver.py" to run the solver and return a result – SAT or UNSAT.
- **SolutionBuilder** – Gets the solution and converts the satisfied variables list to steps list (path) that represents the game solution.

Resources

- **CnfFileGenerator** – Get the cnf formula and Generate from it CNF file in "cnf" format as input for the satSolver.py script.

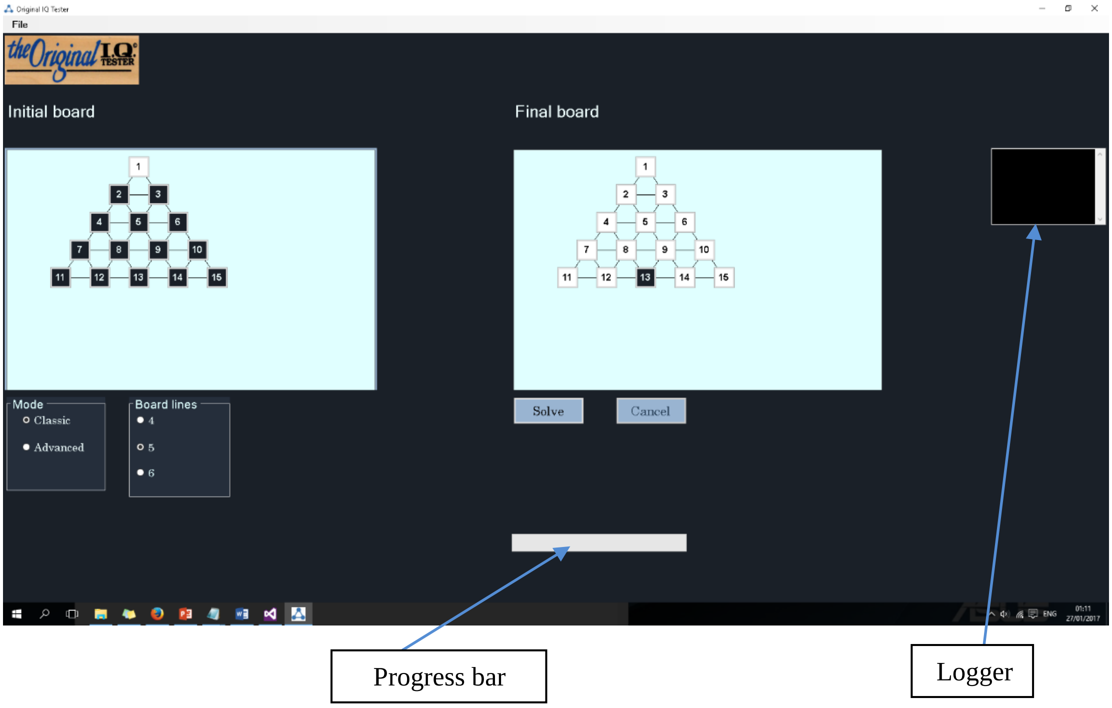
Utilities

- Constants – Contains constants attributes for all the application components
- Logger – contains logs from the SAT solver.

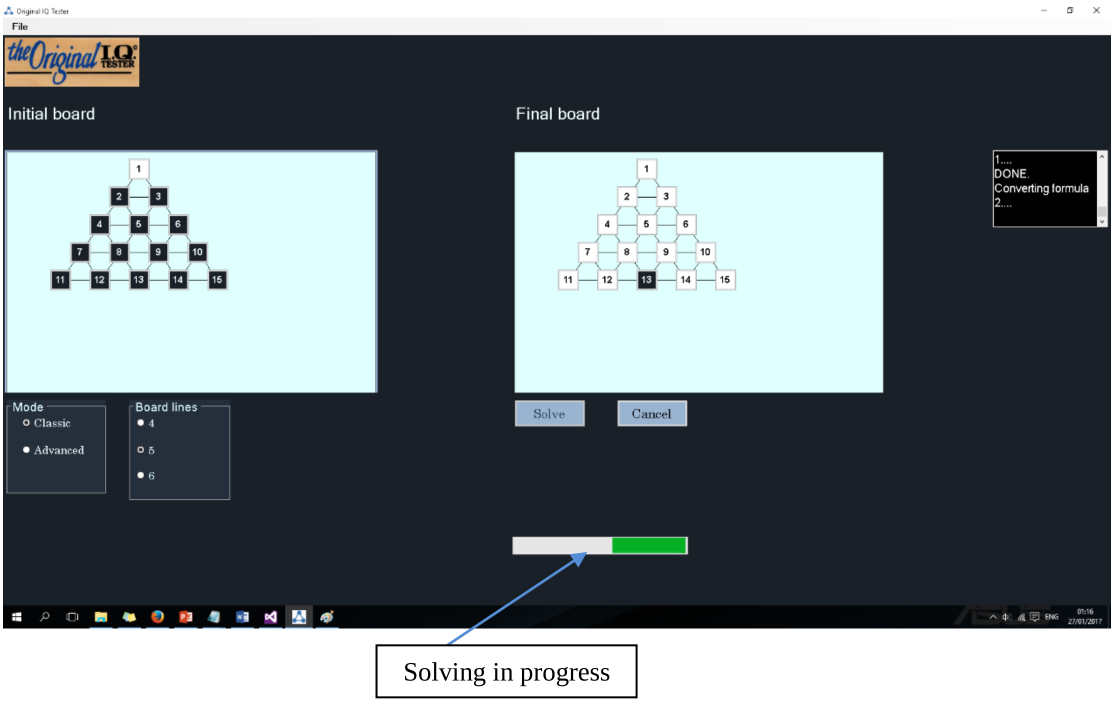
satSolver.py – python script that uses Pycosat to run SAT solver. It gets as input a cnf file and returns a list of satisfied variable identifiers.

Application

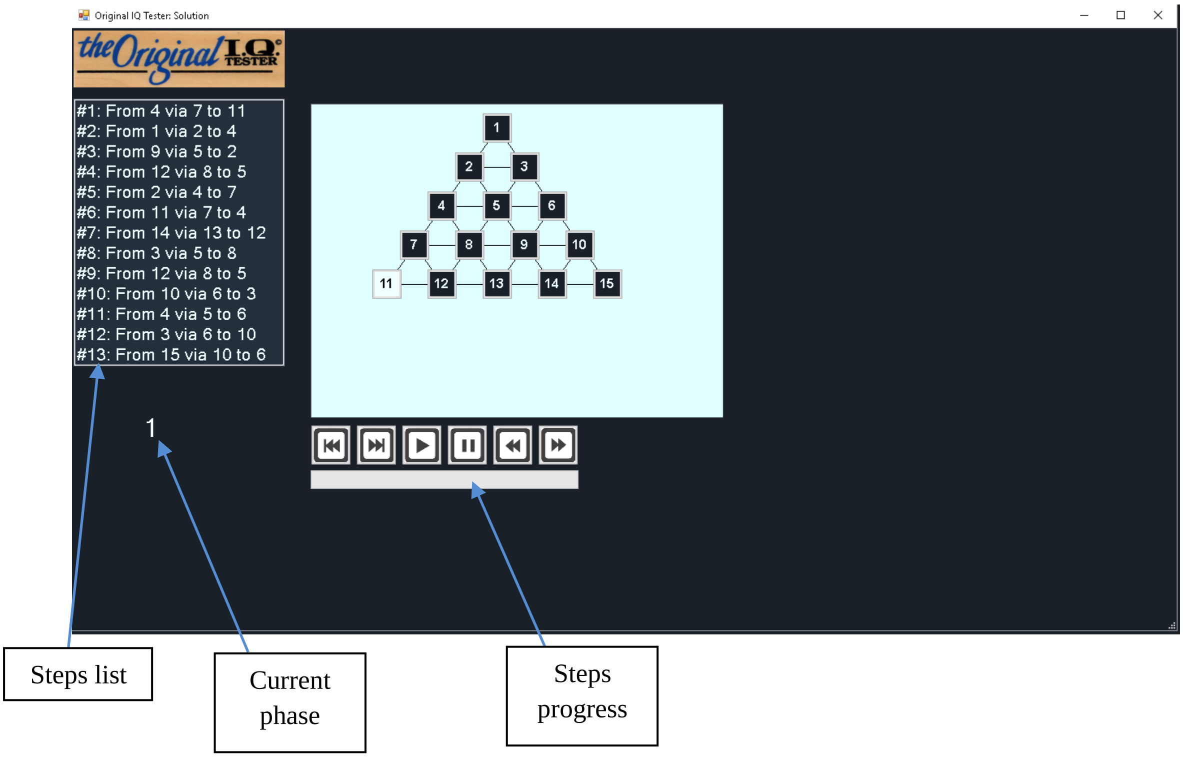
Main Screen



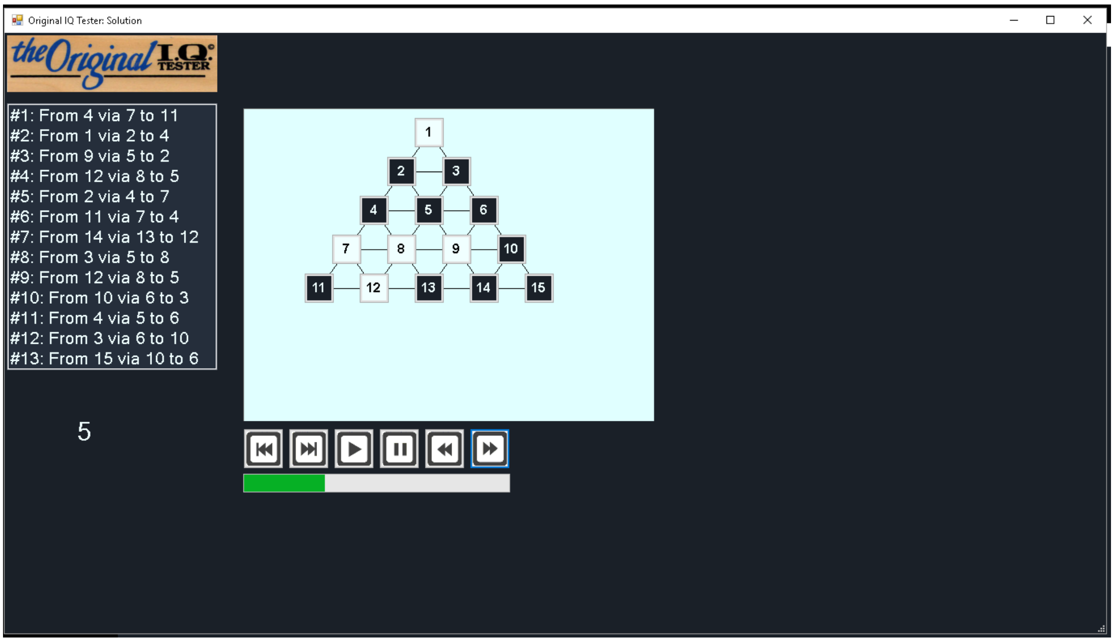
1. Set initial and final boards.
2. Press Solve to start the solver.
3. Press Cancel to cancel operation.



Solution Screen



This how it looks while stepping into the solution.



Steps controls:

- Go to the first phase.
- Go to the last phase.
- Auto-play solution steps.
- Pause the auto-playing.
- Go to the previous phase.
- Go to the next phase.