Boston College Computer Science Senior Thesis

USING QBF SOLVERS TO SOLVE GAMES AND PUZZLES

Zhihe Shen

Advisor: Howard Straubing

Abstract

There are multiple types of games, such as board games and card games. Some are multiplayer games and some are single-player games. Many games such as 2-player games are hard to solve because the problem of determining whether a given player has a winning strategy for these games is PSPACE-complete. It is proved that the problem of determining whether a quantified boolean formula is true is also PSPACE-complete. Because of the PSPACE-completeness of TQBF, every problem in PSPACE, in particular these games, can be encoded as an instance of TQBF. Thus, one way to understand the complexity of a game is to encode it as a quantified Boolean formula.

This thesis aims to investigate the computational complexity of different kinds of games. We choose to work on games played between two agents, for example, simple geography games. Because they are in PSPACE, we convert them into non-clausal quantified Boolean formulas based on the rules of each game. By solving those formulas, we can find a winning strategy for either player. One way to solve these formulas is to use a quantified Boolean formula solver (QBF solver). In this paper, we will use GhostQ to solve the non-clausal quantified Boolean formula.

Contents

1. Introduction

Let us first think about a game like Rush Hour. The goal is to find a sequence of legal moves that would allow a target car to exit the game board. In each round, one of the players can make a legal move based on the rules of the game and state of the board. When the game board is very large, it is hard to solve because we cannot find a way better than a brute-force search of all possible sequences of legal moves. In fact, the problem of solving a generalized $n \times n$ game is proved to be PSPACE-complete [1].

PSPACE-complete problems are important in the way that if we find a solution to one of these problems, we can easily find solutions to other problems in PSPACE. This is because according to the definition of PSPACE-complete, we can easily reduce other problems in PSPACE to the PSPACE-complete one we have solved. However, PSPACE-complete problems are difficult to solve. So how can we have a program to solve them? In a paper in 1972, Meyer and L. Stockmeyer create a PSPACE-complete set based on quantified Boolean formulas [2]. We follow the approach of Sipser to find out that the problem of determining whether a quantified Boolean formula is true is PSPACE-complete. A sketch of the proof of the PSPACE-completeness of TQBF can be found in section 2.8. Following the same approach, we can conclude that the problem of determining whether a given player has a winning strategy for simple geography games is also in PSPACE. Therefore, we can encode these games as instances of quantified Boolean formulas because TQBF is PSPACE complete. The problem of determining which player has a winning strategy in a game can be transformed to the problem of evaluating the value of the corresponding quantified Boolean formula after assigning values that the players have chosen to the variables in the formula.

Then we can use a quantified Boolean formula solver to complete the process of evaluating quantified Boolean formulas. In this paper, we will encode simple geography games as instances of quantified Boolean formulas. In section 3.3, we will introduce the symbols we use and the rules we follow to encode these games. So how can we solve these instances of quantified Boolean formulas? Given that fairly large practical instances of the Boolean satisfiability problem can be solved using SAT solvers, it is reasonable for us to solve the instances of quantified Boolean formulas by a QBF solver. However, there exists a gap between theory and practice. In contrast to SAT solvers, QBF solvers are not widely used in practice. A detailed comparison between SAT solvers and QBF solvers is in section 2.10. In this paper, we will use a QBF solver GhostQ to solve the instances of QBF we generate for geography games. Compared to other solvers, one advantage of GhostQ is that it can solve non-clausal quantified Boolean formulas, which is the reason why we choose GhostQ. A detail description of non-clausal quantified Boolean formulas is in section 2.5.

2. Concept

2.1 Concept of Boolean formula

A **Boolean formula** returns true or false, and is formed by variables, along with the symbols OR $\vee$, AND $\wedge$, NOT $\neg$, and parentheses (,). For example, $(\neg x_1 \vee x_2) \wedge x_3$ is a Boolean formula with variables x_1, x_2, x_3 and $\vee$, $\wedge$, $\neg$, (,).

2.2 Concept of Conjunctive Normal Form (CNF)

A formula is in **conjunctive normal form** if it is a conjunction of disjunctions. For example, $(\neg x_1 \vee x_2) \wedge (x_3 \vee x_4)$ is in conjunctive normal form because it is a conjunction of disjunction $(\neg x_1 \vee x_2)$ and disjunction $(x_3 \vee x_4)$.

2.3 Concept of Boolean Satisfiability Problem (SAT)

The **Boolean Satisfiability Problem (SAT)** is to determine whether a formula in conjunctive normal form (CNF) has a satisfying assignment. The formula is satisfiable if there exists a true/false assignment for variables in the formula that can make the formula true.

For example, to determine whether the formula $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_1)$ has a satisfying assignment is a Boolean satisfiability problem since $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_1)$ is in conjunctive normal form. The formula $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_1)$ is satisfiable because $x_2 =$ true and $x_1 =$ true is a satisfying assignment for the formula. After assigning the values to variables x_1 and x_2, the formula becomes (false $\vee$ true) $\wedge$

(true $\vee$ true) $\equiv$ true.

2.4 Concept of Quantified Boolean Formulas (QBF)

Quantified Boolean formulas (QBF) extend propositional formulas by allowing explicit quantification ($\exists, \forall$) over the propositional variables [5].

Syntax:
Boolean formulas together with quantifiers $\forall$ (for all) and $\exists$ (there exists) are called **quantified Boolean formulas**. If all the variables in a formula are within the scope of some quantifier, then the formula is fully quantified.

Semantics:
$\forall x\ \varphi$: we need to check for both possible values of 'x' in φ to see if φ is true or false. If for both possible values (i.e. 0 and 1) for the variable x, the statement φ is true, then $\forall x\ \varphi$ is true. Otherwise, $\forall x\ \varphi$ is false.
$\exists x\ \varphi$: we need to check if there exists a value for 'x' such that φ is true. If for some value of the variable x, the statement φ is true, then $\exists x\ \varphi$ is true. Otherwise, $\exists x\ \varphi$ is false.

Example:
$\phi = \forall x_1 \exists x_2 (x_1 \bigoplus x_2)$ is an example of a quantified Boolean formula. Here, ϕ is true if and only if for all values of x_1, there exists a value of x_2 such that exactly one of x_1 and x_2 is true. In addition, the formula is also fully quantified because x_1 is within the scope of the quantifier $\forall$ and x_2 is within the scope of the quantifier $\exists$. To determine whether the formula ϕ is true or false, we first assume x_1 to be true. In this case, we can let x_2 be false, so exactly one of x_1 and x_2 is true, which

means ϕ is true. Then we assume x_1 to be false, so we can let x_2 be true to make the statement ϕ true. Since for all possible values of x_1, we can find a value of x_2 to make the statement ϕ true. We conclude that the statement is always true.

2.5 Clausal vs. Non-Clausal Quantified Boolean Formulas

A **Clausal quantified Boolean formula** is constructed by one or more quantifiers followed by a Boolean formula in conjunctive normal form.

For example, $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \exists z3\ ((\neg x_1 \vee x_2) \wedge (y_2 \vee y_3) \wedge z3)$ is a clausal quantified Boolean formula.

However, a **non-clausal quantified Boolean formula** does not have this restriction of order. It can be constructed by one or more quantifiers followed by a propositional expression which is followed by quantifiers followed by another propositional expression and so on.

For example, $\exists x_1 \exists x_2\ ((x_1 \vee x_2) \wedge \forall y_1 \forall y_2 (\neg y_1 \vee y_2 \rightarrow x_1))$ is a non-clausal quantified Boolean formula.

2.6 Concept of PSPACE and PSPACE-complete

Decision problems are problems with yes-no answers. **PSPACE** problems are decision problems which require an amount of space polynomial in the size of the instance. For example, the problem of determining whether a given player has a winning strategy for the Japanese game go-moku is PSPACE. To show that, we can loop through all possible moves of the two players. Assume there exists a game tree. For an $n \times n$ game board, we need at most $O(n^2)$ space to store the board, so each level of the recursion stack uses at most $O(n^2)$ space. Then we need to keep track of the moves that have been examined. The height of the recursion stack is less than or equal to the depth of the game tree, which is n^2. Thus, the algorithm to solve go-moku runs in space $O(n^4)$, which is polynomial in the input length.

A language is **PSPACE-complete** if 1. it is in PSPACE, 2. Every language in PSPACE is polynomial time reducible to it [3]. For example, the problem of determining whether a quantified Boolean formula is true is PSPACE-complete. The introduction of true quantified Boolean formula is in section 2.7 and the sketch of the proof is in section 2.8.

2.7 Concept of TQBF

Syntax: Our approach to define **true quantified Boolean formula** (TQBF) follows that of Sipser [3]. We encode problem instances as strings, so if φ is an instance of QBF - that is, a quantified Boolean formula - then $\langle\varphi\rangle$ denotes the string encoding φ.

We set TQBF = $\{\langle\varphi\rangle |$ φ is a true fully quantified Boolean formula.$\}$

Semantics: A fully quantified Boolean formula is either true or false because every variable of such formula is within the scope of some quantifier. The TQBF problem is to determine whether a fully quantified Boolean formula is true or false.

2.8 Sketch of the proof that TQBF is PSPACE-complete

The problem of determining whether a quantified Boolean formula is true is PSPACE-complete. Our approach to prove the statement follows that of Sipser [3]. The proof consists of two parts.

First, we use a recursive algorithm to show that TQBF is in PSPACE. Let T be a polynomial space algorithm that decides TQBF and $\langle\theta\rangle$ - a fully quantified Boolean formula - be the input of the algorithm T:
1. If θ contains no quantifiers i.e. $\forall$, $\exists$, then we evaluate θ directly because the expression only contains constants. If θ is true, then accept. Otherwise, reject.
2. If θ equals $\forall x\ \varphi$, then we recursively call T on φ because variable x can have different values. That is to say, we replace variable x with 1 and 0 to evaluate φ. According to the concept of QBF, if in either case, the result is accept, then accept. Otherwise, reject.
3. If θ equals $\forall\varphi$, then we recursively call T on φ because variable x can have different values. That is to say, we replace variable x with 1 and 0 to evaluate φ. According to the concept of QBF, if in both case, the result is accept, then accept. Otherwise, reject.

Then we need to show that the algorithm T runs in polynomial time. In fact, algorithm T runs in linear space, which is even stronger than the proposition that it runs in polynomial time. To show T runs in linear space, let us assume that the number of variables in θ is m. Then the maximum number of recursive calls would be m, and at each level of the recursion, we only need to store the value of one variable in θ. Thus, the total space complexity is $o(m)$, which is linear.

Second, we need to show that TQBF is PSPACE-hard. We need to construct a quantified Boolean formula ϕ that is true if and only if a Turing Machine M accepts the input string w. To get an idea of how to construct ϕ, we first construct a formula $\phi_{c_1,c_2,t}$ where c_1, c_2 are two configurations and t is a positive number. We let the formula to be true if and only if M can go from c_1 to c_2 in at most t step.

If t = 1, we can construct $\phi_{c_1,c_2,t}$ such that one of the following two conditions is true: 1. c_1 equals c_2 2. M can go from c_1 to c_2 in one step
If t>1, we construct $\phi_{c_1,c_2,t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}[\phi_{c_3,c_4.t/2}]$, where m_1 is a configuration of M. This formula indicates that the variable representing the configurations c_3, c_4 can take either the values of the variables of c_1 and m_1 or m_1 and c_2. In either case, the formula $\phi_{c_3,c_4.t/2}$ is true, which means that M can go from c_3 to c_4 in at most $t/2$ steps. To convert the formula $\phi_{c_1,c_2,t}$ into a quantified Boolean formula, we replace $\forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}$ by $\forall (c_3, c_4)$ $[(c_3, c_4) = (c_1, m_1)[...] \vee (c_3, c_4) = (m_1, c_2) \to ...]$

The formula $\phi_{c_{start},c_{accept},h}$, where $h = 2^{df(n)}$, and d is a constant. When $t > 1$, we construct ϕ recursively. The size of each level of recursion is $O(f(n))$, and the number of levels of recursion is also $O(f(n))$. Thus, the formula we get after recursive calls is of size $O(f^2(n))$, which is polynomially large.

2.9 SAT vs. QBF

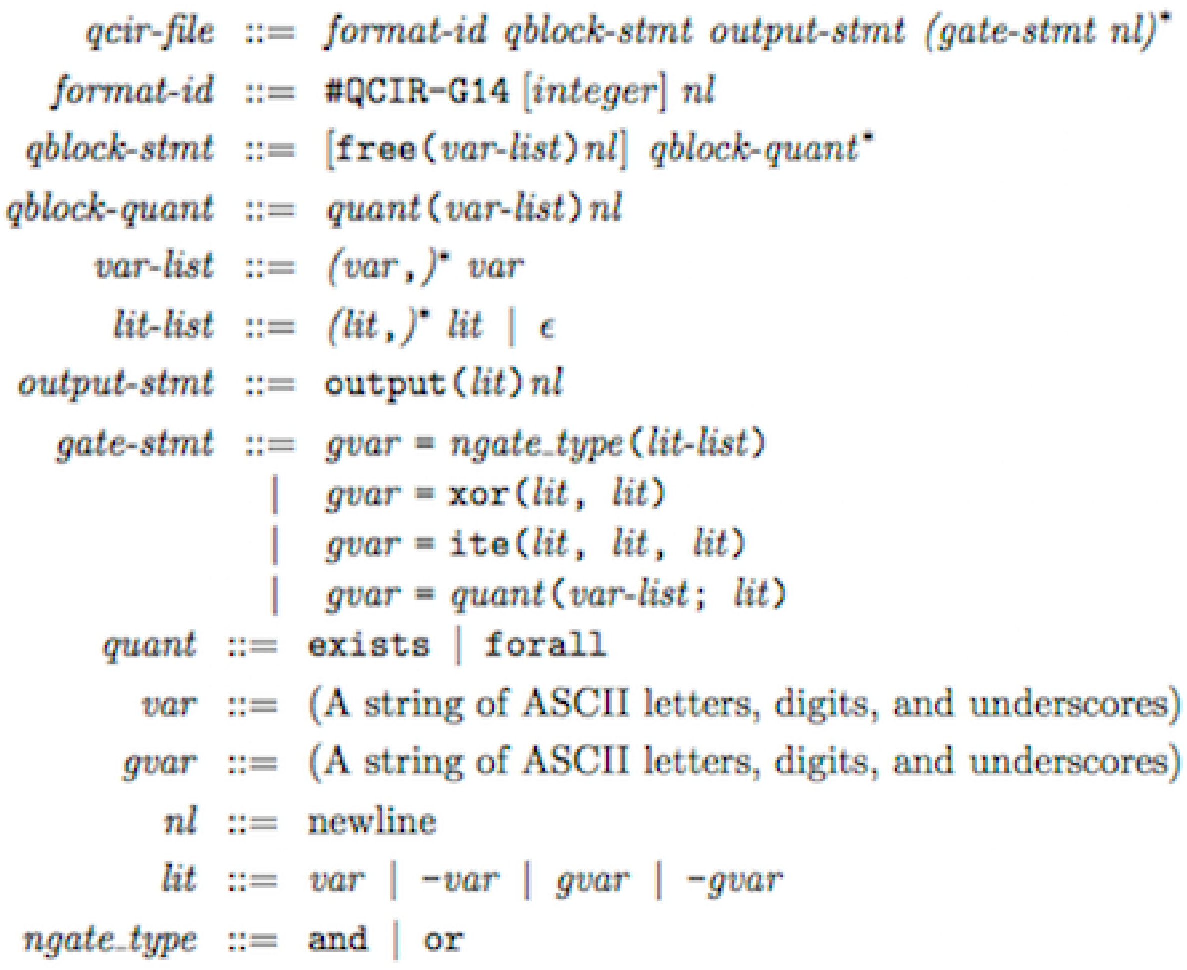
Boolean Satisfiability Problem (SAT) are hard to solve. It is believed that no algorithm can solve all Boolean Satisfiability Problems efficiently. According to Cook-Levin theorem, the Boolean Satisfiability Problem is NP-complete, which means that any problem in class NP is polynomial time reducible to the Boolean Satisfiability Problem. However, the decision problem of QBF is PSPACE-complete, as shown in the previous section. Thus, according to the definition of PSPACE-complete, the decision problem of QBF is in PSPACE and is PSPACE hard. Since NP $\subset$ PSPACE and NP is believed to be not equal to PSPACE, we know that PSPACE problems are harder than NP problems. That is to say, the decision problems of QBF are even harder than satisfiability problems.

2.10 QBF solvers vs. SAT solvers

SAT solvers produce a satisfying assignment for a formula in conjunctive normal form (CNF) if such an assignment exists [4]. SAT solvers are widely used to solve NP problems. Compared to SAT solvers, QBF solvers are not widely used in practice. Both SAT and TQBF are believed to be computationally hard problems, so what makes the difference in application? In fact, SAT solvers are successful in practice because the hard instances don't seem to arise. While SAT solvers need to solve satisfiability checking problems, a QBF solver also has to solve validity checking problems which depend on the variable quantification. SAT solvers require input in conjunctive normal form (CNF). Thus, one way to construct QBF solvers is to extend the functions of SAT solvers. By doing so, the QBF solvers would require that formulas be converted into prenex conjunctive normal form. However, typically, it is hard to convert QBF into CNF. Translating a formula to CNF would introduce new existentially quantified variables but not unversally quantified variables, which makes it hard for QBF solvers to detect when a formula becomes true [8]. Non-CNF format is more flexible and allows more freedom to encode decision problems of QBF. A detailed comparison of CNF and non-CNF is in section 3.10. Thus, QBF solvers which read in non-clausal input are introduced. Though using non-clausal input might be less satisfying in generating the result, it is believed that the advantage in encoding of these new type of QBF solvers outweighs the disadvantage in practice [6]. Thus, in this paper, we use GhostQ, a QBF solver which accepts non-CNF input, to solve decision problems of QBF.

2.11 A QBF solver: GhostQ

Syntax: The input to the QhostQ solver is a QCIR formula. QCIR formulas are defined by the BNF grammar below. (The listing of the grammar is reproduced from [7].)



```
   qcir-file ::= format-id qblock-stmt output-stmt (gate-stmt nl)*
   format-id ::= #QCIR-G14 [integer] nl
 qblock-stmt ::= [free(var-list)nl] qblock-quant*
qblock-quant ::= quant(var-list)nl
    var-list ::= (var,)* var
    lit-list ::= (lit,)* lit | ε
 output-stmt ::= output(lit)nl
   gate-stmt ::= gvar = ngate_type(lit-list)
               | gvar = xor(lit, lit)
               | gvar = ite(lit, lit, lit)
               | gvar = quant(var-list; lit)
       quant ::= exists | forall
         var ::= (A string of ASCII letters, digits, and underscores)
        gvar ::= (A string of ASCII letters, digits, and underscores)
          nl ::= newline
         lit ::= var | -var | gvar | -gvar
  ngate_type ::= and | or
```

The output is a winning strategy for the first player if one exists. We will see later the exact form the output takes.

Example:

Take the formula $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \ ((x_1 \oplus x_2) \wedge ((y_1 \wedge y_2) \vee (\neg y_1 \wedge \neg y_2) \vee (y_1 \wedge \neg x_1) \vee (y_2 \wedge \neg x_1)))$ as an example. We encode the formula in QCIR format as the following:

```
exists(x1,x2)
forall(y1,y2)
output(g1)
g2=xor(x1,x2)
u1=and(y1,y2)
u2=and(-y1,-y2)
u3=and(y1,-x1)
u4=and(y2,-x1)
g3=or(u1,u2,u3,u4)
g1=and(g2,g3)
```

Here, $g1$ is the output gate variable and $u1, u2, u3, u4, g2, g3$ are intermediate gate variables. This encoding follows the rules of QCIR format. [7]

The output contains the winning strategy for the first player: list(list(x1, false()), list(x2, true())) This means that x_1 = false, x_2 = true is a solution to the decision problem of QBF. To show that the solution is valid, we need to check if $((x_1 \oplus x_2) \wedge ((y_1 \wedge y_2) \vee (\neg y_1 \wedge \neg y_2) \vee (y_1 \wedge \neg x_1) \vee (y_2 \wedge \neg x_1)))$ is true for all possible values of y_1 and y_2 when x_1 = false, x_2 = true. There are 4 possible cases:

(1) $y_1 = \text{true}$, $y_2 = \text{true}$

(2) $y_1 = \text{true}$, $y_2 = \text{false}$

(3) $y_1 = \text{false}$, $y_2 = \text{true}$

(4) $y_1 = \text{false}$, $y_2 = \text{false}$

Since $x_1 = \text{false}$, $x_2 = \text{true}$, we have $x_1 \oplus x_2 \equiv \text{true}$. Thus, we only need to check if $\phi = (y_1 \wedge y_2) \vee (\neg y_1 \wedge \neg y_2) \vee (y_1 \wedge \neg x_1) \vee (y_2 \wedge \neg x_1)$ is true in the 4 cases above when $x_1 = \text{false}$, $x_2 = \text{true}$. Since ϕ is a disjunction of 4 conjunctions, ϕ would be true if the value of one of its conjunction is true.

(1) If $y_1 = \text{true}$, $y_2 = \text{true}$, then $y_1 \wedge y_2 \equiv \text{true}$. Thus, ϕ is true.

(2) If $y_1 = \text{true}$, $y_2 = \text{false}$, since x_1 is false, then $y_1 \wedge \neg x_1 \equiv \text{true}$. Thus, ϕ is true.

(3) If $y_1 = \text{false}$, $y_2 = \text{true}$, since x_1 is false, then $y_2 \wedge \neg x_1 \equiv \text{true}$. Thus, ϕ is true.

(4) If $y_1 = \text{false}$, $y_2 = \text{false}$, then $\neg y_1 \wedge \neg y_2 \equiv \text{true}$. Thus, ϕ is true.

Thus, $x_1 = \text{false}$, $x_2 = \text{true}$ is a solution to the formula $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \ ((x_1 \oplus x_2) \wedge ((y_1 \wedge y_2) \vee (\neg y_1 \wedge \neg y_2) \vee (y_1 \wedge \neg x_1) \vee (y_2 \wedge \neg x_1)))$.

3. Solve simple geography games

3.1 Introduction

In a geography game, two players take turns to name cities from all over the world. At each round, the player must choose a city beginning with the letter that is the same as the last letter of the previous city's name given by the other player. Repetition is not allowed. The game starts with some designated city and ends if one of the player is unable to give a city name to continue the game [3].

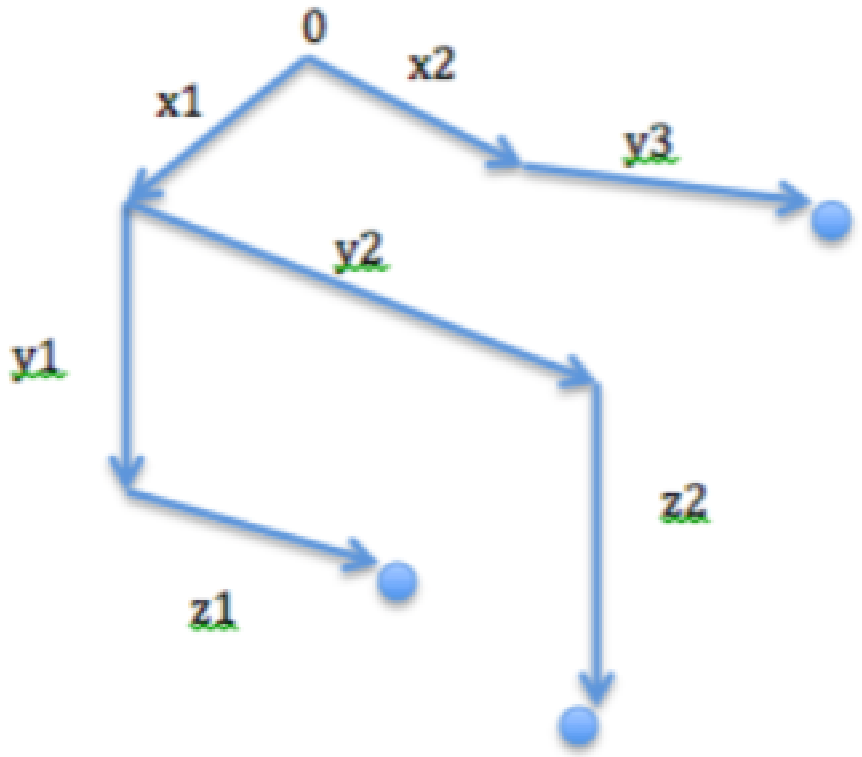
3.2 Methodology

As we discussed in section 2.8, TQBF is PSPACE-complete. Therefore, according to the definition of PSPACE-complete, every PSPACE problem can be encoded as instances of quantified Boolean formula. Since the problem of determining whether a given player has a winning strategy for geography games is in PSPACE, we can encode these games as instances of quantified Boolean formulas and use a QBF solver to solve the instances. In the following sections, we will discuss how to encode and solve a geography game.

3.3 Encoding

A geography game can be modeled by a directed graph with a designated starting node. In the directed graph, the nodes are the cities. Suppose node 1 is city X and node 2 is city Y. If city X ends with the same letter that begins city Y, then there's an edge from node 1 to node 2.

For example, the following is a directed graph with vertex 0 as the starting node.



The graph has starting node 0 and three ending nodes. The maximum number of rounds the game can last is 3. We use x, y, z to denote possible moves at the first, second and third round of the geography game. Let 1,2 3 be the index of possible ways of move at each round. Let's assume player 1 plays the first step of the game. The purpose of the game is to decide whether the first player has a winning strategy. If he has a strategy to win, then we need to find the winning strategy. For geography games, a winning strategy for player 1 means that player 1 can successfully reply to all of player 2's replies. From the graph, we could see that at the first round, player 1 has 2 possible moves: x_1 or x_2. If player 1 moves along x_1, then at the second round, player 2 has 2 possible moves: y_1 or y_2. If player 1 moves along x_2, then player 2 has 1 possible move y_3. Since y_3 leads to one of the ending node of the graph, payer 2 wins the game. If player 2 moves along y_1 at the second round, then player 1 has 1 possible move at the third round: z_1. Since z_1 leads to one of the ending node of the graph, payer 1 wins the game. Similarly, if player 2 moves along y_2 at the second round, then player 1 will win the game by moving along z_2 to reach the ending node.

Rules of the Game to encode:

1. Players can visit exactly one city at each round of the games. That is to say, exactly one node in the directed graph is visited at each round of the game.
2. Players cannot visit the cities which have been visited before. That is to say, the node visited at each round was not visited at the previous rounds of the game.
3. Players should visit the city adjacent to the city being visited at the previous round. That is to say, the node visited at each round is adjacent to the node visited at the previous round.
4. Let node 0 be the starting city. Then the node visited at the first round should be adjacent to node 0.

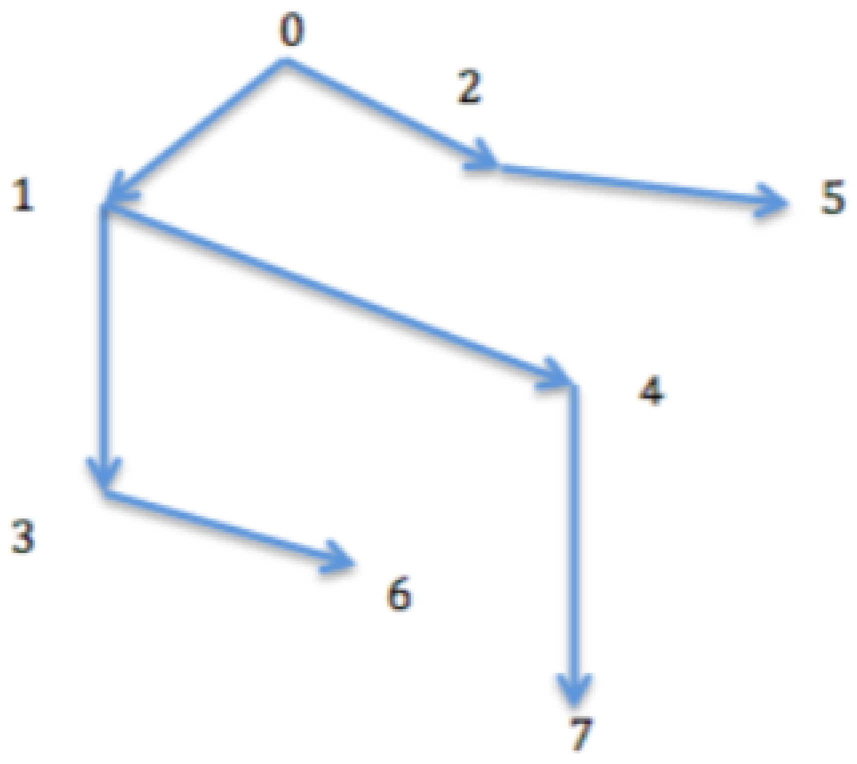
3.4 Implementation

As mentioned in section 2.11, the QBF solver ghostQ reads in a QCIR specification file and outputs a file containing a winning strategy for the first player if one exists. Thus, our goal is to write a program which can generate the QCIR specification file for arbitrary depth quantified Boolean formulas of geography game played on a graph automatically. After that, we can use the specification file to run ghostQ which would generate the output file for the geography game. The following is how we write the program to generate QCIR specification files.

Input:

The program will read in the information of a graph from a .txt file and convert the information into a Python dictionary. The dictionary will consist of key value pairs where each key is a vertex, and the value associated with the key is a list of the neighbors of the vertex. By convention, the vertices are numbered 0, .., n and 0 is the start vertex. To generate the QCIR specification file for arbitrary depth quantified Boolean formulas of geography game, we need to determine the variables for each round. Because there are at most 26 alphabet letters, we cannot represent each round with a new letter when the maximum number of rounds a geography game can last exceeds 26. Thus, we cannot apply the way of encoding the variables described in section 3.3. To solve this problem, we instead encode the variables for each round of the game as the following: first-round variables are $x_{1\_1}, x_{1\_2}, x_{1\_3}, ...$; second-round variables are $x_{2\_1}, x_{2\_2}, x_{2\_3}, ...etc.$ For the meaning of each variable, we let x_{i_j} be true if on round i the player moves to vertex j. In addition, for all the games, we suppose that player 1 plays first. Furthermore, the QCIR specification file needs to have an output gate variable g and intermediate gate variables $g_1, g_2, g_3, ...$ In our program, we use a global count to keep track of the next gate variable to use.

The new directed graph that represents the same geography game as described on page 17 is shown on the next page.



The input .txt file for our Python program looks like:

```
3
0 1 2
1 0 3 4
2 0 5
3 1 6
4 1 7
5 2
6 3
7 4
```

In the first row, 3 is the maximum number of rounds the game (represented by the above directed graph) can last. The second row means that from node 0, players can go to node 1 or node 2. The rest rows in the .txt file can be interpreted in similar way.

We covert the input in .txt into a Python dictionary and a number to make it easier to retrieve information and modify the graph. The dictionary indicates the nodes and edges in the directed graph. The number indicates the maximum number of rounds the game can last. The following is what we get after converting the .txt file on the previous page:
{0:[1,2], 1:[0,3,4], 2:[0, 5], 3:[1, 6], 4:[1, 7] , 5:[2], 6:[3], 7:[4]}, 3

Constraints:
Let us take the game represented by the directed graph on the previous page as an example to show how we encode the 4 constraints described on page 18. Recall that we let x_{i_j} be true if on round i the player moves to node j.

a. Exactly one of the variables at a given level is true i.e. exactly one node is visited at each level

In order to make sure that exactly one node at this level is visited, the following 2 conditions need to be satisfied: 1. at least one node at this level is visited
2. no two nodes at this level can both be visited

Condition 1 can be encoded as the following:
$(x_{1\_1} \vee x_{1\_2} \vee x_{1\_3} \vee x_{1\_4} \vee x_{1\_5} \vee x_{1\_6} \vee x_{1\_7}) \wedge$
$(x_{2\_1} \vee x_{2\_2} \vee x_{2\_3} \vee x_{2\_4} \vee x_{2\_5} \vee x_{2\_6} \vee x_{2\_7}) \wedge$

$(x_{3\_1} \vee x_{3\_2} \vee x_{3\_3} \vee x_{3\_4} \vee x_{3\_5} \vee x_{3\_6} \vee x_{3\_7})$

Condition 2 can be encoded as the following:
$(-x_{1\_1} \vee -x_{1\_2}) \wedge (\neg x_{1\_1} \vee \neg x_{1\_3}) \wedge (\neg x_{1\_1} \vee \neg x_{1\_4}) \wedge (\neg x_{1\_1} \vee \neg x_{1\_5}) \wedge (\neg x_{1\_1} \vee \neg x_{1\_6}) \wedge (\neg x_{1\_1} \vee \neg x_{1\_7}) \wedge (\neg x_{1\_2} \vee \neg x_{1\_3}) \wedge (\neg x_{1\_2} \vee \neg x_{1\_4}) \wedge (\neg x_{1\_2} \vee \neg x_{1\_5}) \wedge (\neg x_{1\_2} \vee \neg x_{1\_6}) \wedge (\neg x_{1\_2} \vee \neg x_{1\_7}) \wedge (\neg x_{1\_3} \vee \neg x_{1\_4}) \wedge (\neg x_{1\_3} \vee \neg x_{1\_5}) \wedge (\neg x_{1\_3} \vee \neg x_{1\_6}) \wedge (\neg x_{1\_3} \vee \neg x_{1\_7}) \wedge (\neg x_{1\_4} \vee \neg x_{1\_5}) \wedge (\neg x_{1\_4} \vee \neg x_{1\_6}) \wedge (\neg x_{1\_4} \vee \neg x_{1\_7}) \wedge (\neg x_{1\_5} \vee \neg x_{1\_6}) \wedge (\neg x_{1\_5} \vee \neg x_{1\_7}) \wedge (\neg x_{1\_6} \vee \neg x_{1\_7})$

b. The node visited at a certain level was not visited at all previous levels
This constraint can be encoded as the following:
$(\neg x_{1\_1} \vee \neg x_{2\_1}) \wedge (\neg x_{1\_2} \vee \neg x_{2\_2}) \wedge (\neg x_{1\_3} \vee \neg x_{2\_3}) \wedge (\neg x_{1\_4} \vee \neg x_{2\_4}) \wedge (\neg x_{1\_5} \vee \neg x_{2\_5}) \wedge (\neg x_{1\_6} \vee \neg x_{2\_6}) \wedge (\neg x_{1\_7} \vee \neg x_{2\_7}) \wedge (\neg x_{1\_1} \vee \neg x_{3\_1}) \wedge (\neg x_{1\_2} \vee \neg x_{3\_2}) \wedge (\neg x_{1\_3} \vee \neg x_{3\_3}) \wedge (\neg x_{1\_4} \vee \neg x_{3\_4}) \wedge (\neg x_{1\_5} \vee \neg x_{3\_5}) \wedge (\neg x_{1\_6} \vee \neg x_{3\_6}) \wedge (\neg x_{1\_7} \vee \neg x_{3\_7}) \wedge (\neg x_{2\_1} \vee \neg x_{3\_1}) \wedge (\neg x_{2\_2} \vee \neg x_{3\_2}) \wedge (\neg x_{2\_3} \vee \neg x_{3\_3}) \wedge (\neg x_{2\_4} \vee \neg x_{3\_4}) \wedge (\neg x_{2\_5} \vee \neg x_{3\_5}) \wedge (\neg x_{2\_6} \vee \neg x_{3\_6}) \wedge (\neg x_{2\_7} \vee \neg x_{3\_7})$

c. The node visited at a certain level is adjacent to the node visited at the previous level
This constraint can be encoded as the following:
$(x_{2\_1} \rightarrow (x_{1\_3} \vee x_{1\_4})) \wedge (x_{2\_2} \rightarrow x_{1\_5}) \wedge (x_{2\_3} \rightarrow (x_{1\_1} \vee x_{1\_6})) \wedge (x_{2\_4} \rightarrow (x_{1\_1} \vee x_{1\_7})) \wedge (x_{2\_5} \rightarrow x_{1\_2}) \wedge (x_{2\_6} \rightarrow x_{1\_3}) \wedge (x_{2\_7} \rightarrow x_{1\_4}) \wedge (x_{3\_1} \rightarrow (x_{2\_3} \vee x_{2\_4})) \wedge (x_{3\_2} \rightarrow x_{2\_5}) \wedge (x_{3\_3} \rightarrow (x_{2\_1} \vee x_{2\_6})) \wedge (x_{3\_4} \rightarrow (x_{2\_1} \vee x_{2\_7})) \wedge (x_{3\_5} \rightarrow x_{2\_2}) \wedge (x_{3\_6} \rightarrow x_{2\_3}) \wedge (x_{3\_7} \rightarrow x_{2\_4})$

d. The node visited at the first level is adjacent to 0
This constraint can be encoded as $(x_{1\_1} \vee x_{1\_2})$

3.5 Result

a. Output

We run GhostQ with the QCIR file we have generated for the directed graph on page 19 and get a cqbf file which contains the following:

Seed: 1. true. Bt: 1, D: 5. R: 0, P: 378, w: 448, C: 0, T: 0.000 s. true()

Interpretation: The first "true" after "Seed" means that there exists a winning strategy for player 1 for the geography game.

To find out the winning strategy, we run GhostQ with the cqbf file to generate the file which contains the strategy:

list(list(x_{1_1}, true()), list(x_{1_2}, false()), list(x_{1_3}, false()),
list(x_{1_4}, false()), list(x_{1_5}, false()), list(x_{1_6}, false()),
list(x_{1_7}, false()), list(x_{3_1}, false()), list(x_{3_2}, false()),
list(x_{3_3}), list(x_{3_4}, false()), list(x_{3_5}, false()),
list(x_{3_6}, ite(x_{2_4}, false(), true())),
list(x_{3_7}, ite(x_{2_4}, true(), false())))

b. Interpretation

According to the output above, the winning strategy for player 1 is the following:
x_{1_1}, true() means that player1 should go to node 1 at the first round.
list(x_{3_6}, ite(x_{2_4}, false(), true())) means that if player 2 does not go to node 4 at the second round, then player 1 should go to node 6 at the second round. This is the same as if player 1 goes to node 3 at the second round, then player 1 should go to node 6 at the third round because at the second round, player 2 can only go to node 3 if he does not go to node 4 given that player 1 goes to node 1 at the first

round.
list(x_{3_7}, ite(x_{2_4}, true(), false())) means that if player 2 goes to node 4 at the second round, then player 1 should go to node 6 at the second round.
After verification, we can conclude that the above strategy is a winning strategy for player 1 for the geography game represented by the directed graph on page 20.

c. Evaluation

We run GhostQ with different geography games to evaluate the performance of GhostQ with our QCIR files as input. The 4 columns are the number of nodes in the directed graph, the maximum number of rounds that the geography game can last, the number of variables (including the gate variables) in the QCIR specification file, and the number of lines in the QCIR specification file.

num of nodes	max number of rounds	num of variables	num of lines in spec file
5	3	103	92
8	4	213	190
8	6	297	262
10	5	382	343
20	10	2524	2328

We could see that when the number of rounds and the number of nodes become large, the number of lines in the QCIR specification file grows quickly.

Then we test the performance of GhostQ. When the maximum number of rounds a geography game can last increases to around 10, GhostQ starts to run slowly with our QCIR input. When the QCIR file becomes too large, GhostQ may fail due to the lack of stack space.

3.6 Game Interface

We write a Python program, so we can play the geography game interactively. Given a geography game, we let the computer be the first player. We run GhostQ to generate the strategy for the computer to win the game if winning strategy exists, and then extract the first step for the computer to take. Suppose computer goes to node i, then we let node i be the new starting node and modify the directed graph accordingly. For example, we need to remove from the graph the original starting node and the edges going from the original starting node . With the new graph and the new starting point, we then ask the human player to choose which node he wants to go to. Then we modify the directed graph accordingly. After that, we run GhostQ again using the new graph and generate the strategy for the computer...In this way, we are able to play the geography games interactively.

4. References

[1] Gary William Flake and Eric B. Baum. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". Theoretical Computer Science, 270(1-2):895-911, January 2002.

[2] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In Proceedings of the 13th IEEE Symposium on Switching and Automata Theory, pages 125-129. IEEE, New York, 1972.

[3] Michael Sipser, Introduction to the Theory of Computation, Second Edition, Thomson Course Technology, Boston, 2006.

[4] Frank van Harmelen , Vladimir Lifschitz , Bruce Porter, Handbook of Knowledge Representation, Elsevier Science, San Diego, 2007.

[5] Kleine Buning, H., and Bubeck, U. 2009. Theory of quantified Boolean formulas. In Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press. 735-760.

[6] Non-CNF QBF Solving with QCIR. Charles Jordan, Will Klieber, and Martina Seidl. In Beyond NP 2016.

[7] QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas, QBF Gallery 2014

[8] Formal Verification Using Quantified Boolean Formulas (QBF), William Klieber, 2014