Bundik Bettina Vivien - 5118FG13-8

Reliable Software Assignment 3 - 2018/8/9

Knight's Tour Problem

Problem analysis

The Knight's Tour problem is the mathematical problem of finding a Knight's Tour. This tour can be defined as a specific sequence of moves of the knight on a chessboard where every square on the board is visited only once.

If the square, that the knight stepped on last, is one move away from the beginning square, then its tour is closed, otherwise open. Here we only consider the solution for an open tour.

By using the following encoding of the problem:

With indexes $i = 1 \dots N^2,\ j, k = 1 \dots N$

If the knight is at square (j, k) at time i $\quad x_{100i+10j+k} = 1$

Otherwise $\quad x_{100i+10j+k} = 0$

We can define constraints for these CNF variables such as:

1.

At time i, the knight is at one square only and not at two or more positions at the same time. This constraint will result in a list of 2-CNF variables and can be defined as follows:

For every $i = 1 \dots N^2$

$$\neg x_{ijk} \vee \neg x_{ij'k'} = True$$

For every $(j, k), (j', k')$ pairs where $j, k, j', k' = 1 \dots N$ and $j \neq j' \wedge k \neq k'$.

2.

Considering the knight's valid movements on the chessboard, there are maximum 8 options.

If the knight at time i is at position (j, k) , then at time $i + 1$, its next position is decided according to these 8 possible moves.

This constraint can generate a list of variables of 3-CNF to 9-CNF.

For every $i = 1 \dots N^2 - 1$

$$\neg x_{ijk} \vee (\vee x_{i+1j'k'}) = True$$

For every $(j, k), (j', k')$ pairs where $j, k = 1 \dots N$ and (j', k') pairs come from the following 8 rules (if the step is valid):

$(j', k') = (j + 2, k + 1)$

$(j', k') = (j + 1, k + 2)$

$(j', k') = (j - 1, k + 2)$

$(j', k') = (j - 2, k + 1)$

$(j', k') = (j - 2, k - 1)$

$(j', k') = (j - 1, k - 2)$

$(j', k') = (j + 1, k - 2)$

$(j', k') = (j + 2, k - 1)$

3.

At each time, every square is visited only once, meaning that we need to iterate the time in this case, not the position.

This will result in a list of 2-CNF variables again.

For every $j, k = 1 \dots N$

$$\neg x_{ijk} \vee \neg x_{i'jk} = True$$

For every i and i' where $i, i' = 1 \dots N^2$ and $i \neq i'$.

4.

Another important rule is that no square is left unvisited so we have to exclude the case where there is no knight and the sequence of moves is empty.

This will generate a list of N^2CNF-variables.

$$\bigvee x_{ijk} = True$$

For every $i = 1 \dots N^2, j, k = 1 \dots N$.

For constraints 2 and 4, we transform the CNF variables into 3-CNF, where necessary, with Tseytin Transformation.

Appendix

Used language and version:
C#
.NET Framework v4.5.2

Used SAT solver
MiniSat 2.2

Program.cs

```
using System;

namespace KnightsTour
{
    class Program
    {
        static void Main(string[] args)
        {
            KnightsModel model = new KnightsModel(4);
            model.GenerateDIMACS();
        }
    }
}
```

KnightsModel.cs

```
using System;
using System.Collections.Generic;

namespace KnightsTour
{
    public class Position
    {
        public Int32 x;
        public Int32 y;

        public Position()
        {
            x = 0;
            y = 0;
        }

        public Position(Int32 _x, Int32 _y)
        {
            x = _x;
            y = _y;
        }
    }

    public class Variable
    {
        public Int32 i;
        public Int32 j;
        public Int32 k;
```

```
    public Boolean value;

    public Variable(Int32 _i, Int32 _j, Int32 _k, Boolean _value)
    {
        i = _i;
        j = _j;
        k = _k;
        value = _value;
    }
}

public class KnightsModel
{
    private Int32 n;
    private List<List<Variable>> cnf1;
    private List<List<Variable>> cnf2;
    private List<List<Variable>> cnf3;
    private List<List<Variable>> cnf4;
    private List<Int32> move_x;
    private List<Int32> move_y;

    public KnightsModel (Int32 _n)
    {
        n = _n;

        cnf1 = new List<List<Variable>>();
        cnf2 = new List<List<Variable>>();
        cnf3 = new List<List<Variable>>();
        cnf4 = new List<List<Variable>>();

        move_x = new List<Int32>();
        move_y = new List<Int32>();

        move_x.Add(2);
        move_x.Add(1);
        move_x.Add(-1);
        move_x.Add(-2);
        move_x.Add(-2);
        move_x.Add(-1);
        move_x.Add(1);
        move_x.Add(2);

        move_y.Add(1);
        move_y.Add(2);
        move_y.Add(2);
        move_y.Add(1);
        move_y.Add(-1);
        move_y.Add(-2);
        move_y.Add(-2);
        move_y.Add(-1);
    }

    public void GenerateDIMACS()
    {
        CNF_Constraint1();
```

```
            CNF_Constraint2();
            CNF_Constraint3();
            CNF_Constraint4();
            WriteAll();
        }

        private Boolean ValidCell(Position p)
        {
            return p.x >= 1 && p.y >= 1 && p.x <= n && p.y <= n;
        }

        private Int32 ConvertToNumber(Int32 i, Int32 j, Int32 k)
        {
            return 100 * i + 10 * j + k;
        }

        private void ConvertTo_3CNF(ref List<List<Variable>> cnf)
        {
            List<List<Variable>> cnf_new = new List<List<Variable>>();
            Int32 count_i = n * n + 1;
            Int32 count_j = n + 1;
            Int32 count_k = n + 1;

            for (int i = 0; i < cnf.Count; i++)
            {
                if (cnf[i].Count <= 3)
                    cnf_new.Add(cnf[i]);
                else
                {
                    List<Variable> tmp = new List<Variable>();
                    tmp.Add(cnf[i][0]);
                    tmp.Add(cnf[i][1]);
                    tmp.Add(new Variable(count_i, count_j, count_k, true));
                    cnf_new.Add(tmp);

                    for (int j = 2; j < cnf[i].Count; j++)
                    {
                        List<Variable> tmp2 = new List<Variable>();

                        count_i++;
                        count_j++;
                        count_k++;

                        tmp2.Add(new Variable(count_i, count_j, count_k, false));
                        tmp2.Add(cnf[i][j]);
                        tmp2.Add(new Variable(count_i + 1, count_j + 1, count_k +
1, true));

                        cnf_new.Add(tmp2);
                    }
                }
            }

            cnf = new List<List<Variable>>(cnf_new);
        }
```

```
        public void CNF_Constraint1()
        {
            for (int i = 1; i <= n * n; i++)
            {
                Variable knightpos = new Variable(i, (int)Math.Ceiling((double)i /
n), i % n, false);

                for (int j = 1; j <= n; j++)
                {
                    for (int k = 1; k <= n; k++)
                    {
                        if (j != knightpos.j && k != knightpos.k)
                        {
                            List<Variable> tmp = new List<Variable>();
                            tmp.Add(knightpos);
                            tmp.Add(new Variable(i, j, k, false));
                            cnf1.Add(tmp);
                        }
                    }
                }
            }
        }

        public void CNF_Constraint2()
        {
            for (int i = 1; i <= n * n - 1; i++)
            {
                Variable knightpos = new Variable(i, (int)Math.Ceiling((double)i /
n), i % n, false);

                List<Variable> tmp = new List<Variable>();
                tmp.Add(knightpos);

                for (int m = 0; m < move_x.Count; m++)
                {
                    Position next = new Position(knightpos.j + move_x[m],
knightpos.k + move_y[m]);
                    if (ValidCell(next))
                        tmp.Add(new Variable(i + 1, next.x, next.y, true));
                }

                cnf2.Add(tmp);
            }

            ConvertTo_3CNF(ref cnf2);
        }

        public void CNF_Constraint3()
        {
            for (int j = 1; j <= n; j++)
            {
                for (int k = 1; k <= n; k++)
                {
                    for (int i1 = 1; i1 <= n * n; i1++)
```

```
                    {
                        Variable x = new Variable(i1, j, k, false);
                        for (int i2 = i1 + 1; i2 <= n * n; i2++)
                        {
                            List<Variable> tmp = new List<Variable>();
                            tmp.Add(x);
                            tmp.Add(new Variable(i2, j, k, false));
                            cnf3.Add(tmp);
                        }
                    }
                }
            }
        }

        public void CNF_Constraint4()
        {
            List<Variable> tmp = new List<Variable>();
            for (int i = 1; i <= n * n; i++)
            {
                for (int j = 1; j <= n; j++)
                {
                    for (int k = 1; k <= n; k++)
                    {
                        tmp.Add(new Variable(i, j, k, true));
                    }
                }
            }

            cnf4.Add(tmp);

            ConvertTo_3CNF(ref cnf4);
        }

        private void ConvertToDIMACS(List<List<Variable>> cnf, ref List<String>
dimacs)
        {
            for (int i = 0; i < cnf.Count; i++)
            {
                String row = "";
                for (int v = 0; v < cnf[i].Count; v++)
                {
                    String variable;
                    if (cnf[i][v].value)
                        variable = ConvertToNumber(cnf[i][v].i, cnf[i][v].j,
cnf[i][v].k).ToString();
                    else
                        variable = "-" + ConvertToNumber(cnf[i][v].i, cnf[i][v].j,
cnf[i][v].k).ToString();

                    row += variable + " ";
                }

                row += "0";
                dimacs.Add(row);
            }
```

```
        }

        public void WriteAll()
        {
            List<String> dimacs = new List<String>();

            ConvertToDIMACS(cnf1, ref dimacs);
            ConvertToDIMACS(cnf2, ref dimacs);
            ConvertToDIMACS(cnf3, ref dimacs);
            ConvertToDIMACS(cnf4, ref dimacs);

            using (System.IO.StreamWriter file =
                new System.IO.StreamWriter("knightstour.dimacs"))
            {
                file.WriteLine("p cnf 1 2");
                for (int i = 0; i < dimacs.Count; i++)
                {
                    file.WriteLine(dimacs[i]);
                }
            }
        }
    }
}
```