GV HƯỚNG DẪN:

[Email address]

Group: AI00

18127008 – Lê Mạnh Hoàng  
19127197 – Hoàng Thị Quỳnh Liên  
19127273 – Huỳnh Thị Mỹ Thanh

REPORT PROJECT 2

Introduction to AI

COLORING PUZZLE



**I. Generating CNFs:**

**The logical principles for generating CNFs:**

Based on the digit in the cell to calculate its combinatorics of surrounding cells which is the number of clauses of that cell.

In each clause of the main cell: The number of surrounding cells that are blue is equal to the digit in the main cell and other surrounding cells are red.

For example:

3

2

1

3

2

1

|  |  |  |
| --- | --- | --- |
| . | . | . |
| . | 3 | . |
| . | . | . |

The combinatorics of the cell (2,2) would be:

Assume that we make 3 cells: (1,1), (1,2), (2,1) are blue, others (1,3), (2,2), (2,3), (3,1), (3,2), (3,3) must be red. In contrast, if 6 cells were colored red, others would be blue.

Clause: ((1,1) ∧ (1,2) ∧ (2,1)) (~(1,3) ∧ ~(2,2) ∧ ~(2,3) ∧ ~(3,1) ∧ ~(3,2) ∧ ~(3,3)).

Convert to CNF:

( )

()

[ ∨ ∨ ∨ (

[ ∨ ( ) ]

[ ∨ ∨ ∨

[∨

)]

**Algorithm for generating CNF:**

For every cell that has a digit k and n adjacent cells, we need to generate clauses for both the red and blue case:

**1. Blue:**

We need to generate clauses. Each clause we choose a set of k – 1 adjacent cells and make sure they do not duplicate with any set. After that we have a list of the sets, we need to consider 2 ways: A ⬄ B

* With A => B, that means if the k – 1 adjacent cells are blue, the others must be red.

For example, we have the table below:

|  |  |  |
| --- | --- | --- |
|  | **2** |  |
|  |  |  |

The cell 2 contains digit 2, and the adjacent cells of that are 1, 3, 4, 5, 6.

Assume that we choose a set has the cell 1 (that means k – 1 (2-1) adjacent cells)

Then we have clauses:

(1 2) => (~3 ~4 ~5 ~6)

~1 ~2 (~3 ~4 ~5 ~6)

(~1 ~2 ~3) (~1 ~2 ~4) ~5) (~1 ~2 ~6)

That mean if the cell 2 and the adjacent cell 1 are blue, the others must be red.

* With B => A, if some adjacent cells are red, the k – 1 adjacent cells are blue. We have k clauses divided in 2 types:

+ The positive value of the cell not in the set.

+ The positive value of the cell in the set.

For example, we have the table below:

|  |  |  |
| --- | --- | --- |
|  | **2** |  |
|  |  |  |

The cell 2 contains digit 2, and the adjacent cells of that are 1, 3, 4, 5, 6.

Assume that we choose a set has the cell 1 in the list (that means k – 1 (2-1) adjacent cells)

Then we have clauses:

(~3 ~4 ~5 ~6) => (1 2)

(3 4 5 6) (1 2)

(3 4 5 6 (3 4 5 6

That mean if the cell 3, 4, 5, 6 are red, the others (2 and 1) must be blue.

**2. Red:**

We need to generate clauses. Each clause we choose a set of k adjacent cells and make sure they do not duplicate with any set. After that we have a list of the sets, we need to consider 2 ways: A ⬄ B

* With A => B, that means if the k – 1 adjacent cells are red, the others must be blue. We have n – k CNF clauses divided in 2 types:

+ The negative value of the cell, the negative values of cells in the set.

+ The negative values of cells in the set and a negative value of a cell not in the set. We have n-k-1(k is the number of the cells in the set, 1 is the number of the cells we are working) cells not in the set, so we have n-k-1 clauses for the second type.

For example, we have the table below:

|  |  |  |
| --- | --- | --- |
|  | **2** |  |
|  |  |  |

The cell 2 contains digit 2, and the adjacent cells of that are 1, 3, 4, 5, 6.

Assume that we choose a set has the cell 1 and 3

Then we have clauses:

(1 3) => (~2 ~4 ~5 ~6)

~1 ~3 (~2 ~4 ~5 ~6)

(~1 ~3 ~2) (~1 ~3 ~4) ~5) (~1 ~3 ~6)

That mean if the cell 3 and the adjacent cell 1 are blue, the others must be red.

* With B => A, if n – k – 1 adjacent cells are red, the k others must be blue.

For example, we have the table below:

|  |  |  |
| --- | --- | --- |
|  | **2** |  |
|  |  |  |

The cell 2 contains digit 2, and the adjacent cells of that are 1, 3, 4, 5, 6.

Assume that we choose a set has the cell 1 and 3

Then we have clauses:

(~2 ~4 ~5 ~6) => (1 3)

(2 4 5 6) (1 3)

(2 4 5 6 (2 4 5 6

That mean if the cell 2, 4, 5, 6 are red, the others (3 and 1) must be blue.

**The special case for the value of the cell:**

* If the digit is zero, we just need to generate clauses for the case which is the

cell in red color.

For example:

|  |  |  |
| --- | --- | --- |
|  | **0** |  |
|  |  |  |

We have adjacency cells 1,3,4,5,6 so that we have multiple clauses such as (~1 ~2 ~3 ~4 ~5 ~6).

* If the value is 1 and the cell is blue, we just have multiple clauses that satisfy the

condition which is (that cell) ^ ~ [ each cell in adjacency cells].

For example:

|  |  |  |
| --- | --- | --- |
|  | **1** |  |
|  |  |  |

* If the value of the cell is greater than the number of the adjacency cells, we do not

consider the red case.

* If the value of the cell subtracts the number of the adjacency cells greater than 1, we do not consider the green case.

A\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Input** | **Output** | **Time** | **Evaluate** |
| 3x3 | 3 5 3  4 6 4  2 3 2 | 1 1 1  1 0 1  0 1 0 | 0.007000 s |  |
| 5x5 | -1 5 6 -1 3  5 7 8 7 -1  4 6 8 7 5  5 6 7 7 -1  -1 4 5 5 -1 | 1 1 1 1 1  0 1 1 1 0  1 1 0 1 1  0 1 1 1 1  1 1 0 1 1 | 0.127009 s |  |
| 7x7 | -1 5 6 -1 3 2 -1  5 7 8 7 -1 3 1  4 6 8 7 5 2 -1  5 6 7 7 -1 3 0  -1 4 5 5 -1 2 0  3 3 2 2 2 2 -1  -1 1 0 0 -1 1 1 | 1 1 1 1 1 0 1  0 1 1 1 0 0 0  1 1 0 1 1 0 0  0 1 1 1 1 0 0  1 1 0 1 1 0 0  0 0 0 0 0 0 0  1 0 0 0 0 0 1 | 2.831627 s |  |
| 10x10 | -1 2 3 -1 -1 0 -1 -1 -1 -1  -1 -1 -1 -1 3 -1 2 -1 -1 6  -1 -1 5 -1 5 3 -1 5 7 4  -1 4 -1 5 -1 5 -1 6 -1 3  -1 -1 4 -1 5 -1 6 -1 -1 3  -1 -1 -1 2 -1 5 -1 -1 -1 -1  4 -1 1 -1 -1 -1 1 1 -1 -1  4 -1 1 -1 -1 -1 1 -1 4 -1  -1 -1 -1 -1 6 -1 -1 -1 -1 4  -1 4 4 -1 -1 -1 -1 4 -1 -1 | No solutions | 21.354550s |  |
|  |  |  |  |  |

Brute-force

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Input** | **Output** | **Time** | **Evaluate** |
| 3x3 | 3 5 3  4 6 4  2 3 2 | 1 1 1  1 0 1  0 1 0 | 0.002000 s |  |
| 5x5 | -1 5 6 -1 3  5 7 8 7 -1  4 6 8 7 5  5 6 7 7 -1  -1 4 5 5 -1 | 1 1 1 1 1  0 1 1 1 0  1 1 0 1 1  0 1 1 1 1  1 1 0 1 1 | 3.636741 s |  |
| 7x7 | -1 5 6 -1 3 2 -1  5 7 8 7 -1 3 1  4 6 8 7 5 2 -1  5 6 7 7 -1 3 0  -1 4 5 5 -1 2 0  3 3 2 2 2 2 -1  -1 1 0 0 -1 1 1 | 1 1 1 1 1 0 1  0 1 1 1 0 0 0  1 1 0 1 1 0 0  0 1 1 1 1 0 0  1 1 0 1 1 0 0  0 0 0 0 0 0 0  1 0 0 0 0 0 1 | 439.243063 s |  |
|  |  |  |  |  |
|  |  |  |  |  |

Backtracking

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Input** | **Output** | **Time** | **Evaluate** |
| 3x3 | 3 5 3  4 6 4  2 3 2 | 1 1 1  1 0 1  0 1 0 | 0.000996 s |  |
| 5x5 | -1 5 6 -1 3  5 7 8 7 -1  4 6 8 7 5  5 6 7 7 -1  -1 4 5 5 -1 | > 10 minutes | 600.000273 s |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
|  | Pros | Cons |
| A\* algorithm | - It is complete and optimally efficient.  - It is used to solve very complex search problems.  - It is one of the best heuristic search techniques. | - The speed execution of A\* search is dependent on the accuracy of the heuristic algorithm that is used to compute h(n).  - It is complete if the branching factor is finite and every action has a fixed cost.  - It has complexity problems. |
| Brute-force algorithm | - It is not limited to any specific domain of problems.  - It is used to solve simple and small problems. | - Brute force algorithms are slow.  - For real-time problems, algorithm analysis often goes above on O(N!) order of growth. |
| Backtracking algorithm | - If there exists any problems, it finds all the existing solutions.  - It is used to solve any problems. | - It is very slow.  - Large space complexity because of using recursion. |