# 人工智能-第二次课程作业报告

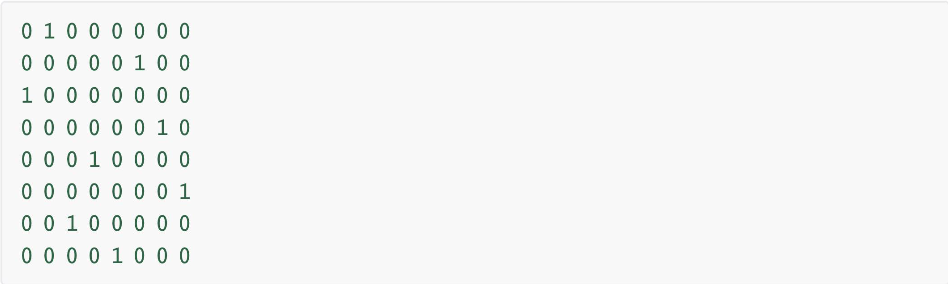
授课教师：张宇 作者：陆文韬-58122231

## 1 问题描述

### 1.1题目介绍

该问题为一个 8\*8 的棋盘，在该棋盘上摆放 8 个皇后，使其不能互相攻击，即任 意两个皇后都不能处于同一行、同一列或同一斜线(包含主对角线和次对角线)上。

最终会有多种摆法方案可以满足上述条件，例如:



### 1.2任务说明

本作业基于 Eight Queen Puzzle，目标是基于回溯搜索(backtrackSearch)和最小冲突搜索(minconflict)算法求该问题的解。该部分内容对应《Artificial Intelligence: A Modern Approach 3rd》中第六章的内容:Constraint Satisfaction Problems。

在项目 Assignment2 中实现 5 个函数，分别是对应实现回溯搜索算法的 backtrack 函数和实现最小冲突搜索的 minConflict 函数，以及辅助完成 minConflict 函数的 getConflicts 函数、chooseConflictVariable 函数和 getMinConflictValue 函数。

### 1.3实验环境

MacOS +Intel Core i5-5786U +Xcode

### 1.4评价标准

1、搜索算法的正确性: 算法得出的结果可以通过 main.cpp 中 searchTest 函数的验证。  
2、搜索算法的搜索时间：回溯搜索可以在 5 秒内跑出 puzzle 的结果，最小冲突搜索大部分可以在 200 步之内得出结果。

3、内存管理: 算法的内存消耗需要符合该算法应有的空间复杂度。

## 2 实验方案

1. 回溯搜索算法实现如下:

std::vector<Queen\*> search::backtrack(std::vector<Queen\*> assignment, Csp& csp)

{

/\*

\* TODO

\* Algorithm (Reference: Figure 6.5):

function BACKTRACK(assignment, csp) returns a solution, or failure

if assignment is complete then return assignment (use this condition: assignment.size() == csp.variables.size())

var<-SELECT-UNASSIGNED-VARIABLE(csp)

for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do

record csp state # csp.recode() require two variables, you need to create two local variables to store the state

if value is consistent with assignment then

assign value to var # use var->assign(value)

add var to assignment

inferences<-INFERENCE(csp, var, value) # use makeInference function here

if inferences != failure then

add inferences to assignment

result<-BACKTRACK(assignment, csp)

if result != failure then

return result

recover csp state (csp.recover)

remove {var = value} and inferences from assignment # use refresh(assignment)

return failure

\*/

**if**(assignment.size()==csp.variables.size())**return** assignment;

Queen\* var = search::selectUnassignedVariable(csp);

**for**(**auto** position:search::orderDomainValues(var, assignment, csp))

{

std::vector<Position> lastPositions = {};

std::vector<std::vector<Position>> lastDomains = {};

csp.record(lastPositions, lastDomains);

**if**(csp.consistent(position, assignment))

{

var->assign(position);

assignment.push\_back(var);

std::vector<Queen\*> inferences = search::makeInference(csp, var, position);

**if**(inferences != std::vector<Queen\*>({**NULL**}))

{

**for**(**auto** inference : inferences)

{

assignment.push\_back(inference);

}

std::vector<Queen\*> result = search::backtrack(assignment, csp);

**if**(result != std::vector<Queen\*>({**NULL**}))

{

**return** result;

}

}

}

csp.recover(lastPositions, lastDomains);

refresh(assignment);

}

**return** std::vector<Queen\*>({**NULL**});

}

2、最小冲突搜索算法实现如下:

std::vector<Queen\*> search::minConflict(Csp& csp, **int** maxSteps)

{

/\*

\* TODO

\* Algorithm (Reference: Figure 6.8):

function MIN-CONFLICTS(csp,max steps) returns a solution or failure

inputs: csp, a constraint satisfaction problem

max steps, the number of steps allowed before giving up

current<-an initial complete assignment for csp

for i = 1 to max steps do

if current is a solution for csp then # use isSolution

print how many steps used here

return current

var <- a randomly chosen conflicted variable from csp.VARIABLES # use chooseConflictVariable

value <- the value v for var that minimizes CONFLICTS(var, v, current , csp) # use getMinConflictValue

set var =value in current # use var->position = value

return failure

\*/

std::vector<Queen\*> current = {};

**for**(Queen\* q : csp.variables)

{

current.push\_back(q);

}

**for**(**int** i = 1;i <= maxSteps;++i)

{

**if**(isSolution(csp, current))

{

std::cout<<i<<"steps used here!"<<std::endl;

**return** current;

}

Queen\* var = search::chooseConflictVariable(csp);

Position value = search::getMinConflictValue(csp, var);

var->position = value;

}

**return** std::vector<Queen\*>({**NULL**});

}

**int** search::getConflicts(Csp& csp, Position& position)

{

/\*

\* TODO

\* 得到一个position在当前棋盘上的冲突数量

\* 注意：与position在同一列的queen的冲突不应该计算

\* 样例：

\* 0 1 0 0

1 0 0 0

0 0 1 0

0 0 0 1

\* Position{0, 0}的冲突数应该为3，因为它与{0, 1},{2, 2},{3, 3}冲突

\* Position{1, 0}的冲突数量应该为1，因为它与{0, 1}冲突

\*/

**int** numofConflicts = 0;

**for**(**auto** p : csp.variables)

{

**if**(p->position.col == position.col)

**continue**;

**if**(p->position.row == position.row)

++numofConflicts;

**if**(abs(p->position.row - position.row) == abs(p->position.col - position.col))

++numofConflicts;

}

**return** numofConflicts;

}

Queen\* search::chooseConflictVariable(Csp& csp)

{

/\*

\* TODO

\* 返回一个目前赋值的冲突数大于0的variable

\* 注意：冲突数大于0的variable可能有多个，需要随机选择

\* 样例：

\* 0 1 0 0

1 0 0 0

0 0 1 0

0 0 0 1

\* Queen1-4的冲突数都大于0，随机选择一个作为该函数的返回结果

\*/

std::vector<Queen\*> conflictVariable = {};

**for**(**auto** var : csp.variables)

{

**if**(getConflicts(csp, var->position) > 0)

{

conflictVariable.push\_back(var);

}

}

**return** conflictVariable[rand()%conflictVariable.size()];

**return** **NULL**;

}

Position search::getMinConflictValue(Csp& csp, Queen\* var)

{

/\*

\* TODO

\* 返回var的domian中，可以使冲突数最小的值

\* 注意：使冲突数最小的值可能有多个，需要随机选择，如果不随机选择问题可能会陷入局部稳定点并且该稳定点不是解

\* 样例：

\* 1 1 0 0

0 0 0 0

0 0 1 0

0 0 0 1

\* Queen1所在的位置的冲突数为3，它的domain为{[0-3], 0}。{1, 0},{2, 0},{3, 0}的冲突数都为1。

\* 需要从中随机选取一个作为返回值。

\*/

std::vector<Position> minConflict = {};

**for**(**auto** position : var->domain)

{

**if**(minConflict.empty())

{

minConflict.push\_back(position);

**continue**;

}

**if**(getConflicts(csp, position) == getConflicts(csp, minConflict.front()))

{

minConflict.push\_back(position);

}

**if**(getConflicts(csp, position) < getConflicts(csp, minConflict.front()))

{

minConflict.clear();

minConflict.push\_back(position);

}

}

**return** minConflict[rand()%minConflict.size()];

**return** Position::getUnassigned();

}

## 3实验结果

在进行了多次回溯搜索实验之后，我发现仅当问题规模是 2 或 3 时无法得到解决方案（这种情况下实际上不存在解决方案），此时结果显示为“No Valid Solution”。在其他所有情况下，我可以迅速找到正确的解决方案。在进行最小冲突搜索实验时，部分情况由于本身不存在解决方案，因此所有实验结果均显示为“No Valid Solution”。在其他情况下，当问题规模为 8 时，我能找到解决方案，具体所需步骤取决于实际情况。在这个实验环境下，两种方法没有明显差异，都能非常快速地给出结果。

## 4实验分析

在进行八皇后问题的解决时，若采用穷举法，我们需要尝试高达 16,777,216 种不同的布局。这种方法从将所有皇后放置在第一行的布局开始，并检查皇后之间是否存在冲突。如果存在冲突，就将最后一列的皇后下移一行，然后继续检查。如果最后一列的皇后已经在最后一行，则将前一列的皇后下移一行，并重新开始尝试最后一列。这种策略效率极低，因为它不是针对冲突进行调整，而是盲目地遍历所有可能的布局。

而在这次实验中使用的回溯算法则更为高效。其流程如下：

1. 如果所有变量都已经赋值，则结束；

2. 否则，选择一个未赋值的变量 v，并对其赋值。

3. 如果当前赋值不产生冲突，则继续赋值下一个变量；

4. 如果产生冲突，则尝试对 v 赋予另一个值。

5. 如果 v 的所有可能值都会产生冲突，那么回溯到前一个变量 u，并尝试对 u 重新赋值。

最小冲突算法的过程可以这样概括：

1. 首先生成一个随机的完整赋值。

2. 然后重复以下步骤，直至消除所有冲突。

3. 随机选择一个产生冲突的变量 v。

4. 为 v 重新赋值，选择使得与其他变量冲突最小的赋值。如果有多个赋值使冲突相同，则随机选择一个。

## 5 结论

解决八皇后问题的回溯算法遵循以下逻辑：遇到冲突就解决，没有冲突则继续前进，走投无路就回头，一直走到终点就得到答案。为了提高检测冲突的效率，可以创建标志数组来追踪每一行以及每条斜线上是否放置了皇后。放置一个新的皇后时，就设定标志；回溯时，移动一个已有的皇后并清除其标志。

另一方面，最小冲突搜索算法会选择系统中的一个特定局部来进行评估，并在这个局部范围内考察不同的调整方案，寻找局部的最佳解。然后它会转移到下一个局部，重复这一过程，直至找到一个全局满足条件的解决方案，这就是最终的答案。