# 人工智能-第2次课程作业报告

授课教师：张宇 作者：蒋雨初-58121102

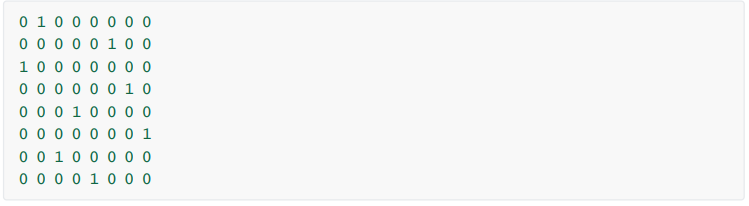
# 1 问题描述

## 1.1题目介绍

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 Approach3rd》中的第六章内容： Constraint Satisfaction Problems.。Eight Queen Puzzle

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（主对角线、次对角线）上。

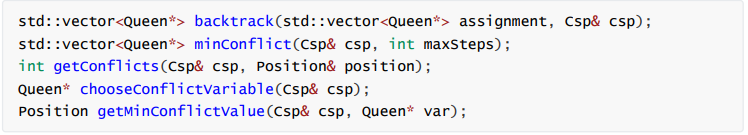
有多种摆放方案可以满足上述条件，例如



## 1.2任务说明

完成 Assignement2项目中的五个函数

search.cpp中

其中 minConflict函数依赖于最后三个。

## 1.3实验环境

### 设备规格

设备名称 LAPTOP-TFMBQKQ8

处理器 AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

机带 RAM 16.0 GB (15.4 GB 可用)

设备 ID 6CC35513-8821-49C9-A60B-C44F31F302A7

产品 ID 00342-35891-56086-AAOEM

系统类型 64 位操作系统, 基于 x64 的处理器

笔和触控 没有可用于此显示器的笔或触控输入

### 系统规格

版本 Windows 11 家庭中文版

版本 22H2

安装日期 ‎2022/‎9/‎28

操作系统版本 22621.521

体验 Windows Feature Experience Pack 1000.22634.1000.0

### 开发环境

Microsoft Visual Studio Professional 2022

Version 17.2.4

VisualStudio.17.Release/17.2.4+32602.215

Microsoft .NET Framework

Version 4.8.09032

Installed Version: Professional

Visual C++ 2022 00483-00000-00004-AA929

Microsoft Visual C++ 2022

### 语言标准

C++20 or latest.

## 1.4评价标准

运行时间

# 2 实验方案

## 2.1 回溯搜索

当CSP不能只用推理来求解时，我们必须通过搜索来求解。如果行动的先后顺序对结果没有影响，那么问题就是可交换的。而CSP恰是可交换的，因为给变量赋值的时候不需考虑赋值的顺序。因此，只需考虑搜索树一个结点的单个变量。这也就不会出现搜索树大小指数增长的问题。

术语**回溯搜索**用于深度优先搜索中，它每次为一变量选择一个赋值，当没有合法的值可以赋给某变量时就回溯。算法见图。它不断选择未赋值变量，轮流尝试变量值域中的每一个值，试图找到一个解。一旦检测到不相容， BACKTRACK失败，返回上一次调用尝试另一个值。此外，因为CSP表示是标准化的，不需要给 BACKTRACKING- SEARCH提供领域专用的初始状态、行动函数、转移模型或目标测试。

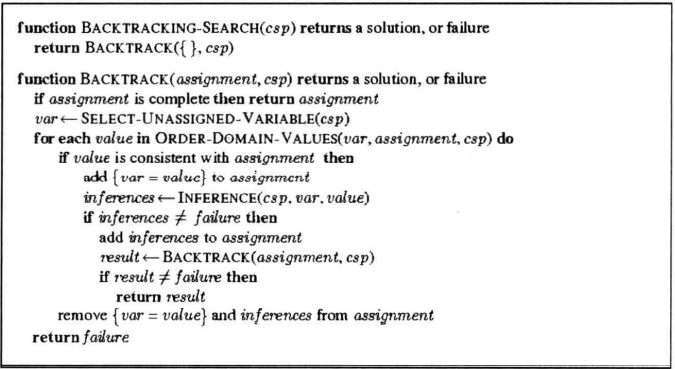


Fig. 回溯搜索的伪代码

回溯算法包括这样一行：



SELECT- UNASSIGNED- VARIABLE最简单的策略是按照列表顺序选择未赋值变量。这种静态的变量排序很少能使得搜索高效，所以我们意图探究出一种启发式函数来帮助得到更好的选择。

在搜索树的上下文中，它的目的是帮助更快地找到答案，我们通过生成或调查更少的后继者来“修剪搜索”来做到这一点，或者通过切断没有希望的方向等。这里我们调用一个简单的启发式，减少在任何搜索步骤中生成的直接后继者的数量，从而使搜索树更细，在下一级探索的状态更少。事实证明，这种简单的启发式方法不会对 N-Queens 产生负面影响（比如使搜索不完整）。

新想法是 SELECT-UNASSIGNED-VARIABLE不是以固定或随机顺序，而是以一种随搜索动态变化的原则方式。最小剩余值（MRV）启发式是选择剩余合法行数最少的列将Queen插入（最小化分支因子）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 经典回溯 | 回溯+MRV | 向前检验 | 向前检验+MRV | 最小冲突 |
| N-Queens | (>40,000K) | 13,500K | (>40,000K) | 817K | 4K |

Table N-皇后问题的多种算法性能测试，其中数字代表操作步数

要实现这个启发式，要记录额外信息——只需添加另一个长度为N的向量维护每列的合法行数，并在插入任意一个Queen的时候更新所有列时。正如表格所示，这将“N皇后问题”（测试范围为从2~50个Queen）从“不可能”（限制为 4000 万次操作）变为可能。

MRV启发式也被称为“最受约束变量”或“失败优先”启发式，之所以被称为后者是因为它选择了最可能很快导致失败的变量，从而对搜索树剪枝。如果变量X没有可选的合法取值，那么MRV启发式将选择X并马上检测到失败——避免其他无意义的搜索继续进行。相比随机的或静态的排序，MRV启发式通常性能更好，取决于问题不同，它的性能有时要好到1000倍以上。

一旦一个变量被选定，算法需要决定检验它的取值顺序。为此，有时最少约束值启发式很有效。它优先选择的值是给邻居变量留下更多的选择。一般来说，启发式应该试图为剩余变量赋值留下最大的空间。当然，如果试图找到问题的所有解，而不只是第一个解，那么这个排序毫无意义，因为无论如何要考查所有情况。当问题没有解的情况也是一样。

为什么变量选择是失败优先而值的选择是失败最后呢？现在的发现是，对于各种各样的问题，选择具有最少剩余值的变量通过早期的有效剪枝而有助于最小化搜索树中的结点数。而对于值的排序，窍门是只需找到一个解；因此首先选择最有可能的值是有意义的。如果需要枚举所有的解而不是只找一个解，值的排序毫无意义。

## 2.2 局部搜索算法：MinConflicts

局部搜索算法对求解许多CSP都是很有效的。它们使用完整状态的形式化：初始状态是给每个变量都赋一个值，搜索过程是一次改变一个变量的取值。例如，在八皇后问题中，初始状态是8个皇后在8列上的一个随机布局，然后每步都是选择一个皇后并把它移动到该列的新位置上。典型地，初始布局会违反一些约束。局部搜索的目的就是要消除这些矛盾。在为变量选择新值的时候，最明显的启发式是选择与其他变量冲突最少的值——最少冲突启发式。Fig. 2给出了该算法，Fig. 3则将算法应用于八皇后问题。

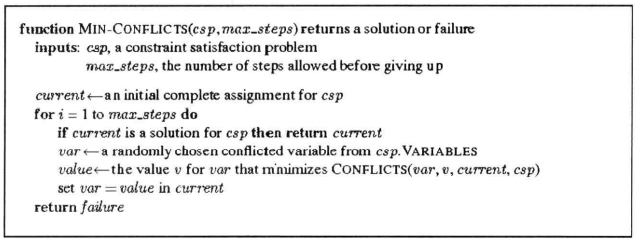


Fig. 用局部搜索解决CSP的MIN-CONFLICTS

令人惊讶的是最小冲突对许多CSP都有效。神奇的是在n后问题中，如果不依赖于皇后的初始放置情况，最少冲突算法的运行时间大体上独立于问题的规模。它甚至能在平均(初始赋值之后)50步之内求解百万皇后问题。

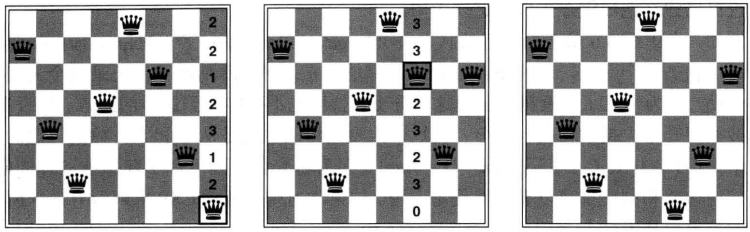
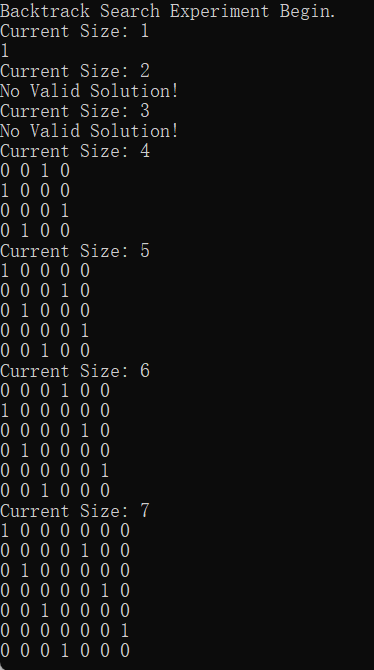
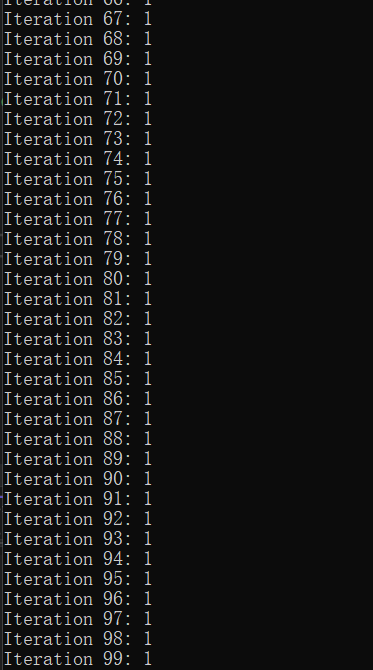


Fig. 用MIN-CONFLICTS解决八皇后问题的两步解

每一步选择一个皇后，在它所在列重新分配位置。方格中的数字表示冲突的个数（在这个问题中是能攻击到的皇后的个数)。算法将皇后移到最小冲突的方格里，随机地打乱平衡

# 3实验结果

## 3.1 回溯搜索

篇幅受限，左图为1~7皇后问题的解的情况。右图为8皇后问题搜索测试，在100次测试中均能得到正确结果。实际上，单独运行一百次来看是否成功是没有意义的，因为回溯法会从头开始逐个摆放Queen，每一次形成的解都是相同的。

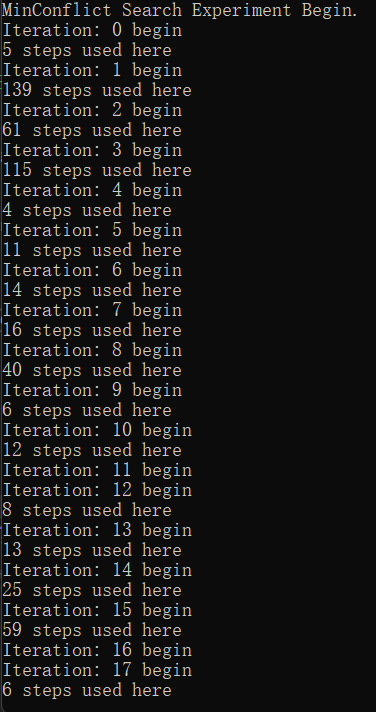
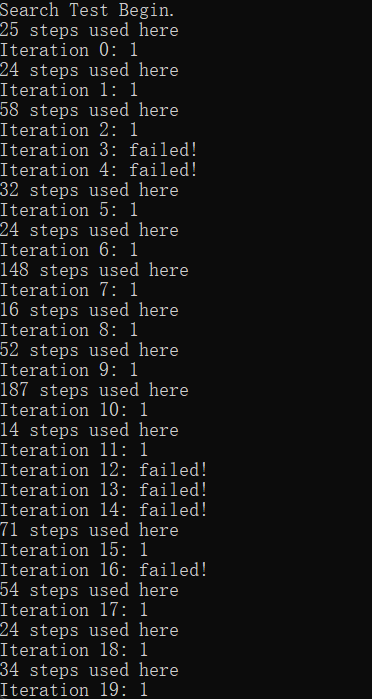
下面又测试了N（，当N=1,2,3时的探究是没有意义的）皇后问题的N-时间关系曲线，每一个N都对应100次相同的求解，再求平均值，以此得到运行时间。



Table 回溯法求解N-皇后问题测试结果

Fig. 回溯法求解N-皇后问题N-t关系曲线

## 3.2 局部搜索算法：MinConflicts

篇幅受限，左图为8皇后问题的解的情况。右图为8皇后问题搜索测试，在100次测试中，有大约84%的情况均能得到正确结果。

下面又测试了N（）皇后问题下MinConflicts算法的表现。设置max step为200，每一个N都对应100次相同的求解，再求平均值，以此得到各组数据。其中，没有成功得到解的情况并未计入时间和步数统计。



Table 局部搜索算法MinConflicts求解N-皇后问题测试结果

Fig. 局部搜索算法MinConflicts求解N-皇后所得N与平均步数和所用时间的图线

# 4实验分析

## 4.1 回溯搜索

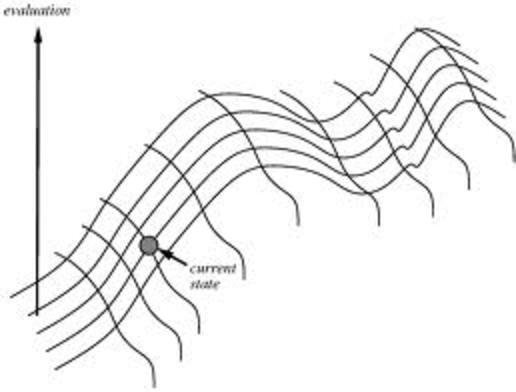
观察结果，回溯法在的时候表现还算良好，而一旦突破这个阈值，其运行时间就开始快速膨胀了。此外，也不难注意到，时所需要的时间总是比时需要的少。由于我自身水平的限制，我不能够解释这个现象。

回溯法总是从头开始摆放Queen，这致使回溯法总会得到一模一样的解。不过，不难想到，如果目的是为了求出所有的解，回溯法可能是比较理想的选择。

## 4.2局部搜索算法：MinConflicts

从实验结果来看，MinConflicts在“6皇后问题”的表现出奇的差，而在整体上上看，无论是运行时间还是平均需要步骤都是线性增长的。

我们的操作要么是“插入Queen”，要么是（隐含地失败时）“移除queen”。假设我们重新考虑——假设初始状态是 N 个皇后的（随机，最佳猜测，......）放置，每列一个，并且唯一的操作是“移动Queen”（在其列中向上或向下）。那么我们现在的工作听起来有点不同：在这里，我们总是有一个完整的解决方案候选者，工作是改进它，而不是创造它。这种策略是一种“爬山”搜索。



在图像中，状态是二维空间中的一个点，状态的值是垂直高度。在这里，沿着一个维度移动并没有像沿着另一个维度移动那样改善问题，而且我们还看到了局部最大值，这也是为什么MinConflicts哪怕到达了max steps也未能找到正确的解——它陷入了局部最优解（一个小的凹槽），然而这个解并不是全局的。

说到这里，它其实有些类似于机器学习中的梯度下降法，那么如果要改进这种算法，也许可以引入随机操作，当陷入到局部最优解的时候，就使得这个点“跳出”小凹槽，直到找到全局最优解。

# 5 结论

正如前面给出的表格所式，对于N皇后问题，在MRV的帮助下，往往可以极大地减小问题求解的规模（通过剪枝），而如果再利用上向前检验或向后检验，则性能又会有很大的提高。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 经典回溯 | 回溯+MRV | 向前检验 | 向前检验+MRV | 最小冲突 |
| N-Queens | (>40,000K) | 13,500K | (>40,000K) | 817K | 4K |

# 6 代码实现

## 回溯算法

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

if (assignment.size() == csp.variables.size())

return assignment;

Queen\* var = selectUnassignedVariable(csp);

for (auto&& value : orderDomainValues(var, assignment, csp)) {

std::vector<Position> lastPositions;

std::vector<std::vector<Position>> lastDomains;

csp.record(lastPositions, lastDomains);

if (csp.consistent(value, assignment)) {

var->assign(value);

assignment.push\_back(var);

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

if (!failed(inferences)) {

assignment.insert(assignment.end(), inferences.begin(), inferences.end());

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

if (!failed(result))

return result;

}

}

csp.recover(lastPositions, lastDomains);

refresh(assignment);

}

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

}

## 局部搜索算法：MinConflicts

注：名字空间rng为namespace ranges

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

std::vector<Queen\*> current = csp.variables;

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

if (isSolution(csp, current)) {

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

return current;

}

Queen\* var = chooseConflictVariable(csp);

Position value = getMinConflictValue(csp, var);

var->position = value;

}

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

}

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

return rng::count\_if(csp.variables, [&](Queen\* q) {

return position.col != q->position.col && csp.constraints(q->position, position);

});

}

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

std::random\_device rd;

std::mt19937 gen{ rd() };

Queen\* res;

rng::sample(csp.variables | std::views::filter([&](Queen\* q) {

return getConflicts(csp, q->position);

}), &res, 1, gen);

return res;

}

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

std::random\_device rd;

std::mt19937 gen{ rd() };

std::multimap<int, Position> conflicts;

std::pair<int, Position> res;

rng::for\_each(var->domain, [&](Position& pos) {

conflicts.emplace(getConflicts(csp, pos), pos);

});

auto [first, last] = conflicts.equal\_range(conflicts.begin()->first);

std::sample(first, last, &res, 1, gen);

return res.second;

}