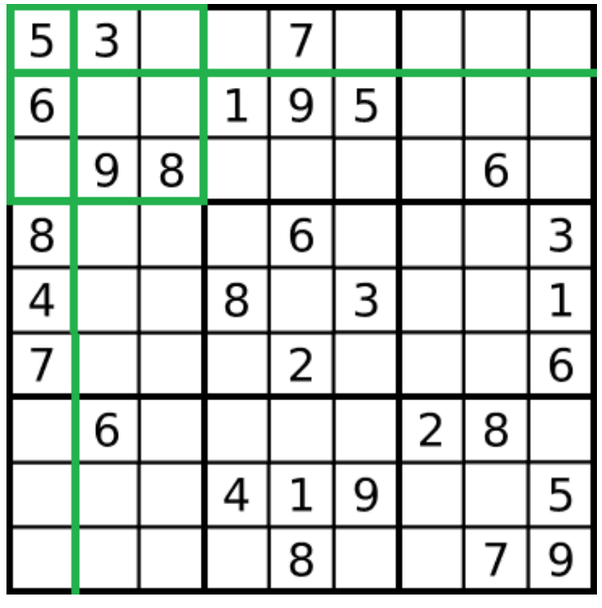
人工智能实验 2 ——数独求解

UESTC · 2021 fall

了解数独

- 数独游戏的目标是用数字填充9x9的宫格，让每一列，每一行和每个3x3小九宫部分都包含1到9之间的数字。在游戏开始时，9x9的宫格中会有一些方格已填上数字。你要做的是填上缺失的数字并完成宫格。如果出现以下情况，表示填法不正确：
  - 任意一行中，有多个相同的1到9中的数字
  - 任意一列中，有多个相同的1到9中的数字
  - 任意一个3x3小宫格中，有多个相同的1到9中的数字

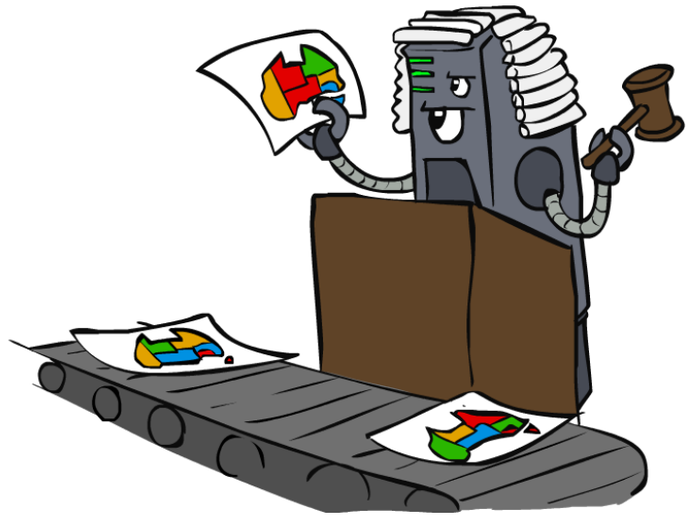


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

数独是一个约束满足问题

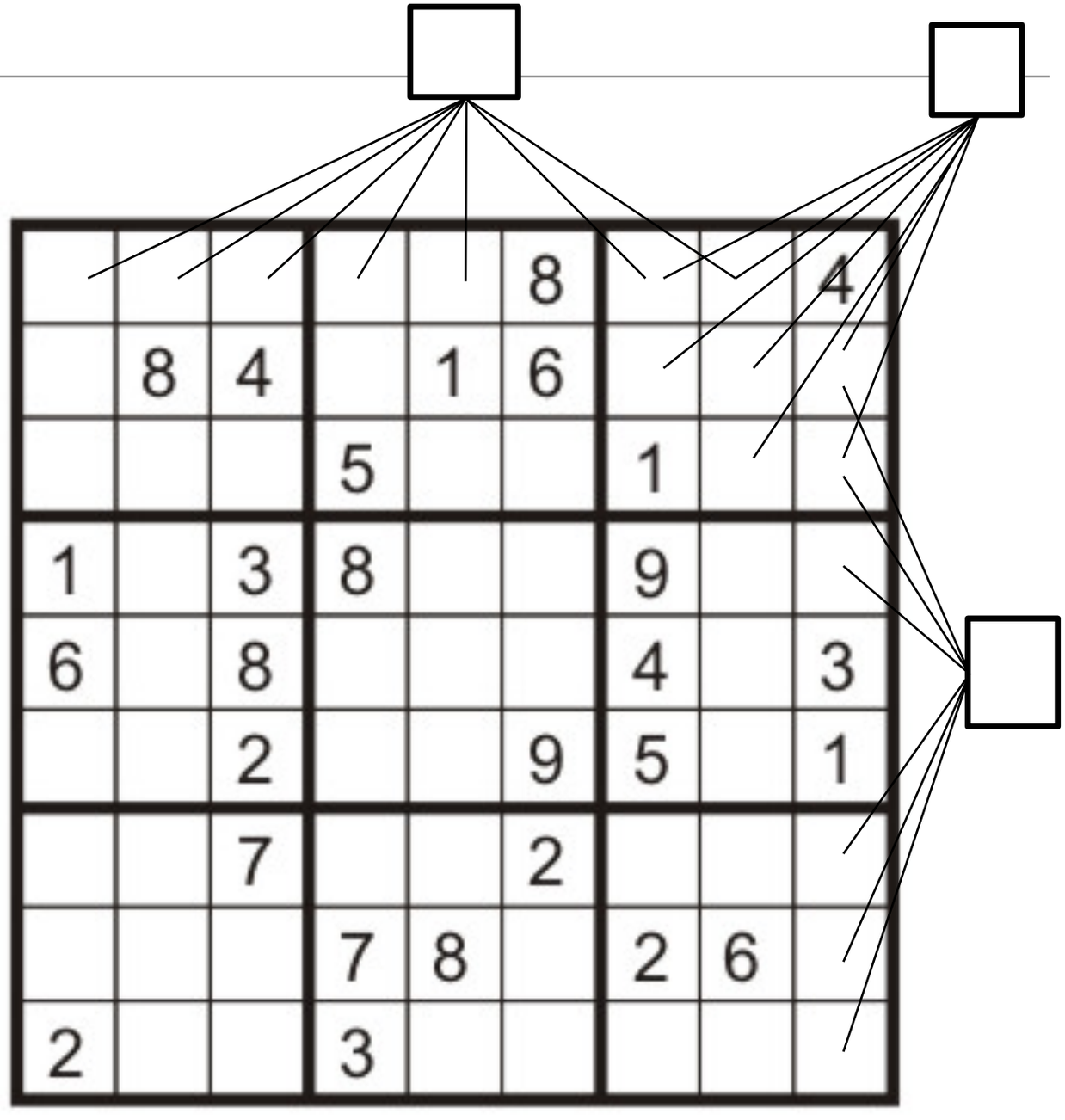
- 标准的搜索问题:
  - 状态是任意的数据结构
  - 目标测试可以是任何函数
  - 后继函数也可以是任意的



- 约束满足问题:
  - 是搜索问题的一个特殊子集
  - 状态由若干个变量X_i 组成，每个变量有一个取值域 D（有时D与i相关）
  - 目标测试是一组约束，指定变量子集的值的允许组合

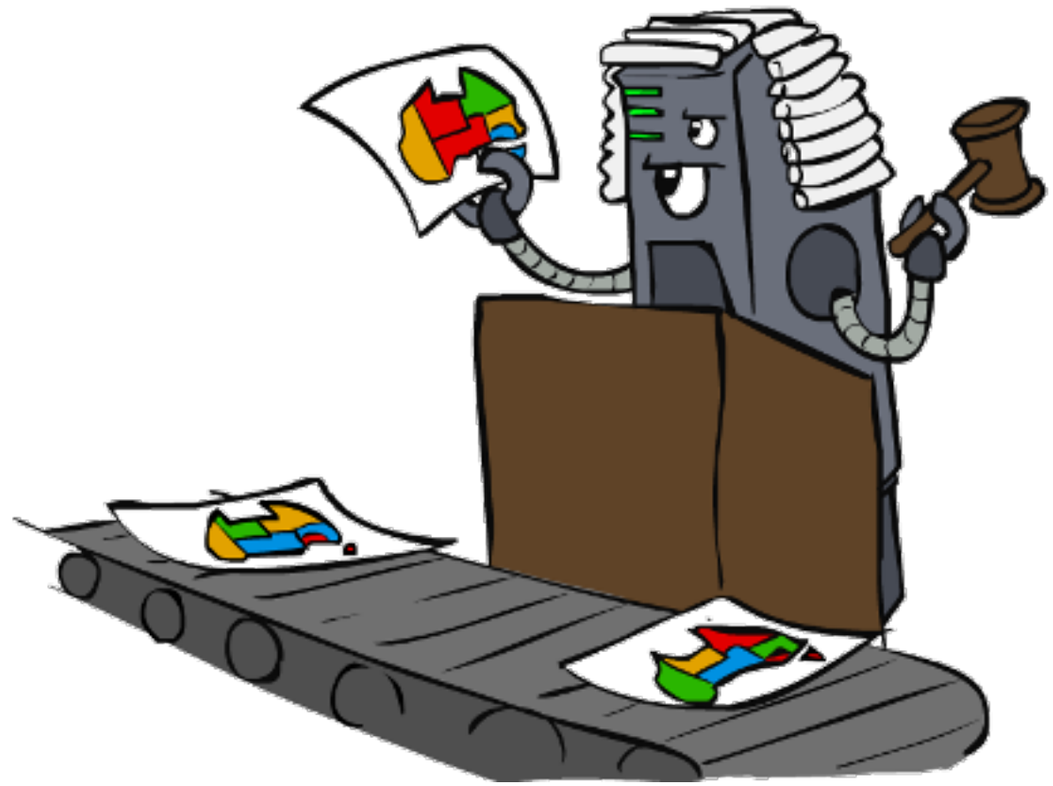
数独(Sudoku)

- 变量:
  - 每一个 空白方格
- 值域:
  - {1,2,...,9}
- 约束条件:
  - 每列9个数字都不同
  - 每行9个数字都不同
  - 每个9x9大方格里的9个数字都不同



按标准的搜索问题来解决CSP

- 状态反映了变量赋值的当前情况(部分赋值)
  - 初始状态: 没有赋值, {}
  - 行动集合(s): 分配一个值给一个未赋值的变量
  - 结果状态(s,a)(即转换模型): 该变量被赋了这个值
  - 目标-检测(s): 是否所有变量已被赋值 并且满足所有约束条件
- 我们开始将用最直接的方法，然后逐步改进



回溯搜索

- 回溯搜索是基本的无启发式信息的算法，用来求解CSP问题
- 想法 1: 一次探索一个变量
  - 变量赋值是可交换的，所以选择一个顺序固定下来
  - 例如., [WA = red then NT = green] 和 [NT = green then WA = red] 是一样的
  - 在每一步只需考虑给一个变量配值: 减少分支因子数 b 从 nd 到 d
- 想法 2: 一边探索一边检查约束条件
  - 探索过程中检查当前的变量赋值是否满足约束条件，和之前已赋值的不冲突
  - 也许需要花费一些计算来检查约束条件是否满足
  - 相当于 “逐步增加的目标测试”
- 深度优先搜索结合这两点改进，就叫作 回溯搜索

```python
if __name__ == "__main__":
    # 从文件中读取puzzle
    puzzle = load_puzzle(sudoku_file_loc)

    # 打印初始puzzle
    print("-" * 5, "puzzle", "-" * 4)
    print_puzzle(puzzle)
    print("-" * 17)

    start = datetime.now()

    # 初始化puzzle
    puzzle = initialise(puzzle)

    # 回溯搜索
    succ, puzzle = backtrack(puzzle)

    if succ:
        end = datetime.now()
        print("Solve the puzzle within :", end - start, ", with", num_of_iteration, "iterations.")

        # 做一下结果的转换，并且打印结果
        ans = convert_to_output(puzzle)
        print("-" * 5, "answer", "-" * 4)
        print_puzzle(ans)
        print("-" * 17)

        # 结果保存到文件中
        write_solution(ans, soln_file_loc)
```

```
# 将puzzle中的所有值初始化为一组值.
# 如果存在预先存在的值，我们将其设置为单个值的集合.
# 如果值为0，我们将其设置为一个包含了1-9数值的集合，也就是该位置的所有可能取值.
def initialise(puzzle):
    initial_neighours()
    new_puzzle = [[{1, 2, 3, 4, 5, 6, 7, 8, 9} for i in range(9)] for j in range(9)]
    for row in range(9):
        for col in range(9):
            if puzzle[row][col] != 0:
                assign(new_puzzle, (row, col), puzzle[row][col])
    puzzle = new_puzzle
    return puzzle

def initial_neighours():
    for row in range(9):
        for col in range(9):
            neighbour = get_connecting_cells_positions((row, col))
            neighbours[(row, col)] = neighbour
    return neighbours
```

```
# 回溯搜索
def backtrack(puzzle):
    # 检查每行、每列和每宫(3x3框)是否合法
    if not analyse_domains(puzzle):
        return False, puzzle

    # 遍历puzzle，找出一个包含了多个候选数字的位置
    h_row = -1
    h_col = -1
    for row in range(9):
        for col in range(9):
            if len(puzzle[row][col]) > 1:
                h_row = row
                h_col = col
    # 先选择哪一个候选位置？这里可以应用最小剩余值 (Minimum remaining values -- MRV)原则

    # puzzle中每个位置都只有1个候选数字，则该puzzle已经成功求解了
    if h_row == -1 and h_col == -1:
        return True, puzzle

    # 候选赋值集合
    h_values = puzzle[h_row][h_col].copy()

    # 尝试按照候选赋值集合中的值对该位置进行赋值：
    # 先选择哪一个value？这里可以应用最小制约的值 (Least Constraining Value -- LCV)原则
    for value in h_values:
        current_puzzle = copy.deepcopy(puzzle)
        current_puzzle[h_row][h_col] = set()
        current_puzzle[h_row][h_col].add(value)
        # 将current_puzzle中的位置(h_row, h_col)赋值为value
        # 赋值会失败吗？
        if not assign(current_puzzle, (h_row, h_col), value):
            continue

        # 迭代进行回溯搜索
        result = backtrack(current_puzzle)
        succ, current_puzzle = result
        if succ:
            return succ, current_puzzle

    # 所有尝试都没有return，说明尝试失败了，得回溯
    return False, current_puzzle
```

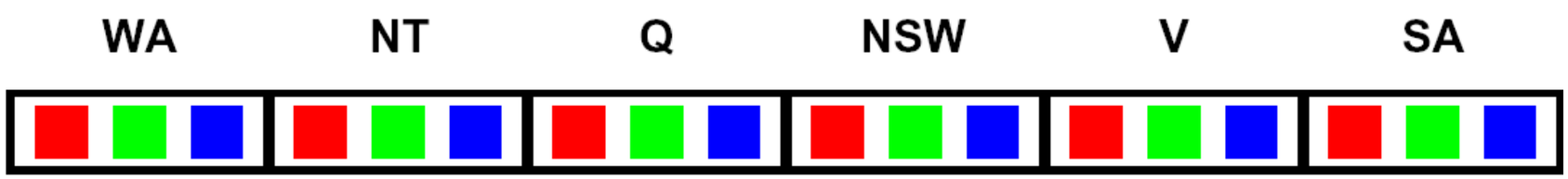
```
# 将puzzle中的位置position赋值为value
def assign(puzzle, position, value):
    global num_of_iteration
    num_of_iteration = num_of_iteration + 1
    row, col = position
    puzzle[row][col] = set()
    puzzle[row][col].add(value)

    # 赋值完了可以做前向检查(forward checking)，减少puzzle中可取值集合的大小，
    # 也有可能导致赋值失败（某个位置的可选value集合为空），从而更快地进行回溯
    # 检查完了还可以做约束传播(Constraint Propagation)，进一步减少puzzle中可取值集合的大小，
    # 也有可能更快地导致赋值失败，从而更快地进行回溯

    return True
```

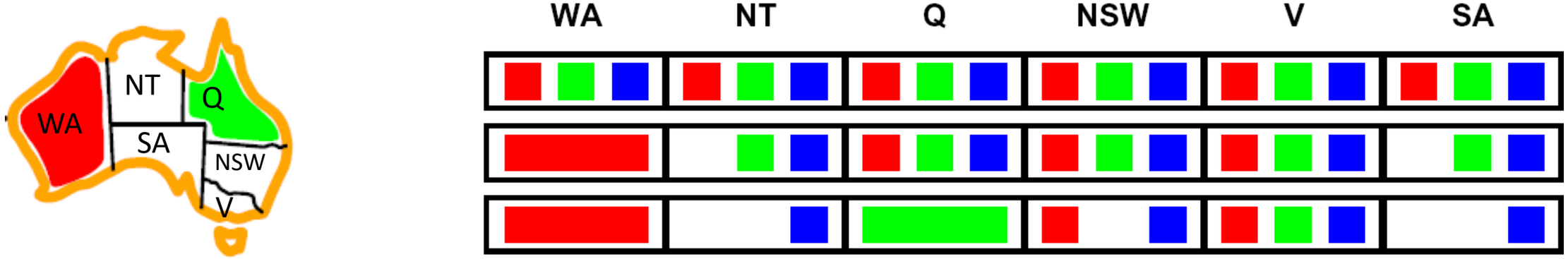
过滤: 前向检查法(Forward Checking)

- 过滤:搜索中, 持续检测未赋值变量的值域，去掉违反约束条件的值
- 简单的过滤: 向前检查法
  - 当添加对一个变量的赋值后，划掉剩下变量值域中违反约束条件的值



过滤:约束传播(Constraint Propagation)

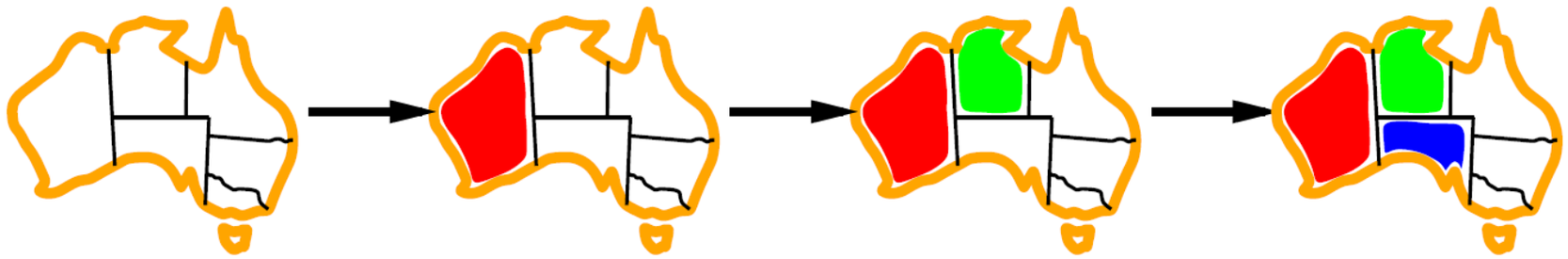
- 前向检查传播已分配到未分配变量的信息，但不提供对所有故障的早期检测:



- NT和SA不可能都是蓝色的!
- 为什么我们还没发现呢?
- 约束传播:从约束到约束的推理

变量排序

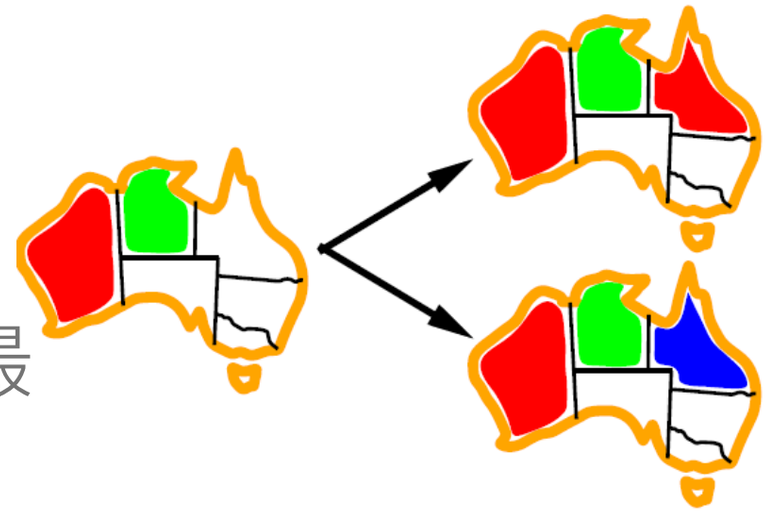
- 变量排序: 最小剩余值 (Minimum remaining values -- MRV)原则:
  - 先选择其值域中所剩合理可选的值最少的变量

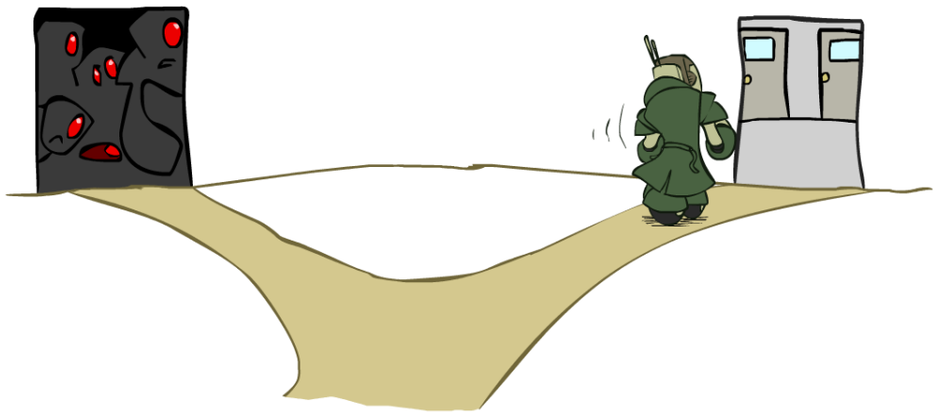


  - 为什么是最少而不是最大?
  - “快速失败” 排序
  - 使用连接度数启发信息打破一样的情况
    - 选择和其他变量连接数最多的变量(受约束最多的)

对值的排序选择

- 选择最小制约的值 (Least Constraining Value -- LCV)
  - 选择对剩下的变量在选值的时候约束最小的值
  - 这可能需要花费些计算时间!
- 为什么最小而不是最大制约的?
- 结合这些排序上的改进，能够解决1000-皇后问题





实验要求

- 改进基于回溯搜索的数独求解算法：
  - 加入前向检查
  - 加入约束传播
  - 加入MRV和LCV
- 尽可能快地求解数独问题