人工智能实验报告

姓名：孟衍璋 学号：16337183

一、 实验目的

用 brute force、back-tracking、forward-checking、enforce Generalized Arc Consistency 求解数独。

二、 实验内容

写一个 python 程序，输入一个数独问题，返回它的计算结果。使用几种不同的算法来解决数独问题，返回数独的解决方案。几种算法如下：

- brute force (exhaustive search) method
- back-tracking (Constraint Satisfaction Problem (CSP)
- forward-checking with Minimum Remaining Values (MRV) heuristics
- enforce Generalized Arc Consistency (GAC) algorithm

主程序文件必须被命名为 SudokuSolver.py，数独谜题存在 puzzle.txt 文件中，解决方案存在 solution.txt 文件中。

程序必须使用命令行输入，样例如下:

Python SudokuSolver puzzle1.txt BF

Python SudokuSolver puzzle2.txt BT

Python SudokuSolver puzzle3.txt FC-MRV

Python SudokuSolver puzzle4.txt GAC

输出需要包含总运行时间，搜索时间，遍历的结点个数。将这些信息存在 performance 文件中。

三、 实验步骤

1. 实现 brute_force

主要由七个函数构成，首先是读取文件的函数：

```
def read_puzzle(file):
    with open(file) as f:
        lines = f.readlines()
        for data in lines:
            # print(data.split())
            puzzle.append([int(x) for x in data.split()])
    return puzzle
```

第二个是判断某一位置还能填什么数字的函数：

```python
# 判断某一个位置还能填什么数字，返回一个list
def can_fill(x,y,puzzle):
    a = []
    # 如果当前元素不为0，直接返回一个空列表
    if puzzle[x][y] != 0:
        return a

    appear = {1:False, 2:False, 3:False, 4:False, 5:False, 6:False, 7:False, 8:False, 9:False}
    # 当前行出现过的元素都置True
    for i in range(9):
        if(puzzle[x][i] != 0):
            appear[puzzle[x][i]] = True
    # 当前列出现过的元素都置True
    for i in range(9):
        if(puzzle[i][y] != 0):
            appear[puzzle[i][y]] = True
    # 当前九宫格出现过的元素都置True
    if (x >= 0 and x <= 2):
        row = 0
    elif (x >= 3 and x <= 5):
        row = 1
    elif (x >= 6 and x <= 8):
        row = 2
    if (y >= 0 and y <= 2):
        col = 0
    elif (y >= 3 and y <= 5):
        col = 1
    elif (y >= 6 and y <= 8):
        col = 2
    for j in range(3):
        for k in range(3):
            if (puzzle[row*3+j][col*3+k] != 0):
                appear[puzzle[row*3+j][col*3+k]] = True

    # 将还能填的数字存在一个list中
    for i in appear:
        if(appear[i] == False):
            a.append(i)
    return a
```

然后是打印并写入 solution 文件的函数：

```python
def write_solution(file, puzzle):
    solution = []
    # 将结果存入solution这个列表
    for lines in puzzle:
        solution.append(lines)
    # 将结果打印出来
    print('solution:')
    for lines in solution:
        print(lines)
    # 将结果写入文件
    with open(file, 'w') as f:
        for lines in solution:
            for data in lines:
                f.write(str(data))
                f.write(' ')
            f.write('\n')
```

然后是打印并记录 performance 文件的函数：

```
def write_performance(file, start_time, end_time, search_start_time, search_end_time, count):
    print('Total clock time:', end_time - start_time)
    print('Search clock time:', search_end_time - search_start_time)
    print('Number of nodes generated:', count)
    with open(file, 'w') as f:
        f.write('Total clock time: ')
        f.write(str(end_time - start_time))
        f.write('\nSearch clock time: ')
        f.write(str(search_end_time - search_start_time))
        f.write('\nNumber of nodes generated: ')
        f.write(str(count))
```

然后是判断数独是否已经填完的函数：

```
def not_done(puzzle):
    return True in [0 in r for r in puzzle]
```

然后是依次填入的函数：

```
def go_around(puzzle):
    ans = []
    global count
    for index_r, r in enumerate(puzzle):
        row = []
        for index_c, c in enumerate(r):
            if 0 == c:
                count += 1
                maybe_ans = can_fill(index_r, index_c, puzzle)
                row.append(maybe_ans[0] if len(maybe_ans) == 1 \
                           else 0)
            else:
                row.append(c)
        ans.append(row)
    return ans
```

最后则是实现 brute_force 的函数：

```
def bf(puzzle):
    search_start_time = time.time()
    while not_done(puzzle):
        halfway = time.time()
        #  如果搜索时间大于30秒，就退出程序
        if halfway - search_start_time >= 30:
            print('Number of nodes generated:', count)
            sys.exit()
        puzzle = go_around(puzzle)
    search_end_time = time.time()
    end_time = time.time()
    write_solution('solution_bf.txt', puzzle)
    write_performance('performance_bf.txt', start_time, end_time, search_start_time, search_end_time, count)
```

在该函数中，设置了如果搜索时间大于 30 秒就退出程序的操作。

2. 实现 backtracking

主要由六个函数构成，首先是读取文件的函数：

```
def read_puzzle(file):
    with open(file) as f:
        lines = f.readlines()
        for data in lines:
            # print(data.split())
            puzzle.append([int(x) for x in data.split()])
    return puzzle
```

然后是判断冲突的函数，即在数独的空余位置填上一个数字之后，判断是否会与同行、同列、同九宫格的元素冲突：

```
# (x,y)为当前点位置，第x行，第y列
def confict(x,y,num):
    # 判断当前行的元素与新填的元素有无冲突
    for i in range(9):
        if (puzzle[x][i] == num):
            return True
    # 判断当前列的元素与新填的元素有无冲突
    for i in range(9):
        if (puzzle[i][y] == num):
            return True
    # 判断当前九宫格的元素与新填的元素有无冲突
    if (x >= 0 and x <= 2):
        row = 0
    elif (x >= 3 and x <= 5):
        row = 1
    elif (x >= 6 and x <= 8):
        row = 2
    if (y >= 0 and y <= 2):
        col = 0
    elif (y >= 3 and y <= 5):
        col = 1
    elif (y >= 6 and y <= 8):
        col = 2
    for j in range(3):
        for k in range(3):
            if (puzzle[row*3+j][col*3+k] == num):
                return True
    # 没有冲突则返回False
    return False
```

然后是打印并写入 solution 文件的函数：

```python
def write_solution(file, puzzle):
    solution = []
    # 将结果存入solution这个列表
    for lines in puzzle:
        solution.append(lines)
    # 将结果打印出来
    print('solution:')
    for lines in solution:
        print(lines)
    # 将结果写入文件
    with open(file, 'w') as f:
        for lines in solution:
            for data in lines:
                f.write(str(data))
                f.write(' ')
            f.write('\n')
```

然后是打印并记录 performance 文件的函数：

```python
def write_performance(file, start_time, end_time, search_start_time, search_end_time, count):
    print('Total clock time:', end_time - start_time)
    print('Search clock time:', search_end_time - search_start_time)
    print('Number of nodes generated:', count)
    with open(file, 'w') as f:
        f.write('Total clock time: ')
        f.write(str(end_time - start_time))
        f.write('\nSearch clock time: ')
        f.write(str(search_end_time - search_start_time))
        f.write('\nNumber of nodes generated: ')
        f.write(str(count))
```

然后是 start_point 函数，用来指定每次递归访问的位置：

```python
def start_point(puzzle):
    for i in range(9):
        for j in range(9):
            if not puzzle[i][j]:
                return i,j
    return i,j
```

最后是 backtracking 函数，用递归的方法实现：

```python
search_start_time = time.time()
def bt(puzzle):
    global count
    count += 1
    i,j = start_point(puzzle)
    # 如果i，j均为8且该位置的数不为0，说明已经填完
    if i == 8 and j == 8 and puzzle[8][8]:
        search_end_time = time.time()
        end_time = time.time()
        write_solution('solution.txt', puzzle)
        write_performance('performance.txt', start_time, end_time, search_start_time, search_end_time, count)
        return True

    for value in range(1,10):
        # 判断有无冲突
        if conflict(i, j, value) == 0:
            # 如果当前位置没有冲突，则令puzzle[i][j]=value
            puzzle[i][j] = value
            if bt(puzzle) == 0:
                # 如果后面的递归搜索不满足要求，令puzzle[i][j] = 0
                puzzle[i][j] = 0
            else:
                return True
    # 如果该点遍历1-9都不符合要求，则表示上游选值不当，回溯
    return False
```

每递归调用一次函数，全局变量 count 就加 1，这个值就是访问的节点数。

函数中先用 start_point 函数确定访问的位置，如果已经到了第 9 行第 9 列且当前数字不为 0，说明已经将数独填完，记录下解决方案和算法表现。

如果是在平常的位置，用之前的冲突函数判断，如果没有冲突，从 1-9 依次填入数字。如果递归搜索不满足条件，则将刚刚填的值置为 0。如果某个点遍历 1-9 都不符合要求，则表示之前选值不当，回溯。

3. 实现 forward_checking

主要由六个函数构成，第一个是读取数独的函数：

```python
def read_puzzle(file):
    with open(file) as f:
        lines = f.readlines()
        for data in lines:
            # print(data.split())
            puzzle.append([int(x) for x in data.split()])
    return puzzle
```

第二个是判断某一个位置还能填什么数字的函数：

```python
# 判断某一个位置还能填什么数字，返回一个list
def can_fill(x,y,puzzle):
    a = []
    # 如果当前元素不为0，直接返回一个空列表
    if puzzle[x][y] != 0:
        return a

    appear = {1:False, 2:False, 3:False, 4:False, 5:False, 6:False, 7:False, 8:False, 9:False}
    # 当前行出现过的元素都置True
    for i in range(9):
        if(puzzle[x][i] != 0):
            appear[puzzle[x][i]] = True
    # 当前列出现过的元素都置True
    for i in range(9):
        if(puzzle[i][y] != 0):
            appear[puzzle[i][y]] = True
    # 当前九宫格出现过的元素都置True
    if (x >= 0 and x <= 2):
        row = 0
    elif (x >= 3 and x <= 5):
        row = 1
    elif (x >= 6 and x <= 8):
        row = 2
    if (y >= 0 and y <= 2):
        col = 0
    elif (y >= 3 and y <= 5):
        col = 1
    elif (y >= 6 and y <= 8):
        col = 2
    for j in range(3):
        for k in range(3):
            if (puzzle[row*3+j][col*3+k] != 0):
                appear[puzzle[row*3+j][col*3+k]] = True

    # 将还能填的数字存在一个list中
    for i in appear:
        if(appear[i] == False):
            a.append(i)
    return a
```

它访问数独中某一个元素，然后统计与它同行、同列、同九宫格中都出现过哪些元素，将剩下的元素存为一个列表。

第三第四个是写 solution 与 performance 的函数，同 backtracking 一样。

第五个是决定该访问哪一个节点的函数：

```python
# 选取remaining中可以选择的值最少的地方开始访问
def start_point(remaining,puzzle):
    i = j = 0
    x = y = 0
    smallest = float("inf")
    for i in range(9):
        for j in range(9):
            if len(remaining[i][j]) <= smallest and len(remaining[i][j]) != 0:
                smallest = len(remaining[i][j])
                x = i
                y = j
    return x,y
```

该函数与 backtracking 的有些不同，FC-MRV 是先选择约束

条件最多的位置，也即剩余能填的数字最少的位置。然后返回这个位置的坐标。

最后是 forward_checking 的实现：

```
count = 0
search_start_time = time.time()
# forward_checking
def fc_mrv(remaining,puzzle):
    global count
    count += 1
    x,y = start_point(remaining,puzzle)
    # 如果所有的空都填上了
    judge = 1
    for i in range(9):
        for j in range(9):
            if puzzle[i][j] == 0:
                judge = 0
    if judge:
        search_end_time = time.time()
        end_time = time.time()
        write_solution('solution.txt', puzzle)
        write_performance('performance.txt', start_time, end_time, search_start_time, search_end_time, count)
        # sys.exit()
        return True

    for i in remaining[x][y]:
        new_remaining = copy.deepcopy(remaining)
        new_puzzle = copy.deepcopy(puzzle)
        new_remaining[x][y] = []
        new_puzzle[x][y] = i
        # remaining[x][y] = []
        # puzzle[x][y] = i
        # 更新remaining
        for j in range(9):
            if i in new_remaining[x][j] and j != y:
                new_remaining[x][j].remove(i)
            if i in new_remaining[j][y] and j != x:
                new_remaining[j][y].remove(i)
        if (x >= 0 and x <= 2):
            row = 0
        elif (x >= 3 and x <= 5):
            row = 1
        elif (x >= 6 and x <= 8):
            row = 2
        if (y >= 0 and y <= 2):
            col = 0
        elif (y >= 3 and y <= 5):
            col = 1
        elif (y >= 6 and y <= 8):
            col = 2
        for j in range(3):
            for k in range(3):
                if i in new_remaining[row*3+j][col*3+k] and x != (col*3+j) and y != (col*3+j):
                    new_remaining[row*3+j][col*3+k].remove(i)

        if fc_mrv(new_remaining,new_puzzle) == 0:
            # 如果后面的递归搜索不满足要求，令new_puzzle[i][j] = 0
            new_puzzle[x][y] == 0
        else:
            return True
    # 如果该点遍历1-9都不符合要求，则表示上游选值不当，回溯
    return False
```

与 backtracking 不一样的地方在于，选取访问节点的方式不同，而且不需要判断冲突，利用 can_fill 函数得出的 remaining 列表，就能得到每个位置能填的数字。然后每填一个数字，对 remaining 进行一次更新。

其余操作便和 backtracking 差不了太多。

4. 实现 generalized arc consistency

主要由六个函数构成，第一个是读取数独的函数：

```
def read_puzzle(file):
    with open(file) as f:
        lines = f.readlines()
        for data in lines:
            # print(data.split())
            puzzle.append([int(x) for x in data.split()])
    return puzzle
```

第二个是判断某个位置还能填什么数字的函数，用于生成 remaining 列表：

```
# 判断某一个位置还能填什么数字，返回一个list
def can_fill(x,y,puzzle):
    a = []
    # 如果当前元素不为0，直接返回一个空列表
    if puzzle[x][y] != 0:
        return a

    appear = {1:False, 2:False, 3:False, 4:False, 5:False, 6:False, 7:False, 8:False, 9:False}
    # 当前行出现过的元素都置True
    for i in range(9):
        if(puzzle[x][i] != 0):
            appear[puzzle[x][i]] = True
    # 当前列出现过的元素都置True
    for i in range(9):
        if(puzzle[i][y] != 0):
            appear[puzzle[i][y]] = True
    # 当前九宫格出现过的元素都置True
    if (x >= 0 and x <= 2):
        row = 0
    elif (x >= 3 and x <= 5):
        row = 1
    elif (x >= 6 and x <= 8):
        row = 2
    if (y >= 0 and y <= 2):
        col = 0
    elif (y >= 3 and y <= 5):
        col = 1
    elif (y >= 6 and y <= 8):
        col = 2
    for j in range(3):
        for k in range(3):
            if (puzzle[row*3+j][col*3+k] != 0):
                appear[puzzle[row*3+j][col*3+k]] = True

    # 将还能填的数字存在一个list中
    for i in appear:
        if(appear[i] == False):
            a.append(i)
    return a
```

第三个和第四个是第三第四个是写 solution 与 performance 的函数，同之前一样。

第五个是 start_point 函数，和 forward_checking 的一样，选取 remaining 中可以选择的值最少的地方开始访问。

第六个便是 gac 函数，其余部分都和 FC 一样，不同的是加了这样一个约束条件：

```
# 如果当前点所在的行有n个空位，这n个空位能填的数字取并集，若这个并集中的元素个数小于n，则不符合要求，回溯
un = set()
vacant_num = 0
for i in range(9):
    if puzzle[x][i] == 0:
        vacant_num += 1
        for e in remaining[x][i]:
            un.add(e)
if len(un) < vacant_num:
    return False

# 如果当前点所在的列有n个空位，这n个空位能填的数字取并集，若这个并集中的元素个数小于n，则不符合要求，回溯
un = set()
vacant_num = 0
for i in range(9):
    if puzzle[i][y] == 0:
        vacant_num += 1
        for e in remaining[i][y]:
            un.add(e)
if len(un) < vacant_num:
    return False

# 如果当前点所在的九宫格有n个空位，这n个空位能填的数字取并集，若这个并集中的元素个数小于n，则不符合要求，回溯
un = set()
vacant_num = 0
if (x >= 0 and x <= 2):
    row = 0
elif (x >= 3 and x <= 5):
    row = 1
elif (x >= 6 and x <= 8):
    row = 2
if (y >= 0 and y <= 2):
    col = 0
elif (y >= 3 and y <= 5):
    col = 1
elif (y >= 6 and y <= 8):
    col = 2
for j in range(3):
    for k in range(3):
        if puzzle[row*3+j][col*3+k] == 0:
            vacant_num += 1
            for e in remaining[row*3+j][col*3+k]:
                un.add(e)
if len(un) < vacant_num:
    return False
```

上述约束条件实际上是指，对于当前访问的位置，计算与其同行或者同列、或者同九宫格剩余的空位能填的数字，取一个并集，如果这个并集中的数字个数小于空位的个数，则之前填的方案一定是错误的，需要回溯。

5. 模块化

因为要实现命令行输入，所以将之前的几种算法都分别存为单独的文件，在同一个目录下新建一个__init__.py 文件，表示这个目录是一个包。

利用 sys.argv 参数，将命令行的输入传入程序中，再调用各个模块的函数实现数独的解答。

四、 实验结果

1. brute_force

在数独简单的情况下，可以很快算出答案：

```
C:\Users\myz\Desktop\人工智能>python SudokuSolver.py puzzle.txt BF
solution:
[8, 6, 1, 2, 3, 4, 9, 5, 7]
[4, 7, 9, 5, 6, 1, 2, 8, 3]
[3, 2, 5, 9, 7, 8, 1, 6, 4]
[9, 5, 8, 1, 4, 3, 6, 7, 2]
[7, 1, 2, 8, 5, 6, 3, 4, 9]
[6, 3, 4, 7, 2, 9, 5, 1, 8]
[5, 9, 6, 4, 8, 2, 7, 3, 1]
[1, 4, 3, 6, 9, 7, 8, 2, 5]
[2, 8, 7, 3, 1, 5, 4, 9, 6]
Total clock time: 0.015624284744262695
Search clock time: 0.0
Number of nodes generated: 95
```

在数独复杂的时候，就需要很长时间，我在程序里设置了如果 30 秒算不出来就退出的操作。

```
C:\Users\myz\Desktop\人工智能>python SudokuSolver.py puzzle1.txt BF
Number of nodes generated: 1152896
```

2. backtracking

```
C:\Users\myz\Desktop\人工智能>python SudokuSolver.py puzzle2.txt BT
solution:
[5, 3, 2, 1, 7, 8, 6, 9, 4]
[7, 6, 1, 3, 4, 9, 5, 2, 8]
[4, 8, 9, 5, 2, 6, 7, 1, 3]
[9, 4, 5, 2, 1, 3, 8, 6, 7]
[8, 2, 3, 4, 6, 7, 9, 5, 1]
[1, 7, 6, 8, 9, 5, 3, 4, 2]
[2, 5, 7, 9, 3, 1, 4, 8, 6]
[3, 9, 4, 6, 8, 2, 1, 7, 5]
[6, 1, 8, 7, 5, 4, 2, 3, 9]
Total clock time: 0.04024076461791992
Search clock time: 0.04024076461791992
Number of nodes generated: 912
```

3. forward_checking_MRV

```
C:\Users\myz\Desktop\人工智能>python SudokuSolver.py puzzle3.txt FC-MRV
solution:
[5, 3, 7, 9, 2, 8, 6, 4, 1]
[1, 9, 4, 6, 7, 2, 8, 5, 3]
[6, 4, 8, 1, 5, 3, 2, 9, 7]
[2, 5, 1, 7, 3, 9, 4, 6, 8]
[4, 8, 6, 2, 1, 5, 3, 7, 9]
[3, 7, 9, 4, 8, 6, 1, 2, 5]
[7, 2, 5, 3, 6, 1, 9, 8, 4]
[8, 1, 2, 5, 9, 4, 7, 3, 6]
[9, 6, 3, 8, 4, 7, 5, 1, 2]
Total clock time: 0.5124454498291016
Search clock time: 0.5124454498291016
Number of nodes generated: 987
```

4. generalized_arc_consistency

```
C:\Users\myz\Desktop\人工智能>python SudokuSolver.py puzzle4.txt GAC
solution:
[3, 7, 6, 5, 1, 9, 8, 2, 4]
[9, 2, 4, 7, 3, 8, 5, 1, 6]
[5, 8, 1, 4, 2, 6, 7, 3, 9]
[2, 4, 3, 6, 7, 1, 9, 8, 5]
[1, 6, 9, 3, 8, 5, 2, 4, 7]
[8, 5, 7, 2, 9, 4, 3, 6, 1]
[7, 1, 5, 8, 6, 2, 4, 9, 3]
[4, 9, 8, 1, 5, 3, 6, 7, 2]
[6, 3, 2, 9, 4, 7, 1, 5, 8]
Total clock time: 0.25340890884399414
Search clock time: 0.25340890884399414
Number of nodes generated: 429
```

五、 实验心得

这次实验是用程序解数独，感觉比较有趣，实验要求中需要用四种方式求解，第一种是暴力搜索，第二种是回溯法，第三种是带最小剩余值的前向检查，第四种是广义弧一致性。首先需要理解这些方法的不同之处，和它们分别是怎样实现的。

第一种暴力搜索就是穷举，没有什么技术含量。第二种回溯法，就是依次遍历每个空位，每个空位都依次填入 1-9 的数字，如果判断出有冲突，就依次加 1，如果 1-9 填入了之后都有冲突，说明之前一位填得有问题，需要回到之前一位，将其加 1。

第三种方法是带最小剩余值的前向检查，与回溯法不同的地方在于，不需要判断冲突，在进行递归之前，就将每个空格中除去冲突能填的数字列出来，然后访问到每个位置的时候，就依次填入这些数字，每填入一个数字之后，需要更新这个列表。带最小剩余值的意思则是，在决定访问什么位置的时候，先访问那些可填值列表中最少的位置，这样先满足了约束最多的一项，后面的约束也就会更少。

第四种方法是广义弧一致性，开始没有太理解，后来问了同学老师之后，在带最小值的前向检查的基础上，又加了一个约束条件。如果对于当前访问的位置，计算与其同行或者

同列、或者同九宫格剩余的空位能填的数字，取一个并集，如果这个并集中的数字个数小于空位的个数，则之前填的方案一定是错误的，需要回溯。

最后还要考虑在控制台输出的问题，题目要求中需要按照类似 Python SudokuSolver puzzle1.txt BF 这样的形式来。经过查询之后知道控制台的输入可以用 sys.argv 来获取，第一个参数下标是 0，第二个参数下标是 1，依次类推，然后再将之前写的四种算法存为四个文件，组织成模块的形式，便能在主程序中调用。

总而言之，这次实验十分有趣，而且让我学习到了很多东西，感觉收获很多。