

华中科技大学

课程实验报告

课程名称：计算机算法基础

院 系：计算机科学与技术

专业班级：CS1409

学 号：U201414797

姓 名：张丹朱

指导教师：何琨

2017 年 1 月 5 日

目 录

实验一.QUICKSORT 的迭代模型.....	1
1. 实验目的与要求.....	1
2. 算法设计.....	1
3. 实验环境.....	4
4. 实验过程.....	4
5. 实验结果.....	5
6. 结果分析.....	5
实验二.基于二次取中的选择算法.....	6
1. 实验目的与要求.....	6
2. 算法设计.....	6
3. 实验环境.....	8
4. 实验过程.....	8
5. 实验结果.....	9
6. 结果分析.....	9
实验三.DIJKSTRA 算法.....	10
1. 实验目的与要求.....	10
2. 算法设计.....	10

3. 实验环境.....	12
4. 实验过程.....	12
5. 实验结果.....	12
6. 结果分析.....	12
附加一.每对结点之间的最短路径长度.....	14
1. 实验目的与要求.....	14
2. 算法设计.....	14
3. 实验环境.....	15
4. 实验过程.....	15
5. 实验结果.....	15
6. 结果分析.....	16
附加二.N 皇后求解.....	17
1. 实验目的与要求.....	17
2. 算法设计.....	17
3. 实验环境.....	18
4. 实验过程.....	18
5. 实验结果.....	19
6. 结果分析.....	19

附录一.分治法源代码.....	20
附录二.单源点最短路径源代码.....	25
附录三.每对结点最短路径源代码.....	29
附录三.N 皇后问题源代码.....	31

实验一.QUICKSORT 的迭代模型

1. 实验目的与要求

编程实现算法 4.14:QUICKSORT 的迭代模型(P.90)，编制以下过程:

PARTITION

QUICKSORT2

要求:

- a. 栈可用数组实现,但要有溢出检测
- b. 编制测试数据,给出实验结果
- c. 分析该算法的空间复杂性,特别是需要说明为什么说其空间复杂度表达式

是:

$$S(n) = \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

2. 算法设计

1. INTERCHANGE 过程

设计思路：通过中间变量，将参数 a 和参数 b 的值互换。即先将 a 的值赋给中间变量 temp，再将 b 的值赋给 a，最后将 temp 的值赋给 b，由此完成两数值的交换。

2. PARTITION 过程

设计思路：假定第一个元素为划分元素，i，p 分别指向第一个元素与最后元素的下一个元素（该元素的值被定义为大于所有的元素）。i 由左向右移直至

找到不小于划分元素的元素， p 由右向左移直至找到不大于划分元素的元素。
 若此时 $i < p$ ，则交换 $A(i)$ 和 $A(p)$ ，继续循环，否则结束循环，划分元素到位置 p 。

其流程图如下图 1.1 所示。

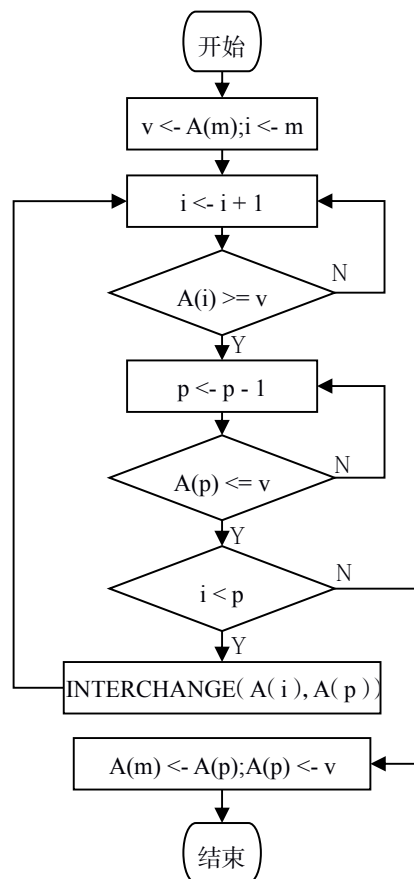


图 1.1 PARTITION 流程图

3. QUICKSORT2 过程

设计思路：当 PARTITION 把文件 $A(p:q)$ 分成两个子文件 $A(p:j-1)$ 和 $A(j+1:q)$ 之后，总是先对其中较小的那个子文件分类。对于原算法中的递归调用则用栈来代替，先将较大的那个子文件起始和结束的下标入栈，当对较小那个文件分类结束后再出栈继续分类。

其流程图如下图 1.2 所示。

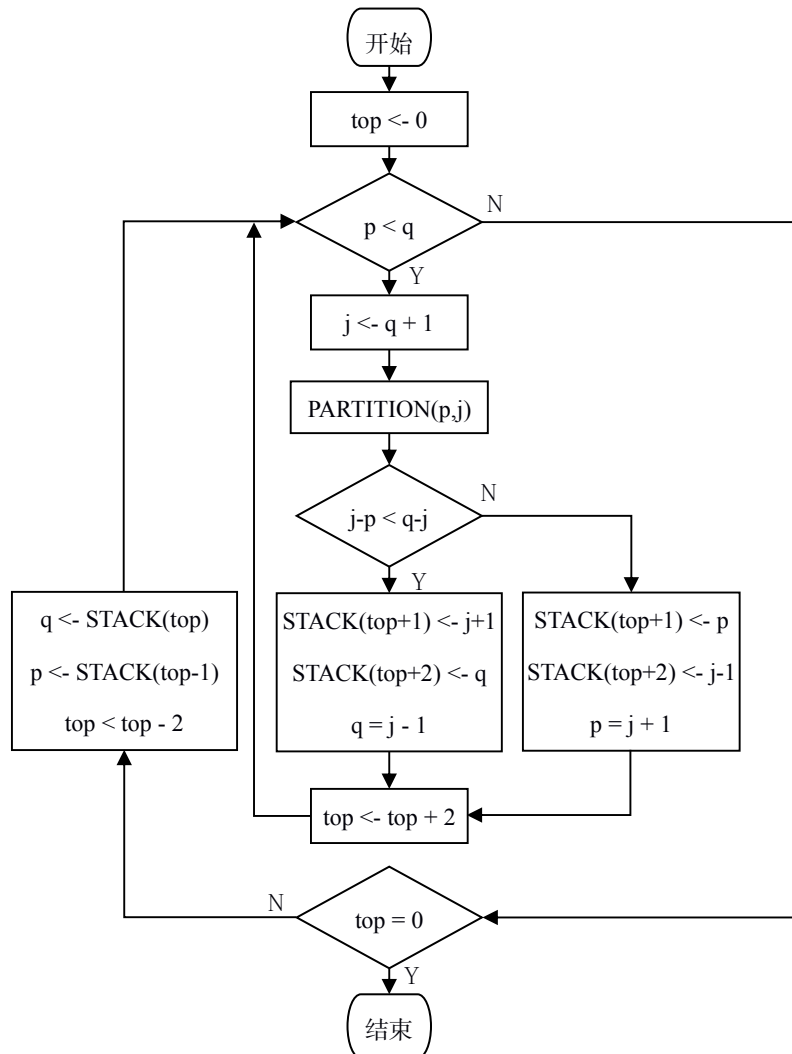


图 1.2 quicksort 的迭代模型流程图

3. 实验环境

操作系统：ubuntu16.04

编程语言：C 语言

编译工具：gcc

4. 实验过程

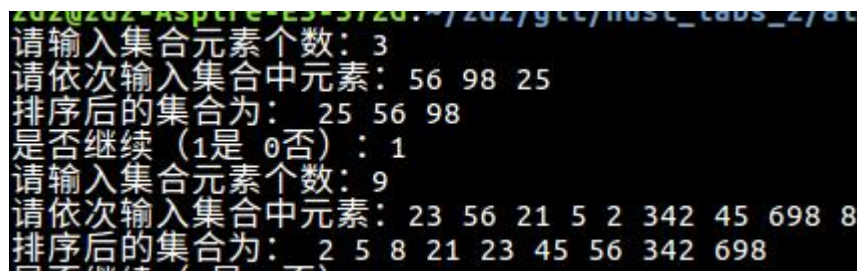
考虑到实验二(基于二次取中的选择算法)也需用到 PARTITION 和 INTERCHANGE，故将这两个过程单独用文件实现，最后藉由头文件完成实验

一的快速排序算法。另写 test_quicksort2.c 进行测试（手动输入数据）。

源程序见附录一。

5. 实验结果

取几组不同的数据皆能得到正确的排序结果，以下图 1.3 仅取两个例子进行展示。



```
202@202-ASP-10-ES-5720:~/202/gcc/m03c_c005_2/8c
请输入集合元素个数: 3
请依次输入集合中元素: 56 98 25
排序后的集合为: 25 56 98
是否继续 (1是 0否): 1
请输入集合元素个数: 9
请依次输入集合中元素: 23 56 21 5 2 342 45 698 8
排序后的集合为: 2 5 8 21 23 45 56 342 698
是否继续 (1是 0否):
```

图 1.3 quicksort 结果截图

符合实验预期。

6. 结果分析

快速分类算法的最坏情况时间是 $O(n^2)$ ，而平均情况时间是 $O(n \log n)$ 。

接着考虑空间复杂度，若采用递归，其最大深度可达到 $n-1$ ，所需的栈空间是 $O(n)$ 。现迭代模型由于每次都是把文件分为了两部分，递归深度也变为约原来的二分之一，故其所需的栈空间缩小，可以得出空间复杂度表达式是：

$$S(n) = \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

实验二.基于二次取中的选择算法

1. 实验目的与要求

编程实现算法 4.17:基于二次取中的选择算法, 编制以下过程:

PARTITION

INSERTIONSORT

INTERCHANGE

SELECT2

编制测试数据,给出实验结果:给出几个不同的数据集,并选择几个不同 r 值进行测试。

2. 算法设计

1. PARTITION 和 INTERCHANGE 过程见实验一算法设计。

2. INSERTIONSORT 过程

设计思路:

从第二个元素开始, 将其移到左边第一个不大于它的元素之后并同时整体移动数组。移完之后继续对下一个进行同样的操作直至所有元素都移到比其小的元素左边。

其流程图见下图 2.1 所示。

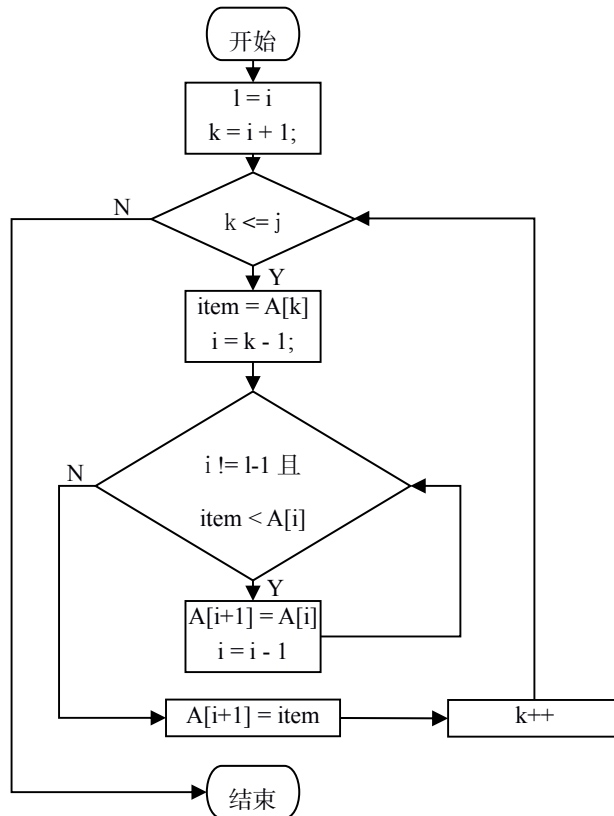


图 2.1 insertionsort 过程流程图

3. SELECT2 过程

设计思路：

对于 $A[m:p]$ 要查找第 k 个元素，若元素数不超过一组的 (r 个)，则直接使用 insertionsort 进行排序，返回 $m+k-1$ 。

否则计算各组中间值（为方便后面计算，将中间值收集到 $A[m:p]$ 的前部），再计算这些中间值的中间值并将其作为划分元素调用 partition 对 $A[m:p]$ 进行划分，若中间值恰为第 k 个，则返回此时中间值的下标，若偏大则在小的一堆中进行查找，偏小的话反之。

其流程图如下图 2.2 所示：

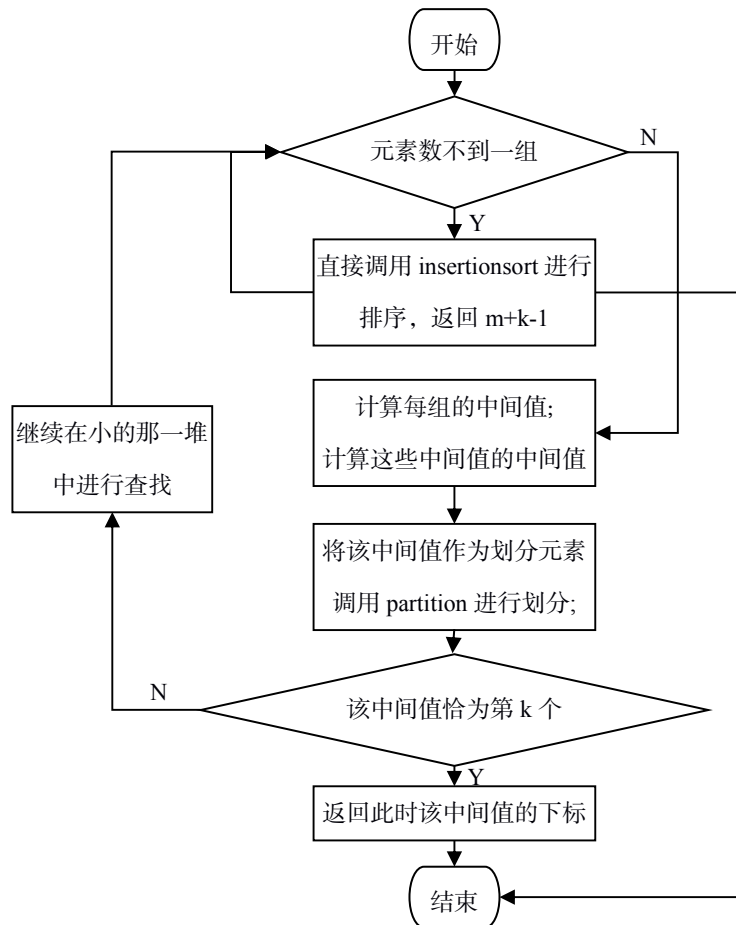


图 2.2 select2 过程流程图

3. 实验环境

操作系统：ubuntu16.04

编程语言：C 语言

编译工具：gcc

4. 实验过程

部分相关源程序在实验一中以完成，新完成 insertionsort.c 和 select2.c。另外编写 test_select2.c 进行测试（手动输入数据）。

源程序见附录一。

5. 实验结果

结果（选取部分）如下图 2.3 所示。

```
请输入集合元素个数：5
请依次输入集合中元素：1 2 3 4 5
请输入要找第几小元素（1 到 5）：4
请输入每组元素个数(大于1的整数)：3
第4小元素为：4
是否继续（1是 0否）：1
请输入集合元素个数：9
请依次输入集合中元素：25 36 15 48 2 36 56 9 87
请输入要找第几小元素（1 到 9）：3
请输入每组元素个数(大于1的整数)：15
第3小元素为：15
是否继续（1是 0否）：1
请输入集合元素个数：4
请依次输入集合中元素：56 25 12 36
请输入要找第几小元素（1 到 4）：2
请输入每组元素个数(大于1的整数)：2
第2小元素为：25
```

图 2.3 select2 结果截图

符合实验预期。

6. 结果分析

该算法每步至多需要 $O(n)$ 时间，故其最坏情况时间复杂度也是 $O(n)$ 。

由于用迭代代替了递归，故其空间复杂度比其原算法也会优越许多。

实验三.Dijkstra 算法

1. 实验目的与要求

算法 5.10(Dijkstra) SHORTEST-PATHS 求出了 v_0 至其它各结点的最短路径,但是没有给出这些最短路径。

补充算法 5.10,使新算法在找出这些最短路径长度的同时,也能求出路径上的结点序列。

要求:

- 给出新算法的描述
- 编写该算法的程序
- 用书上的实例(或自行设计测试数据)测试程序,输出测试结果。基本形式如下:

start	end	length	nodes list
v_1	v_2	20	v_1v_2
v_1	v_4	30	v_1v_4
v_1	v_6	80	$v_1v_2v_3v_5v_6$

2. 算法设计

设计思路: 首先将源点 v 计入 S , 初始化各结点到源点的最短路径长度, 同时初始化各最短路径 (此时只有 v 到当前结点)。

接着找出最小的 $DIST[u]$, 结点 u 计入 S , 更新剩下的结点 w 到源点的最短路径与长度, 更新过程: 若 $DIST[u]+DIST[w]$ 比原 $DIST[w]$ 更小, 则最短路径长度更新为 $DIST[u]+DIST[w]$, 同时将 w 的最短路径改为源点到 u 的最短路径再加

上 u 到 w 。

其流程图如下所示：

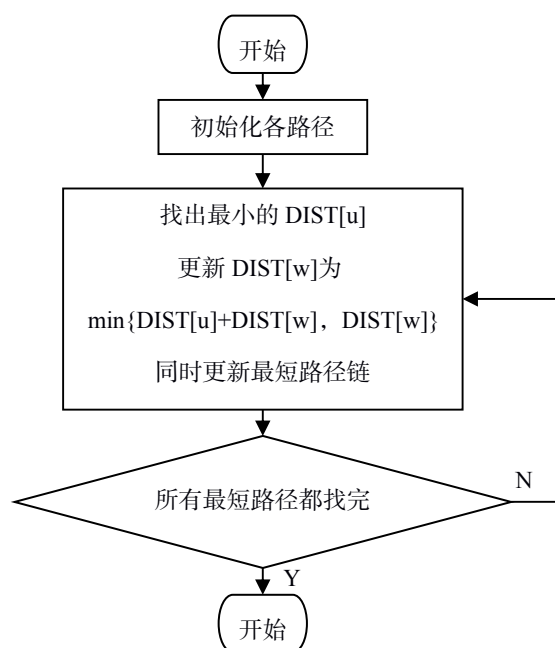


图 3.1 Dijkstra 算法流程图

3. 实验环境

操作系统：ubuntu16.04

编程语言：C 语言

编译工具：gcc

4. 实验过程

编写 shortestpath.c, 在其中写 main 函数进行测试(使用书上图 5.10 的数据)。

源程序见附录二。

5. 实验结果

结果如下图 3.2 所示，与书上的数据相同，符合实验预期(其他测试也符合，在此不一一列出)。

```
zdz@zdz-Aspire-E5-572G:~/zdz/gcc/hust_lab5_
```

start	end	length	nodes list
V1	V2	20	V1 V2
V1	V3	45	V1 V2 V3
V1	V4	30	V1 V4
V1	V5	70	V1 V2 V3 V5
V1	V6	80	V1 V2 V3 V5 V6
V1	V7	130	V1 V2 V3 V5 V6 V7

```
zdz@zdz-Aspire-E5-572G:~/zdz/gcc/hust_lab5_
```

图 3.2 Dijkstra 算法结果截图

6. 结果分析

Dijkstra 算法是一种贪心算法，容易看出这个算法是正确的。

在 n 结点图上，算法的时间复杂度是 $O(n^2)$ 。为了算出每一条路径，主要循环过程要执行 $n-2$ 次，每次修改 DIST 的时间复杂度又为 $O(n)$ ，故总的时间复杂度为 $O(n^2)$ 。

任何最短路径算法都必须至少检查这个图中的每条边一次，故算法的最小时间便是 $O(e)$ 。由于用邻接矩阵来表示图，要确定哪些边在 G 中正好需要 $O(n^2)$ 时间，故使用这种表示法的任何最短路径算法必定花费 $O(n^2)$ 时间。于是，对于这种表示法，该算法在一个常因子范围内是最优的。

附加一.每对结点之间的最短路径长度

1. 实验目的与要求

利用动态规划的方法求取带权图上每对结点之间的最短路径长度。

2. 算法设计

设计思路：依次考虑最高下标为 k 的结点的路径 (k 从 1 到 n)，对于所有可能的结点对，更新其最短路径长度为 $\min\{A(i,j), A(i,k)+A(k,j)\}$ ，即原最短路径长度与经由 k 的路径长度中的较小者。其流程图如下图 4.1 所示：

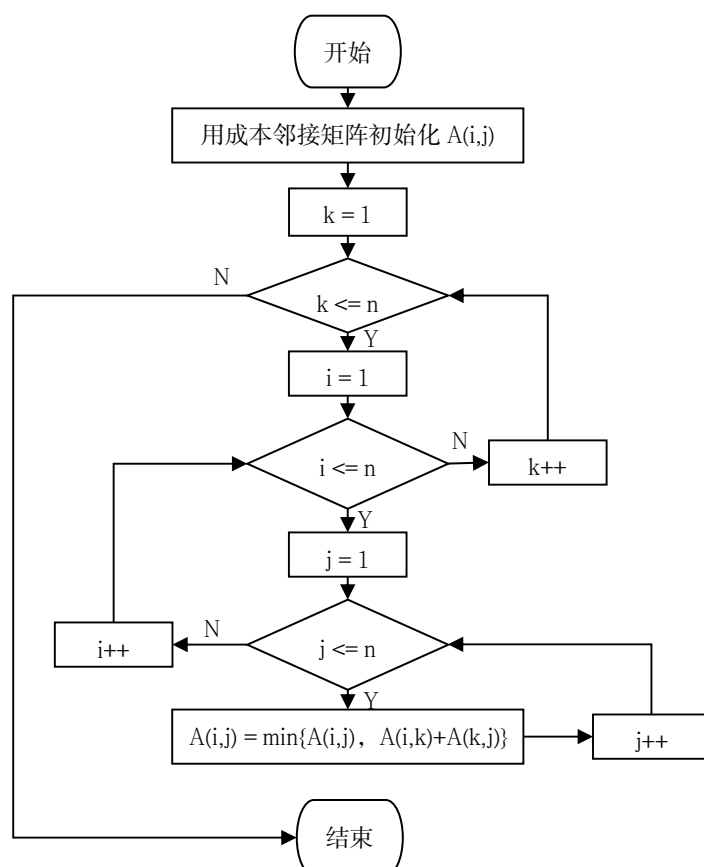


图 4.1 每对结点之间最短路径算法流程图

3. 实验环境

操作系统：ubuntu16.04

编程语言：C 语言

编译工具：gcc

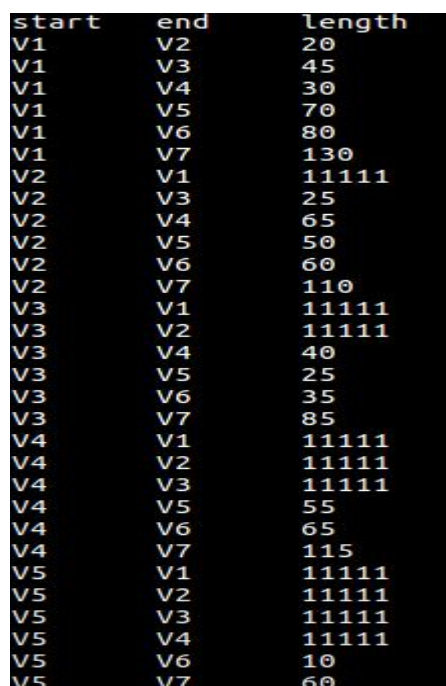
4. 实验过程

编写 all_paths.c，在其中写 main 函数进行测试（使用书上图 5.10 的数据）。

源程序见附录三。

5. 实验结果

结果如下图 4.2 所示，与书上的数据相同，符合实验预期（其他测试也符合，在此不一一列出）。(因结果太长，仅截取了部分，其中 11111 表示无穷，即到不了)



start	end	length
V1	V2	20
V1	V3	45
V1	V4	30
V1	V5	70
V1	V6	80
V1	V7	130
V2	V1	11111
V2	V3	25
V2	V4	65
V2	V5	50
V2	V6	60
V2	V7	110
V3	V1	11111
V3	V2	11111
V3	V4	40
V3	V5	25
V3	V6	35
V3	V7	85
V4	V1	11111
V4	V2	11111
V4	V3	11111
V4	V5	55
V4	V6	65
V4	V7	115
V5	V1	11111
V5	V2	11111
V5	V3	11111
V5	V4	11111
V5	V6	10
V5	V7	60

图 4.2 每对结点之间最短路径结果部分截图

6. 结果分析

对比于用 Dijkstra 算法求取每对结点之间的最短路径，虽然两个算法的时间复杂度都是 $O(n^3)$ ，但 Dijkstra 算法需要维护 S （已加入的结点）数组，同时还要比较当前考虑的结点是否在 S 中，这就增加了时间与空间开销。所以虽然两算法的时间复杂度相同，但是无论从时间还是空间方面来考虑，使用该算法都比用 Dijkstra 算法多次更改源点算出每对结点之间的最短路径要来的高效。

附加二.n 皇后求解

1. 实验目的与要求

使用回溯的方法求出 n 皇后问题有多少个解。

2. 算法设计

设计思路：

使用回溯的策略，将 n 个皇后放在不同的 n 行，从第一行第一列开始依次考虑每一行放在哪一列可行。

若是在当前行可以找到符合要求的列来放置该行的皇后，则放置在该列并继续下一行，直至全部放完（此时解的个数加一）或有一行找不到可以放置的列。

否则若是在该行找不到可以放置的列，则返回上一行，上一行继续找可以放置的列。

对于能否放置的判断，一是不能与前面的列重复，二是行之差的绝对值不能等于列之差的绝对值（即不能在同一斜对角）。

不断查找直到把所有可能的摆放方式全部找完，此时计数器为解的总个数。

其流程图如下图 5.1 所示：

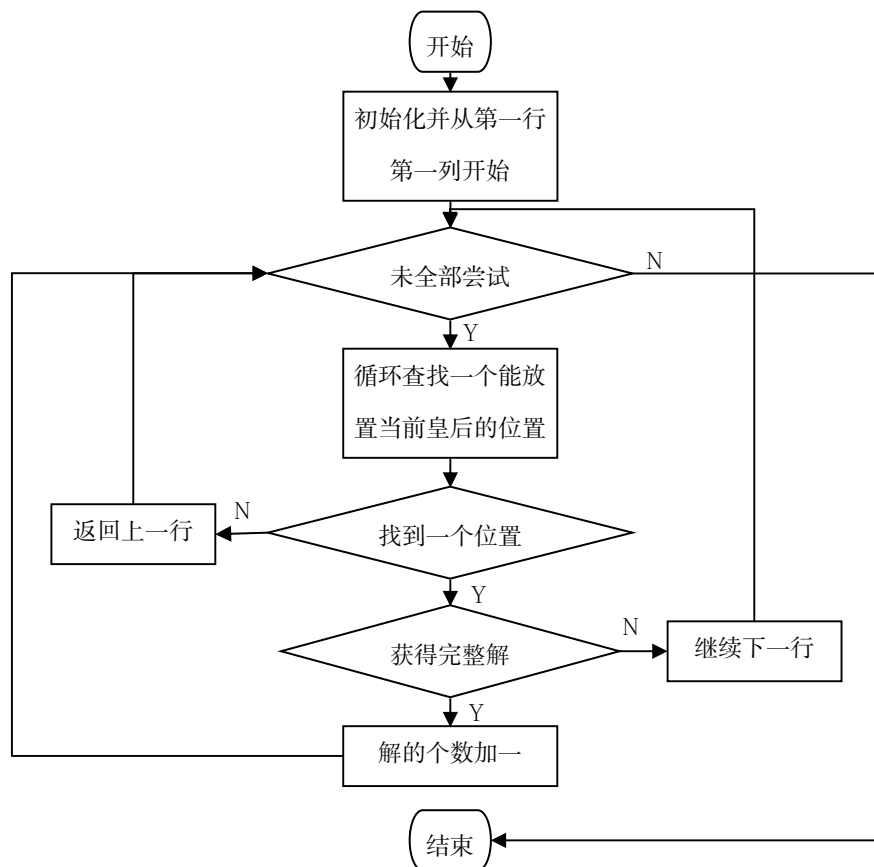


图 5.1 n 皇后问题流程图

3. 实验环境

操作系统：ubuntu16.04

编程语言：C 语言

编译工具：gcc

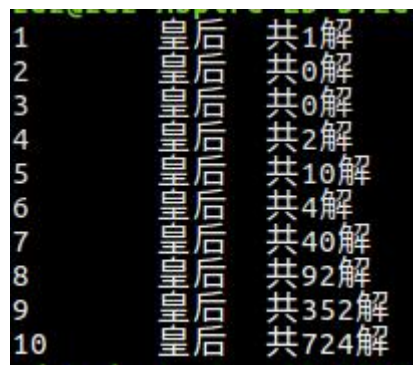
4. 实验过程

编写 nqueens.c 并在 main 函数中测试 1 到 10 皇后的解的个数（再多则计算时间稍长在此不列出）。

源程序见附录四。

5. 实验结果

实验结果如下图 5.2 所示。符合实际。



1	皇后	共1解
2	皇后	共0解
3	皇后	共0解
4	皇后	共2解
5	皇后	共10解
6	皇后	共4解
7	皇后	共40解
8	皇后	共92解
9	皇后	共352解
10	皇后	共724解

图 5.1 n 皇后问题结果截图

6. 结果分析

采用回溯算法将所有可能的情况情况检查一遍虽然效率较低,但是不失为一个求取答案的好策略。同时,因为加入了限界函数,回溯法并不是一味的暴力穷举,对于某些错误的放置会立即停止继续往下的搜索而直接回溯,故而其实际效率比想象的要高。

附录一.分治法源代码

1.头文件 divide.h

```
#ifndef DIVIDE_H
#define DIVIDE_H
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// 函数声明
void partition(int m, int *p, int *A);
void quicksort2(int p, int q, int *A, int n);
void interchange(int *x, int *y);
void insertionsort(int *A, int i, int j);
int select2(int *A, int m, int p, int k, int r);
#endif
```

2.partition.c

```
#include "divide.h"

/**
 * @brief 用 A[m]划分集合 A[m:p-1]
 * @arg m: int 型, 选取划分元素 A[m]
 * @arg p: int *型, 选取 A[p-1],其带着划分元素所在下标退出
 * @arg A: int *型, 指向待划分集合
 * @retval void
 */
void partition(int m, int *p, int *A)
{
    int i = m; // i 用于记录下标,初始为 m
    int v = A[m]; // v 用于暂存划分元素

    while (1)
    {
        // i 由左向右移, 直到找到大于划分元素的
        do {
            i += 1;
        } while (i != *p && A[i] < v);
        // p 由右向左移, 直到找到小于划分元素的
        do {
            *p -= 1;
        } while (A[*p] > v);

        if (i < *p)
        {
            interchange(&A[i], &A[*p]);
        }
    }
}
```

```

        }
    else
    {
        break;
    }
}
A[m] = A[*p];
A[*p] = v;
}

```

2.interchange.c

```

#include "divide.h"
/**
 * @brief 交换元素的值
 * @arg x: int *型, 代交换元素 x 地址
 * @arg y: int *型, 待交换元素 y 地址
 * @retval void
 */
void interchange(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

```

3.insertionsort.c

```

#include "divide.h"
/**
 * @brief 插入排序(非降次序)
 * @arg A: int *型, 指向待排序集合
 * @arg i: int 型, 起始下标
 * @arg j: int 型, 截止下标
 * @retval void
 */
void insertionsort(int *A, int i, int j)
{
    int k;
    int l = i;
    int item;
    for (k = i + 1; k <= j; k++)
    {
        item = A[k];
        i = k - 1;
        while ((i != (l-1)) && item < A[i])
        {
            A[i+1] = A[i];
            i = i - 1;
        }
        A[i+1] = item;
    }
}

```



```

    }
}

```

4.select2.c

```

#include "divide.h"
/**
 * @brief 基于二次取中的选择算法
 * @arg A: int *型, 指向元素集合
 * @arg m: int 型, 起始下标
 * @arg p: int 型, 截止下标
 * @arg k: int 型, 要查第 k 小元素
 * @arg r: int 型, 每组元素个数
 * @retval int
 * @return 第 k 小元素下标
 */
int select2(int *A, int m, int p, int k, int r)
{
    int n, i, j;
    while (1)
    {
        if ((p-m+1) <= r) // 只有一组则直接用 insertionsort
        {
            insertionsort(A, m, p);
            return (m + k - 1);
        }
        n = p - m + 1; //元素数

        for (i = 1; i <= floor((double)n/r); i++)
        {
            // 计算中间值
            insertionsort(A, m + (i-1) * r, m + i * r - 1);
            // 将中间值收集到 A[m, p]的前部
            interchange(&A[m+i-1], &A[m + (i-1) * r + (int)ceil((double)r/2) - 1]);
        }
        // 二次取中
        j = select2(A, m, m + floor((double)n/r) - 1, ceil(floor((double)n/r) / 2), r);
        interchange(&A[m], &A[j]); // 产生划分元素
        j = p + 1;
        partition(m, &j, A);
        if ((j-m+1) == k)
        {
            return j;
        }
        else if ((j-m+1) > k) // 偏大则在小的一堆中查
        {
            p = j - 1;
        }
        else
        {

```

```

        k = k - (j-m+1);
        m = j + 1;
    }
}
}

```

4.quicksort2.c

```

#include "divide.h"
/**
 * @brief  quicksort 迭代模型
 * @arg     p:      int 型，表示从下标为 p 的元素开始
 * @arg     q:      int 型，表示到下标为 q 的元素结束
 * @arg     A:      int *型，指向待划分集合
 * @arg     n:      int 型，数组元素个数
 * @retval  void
 */
void quicksort2(int p, int q, int *A, int n)
{
    int j;
    int top = -1;
    int *STACK = (int *)malloc(sizeof(int) * (int)floor(log((double)n) / log((double)2))); // 用数组实现栈
    while (1)
    {
        while (p < q)
        {
            j = q + 1;
            partition(p, &j, A);
            if ((j - p) < (q - j)) // 先对较小的子文件进行分类
            {
                STACK[top+1] = j + 1;
                STACK[top+2] = q;
                q = j - 1;
            }
            else
            {
                STACK[top+1] = p;
                STACK[top+2] = j - 1;
                p = j + 1;
            }
            top = top + 2;
        }
        if (top == -1)
        {
            return ;
        }
        // 再对较大的进行分类
        q = STACK[top];
        p = STACK[top-1];
        top = top - 2;
    }
}

```

```
}
```

```
// 相关测试文件略去
```

附录二.单源点最短路径源代码

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a,b) (a<b)?a:b
#define MAXD 11111

typedef struct node {
    int v; // 结点号
    struct node *next;
}node;

typedef struct nodelist {
    struct node *head; // 指向链表头
    struct node *tail; // 指向链尾
}nodelist;

/**
 * @brief 获取并 old 的路径并在此基础上增加自身结点
 * @arg new: nodelist *型, 指向待修改 nodelist
 * @arg old: nodelist *型, 指向添加进的 nodelist
 * @arg v: int 型, 指向添加的结点号
 * @retval void
 */
void get_add_nl(nodelist *new, nodelist *old, int v)
{
    node *tempnode = (node *)malloc(sizeof(node));
    node *temp = old->head;
    tempnode->next = NULL;
    if (temp != NULL)
    {
        tempnode->v = temp->v;
        new->head = tempnode;
        new->tail = tempnode;
        temp = temp->next;
        while (temp != NULL)
        {
            tempnode = (node *)malloc(sizeof(node));
            tempnode->v = temp->v;
            tempnode->next = new->tail->next;
            new->tail->next = tempnode;
            new->tail = new->tail->next;
            temp = temp->next;
        }
        tempnode = (node *)malloc(sizeof(node));
        tempnode->v = v;
        tempnode->next = new->tail->next;
```

```

        new->tail->next = tempnode;
        new->tail = new->tail->next;
    }
}

/**
 * @brief  Dijkstra 算法生成最短路径
 * @arg      v:      int 型, 表示源点标号
 * @arg      COST:   int *型, 指向成本邻接矩阵
 * @arg      DIST:   int *型, 存放结点 v 到结点 j 的最短路径长度
 * @arg      n:      结点个数
 * @retval
 */
void shortest_paths(int v, int *COST, int *DIST, int n, nodelist *NODELIST)
{
    int *S = (int *)malloc(sizeof(int) * n);
    int u, num, i, w;
    node *tempnode = (node *)malloc(sizeof(node));
    for (i = 0; i < n; i++)
    {
        S[i] = 0;
        DIST[i] = COST[v*n+i];    // 即 COST[v][i]
    }
    // 结点 v 计入 S
    S[v] = 1;
    DIST[v] = 0;
    // 初始化各路径
    tempnode->v = v+1;
    tempnode->next = NULL;
    NODELIST[v].head = tempnode;
    NODELIST[v].tail = tempnode;
    for (w = 0; w < n; w++)
    {
        if (S[w] == 0)
        {
            get_add_nl(&NODELIST[w], &NODELIST[v], w + 1);
        }
    }

    for (num = 1; num < n; num++)    // 确定由结点 v 出发的 n-1 条路
    {
        u = -1;
        for (w = 0; w < n; w++)        // 找出最小的 DIST[u]
        {
            if (0 == S[w])
            {
                if (-1 == u)
                {
                    u = w;
                }
            }
        }
    }
}

```

```

        else
        {
            if (DIST[w] < DIST[u])
            {
                u = w;
            }
        }
    }
}
S[u] = 1; // 结点 u 计入 S
for (w = 0; w < n; w++)
{
    if (0 == S[w])
    {
        // DIST[w] = MIN(DIST[w], DIST[u] + COST[u*n+w]);
        if ((DIST[u]+COST[u*n+w]) < DIST[w])
        {
            DIST[w] = DIST[u] + COST[u*n+w];
            get_add_nl(&NODELIST[w], &NODELIST[u], w + 1);
        }
    }
}
}

int main(int argc, int *argv[])
{
    int n; // 结点数
    int v; // 源点标号
    int *DIST; // 存放最短路径长度
    int *COST; // 指向成本邻接矩阵
    nodelist *NODELIST; // 存放路径
    int i, j;
    n = 7;
    COST = (int *)malloc(sizeof(int) * n * n);
    DIST = (int *)malloc(sizeof(int) * n);
    NODELIST = (nodelist *)malloc(sizeof(nodelist) * n);
    v = 1;
    int a[49] = {0, 20, 50, 30, MAXD, MAXD, MAXD,
                MAXD, 0, 25, MAXD, MAXD, 70, MAXD,
                MAXD, MAXD, 0, 40, 25, 50, MAXD,
                MAXD, MAXD, MAXD, 0, 55, MAXD, MAXD,
                MAXD, MAXD, MAXD, MAXD, 0, 10, 70,
                MAXD, MAXD, MAXD, MAXD, MAXD, 0, 50,
                MAXD, MAXD, MAXD, MAXD, MAXD, MAXD, 0};
    COST = a;
    shortest_paths(v-1, COST, DIST, n, NODELIST);
    printf("start\tend\tlength\tnodes list\n");
    for (i = 0; i < n; i++)
    {
        if ((i+1) != v)

```

```

    {
        printf("V%d\tV%d\t%d\t", v, i+1, DIST[i]);
        node *temp = NODELIST[i].head;
        while (temp != NULL)
        {
            printf("V%d ", temp->v);
            temp = temp->next;
        }
        printf("\n");
    }
}
return 0;
}

```

附录三.每对结点最短路径源代码

```
#include <stdio.h>
#include <stdlib.h>

#define MIN(a,b) (a<b)?a:b
#define MAXD 11111

/**
 * @brief 计算每对结点之间的最短路径长度
 * @arg COST: int *型, 成本邻接矩阵
 * @arg A: int *型, A[i][i]是结点 Vi 到 Vj 的最短路径的成本
 * @arg n: int 型, 结点个数
 * @retval void
 */
void all_paths(int *COST, int *A, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++) // 将 COST[i][j]复制到 A[i][j]
    {
        for (j = 0; j < n; j++)
        {
            A[i*n+j] = COST[i*n+j];
        }
    }
    for (k = 0; k < n; k++) // 对最高下标为 k 的结点路径
    {
        // 对于所有可能的结点对
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                A[i*n+j] = MIN(A[i*n+j], A[i*n+k]+A[k*n+j]);
            }
        }
    }
}

int main(int argc, int *argv[])
{
    int n; // 结点数
    int *A; // 存放最短路径长度
    int *COST; // 指向成本邻接矩阵
    int i, j, v;
    n = 7;
    COST = (int *)malloc(sizeof(int) * n * n);
```



```

A = (int *)malloc(sizeof(int) * n * n);
// NODELIST = (nodelist *)malloc(sizeof(nodelist) * n);
int a[49] = {0, 20, 50, 30, MAXD, MAXD, MAXD,
             MAXD, 0, 25, MAXD, MAXD, 70, MAXD,
             MAXD, MAXD, 0, 40, 25, 50, MAXD,
             MAXD, MAXD, MAXD, 0, 55, MAXD, MAXD,
             MAXD, MAXD, MAXD, MAXD, 0, 10, 70,
             MAXD, MAXD, MAXD, MAXD, MAXD, 0, 50,
             MAXD, MAXD, MAXD, MAXD, MAXD, MAXD, 0};

COST = a;
all_paths(COST, A, n);
printf("start\tend\tlength\n");
for (v = 0; v < n; v++)
{
    for (i = 0; i < n; i++)
    {
        if (i != v)
        {
            printf("V%d\tV%d\t%d\t", v+1, i+1, A[v*n+i]);
            printf("\n");
        }
    }
}
return 0;
}

```

附录三.n 皇后问题源代码

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

#define MAX 10

double nqueens(int n);
int place(int k, int *X);

/**
 * @brief 判断当前位置能否放置皇后
 * @arg k: int 型, 表示当前行
 * @arg X: int *型, 存储当前放置方式
 * @retval int
 * @return 返回能否放置, 1 能, 0 否
 */
int place(int k, int *X)
{
    int i = 0;
    while (i < k)
    {
        if ((X[i] == X[k]) || (abs(X[i]-X[k]) == abs(i-k)))
        {
            return 0;
        }
        i += 1;
    }
    return 1;
}

/**
 * @brief 求解 n-皇后问题的解的个数
 * @arg n: int 型, 表示几皇后
 * @retval double 型
 * @return 解的个数
 */
double nqueens(int n)
{
    double answer_num = 0; // 存储解的个数
    int k, *X; // k 是当前行, X[k]是当前列
    X = (int *)malloc(sizeof(int) * n);

    X[0] = -1;
    k = 0;
    while (k >= 0)
```

```

{
    X[k] = X[k] + 1; // 移到下一行
    while (X[k] < n && !place(k, X)) // 判断此处能否放这个皇后
    {
        X[k] = X[k] + 1;
    }
    if (X[k] < n)    // 找到一个位置
    {
        if ((n-1) == k) // 是一个完整解
        {
            answer_num++; // 解的个数加 1
        }
        else
        {
            // 转向下一行
            k += 1;
            X[k] = -1;
        }
    }
    else
    {
        k -= 1;    // 回溯
    }
}
return answer_num;
}

int main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < MAX+1; i++)
    {
        printf("%d\t 皇后    共%.0lf 解\n", i, nqueens(i));
    }
    return 0;
}

```