华东师范大学计算机科学与技术专业上机实践报告

课程名称：人工智能	**年级**：2022 级	**上机实践成绩**：
指导教师：	**姓名**：杨明远	**创新实践成绩**：
上机实践名称：搜索与优化	**学号**：12215102401	**上机实践日期**：
上机实践编号：No.1	**组号**：	**上机实践时间**：

A. 八数码问题

(1) 问题介绍

- 问题描述

本代码是为了解决经典的八数码问题（也称为滑动拼图问题），使用广度优先搜索（BFS）算法来找到从初始状态到目标状态的路径。八数码问题是一个解决在 3x3 网格上通过滑动空格（代表为'x'）来重新排列数字以达到目标顺序的难题。

(2) 程序设计与算法分析

1. 算法1

- 算法描述

广度优先搜索（BFS）

- 算法流程

使用队列来持续处理状态，直到找到解决方案或所有可能的状态都被探索完毕。

对于队列中的每个状态，检查是否达到目标状态。如果是，则输出路径并返回成功信号。

否则，尝试将空格 'x' 向四个方向（上下左右）移动，生成新的状态。这一过程中要确保移动的合法性（如不能越界或回到之前的状态）。

生成的新状态若未被访问过，则加入队列继续搜索，并在 visited 映射中标记该状态为已访问，并记录路径。

实验结果与分析

函数 update_position：

此辅助函数用于检查和执行空格 'x' 的移动。

接受当前状态、空格索引、新索引、新路径字符串和移动方向。

确保移动不会导致越界，并在合法的情况下执行交换并更新路径。

特点和优化：

封装与重用：将空格移动的逻辑封装在 update_position 函数中，增强了代码的可读性和可维护性。

清晰的状态管理：使用结构体 State 来管理每个状态的访问情况和路径，避免使用多个独立的数据结构。

无全局变量：不使用全局变量来存储路径或结果，而是直接在找到解决方案时输出路径。如果没有解决方案，则输出“unsolvable”。

```cpp
struct State {
    string path;
    bool visited = false;
};

bool update_position(string& state, int zero_idx, int new_idx, string& new_path, char move)
    if (new_idx < 0 || new_idx >= 9 ||  // out of bounds
        (move == 'l' && zero_idx % 3 == 0) ||  // edge cases
        (move == 'r' && zero_idx % 3 == 2)) {
        return false;
    }
    swap(state[zero_idx], state[new_idx]);
    new_path += move;
    return true;
}

int bfs(const string& aim, queue<string>& q, map<string, State>& visited) {
    while (!q.empty()) {
        string current = q.front();
        q.pop();
        int zero_idx = current.find('x');
        vector<int> shifts = {-3, 3, -1, 1}; // up, down, left, right

        if (current == aim) {
            cout << visited[current].path << endl;
            return 1;
        }

        for (int i = 0; i < 4; i++) {
            string new_state = current;
            string new_path = visited[current].path;
            if (update_position(new_state, zero_idx, zero_idx + shifts[i], new_path, direct
                !visited[new_state].visited) {
                visited[new_state].visited = true;
                visited[new_state].path = new_path;
                q.push(new_state);
            }
        }
    }
    return 0;
}
```

```cpp
int main() {
    string initial_state;
    cin >> initial_state;

    queue<string> q;
    map<string, State> visited;
    visited[initial_state] = {"", true};
    q.push(initial_state);

    if (!bfs(target, q, visited)) {
        cout << "unsolvable" << endl;
    }

    return 0;
}
```

C++17	Accepted: 100	0.396	27.980

B. 路径导航

(1) 问题介绍

- 问题描述

本问题是关于在一个给定的无向图中寻找两个顶点之间的最短路径。这个图由 n 个节点和 m 条边构成，每条边都有一个非负权重，代表两个节点之间的距离。需要编写一个函数来计算和返回图中特定起点（节点 s）到目标点（节点 t）之间的最短路径长度。

- 求解方法介绍

这段代码实现了A*算法，使用了一个优先队列来管理待探索的节点。每个节点都计算了从起点到该节点的实际成本 g 和从该节点到目标的启发式估计成本 h，并将两者相加得到 f 值。节点以 f 值的优先级顺序被探索，以确保算法首先考虑最有可能达到目标的路径。

启发式函数 heuristic 在这里简单地用坐标差的绝对值来估计成本，这只是一个示例。在实际应用中，这个函数需要根据问题的具体情况来设计，以确保它既能提供有效的估计，又不会过分增加计算负担。

(2) 程序设计与算法分析

1. 算法1

A*算法

- 算法描述

为了解决这个问题，我们将采用 A*搜索算法（A-star algorithm）。A* 算法是一种有效的路径搜索和图遍历方法，它具有较高的效率和准确性，尤其适用于路径最短问题。A* 算法结合了最佳优先搜索的高效性和 Dijkstra 算法的实用性，通过一个启发式函数来评估剩余路径的预计成本，从而减少搜索空间。

- 算法流程

1初始化：

对每个节点设置一个距离值 g，表示从起点到该点的已知最短距离。起始点 s 的 g 值设为 0，其他所有点的 g 值设为无穷大。

定义一个启发式函数 h，用于估计任一节点到目标节点 t 的距离。这个函数是启发式的核心，它的选择对算法性能有重要影响。

计算每个节点的 f值，f 值是 g 值和 h 值的和，用于评估从起点经过该节点到达目标点的预估最短距离。

2. 处理队列：

使用一个优先队列（或称为开放列表）来存储待处理的节点，根据节点的 f 值进行排序，以保证每次都处理当前最有希望的节点。

从优先队列中取出一个节点，检查它是否为目标节点 t。如果是，算法结束，返回该节点的 g 值作为最短路径长度。

否则，更新该节点所有可达邻居的 g值和 f 值。如果通过当前节点到达邻居节点的路径比已知的路径更短，更新邻居的 g 值和 f 值，并将其添加到优先队列中。

3. 循环执行上述步骤，直到优先队列为空或找到目标节点。

2. 实验结果与分析

启发式函数 (h)：此函数提供了一个估计值，它是从当前节点到目标节点的预估最小成本。这个示例中使用了坐标差，但可以根据具体的应用场景使用更复杂的计算方法，如在地图应用中使用欧几里得距离或曼哈顿距离。

2. 优先队列：使用自定义比较器，根据 f 值的大小来确定优先级，确保总是先处理最有可能是最短路径的节点。

3. 更新机制：当通过一个节点到达其邻居节点提供了一个更短的路径时，更新该邻居节点的 g 值和 f 值，并重新将其放入优先队列中。

```
14  // 启发式函数 h: 估计点到目标点的最小成本，这里使用简单的坐标差
15  double h(int current, int target) {
16      return fabs(coordinates[current] - coordinates[target]);
17  }
18
19  double A_star(int s, int t) {
20      int n = adjacencyList.size();
21      vector<double> g(n, INT_MAX); // 从起点到该点的最短已知距离
22      unordered_map<int, double> f;  // 记录每个节点的 f 值
23      auto cmp = [&](int left, int right) { return f[left] > f[right]; };
24      priority_queue<int, vector<int>, decltype(cmp)> pq(cmp);
25
26      g[s] = 0;
27      f[s] = g[s] + h(s, t);
28      pq.push(s);
29
30      while (!pq.empty()) {
31          int current = pq.top();
32          pq.pop();
33
34          if (current == t) { // 找到目标点
35              return g[t];
36          }
37
```

```
        for (auto& edge : adjacencyList[current]) {
            int neighbor = edge.first;
            double weight = edge.second;
            double g_new = g[current] + weight;

            if (g_new < g[neighbor]) {
                g[neighbor] = g_new;
                f[neighbor] = g[neighbor] + h(neighbor, t);
                pq.push(neighbor);
            }
        }
    }

    return -1; // 如果没有路径可以到达目标
}
```

C. TSP问题

(1) 问题介绍

- 问题描述
- 求解方法介绍

(2) 程序设计与算法分析

1. 算法1
   - 算法描述

   模拟退火（Simulated Annealing，SA）

   是一种启发式搜索算法，用于找到给定问题的近似全局最优解。
   - 算法流程

1. 初始化（init）：初始化温度、当前解等参数。当前解可以是随机生成的路径。
2. 邻域函数（N）：通过交换路径中的两个节点生成一个邻域解。
3. 能量差计算：计算新旧解的路径长度差异。
4. 接受概率（calc_p）：根据温度和能量差确定是否接受新解。
5. 降温（drop）：按一定的规则降低系统温度。
6. 终止条件（termi）：确定算法何时停止，可以是温度低于某阈值或达到一定迭代次数。

2. 实验结果与分析

1. 从1号城市开始，随机生成一个初始解。
2. 在每次迭代中，通过邻域函数生成一个新解。
3. 如果新解比当前解更好，或者根据接受概率决定接受较差的新解，则将当前解更新为新解。
4. 更新温度，重复上述过程直到满足终止条件。

5. 输出最短路径长度及对应的路径

```
// 初始化参数
void init() {
    srand(time(NULL)); // 设置随机数种子
    t = 11.2; // 初始温度
    times = 0; // 初始化迭代计数器
    k = 110; // 降温前的最大迭代次数
}

// 终止条件判断
bool termi() {
    return t < 1.5; // 当温度低于1.5时终止
}
```

```
vector<int> N(const vector<int>& p) {
    vector<int> ret(p);
    int x1 = rand() % (n - 1) + 1; // 保证不选择第一个元素
    int x2 = rand() % (n - 1) + 1; // 保证不选择第一个元素

    if (x1 > x2) swap(x1, x2); // 确保x1小于x2
    while (x1 < x2) {
        swap(ret[x1], ret[x2]);
        x1++;
        x2--;
    }
    return ret;
}

// 降温函数
double drop() {
    if (times < k * n) {
        times++;
        return t;
    } else {
        times = 0;
        return t * 0.92; // 降温
    }
```

```
// 计算接受概率
double calc_p(double delta) {
    return exp(-delta / t); // 使用Metropolis准则
}
```

2024-03-24 20:04:28	C++17	Partial score: 100	1.178	0.594

D. N皇后

(1) 问题介绍

- 问题描述

N皇后问题要求在一个NxN的棋盘上放置N个皇后，使得她们互不攻击，即没有两个皇后在同一行、同一列或同一对角线上。该问题是经典的回溯算法案例，也可以使用启发式搜索来解决。

- 求解方法介绍

这个程序会在读取整数N后计算出一个皇后的位置方案，并按行打印每个皇后的列位置。如果N是偶数，它会先填满所有偶数列，然后填满所有奇数列。如果N是奇数，它会类似地工作，但是会有一个位置的特殊处理，以避免在最后一行出现冲突。

(2) 程序设计与算法分析

1. 算法1

- 算法描述

为解决N皇后问题，可以采用基于回溯的深度优先搜索（DFS）策略。算法一次在棋盘上放置一个皇后，在确认了不冲突的情况下，递归地放置下一个皇后。如果发现当前的放置导致之后无法成功放置其他皇后，则回溯到上一步，尝试不同的位置。

- 算法流程

1. 偶数列的放置:

当N为偶数时，算法从第一个偶数列开始，在棋盘上顺序放置一半的皇后。具体来说，就是在第1列放置第一个皇后，第3列放置第二个皇后，以此类推，直到所有偶数列上都放置了皇后。

2. 奇数列的放置:

然后，在奇数列上放置剩余的皇后。在第0列放置下一个皇后，在第2列放置随后的皇后，以此类推，直到所有奇数列上都放置了皇后。

3. 奇数N的特殊处理:

如果N是奇数，上述规则会导致最后一个皇后无法放置。因此，最后一个皇后被放置在棋盘的最后一列。

4. 输出结果:

按照上述规则放置所有皇后后，每行输出一个整数，表示皇后所在的列位置。

2. 实验结果与分析

```
4  using namespace std;
5
6  vector<int> placeQueens(int n) {
7      vector<int> positions(n);
8      if (n % 2 == 0) {
9          // 如果 N 是偶数
10         for (int i = 0; i < n / 2; ++i) {
11             positions[i] = 2 * i + 1;
12             positions[i + n / 2] = 2 * i;
13         }
14     } else {
15         // 如果 N 是奇数
16         for (int i = 0; i < n / 2; ++i) {
17             positions[i] = 2 * (i + 1);
18             positions[i + n / 2] = 2 * i + 1;
19         }
20         positions[n - 1] = n - 1; // 最后一行特殊处理
21     }
22     return positions;
23 }
```

2024-04-12 15:48:12	C++17	Wrong answer: 82	0.253	34.344

E. 地图染色

(1) 问题介绍

- 问题描述

对于这个图着色问题的实例，我们可以采用贪心算法来分配颜色，该算法能够确保相邻的节点不会有相同的颜色。考虑到给定的节点和边数较少（节点数n<=30, 边数m<=300），使用贪心算法是合理的。这个问题也可以通过更高级的算法如回溯法来解决，但贪心算法在大多数情况下能提供较好的解并且计算效率高。

- 求解方法介绍

(2) 程序设计与算法分析

1. 算法1
   - 算法描述

     贪心着色算法
   - 算法流程
     1. 初始化所有节点颜色为未着色。
     2. 对每个节点，遍历其所有邻接节点的颜色，并创建一个可用颜色的集合。
     3. 为节点分配第一个可用的颜色。
     4. 重复步骤2和3，直到所有节点都被着色。
     5. 计算并输出所用的颜色数量以及每个节点的颜色。
2. 实验结果与分析

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <numeric>

using namespace std;

const int MAXN = 300;
vector<int> graph[MAXN];
int color[MAXN], degree[MAXN];

// 将节点按度数降序排序
vector<int> sortByDegree(int n) {
    vector<int> nodes(n);
    iota(nodes.begin(), nodes.end(), 0); // 填充从0开始的n个整数
    sort(nodes.begin(), nodes.end(), [&](int a, int b) {
        return degree[a] > degree[b];
    });
    return nodes;
}
```

```cpp
void greedyColoring(int n) {
    set<int> used_colors;
    vector<int> nodes = sortByDegree(n);

    for (int i = 0; i < n; ++i) {
        int node = nodes[i];
        bool available[MAXN] = {false}; // 标记相邻节点使用的颜色

        for (int adj : graph[node]) {
            if (color[adj] != -1) {
                available[color[adj]] = true;
            }
        }

        // 为节点分配第一个未使用的颜色
        for (int c = 0; ; ++c) {
            if (!available[c]) {
                color[node] = c;
                used_colors.insert(c);
                break;
            }
        }
    }
```

```cpp
int main() {
    int n, m;
    cin >> n >> m;

    fill(color, color + n, -1); // 初始化颜色为-1

    for (int i = 0; i < m; ++i) {
        int x, y;
        cin >> x >> y;
        graph[x].push_back(y);
        graph[y].push_back(x);
        degree[x]++; degree[y]++; // 增加度数
    }

    greedyColoring(n);

    cout << *max_element(color, color + n) + 1 << endl; // 输出颜色种类数量
    for (int i = 0; i < n; ++i) {
        cout << color[i] << endl; // 输出每个节点的颜色
    }

    return 0;
}
```

2024-04-24 21:18:08	C++17	Wrong answer: 59	0.003	0.605

F. 字符路径

(1) 问题介绍

- 问题描述

给定一个 n x m的字符网格和一个字符串 road，每个字符表示网格中的一个地点。目标是找到从字

符串 road 第一个字符对应的所有位置到最后一个字符对应的所有位置的最短曼哈顿距离总和。

- 求解方法介绍

使用动态规划方法结合曼哈顿距离计算，解决多源到多目标的最短路径问题。程序首先读取网格和路径字符串，然后为每个字符在网格中的位置建立映射。接着，使用动态规划逐步计算并更新每一步的最小距离，最后输出从起始字符到终止字符的最小距离。

(2) 程序设计与算法分析

1. 算法1

- 算法描述

1.动态规划计算最短路径

2.最短曼哈顿距离

- 算法流程

初始化距离数组：

使用二维数组 start_mid，大小为 maxn x maxn，来记录从 road 中第一个字符到第二个字符的所有可能位置对的曼哈顿距离。

遍历第一个字符的所有位置和第二个字符的所有位置，计算每对位置的曼哈顿距离并更新 start_mid 数组。

4. 动态规划计算最短路径：

对于 road 字符串中的每个字符，从第二个字符开始到最后一个字符，执行以下步骤：

使用二维数组 mid_next，大小同样为 maxn x maxn，记录当前步骤中字符到下一个字符的所有位置对的曼哈顿距离。

计算并更新每个可能的位置对的距离，使用曼哈顿距离公式计算当前位置对的距离，并加上之前步骤的累计最短距离，更新到临时数组 temp。

将 temp 数组的值复制回 start_mid 以用于下一轮的距离计算。

5. 提取最终结果：

在动态规划完成后，start_mid 数组包含了从 road 中第一个字符到最后一个字符的最短路径距离。

遍历 start_mid 数组，找到其中的最小值，即为从 road 字符串中第一个字符到最后一个字符的最短曼哈顿距离。

6. 输出结果：

输出从 road 中第一个字符到最后一个字符的最短曼哈顿距离。

2. 实验结果与分析

```cpp
using namespace std;

const int MAXN = 105;
const int INF = INT_MAX;

struct Point {
    int y, x;
};

char map[MAXN][MAXN];
int dp[MAXN][MAXN];
string path;
int len;
unordered_map<char, vector<Point>> points;

int manhattanDistance(const Point& a, const Point& b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}

void initializeDpArray(const vector<Point>& startPoints, const vector<Point>& endPoints) {
    for (int i = 0; i < startPoints.size(); i++) {
        for (int j = 0; j < endPoints.size(); j++) {
            dp[i][j] = manhattanDistance(startPoints[i], endPoints[j]);
        }
    }
}
```

```cpp
void updateDpArray(const vector<Point>& midPoints, const vector<Point>& nextPoints) {
    int temp[MAXN][MAXN];
    memset(temp, INF, sizeof(temp));

    for (int i = 0; i < midPoints.size(); i++) {
        for (int j = 0; j < nextPoints.size(); j++) {
            int dist = manhattanDistance(midPoints[i], nextPoints[j]);
            for (int k = 0; k < midPoints.size(); k++) {
                temp[i][j] = min(temp[i][j], dp[k][i] + dist);
            }
        }
    }

    memcpy(dp, temp, sizeof(dp));
}
```

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> map[i][j];
            points[map[i][j]].emplace_back(Point{i, j});
        }
    }

    cin >> path;
    len = path.length();

    // Initialize dp array with distances from first to second character positions
    initializeDpArray(points[path[0]], points[path[1]]);

    // Use dynamic programming to update the shortest paths
    for (int i = 1; i < len - 1; i++) {
        updateDpArray(points[path[i]], points[path[i + 1]]);
    }

    // Find the shortest path from the start character to the end character
    int answer = INF;
    auto& startPoints = points[path[0]];
    auto& endPoints = points[path[len - 1]];
    for (int i = 0; i < startPoints.size(); i++) {
        for (int j = 0; j < endPoints.size(); j++) {
            answer = min(answer, dp[i][j]);
        }
    }

    cout << answer << endl;
    return 0;
}
```

2024-04-26 17:33:55	C++17	Accepted: 100