算法分析与设计大作业

算法编程实现

学院 软件与微电子学院

组号 周四第 2 组

学号 2301210367

姓名 潘江

2024 年 1 月 14 日

目录

1 作业地址

- Gitee: https://gitee.com/jiang000/algorithm_-bighomework.git
- Overleaf: https://www.overleaf.com/read/fsqhdgbnqkwy#861616

2 分治算法

分治算法它将原问题划分成彼此独立，规模较小并且结构相同的子问题，递归的去求解所有的子问题并且将子问题的解组合得到原问题的解。自己实现了**二分检索**，**二分归并排序**，**快速排序**，以及**选定第 K 小**，**课后习题 2.19 从数组选最大最小值**，并对算法的使用条件，主要步骤，分析方法进行了详细的说明。

2.1 二分检索

- **适用条件**：适用于**顺序存储**（数组）的**有序**列表的检索，包括升序和降序排列。
- **主要步骤**
  - 以升序为例，给定数组 $A[]$, 起始索引 i, 结束索引 j, 目标数值 $target$ 。
  - 如果 i > j，那么查找结束，返回 - 1。
  - 否则找到现在数组的中间索引 mid，比较下标 mid 中的数值和 target。
  - 如果 A[mid] == target, 查找成功，返回 mid 下标。
  - 如果 A[mid] > target，那么在前半部分查找。
  - 如果 A[mid] < target，那么在后半部分查找。
- **算法分析**
  - **分治策略**：采用分治思想找到中枢位置 mid，划分成长度为原来一半的小数组中查找，查找小数组的最终结果即为整个问题的结果。
  - **时间复杂度**：最坏情况下时间复杂度为 $O(\log n)$，其中 n 为数组长度。
  - **空间复杂度**：空间复杂度为递归的深度，最坏情况下为 $O(\log n)$，n 为数组长度。

$$T(n) = T(n/2) + O(1) \tag{1}$$

- **C++ 实现**

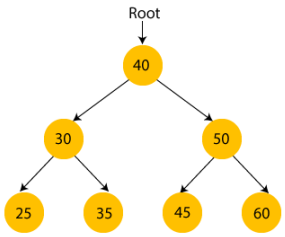

图 1: 二分检索判定树

```cpp
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(vector<int>& nums, int target, int low, int high) {
    if (low > high) {
        return -1;  // 未找到目标值
    }
    int mid = low + (high - low) / 2;
    if (nums[mid] == target) {
        return mid;  // 找到目标值
    } else if (nums[mid] > target) {
        return binarySearch(nums, target, low, mid - 1);// 前半部分
    } else {
        return binarySearch(nums, target, mid + 1, high);// 后半部分
    }
}

int main() {
    vector<int> nums = {25, 30, 35, 40, 45, 50, 60};
    vector<int> targets = {35, 17, 25};
    for(auto target : targets) {
        int result = binarySearch(nums, target, 0, nums.size() - 1);
        cout << target << "  ";
        if (result != -1) {
            cout << "Binary Search: Found target at index " <<
            ↪  result << endl;
        } else {
            cout << "Binary Search: Target not found" << endl;
```

```
        }
    }
}
// 输出结果为:
35  Binary Search: Found target at index 2
17  Binary Search: Target not found
25  Binary Search: Found target at index 0
```

2.2 二路归并排序

- **适用条件**：适用于**线性表**（数组）的**无序**序列的排序，可用于内排序和外排序。
- **主要步骤**
  - 将待排序序列平均分为两部分
  - 递归地对两部分进行归并排序
  - 将两个有序的子序列合并成一个有序序列
- **算法分析**：
  - **分治策略**：采用分治思想，要排序整个的大数组，先划分成较小的数组排序好，再把排序好的小数组归并形成一个较大的有序数组。
  - **稳定性**：稳定的。
  - **时间复杂度**：总体时间复杂度为 $O(n \log n)$，其中 n 为待排序序列长度。
  - **空间复杂度**：需要额外的空间来存储临时数组，空间复杂度为 $O(n)$。
  - **应用**：外排序一般会使用。

$$T(n) = 2T(n/2) + O(n) \tag{2}$$

- **C++ 实现**

```

#include <iostream>
#include <vector>
using namespace std;

void merge(vector<int>& nums, int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;
    vector<int> left(n1);
```

```
    vector<int> right(n2);

    for (int i = 0; i < n1; ++i) {
        left[i] = nums[low + i];
    }
    for (int j = 0; j < n2; ++j) {
        right[j] = nums[mid + 1 + j];
    }

    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            nums[k] = left[i];
            ++i;
        } else {
            nums[k] = right[j];
            ++j;
        }
        ++k;
    }
    while (i < n1) {
        nums[k] = left[i];
        ++i;
        ++k;
    }
    while (j < n2) {
        nums[k] = right[j];
        ++j;
        ++k;
    }
}
void mergeSort(vector<int>& nums, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergeSort(nums, low, mid);
        mergeSort(nums, mid + 1, high);
        merge(nums, low, mid, high);
```

```
    }
}

int main() {
    vector<int> nums = {5, 2, 9, 3, 7, 1, 8, 4, 6};
    mergeSort(nums, 0, nums.size() - 1);
    for(auto num : nums){
        cout << num << "  ";
    }
}
// 输出结果为:
1 2 3 4 5 6 7 8 9
```

2.3 快速排序

- **适用条件**：适用于**线性表**（数组）的**无序**序列的排序
- **主要步骤**
  - 从待排序序列中选择一个基准元素。
  - 将序列中比基准元素小的元素放在基准元素左边，比基准元素大的元素放在基准元素右边。
  - 对基准元素左右两部分分别递归地进行快速排序。
- **算法特点**：
  - **分治策略**：采用分治思想，选取了一个中枢元素，会把一个大的数组划分成两个小的数组，之后再分别对划分之后的小数组进行递归的处理。
  - **不稳定性**：不稳定。
  - **时间复杂度**：平均时间复杂度为 $O(n \log n)$，最坏情况为 $O(n^2)$，其中 n 为待排序序列长度。
  - **空间复杂度**：空间复杂度为 $O(\log n)$，为递归调用的栈空间。
  - **性能好**：快速排序是内部排序算法中时间性能最好的内部排序算法，尤其是当数组长度越来越大的时候，排列的时间性能会显著的优于堆排序和归并排序。
  - **优化**：该算法的性能直接取决于中枢元素的选取，中枢元素选的越中间（1/2 处），那么子数组划分的更均匀，递归的深度越小。所以可以采取三元素取中法，每次随机选取三个元素，把中间一个元素作为中枢元素。

$$T(n) = 2T(n/2) + O(n) \tag{3}$$

- **C++ 实现**

```cpp

#include <iostream>
#include <vector>
using namespace std;

int partition(vector<int>& nums, int low, int high) {
    int pivot = nums[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (nums[j] <= pivot) {
            ++i;
            swap(nums[i], nums[j]);
        }
    }
    swap(nums[i + 1], nums[high]);
    return i + 1;
}

void quickSort(vector<int>& nums, int low, int high) {
    if (low < high) {
        int mid = partition(nums, low, high);
        quickSort(nums, low, mid - 1);
        quickSort(nums, mid + 1, high);
    }
}

int main() {
    vector<int> nums = {5, 2, 9, 3, 7, 1, 8, 4, 6};
    quickSort(nums, 0, nums.size() - 1);
    for(auto num : nums){
        cout << num << "  ";
    }
}

```

```
// 输出结果为：
1 2 3 4 5 6 7 8 9
```

2.4 选择第 k 小元素算法

- **适用条件**：适用于在一个无序序列中查找第 k 小的元素.
- **主要步骤**
  - 从待选择的序列中随机选择一个元素作为基准元素。
  - 将序列中小于基准元素的元素放在基准元素左边，大于基准元素的元素放在右边。
  - 如果基准元素的位置刚好是第 k 小的位置，那么返回基准元素；否则，根据基准元素的位置在左边或右边递归选择。
- **算法特点**：
  - **分治策略**：采用分治思想，将问题拆分为小问题，通过基准元素的位置缩小搜索范围。和快速排序思想类似，但是相比起快速排序，**该问题只需要处理划分子问题的一部分**。
  - **时间复杂度**：平均时间复杂度为 $O(n)$，最坏情况为 $O(n^2)$，其中 n 为待选择序列长度。
  - **空间复杂度**：空间复杂度为 $O(\log n)$，为递归调用的栈空间。
  - **优化**：和快速排序类似，随机选择基准元素，以避免最坏情况的发生。

$$T(n) = T(n/2) + O(n) \tag{4}$$

- **C++ 实现**

```cpp
#include <iostream>
#include <vector>

using namespace std;

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            i++;
```

```
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

int quickSelect(vector<int>& arr, int low, int high, int k) {
    if (k > 0 && k <= high - low + 1) {
        int pi = partition(arr, low, high);
        if (pi - low == k - 1) {
            return arr[pi];
        }
        if (pi - low > k - 1) {
            return quickSelect(arr, low, pi - 1, k);
        }
        return quickSelect(arr, pi + 1, high, k - pi + low - 1);
    }
    return -1;
}

int kthSmallest(vector<int>& arr, int k) {
    int n = arr.size();
    if (k > 0 && k <= n) {
        return quickSelect(arr, 0, n - 1, k);
    }
    return -1;
}

int main() {
    vector<int> arr = {11,222,3333,4,55,666,777,888,-9, 999};
    int k = 4;
    int result = kthSmallest(arr, k);
    if (result != -1) {
        cout << "The " << k << "th smallest element: " << result <<
        ↪  endl;
    } else {
```

```
48          cout << "ERROR" << endl;
49      }
50      return 0;
51  }
52
53  // 输出结果为:
54  The 4th smallest element: 55
```

2.5 课后习题 2.19 选择最大最小值

- **适用条件**：适用于**顺序存储**（数组）的选择最大和最小值。
- **主要步骤**
  - 给定数组 $A[]$，起始索引 i，结束索引 j。
  - 如果 $i = j$，则数组中只有一个元素，直接返回该元素为最大值和最小值。
  - 如果 $i = j - 1$，则数组中有两个元素，比较两个元素的大小，返回较大的为最大值，较小的为最小值。
  - 否则，递归地将数组划分为两半，分别求解左半部分和右半部分的最大值和最小值。
  - 比较左半部分的最大值和最小值，右半部分的最大值和最小值，得到整个数组的最大值和最小值。
- **算法分析**：
  - **分治策略**：采用分治思想，将当前数组通过中枢位置划分成两个规模减半的子问题，分别求解子问题，最后合并得到最后的结果。
  - **时间复杂度**：每次将问题规模减半，时间复杂度为 $O(\log n)$，其中 n 为数组长度。
  - **空间复杂度**：递归深度为 $\log n$，空间复杂度为 $O(\log n)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>

using namespace std;

vector<int> findMinMax(const vector<int>& arr, int start, int end) {
```

```
    vector<int> result(2, 0);
    if (start == end) {
        result[0] = arr[start]; // 数组中只有一个元素
        result[1] = arr[start];
        return result;
    }
    int mid = (start + end) / 2;
    vector<int> leftResult = findMinMax(arr, start, mid);
    vector<int> rightResult = findMinMax(arr, mid + 1, end);
    result[0] = min(leftResult[0], rightResult[0]);
    result[1] = max(leftResult[1], rightResult[1]);
    return result;
}

int main() {
    vector<int> arr = {11,222,3333,4,55,666,777,888,-9, 999};
    // 计算最大值和最小值
    vector<int> result = findMinMax(arr, 0, arr.size() - 1);
    cout << "Maximum value : " << result[1] << endl;
    cout << "Minimum value : " << result[0] << endl;
    return 0;
}

// 输出结果为:
Maximum value : 3333
Minimum value : -9
```

3 动态规划

动态规划是一种用于求解多阶段决策的算法思路，它通过将问题规约成规模相同，结构相同的子问题，建立原问题与子问题之间的依赖关系，即递推表达式。从最小的子问题开始，通过依赖关系求解规模更大的子问题，最终得到原始问题的解。自己实现了**背包问题**，**最长公共子序列**，**最大子段和**，

3.1 背包问题

3.1.1 一维背包问题

- **问题描述**：给定一组物品，每个物品有重量 w_i 和价值 v_i，以及一个最大承重 W 的背包。求在不超过背包承重的前提下，如何选择物品放入背包，使得背包中的物品总价值最大。
- **主要步骤**：
  - 用二维数组 dp 表示状态，其中 $dp[i][j]$ 表示考虑前 i 个物品，背包容量为 j 时能够获得的最大价值。
  - 初始化 dp 数组，$dp[i][j] = 0$ 表示考虑前 i 个物品，背包容量为 j 时的最大价值为 0。
  - 对于第 i 个物品，从 0 到 W 遍历背包容量，更新 $dp[i][j]$：

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i)$$

  - 最终，$dp[n][W]$ 即为所求的最大价值
- **算法分析**：
  - **时间复杂度**：外层循环遍历物品，内层循环遍历背包容量，总时间复杂度为 $O(nW)$
  - **空间复杂度**：使用二维数组 dp 存储状态，空间复杂度为 $O(nW)$。
  - **优化**：可以只使用一维数组来 dp 存储状态，空间复杂度为 $O(n)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
using namespace std;

int compute(int W, vector<int>& weights, vector<int>& values) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j <= W; ++j) {
            if (weights[i - 1] <= j) {
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i
                ↪  - 1]] + values[i - 1]);
```

```
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][W];
}
int main() {
    vector<int> weights = {2, 1, 3, 2};
    vector<int> values = {12, 10, 20, 15};
    int W = 5;
    int result = compute(W, weights, values);
    cout << "Maximum value: " << result << endl;
    return 0;
}


// 输出结果为:
Maximum value: 37
```

3.1.2 二维背包问题

- **问题描述**：给定一组物品，每个物品有两个属性，重量 w_i 和体积 v_i，以及一个最大承重 W 和最大体积 V 的背包。求在不超过背包承重和体积的前提下，如何选择物品放入背包，使得背包中的物品总价值最大。
- **主要步骤**：
  - 用三维数组 $dp[i][j][k]$ 表示状态，其中 $dp[i][j]$ 表示前 i 个物品，背包容量为 j，体积为 k 时能够获得的最大价值。
  - 初始化 dp 数组，$dp[i][j][k] = 0$ 表示前 i 个物品，背包容量为 j，体积为 k 时的最大价值为 0。
  - 对于第 i 个物品，从 W 到 w_i 遍历背包容量，从 V 到 v_i 遍历背包体积，更新 $dp[i][j][k]$。
  - 最终，$dp[N][W][V]$ 即为所求的最大价值。
- **算法分析**：
  - **时间复杂度**：三层循环，外层循环遍历物品，中层循环遍历背包容量，内层循环遍历背包体积，总时间复杂度为 $O(N \cdot W \cdot V)$。

- **空间复杂度**：使用三维数组 dp，空间复杂度为 $O(N \cdot W \cdot V)$。

- **C++ 实现**

```cpp
#include <iostream>
#include <vector>

using namespace std;

int compute(int capacity1, int capacity2, const vector<int>& weights1, const vector<int>& weights2, const vector<int>& values) {
    int n = weights1.size();
    // dp[i][j][k] 表示在容量 1 为 i、容量 2 为 j 时，前 k 个物品的最大价值
    vector<vector<vector<int>>> dp(capacity1 + 1, vector<vector<int>>(capacity2 + 1, vector<int>(n + 1, 0)));
    // 遍历每个物品
    for (int k = 1; k <= n; ++k) {
        for (int w1 = 0; w1 <= capacity1; ++w1) {
            for (int w2 = 0; w2 <= capacity2; ++w2) {
                if (w1 >= weights1[k - 1] && w2 >= weights2[k - 1]) {
                    dp[w1][w2][k] = max(dp[w1][w2][k - 1], dp[w1 - weights1[k - 1]][w2 - weights2[k - 1]][k - 1] + values[k - 1]);
                } else {
                    dp[w1][w2][k] = dp[w1][w2][k - 1];
                }
            }
        }
    }
    return dp[capacity1][capacity2][n];
}

int main() {
    vector<int> weights1 = {2, 3, 4, 5};
    vector<int> weights2 = {1, 2, 3, 4};
```

```
    vector<int> values = {3, 4, 5, 6};
    int capacity1 = 10;
    int capacity2 = 10;
    int result = compute(capacity1, capacity2, weights1, weights2,
    ↪  values);
    cout << "Maximum value: " << result << endl;
    return 0;
}

// 输出结果为:
Maximum value: 13
```

3.1.3 N 维背包

- **问题描述**：给定一组物品，每个物品有 N 个属性，背包有 N 个限制条件，如最大承重、最大体积等。求在不超过背包的限制条件的前提下，如何选择物品放入背包，使得背包中的物品总价值最大。
- **主要步骤**：
  - 用多维数组 $dp[i][j_1][j_2][\ldots][j_N]$ 表示状态，其中 $dp[i][j_1][j_2][\ldots][j_N]$ 表示前 i 个物品，第 N 个属性分别满足 $j_1, j_2, \ldots, j_N$ 时的最大价值。
  - 初始化 dp 数组，$dp[i][j_1][j_2][\ldots][j_N] = 0$ 表示前 i 个物品，第 N 个属性分别满足 $j_1, j_2, \ldots, j_N$ 时的最大价值为 0。
  - 对于第 i 个物品，从第 N 个属性的限制条件到物品的属性值遍历，更新 dp 数组。
  - 最终，$dp[N][j_1][j_2][\ldots][j_N]$ 即为所求的最大价值，其中 N 为物品数量。
- **算法分析**：
  - **时间复杂度**：$N+1$ 层循环，外层循环遍历物品，内层循环遍历每个属性的限制条件，总时间复杂度为 $O(N \cdot L)$，其中 L 为属性的限制条件的层数。
  - **空间复杂度**：使用 $N+1$ 维数组 dp，空间复杂度为 $O(N \cdot L)$。
- **三维背包问题 C++ 实现**

```
#include <iostream>
#include <vector>

using namespace std;
```

```cpp

int compute(int capacity1, int capacity2, int capacity3, const vector<int>& weights1, const vector<int>& weights2, const vector<int>& weights3, const vector<int>& values) {
    int n = weights1.size();
    // dp[i][j][k][l] 表示在容量 1 为 i、容量 2 为 j、容量 3 为 k 时，前 l 个物品的最大价值
    vector<vector<vector<vector<int>>>> dp(capacity1 + 1, vector<vector<vector<int>>>(capacity2 + 1, vector<vector<int>>(capacity3 + 1, vector<int>(n + 1, 0))));
    // 遍历每个物品
    for (int l = 1; l <= n; ++l) {
        for (int w1 = 0; w1 <= capacity1; ++w1) {
            for (int w2 = 0; w2 <= capacity2; ++w2) {
                for (int w3 = 0; w3 <= capacity3; ++w3) {
                    if (w1 >= weights1[l - 1] && w2 >= weights2[l - 1] && w3 >= weights3[l - 1]) {
                        dp[w1][w2][w3][l] = max(dp[w1][w2][w3][l - 1], dp[w1 - weights1[l - 1]][w2 - weights2[l - 1]][w3 - weights3[l - 1]][l - 1] + values[l - 1]);
                    } else {
                        dp[w1][w2][w3][l] = dp[w1][w2][w3][l - 1];
                    }
                }
            }
        }
    }
    return dp[capacity1][capacity2][capacity3][n];
}

int main() {
    // 输入数据
    vector<int> weights1 = {2, 3, 4, 5};
    vector<int> weights2 = {1, 2, 3, 4};
    vector<int> weights3 = {1, 2, 3, 4};
    vector<int> values = {3, 4, 5, 6};
```

```
    int capacity1 = 10;
    int capacity2 = 10;
    int capacity3 = 10;
    int result = compute(capacity1, capacity2, capacity3, weights1,
    ↪  weights2, weights3, values);
    cout << "Maximum value in 3D knapsack: " << result << endl;

    return 0;
}

// 输出结果为:
Maximum value: 13
```

3.2 最长公共子序列（LCS）

- **问题描述**：给定两个序列 X 和 Y，找出它们的最长公共子序列。
- **主要步骤**：
  - 用二维数组 $dp[i][j]$ 表示状态，其中 $dp[i][j]$ 表示序列 X 的前 i 个元素和序列 Y 的前 j 个元素的最长公共子序列的长度。
  - 初始化 dp 数组，$dp[i][j] = 0$ 表示序列 X 的前 i 个元素和序列 Y 的前 j 个元素的最长公共子序列的长度为 0。
  - 逐个考虑 X 和 Y 中的每个元素，更新 dp 数组。
  - 如果 $X[i] = Y[j]$，说明 X 的第 i 个元素和 Y 的第 j 个元素可以构成公共子序列，$dp[i][j] = dp[i-1][j-1] + 1$。
  - 如果 $X[i] \neq Y[j]$，说明 X 的第 i 个元素和 Y 的第 j 个元素不能构成公共子序列，$dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$。
  - 最终，$dp[m][n]$ 即为所求的最长公共子序列的长度，其中 m 和 n 分别为序列 X 和 Y 的长度。
- **算法分析**：
  - **时间复杂度**：$O(m \cdot n)$，其中 m 和 n 分别为序列 X 和 Y 的长度。
  - **空间复杂度**：使用 $m \times n$ 维数组 dp，空间复杂度为 $O(m \cdot n)$。
- **C++ 实现**

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

using namespace std;

int longestCommonSubsequence(const string& str1, const string& str2) {
    int m = str1.length();
    int n = str2.length();
    // dp[i][j] 表示 str1 前 i 个字符和 str2 前 j 个字符的 LCS 长度
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (str1[i - 1] == str2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[m][n];
}

int main() {
    string str1 = "stay hungry";
    string str2 = "stay foolish";
    int result = longestCommonSubsequence(str1, str2);
    cout << "Length of Longest Common Subsequence: " << result << endl;
    return 0;
}


// 输出结果为:
Length of Longest Common Subsequence: 6
```

3.3 最大子段和

- **问题描述**：给定一个整数序列 $A[1, 2, ..., n]$，找出其中连续子序列的最大和。
- **主要步骤**：
  - 用 $dp[i]$ 表示以第 i 个元素结尾的连续子序列的最大和。初始化 dp 数组，$dp[i] = A[i]$。
  - 状态转移方程：$dp[i] = \max(A[i], dp[i-1] + A[i])$
  - 最终答案是 $\max(dp[1], dp[2], ..., dp[n])$，表示以每个元素结尾的连续子序列的最大和中的最大值。
- **算法分析**：
  - **时间复杂度**：$O(n)$，其中 n 为序列的长度。
  - **空间复杂度**：使用一维数组 dp，空间复杂度为 $O(n)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>

using namespace std;

int maxSubarraySum(const vector<int>& nums) {
    int n = nums.size();
    int maxSum = nums[0];
    int currentSum = nums[0];
    for (int i = 1; i < n; ++i) {
        currentSum = max(nums[i], currentSum + nums[i]);
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}

int main() {
    vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    int result = maxSubarraySum(nums);
    cout << "Maximum Subarray Sum: " << result << endl;
    return 0;
```

```
}


// 输出结果为:
Maximum Subarray Sum: 6
```

4 贪心法

贪心法是一种通过每一步的局部最优选择来达到全局最优的算法思路。与动态规划不同，贪心法对子问题的解决方案做出选择后不再进行修改。以下以 Huffman 编码、Prim 算法、Kruskal 算法和 Dijkstra 算法为例，总结贪心法的应用。

4.1 Huffman 编码

- **问题描述**：给定一组权重（频率）为 $w_1, w_2, \ldots, w_n$ 的字符，构建一棵二叉树，使得每个字符的编码长度乘以其权重之和最小。
- **贪心策略**：每次合并权重最小的两个节点，直到构建成一棵树。
- **算法思路**：
  - 构建初始的 n 个节点，每个节点代表一个字符，权重为其频率。
  - 每次选择两个权重最小的节点进行合并，生成一个新节点，其权重为两个节点的权重之和。
  - 重复上述过程，直到所有节点合并成一棵树。
  - 树的结构即为 Huffman 编码树，树的左边为 0，右边为 1，从根到叶子的路径即为字符的编码。
- **算法分析**：
  - **时间复杂度**：$O(n \log n)$，其中 n 为字符的个数。
  - **空间复杂度**：使用二叉树数据结构，空间复杂度为 $O(n)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <queue>
#include <unordered_map>

using namespace std;
```

```cpp
struct HuffmanNode {
    char data;
    unsigned frequency;
    HuffmanNode *left, *right;
    HuffmanNode(char data, unsigned frequency) : data(data), frequency(frequency), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->frequency > b->frequency;
    }
};

void generateCodes(HuffmanNode* root, string code, unordered_map<char, string>& huffmanCodes) {
    if (root) {
        if (!root->left && !root->right) {
            huffmanCodes[root->data] = code;
        }
        generateCodes(root->left, code + "0", huffmanCodes);
        generateCodes(root->right, code + "1", huffmanCodes);
    }
}

void buildHuffmanTree(const string& text) {
    unordered_map<char, unsigned> frequencyMap;
    for (char ch : text) {
        frequencyMap[ch]++;
    }
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> minHeap;

    for (auto& pair : frequencyMap) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }

```

```
    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top();
        minHeap.pop();
        HuffmanNode* right = minHeap.top();
        minHeap.pop();
        HuffmanNode* newNode = new HuffmanNode('$', left->frequency
        ↪  + right->frequency);
        newNode->left = left;
        newNode->right = right;
        minHeap.push(newNode);
    }

    HuffmanNode* root = minHeap.top();
    unordered_map<char, string> huffmanCodes;
    generateCodes(root, "", huffmanCodes);
    cout << "Huffman Codes:\n";
    for (auto& pair : huffmanCodes) {
        cout << pair.first << " : " << pair.second << endl;
    }
}

int main() {
    string text = "stay hungry stay foolish!";
    buildHuffmanTree(text);
    return 0;
}


// 输出结果为:
//Huffman Codes:
//f : 11110
//i : 11011
//a : 1110
//r : 11010
//g : 11001
//n : 11000
//l : 10111
```

```
//! : 10110
//t : 001
//u : 11111
//: 011
//o : 000
//s : 010
//h : 1010
//y : 100

```

4.2 Prim 算法

- **问题描述**：给定一个带权重的无向图，找到一个最小生成树，使得树上所有边的权重之和最小。
- **贪心策略**：每次选择一个未加入生成树的节点，并连接生成树中距离最近的节点。
- **算法思路**：
  - 从任意一个节点开始，将其加入生成树。
  - 每次选择一个未加入生成树的节点，选择一条与生成树连接的最小权重边，将该节点和边加入生成树。
  - 重复上述过程，直到所有节点都加入生成树。
- **算法分析**：
  - **时间复杂度**：$O(V^2)$ 或 $O(E \log V)$，其中 V 为节点数，E 为边数。
  - **空间复杂度**：使用数组存储生成树，空间复杂度为 $O(V + E)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
const int INF = INT_MAX;
int findMinKey(const vector<int>& key, const vector<bool>& mstSet) {
    int minKey = INF, minIndex = -1;
    for (int i = 0; i < key.size(); ++i) {
        if (!mstSet[i] && key[i] < minKey) {
```

```
            minKey = key[i];
            minIndex = i;
        }
    }
    return minIndex;
}

void prim(const vector<vector<int>>& graph, int numNodes) {
    vector<int> parent(numNodes, -1);
    vector<int> key(numNodes, INF);
    vector<bool> mstSet(numNodes, false);
    key[0] = 0; // 选择第一个节点作为起始节点
    for (int count = 0; count < numNodes - 1; ++count) {
        int u = findMinKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < numNodes; ++v) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    // 打印最小生成树的边
    cout << "Edges of the Minimum Spanning Tree:\n";
    for (int i = 1; i < numNodes; ++i) {
        cout << "Edge: " << parent[i] << " - " << i << " Weight: "
        ↪  << graph[i][parent[i]] << endl;
    }
}

int main() {
    vector<vector<int>> graph = {
            {0, 2, INF, 6, INF},
            {2, 0, 3, 8, 5},
            {INF, 3, 0, INF, 7},
            {6, 8, INF, 0, 9},
            {INF, 5, 7, 9, 0}
```

```
    };
    int numNodes = graph.size();
    prim(graph, numNodes);
    return 0;
}

// 输出结果为:
//Edges of the Minimum Spanning Tree:
//Edge: 0 - 1 Weight: 2
//Edge: 1 - 2 Weight: 3
//Edge: 0 - 3 Weight: 6
//Edge: 1 - 4 Weight: 5
```

4.3 Kruskal 算法

- **问题描述**：给定一个带权重的无向图，找到一个最小生成树，使得树上所有边的权重之和最小。
- **贪心策略**：每次选择一条权重最小的边，并且加入生成树，直到生成树中包含所有节点。
- **算法思路**：
  - 将所有边按照权重从小到大排序。
  - 依次遍历排序后的边，将权重最小且不形成环的边加入生成树。
  - 重复上述过程，直到生成树中包含所有节点。
- **算法分析**：
  - **时间复杂度**：$O(E \log E)$，其中 E 为边数。
  - **空间复杂度**：使用数组存储生成树，空间复杂度为 $O(V + E)$。
- **C++ 实现**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Edge {
    int src, dest, weight;
```

```
7       Edge(int src, int dest, int weight) : src(src), dest(dest),
        ↪ weight(weight) {}
8       bool operator<(const Edge& other) const {
9           return weight < other.weight;
10      }
11  };
12  class DisjointSet {
13  public:
14      DisjointSet(int size) : parent(size), rank(size, 0) {
15          for (int i = 0; i < size; ++i) {
16              parent[i] = i;
17          }
18      }
19      int find(int x) {
20          if (parent[x] != x) {
21              parent[x] = find(parent[x]); // 路径压缩
22          }
23          return parent[x];
24      }
25      void Union(int x, int y) {
26          int rootX = find(x);
27          int rootY = find(y);
28          if (rootX != rootY) {
29              if (rank[rootX] < rank[rootY]) {
30                  parent[rootX] = rootY;
31              } else if (rank[rootX] > rank[rootY]) {
32                  parent[rootY] = rootX;
33              } else {
34                  parent[rootX] = rootY;
35                  rank[rootY]++;
36              }
37          }
38      }
39  private:
40      vector<int> parent;
41      vector<int> rank;
42  };
```

```cpp

void kruskal(const vector<Edge>& edges, int numNodes) {
    vector<Edge> sortedEdges = edges;
    sort(sortedEdges.begin(), sortedEdges.end());
    DisjointSet ds(numNodes);
    cout << "Edges of the Minimum Spanning Tree:\n";
    for (const Edge& edge : sortedEdges) {
        int rootSrc = ds.find(edge.src);
        int rootDest = ds.find(edge.dest);
        if (rootSrc != rootDest) {
            cout << "Edge: " << edge.src << " - " << edge.dest << "
            ↪ Weight: " << edge.weight << endl;
            ds.Union(rootSrc, rootDest);
        }
    }
}

int main() {
    vector<Edge> edges = {
            {0, 1, 1},
            {0, 3, 2},
            {1, 2, 3},
            {1, 4, 4},
            {2, 4, 4},
            {2, 5, 4},
            {3, 5, 5},
            {4, 5, 9}
    };
    int numNodes = 6; // 节点数
    kruskal(edges, numNodes);
    return 0;
}

// 输出结果为:
Edges of the Minimum Spanning Tree:
Edge: 0 - 1 Weight: 1
Edge: 0 - 3 Weight: 2
```

```
Edge: 1 - 2 Weight: 3
Edge: 1 - 4 Weight: 4
Edge: 2 - 5 Weight: 4
```

4.4 Dijkstra 算法

- **问题描述**：给定一个带权重的有向图，以及一个起始节点，找到起始节点到其他所有节点的最短路径。
- **贪心策略**：每次选择距离起始节点最近的节点，并更新起始节点到其他节点的距离。
- **算法思路**：
  - 初始化起始节点到所有节点的距离，起始节点的距离为 0，其他节点的距离为正无穷大。
  - 每次选择距离起始节点最近的节点，更新起始节点到其邻居节点的距离。
  - 重复上述过程，直到所有节点都被访问。
- **算法分析**：
  - **时间复杂度**：$O(V^2)$ 或 $O((V+E)\log V)$，其中 V 为节点数，E 为边数。
  - **空间复杂度**：使用数组存储距离，空间复杂度为 $O(V+E)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
using namespace std;
const int INF = INT_MAX; // 代表无穷大
void dijkstra(const vector<vector<int>>& graph, int startNode, int
    ↪ numNodes) {
    vector<int> distance(numNodes, INF);
    vector<bool> visited(numNodes, false);
    distance[startNode] = 0;
    for (int i = 0; i < numNodes - 1; ++i) {
        int minDistance = INF;
        int minIndex = -1;
        for (int j = 0; j < numNodes; ++j) {
            if (!visited[j] && distance[j] < minDistance) {
```

```
                minDistance = distance[j];
                minIndex = j;
            }
        }
        visited[minIndex] = true;
        // 更新通过当前节点的路径，更新到达其它节点的最短距离
        for (int j = 0; j < numNodes; ++j) {
            if (!visited[j] && graph[minIndex][j] != INF &&
            ↪ distance[minIndex] != INF &&
                distance[minIndex] + graph[minIndex][j] <
                ↪ distance[j]) {
                distance[j] = distance[minIndex] +
                ↪ graph[minIndex][j];
            }
        }
    }
    cout << "Shortest distances from node " << startNode << ":\n";
    for (int i = 0; i < numNodes; ++i) {
        cout << "To node " << i << ": " << distance[i] << endl;
    }
}

int main() {
    // 代表无连接的节点使用 INF
    vector<vector<int>> graph = {
            {0, 2, INF, 1, INF},
            {2, 0, 4, INF, 5},
            {INF, 4, 0, 3, INF},
            {1, INF, 3, 0, 6},
            {INF, 5, INF, 6, 0}
    };

    int startNode = 2; // 起始节点
    int numNodes = graph.size();
    dijkstra(graph, startNode, numNodes);
    return 0;
}
```

```
48
49 // 输出结果为:
50 //Shortest distances from node 2:
51 //To node 0: 4
52 //To node 1: 4
53 //To node 2: 0
54 //To node 3: 3
55 //To node 4: 9
```

5 回溯法

回溯法是一种通过试探解空间中的各种可能性来找到问题解的算法思路。在回溯过程中，如果发现当前选择不能达到期望的目标，就进行回退，重新选择其他路径。以下以 N 皇后问题、0-1 背包问题、装载问题、图的 m 着色问题和连续邮资问题为例，详细介绍回溯算法的应用。

5.1 N 皇后问题

- **问题描述**：在 N×N 的棋盘上放置 N 个皇后，使得它们互不攻击，即任意两个皇后都不在同一行、同一列或同一斜线上。
- **回溯策略**：逐行放置皇后，每行只放一个皇后，通过递归实现。
- **算法思路**：
  - 从第一行开始，逐行放置皇后，每行只放一个皇后。
  - 对于每一行，在该行的每一列尝试放置皇后，检查是否符合规则。
  - 如果符合规则，递归到下一行，重复上述过程。
  - 如果某行无法找到合适位置，回溯到上一行，重新选择位置。
  - 直到所有行都放置了皇后，得到一个解。
- **算法分析**：
  - **时间复杂度**：$O(N!)$，其中 N 为棋盘大小。
  - **空间复杂度**：使用递归栈，空间复杂度为 $O(N)$。
  - **应用**：在棋盘上放置不互相攻击的皇后。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
using namespace std;
bool isSafe(const vector<int>& board, int row, int col) {
    for (int i = 0; i < row; ++i) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
        ↪ {
            return false;
        }
    }
    return true;
}
void printBoard(const vector<int>& board) {
    int N = board.size();
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (board[i] == j) {
                cout << "Q ";
            } else {
                cout << ". ";
            }
        }
        cout << endl;
    }
    cout << endl;
}
void solveNQueensUtil(vector<int>& board, int row) {
    int N = board.size();
    if (row == N) {
        // 放置好了所有的皇后，打印整个棋盘。
        printBoard(board);
        return;
    }

    for (int col = 0; col < N; ++col) {
        if (isSafe(board, row, col)) {
            // 放置皇后
```

```
            board[row] = col;
            solveNQueensUtil(board, row + 1);
            board[row] = -1;
        }
    }
}

void solveNQueens(int N) {
    vector<int> board(N, -1);
    solveNQueensUtil(board, 0);
}

int main() {
    int N;
    cout << "Enter one value: ";
    cin >> N ;
    solveNQueens(N);
    return 0;
}

// 输出结果为:
Enter one value:4
 . Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

5.2 0-1 背包问题

- **问题描述**：给定一组物品，每个物品有重量 w_i 和价值 v_i，以及一个最大承重 W 的背包。求在不超过背包承重的前提下，如何选择物品放入背包，使得背包中的物品总价值最大。

- **回溯策略**：逐个考虑每个物品，每个物品可以选择放入或不放入背包，通过递归实现。
- **算法思路**：
  - 用一个索引 i 表示当前考虑的物品。
  - 对于每个物品，可以选择放入或不放入背包，分别递归考虑下一个物品。
  - 递归的终止条件为考虑完所有物品或背包已满。
  - 在递归过程中，记录当前选择的物品，得到一个解。
- **算法分析**：
  - **时间复杂度**：指数级别，$O(2^N)$，其中 N 为物品个数。
  - **空间复杂度**：使用递归栈，空间复杂度为 $O(N)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
using namespace std;
int maxVal = 0;

void backtrack(int W, vector<int>& weights, vector<int>& values, int
    index, int currentWeight, int currentValue) {
    if (index == weights.size() || currentWeight > W) {
        maxVal = max(maxVal, currentValue);
        return;
    }
    backtrack(W, weights, values, index + 1, currentWeight,
        currentValue);
    if (currentWeight + weights[index] <= W) {
        backtrack(W, weights, values, index + 1, currentWeight +
            weights[index], currentValue + values[index]);
    }
}

int compute(int W, vector<int>& weights, vector<int>& values) {
    maxVal = 0;
    backtrack(W, weights, values, 0, 0, 0);
```

```
    return maxVal;
}

int main() {
    vector<int> weights = {2, 1, 3, 2};
    vector<int> values = {12, 10, 20, 15};
    int W = 5;
    int result = compute(W, weights, values);
    cout << "Maximum value: " << result << endl;

    return 0;
}


// 输出结果为:
Maximum value: 37
```

5.3 装载问题

- **问题描述**：有一批集装箱需要装入若干艘轮船，每个集装箱有重量 w_i，每艘轮船的载重有限。要求找到一种装载方式，使得所有集装箱都被装上轮船。
- **回溯策略**：逐个考虑每个集装箱，每个集装箱可以选择装入或不装入轮船，通过递归实现。
- **算法思路**：
  - 用一个索引 i 表示当前考虑的集装箱。
  - 对于每个集装箱，可以选择装入或不装入轮船，分别递归考虑下一个集装箱。
  - 递归的终止条件为考虑完所有集装箱。
  - 在递归过程中，记录当前选择的集装箱，得到一个解。
- **算法分析**：
  - **时间复杂度**：指数级别，$O(2^N)$，其中 N 为集装箱个数。
  - **空间复杂度**：使用递归栈，空间复杂度为 $O(N)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
using namespace std;
void Backtracking(vector<int>& items, int capacity, vector<int>&
    currentBin) {
    static int binCount = 0;
    for (int i = 0; i < items.size(); ++i) {
        if (items[i] <= capacity) {
            currentBin.push_back(items[i]);
            int removedItem = items[i];
            items.erase(items.begin() + i);
            Backtracking(items, capacity - removedItem, currentBin);
            currentBin.pop_back();
            items.insert(items.begin() + i, removedItem);
        }
    }

    if (currentBin.empty()) {
        cout << ++binCount << ": ";
        for (int item : items) {
            cout << item << " ";
        }
        cout << endl;
    }
}

int main() {
    vector<int> items = {4, 8, 1, 4, 2, 5, 3};
    int binCapacity = 10;
    vector<int> currentBin;
    Backtracking(items, binCapacity, currentBin);
    return 0;
}

// 输出结果为:
4 8 1 4 2 5 3
```

5.4 图的 m 着色问题

- **问题描述**：给定一个图，找到一种着色方式，使得相邻的节点颜色不同，并且使用的颜色数最小。
- **回溯策略**：逐个考虑每个节点，每个节点可以选择一种颜色，通过递归实现。
- **算法思路**：
  - 用一个索引 i 表示当前考虑的节点。
  - 对于每个节点，可以选择一种颜色，分别递归考虑下一个节点。
  - 递归的终止条件为考虑完所有节点。
  - 在递归过程中，记录当前选择的颜色，得到一个解。
- **算法分析**：
  - **时间复杂度**：指数级别，$O(m^N)$，其中 m 为颜色数，N 为节点数。
  - **空间复杂度**：使用递归栈，空间复杂度为 $O(N)$。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class Graph {
public:
    Graph(int vertices);
    void addEdge(int u, int v);
    void graphColoring(int m);

private:
    int vertices;
    vector<vector<int>> adjacencyMatrix;

    bool isSafe(int v, vector<int>& color, int c);
};

Graph::Graph(int vertices) : vertices(vertices) {
    adjacencyMatrix.resize(vertices, vector<int>(vertices, 0));
```

```
}

void Graph::addEdge(int u, int v) {
    // 添加边
    adjacencyMatrix[u][v] = 1;
    adjacencyMatrix[v][u] = 1;
}

bool Graph::isSafe(int v, vector<int>& color, int c) {
    for (int i = 0; i < vertices; ++i) {
        if (adjacencyMatrix[v][i] && c == color[i]) {
            return false;
        }
    }
    return true;
}

void Graph::graphColoring(int m) {
    vector<int> color(vertices, 0);
    color[0] = 1;
    // 尝试着色其余节点
    for (int v = 1; v < vertices; ++v) {
        for (int c = 1; c <= m; ++c) {
            if (isSafe(v, color, c)) {
                color[v] = c;
                break;
            }
        }
    }

    // 输出结果
    cout << "Minimum number of colors required: " <<
    ↪  *max_element(color.begin(), color.end()) << endl;
    cout << "Node colors: ";
    for (int i = 0; i < vertices; ++i) {
        cout << color[i] << " ";
    }
```

```
56      cout << endl;
57  }
58
59  int main() {
60      Graph graph(6);
61      graph.addEdge(0, 1);
62      graph.addEdge(0, 2);
63      graph.addEdge(1, 2);
64      graph.addEdge(1, 3);
65      graph.addEdge(0,5);
66      graph.addEdge(1,5);
67      graph.addEdge(2,4);
68      int m = 4; // 颜色的数量
69      graph.graphColoring(m);
70      return 0;
71  }
72
73
74  // 输出结果为:
75  //Minimum number of colors required: 3
76  //Node colors: 1 2 3 1 1 3
```

5.5 连续邮资问题

- **问题描述**：给定邮资面值和信封数量，要求找到一种连续的邮资面值，使得能够覆盖所有信封的邮资。
- **回溯策略**：逐个考虑每个邮资面值，每个面值可以选择使用或不使用，通过递归实现。
- **算法思路**：
  - 用一个索引 i 表示当前考虑的邮资面值。
  - 对于每个邮资面值，可以选择使用或不使用，分别递归考虑下一个邮资面值。
  - 递归的终止条件为考虑完所有邮资面值。
  - 在递归过程中，记录当前选择的邮资面值，得到一个解。
- **算法分析**：
  - **时间复杂度**：指数级别，$O(2^N)$，其中 N 为邮资面值的个数。

- **空间复杂度**：使用递归栈，空间复杂度为 $O(N)$。

- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
using namespace std;
int maxSumIncreasingSubsequence(const vector<int>& values) {
    int n = values.size();
    vector<int> dp(n, 0);
    int result = values[0];
    for (int i = 0; i < n; ++i) {
        dp[i] = values[i];
        for (int j = 0; j < i; ++j) {
            if (values[i] > values[j]) {
                dp[i] = max(dp[i], dp[j] + values[i]);
            }
        }
        result = max(result, dp[i]);
    }
    return result;
}

int main() {
    vector<int> values = {6, 7, 8, 1, 2, 3, 9};
    int result = maxSumIncreasingSubsequence(values);
    cout << "Maximum Sum Increasing Subsequence: " << result <<
    ↪  endl;
    return 0;
}

// 输出结果为:
Maximum Sum Increasing Subsequence: 30
```

6 网络流算法

网络流算法主要用于解决图中的网络流问题，其中包括最大流问题、最小割问题等。以下分别介绍 Ford-Fulkerson 算法和 Dinic 算法，并以 Floyd 算法和 Hitchcock 问题的

匈牙利算法为例详细说明。

6.1 Ford-Fulkerson 算法

- **算法思路**：通过不断增广路径（即在残余图上找一条增广路径），增加流量，直到找不到增广路径为止。
- **算法步骤**：
  1. 初始化流网络，设定初始流量为 0。
  2. 在残余图中找到一条增广路径，更新流量。
  3. 重复步骤 2，直到找不到增广路径。
- **复杂度**：最坏情况下可能需要 $O(VE^2)$ 次迭代，其中 V 为顶点数，E 为边数。
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>
using namespace std;
class FordFulkerson {
public:
    FordFulkerson(int vertices) : V(vertices),
    ↪ residualGraph(vertices, vector<int>(vertices, 0)),
    ↪ parent(vertices, -1) {}
    void addEdge(int u, int v, int capacity) {
        residualGraph[u][v] = capacity;
    }
    int maxFlow(int source, int sink) {
        int maxFlow = 0;
        while (augmentPath(source, sink)) {
            int pathFlow = INT_MAX;
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                pathFlow = min(pathFlow, residualGraph[u][v]);
            }
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
```

```
                residualGraph[u][v] -= pathFlow;
                residualGraph[v][u] += pathFlow;
            }
            maxFlow += pathFlow;
        }
        return maxFlow;
    }

private:
    int V; // Number of vertices
    vector<vector<int>> residualGraph;
    vector<int> parent;
    bool augmentPath(int source, int sink) {
        fill(parent.begin(), parent.end(), -1);
        vector<bool> visited(V, false);
        visited[source] = true;
        parent[source] = source;
        vector<int> stack;
        stack.push_back(source);
        while (!stack.empty()) {
            int u = stack.back();
            stack.pop_back();
            for (int v = 0; v < V; ++v) {
                if (!visited[v] && residualGraph[u][v] > 0) {
                    parent[v] = u;
                    visited[v] = true;
                    stack.push_back(v);
                    if (v == sink) {
                        return true;
                    }
                }
            }
        }

        return false;
    }
};
```

```
59
60 int main() {
61     int vertices = 6;
62     FordFulkerson fordFulkerson(vertices);
63     fordFulkerson.addEdge(0, 1, 16);
64     fordFulkerson.addEdge(0, 2, 13);
65     fordFulkerson.addEdge(1, 2, 10);
66     fordFulkerson.addEdge(1, 3, 12);
67     fordFulkerson.addEdge(2, 1, 4);
68     fordFulkerson.addEdge(2, 4, 14);
69     fordFulkerson.addEdge(3, 2, 9);
70     fordFulkerson.addEdge(3, 5, 20);
71     fordFulkerson.addEdge(4, 3, 7);
72     fordFulkerson.addEdge(4, 5, 4);
73     int source = 0;
74     int sink = 5;
75     int maxFlow = fordFulkerson.maxFlow(source, sink);
76     cout << "Maximum Flow: " << maxFlow << endl;
77     return 0;
78 }
79
80 // 输出结果为:
81 Maximum Flow: 23
```

6.2 Dinic 算法

- **算法思路**：通过层次网络构建，每次在分层图上找一条阻塞流，增加流量。
- **算法步骤**：
  1. 构建分层图，使用 BFS。
  2. 在分层图中找到一条阻塞流，更新流量。
  3. 重复步骤 2，直到找不到阻塞流。
- **复杂度**：最坏情况下时间复杂度为 $O(V^2E)$，其中 V 为顶点数，E 为边数。
- **C++ 实现**

```
1 #include <iostream>
2 #include <vector>
```

```cpp
#include <queue>
#include <climits>
#include <algorithm>
using namespace std;

class Dinic {
public:
    Dinic(int vertices) : V(vertices), level(vertices, -1), adjMatrix(vertices, vector<int>(vertices, 0)) {}
    void addEdge(int u, int v, int capacity) {
        adjMatrix[u][v] = capacity;
    }

    int maxFlow(int source, int sink) {
        int totalFlow = 0;
        while (bfs(source, sink)) {
            vector<int> start(V, 0);
            int pathFlow;
            while ((pathFlow = dfs(source, sink, INT_MAX, start)) != 0) {
                totalFlow += pathFlow;
            }
        }
        return totalFlow;
    }

private:
    int V;
    vector<int> level;
    vector<vector<int>> adjMatrix;
    bool bfs(int source, int sink) {
        fill(level.begin(), level.end(), -1);
        level[source] = 0;
        queue<int> q;
        q.push(source);
        while (!q.empty()) {
            int u = q.front();
```

```
38              q.pop();
39              for (int v = 0; v < V; ++v) {
40                  if (level[v] < 0 && adjMatrix[u][v] > 0) {
41                      level[v] = level[u] + 1;
42                      q.push(v);
43                  }
44              }
45          }
46          return level[sink] >= 0;
47      }
48
49      // Depth-first search for augmenting paths
50      int dfs(int u, int sink, int flow, vector<int>& start) {
51          if (u == sink) {
52              return flow;
53          }
54          for (int v = start[u]; v < V; ++v) {
55              if (level[v] == level[u] + 1 && adjMatrix[u][v] > 0) {
56                  int pathFlow = dfs(v, sink, min(flow,
                    ↪  adjMatrix[u][v]), start);
57                  if (pathFlow > 0) {
58                      adjMatrix[u][v] -= pathFlow;
59                      adjMatrix[v][u] += pathFlow;
60                      return pathFlow;
61                  }
62              }
63              ++start[u];
64          }
65          return 0;
66      }
67  };
68
69  int main() {
70      int vertices = 6;
71      Dinic dinic(vertices);
72      dinic.addEdge(0, 1, 16);
73      dinic.addEdge(0, 2, 13);
```

```
    dinic.addEdge(1, 2, 10);
    dinic.addEdge(1, 3, 12);
    dinic.addEdge(2, 1, 4);
    dinic.addEdge(2, 4, 14);
    dinic.addEdge(3, 2, 9);
    dinic.addEdge(3, 5, 20);
    dinic.addEdge(4, 3, 7);
    dinic.addEdge(4, 5, 4);
    int source = 0;
    int sink = 5;
    int maxFlow = dinic.maxFlow(source, sink);
    cout << "Maximum Flow: " << maxFlow << endl;
    return 0;
}

// 输出结果为:
Maximum Flow: 23
```

6.3 Floyd 算法

- **问题描述**：给定一个带权重的有向图，寻找图中任意两个节点之间的最短路径。。
- **适用条件**：解决有向图中任意两个节点之间的最短路径，可以存在负权值的边，但是不能出现负权环。
- **算法步骤**：
  1. 初始化距离矩阵，直接相连的节点有边则为权值，无边则为无穷大。
  2. 逐步更新距离矩阵，对于每一对节点，考虑经过中间节点的路径是否更短。
  3. 重复步骤 2，直到距离矩阵不再改变。
- **时间复杂度**：$O(V^3)$，其中 V 为顶点数。
- **空间复杂度**：$O(1)$。
- **C++ 实现**

```
#include <iostream>
#include <vector>
using namespace std;
const int INF = INT_MAX;
```

```cpp
void floyd(vector<vector<int>>& graph, int numNodes) {
    vector<vector<int>> distance = graph;
    for (int k = 0; k < numNodes; ++k) {
        for (int i = 0; i < numNodes; ++i) {
            for (int j = 0; j < numNodes; ++j) {
                if (distance[i][k] != INF && distance[k][j] != INF
                ↪ &&
                    distance[i][k] + distance[k][j] <
                    ↪ distance[i][j]) {
                    distance[i][j] = distance[i][k] +
                    ↪ distance[k][j];
                }
            }
        }
    }
    cout << "Shortest distances between all pairs of nodes:\n";
    for (int i = 0; i < numNodes; ++i) {
        for (int j = 0; j < numNodes; ++j) {
            if (distance[i][j] == INF) {
                cout << "INF ";
            } else {
                cout << distance[i][j] << " ";
            }
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> graph = {
        {0, 2, INF, 1, INF},
        {2, 0, 4, INF, 5},
        {INF, 4, 0, 3, INF},
        {1, INF, 3, 0, 6},
        {INF, 5, INF, 6, 0}
    };
    int numNodes = graph.size();
```

```
    floyd(graph, numNodes);
    return 0;
}

// 输出结果为:
Shortest distances between all pairs of nodes:
0 2 4 1 7
2 0 4 3 5
4 4 0 3 9
1 3 3 0 6
7 5 9 6 0
```

7 线性规划

线性规划（Linear Programming，简称 LP）是一种数学优化方法，用于在给定约束条件下，最大化或最小化一个线性目标函数的数学模型。自己实现了简单的单纯形法，用于求解线性规划问题。

7.1 单纯形法

- **问题描述**：一共 n 个变量，目标是在一组约束条件下最小化一个线性目标函数。所有变量的数值都是大于等于 0，并且所有的约束都是小于等于表达式。
- **线性规划**：

$$\begin{aligned} \text{Minimize} \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\ \text{Subject to} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq b_m \\ & x_1, x_2, \ldots, x_n \geq 0 \end{aligned}$$

- **算法输入**：
  - $c_1, c_2, c_3, \ldots, c_n$ 是目标函数的系数
  - a_{ij} 是约束矩阵里的系数
  - $b_1, b_2, b_3, \ldots, b_n$ 是约束条件的右侧系数
  - $x_1, x_2, x_3, \ldots, x_n$ 是基本变量

- **算法分析**：
  - **时间复杂度**：最坏情况下为 $O(2^n)$
- **C++ 实现**

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

// 定义一个结构体表示单纯形表
struct SimplexTable {
    std::vector<std::vector<double>> table;
    std::vector<int> basis; // 存储基变量的索引
};

// 打印单纯形表
void printSimplexTable(const SimplexTable& simplexTable) {
    for (const auto& row : simplexTable.table) {
        for (double value : row) {
            std::cout << value << "\t";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

// 在单纯形表上应用主元素法
void pivot(SimplexTable& simplexTable, int row, int col) {
    int numRows = simplexTable.table.size();
    int numCols = simplexTable.table[0].size();
    // 将主元素所在列变为单位向量
    for (int i = 0; i < numRows; ++i) {
        if (i != row) {
            double ratio = simplexTable.table[i][col] /
            ↪ simplexTable.table[row][col];
            for (int j = 0; j < numCols; ++j) {
                simplexTable.table[i][j] -= ratio *
                ↪ simplexTable.table[row][j];
```

```
            }
        }
    }
    // 将主元素所在行变为单位向量
    double pivotElement = simplexTable.table[row][col];
    for (int j = 0; j < numCols; ++j) {
        simplexTable.table[row][j] /= pivotElement;
    }
    // 更新基变量
    simplexTable.basis[row] = col;
}

// 执行单纯形法
bool simplex(std::vector<std::vector<double>>& coefficients,
    ↪ std::vector<double>& objectiveFunction, std::vector<double>&
    ↪ constraints) {
    int numRows = coefficients.size();
    int numCols = coefficients[0].size();
    SimplexTable simplexTable;
    simplexTable.table.resize(numRows + 1,
        ↪ std::vector<double>(numCols + numRows + 1, 0));
    simplexTable.basis.resize(numRows);
    // 初始化单纯形表
    for (int i = 0; i < numRows; ++i) {
        for (int j = 0; j < numCols; ++j) {
            simplexTable.table[i][j] = coefficients[i][j];
        }
        simplexTable.basis[i] = numCols + i;
        simplexTable.table[i][numCols + i] = 1;
        simplexTable.table[i][numCols + numRows] = constraints[i];
    }
    for (int j = 0; j < numCols; ++j) {
        simplexTable.table[numRows][j] = objectiveFunction[j];
    }
    printSimplexTable(simplexTable);
    // 迭代直到目标函数系数非负
    while (true) {
```

```
66          int enteringCol = -1;
67          for (int j = 0; j < numCols; ++j) {
68              if (simplexTable.table[numRows][j] < 0) {
69                  enteringCol = j;
70                  break;
71              }
72          }
73          if (enteringCol == -1) {
74              // 所有目标函数系数非负，达到最优解
75              break;
76          }
77          int leavingRow = -1;
78          double minRatio = std::numeric_limits<double>::infinity();
79
80          for (int i = 0; i < numRows; ++i) {
81              if (simplexTable.table[i][enteringCol] > 0) {
82                  double ratio = simplexTable.table[i][numCols +
                    ↪  numRows] / simplexTable.table[i][enteringCol];
83                  if (ratio < minRatio) {
84                      minRatio = ratio;
85                      leavingRow = i;
86                  }
87              }
88          }
89          if (leavingRow == -1) {
90              return false;
91          }
92
93          pivot(simplexTable, leavingRow, enteringCol);
94
95          std::cout << "CHOOSE AND CHANGE:\n";
96          printSimplexTable(simplexTable);
97      }
98
99      // 输出最优解和最优值
100     std::cout << "Optimal Solution:\n";
101     for (int i = 0; i < numCols; ++i) {
```

```
            int basisIndex = -1;
            for (int j = 0; j < numRows; ++j) {
                if (simplexTable.basis[j] == i) {
                    basisIndex = j;
                    break;
                }
            }
            if (basisIndex != -1) {
                std::cout << "x" << i + 1 << " = " <<
                ↪  simplexTable.table[basisIndex][numCols + numRows] <<
                ↪  std::endl;
            } else {
                std::cout << "x" << i + 1 << " = 0" << std::endl;
            }
        }
        return true;
    }

    int main() {
        std::vector<std::vector<double>> coefficients = {{2, 1, 0, 1, 0,
        ↪  0}, {-4, -2, 3, 0, 1, 0},{1, -2, 1, 0, 0, 1}};
        std::vector<double> objectiveFunction = {-2, 1, -1, 0, 0, 0};
        std::vector<double> constraints = {10, 10, 14};

        if (simplex(coefficients, objectiveFunction, constraints)) {
            std::cout << "Optimization successful.\n";
        } else {
            std::cout << "Optimization failed (unbounded problem).\n";
        }

        return 0;
    }

    这里是课后 6.10题目的第一小题为例子进行求解
    min -2x1 + x2 - x3
    s.t.   2x1 + x2 <= 10
           -4x1 -2x2 + 3x3 <= 10
```

```
        x1 - 2x2 + x3 <= 14
        x1 x2 x3 >= 0

结果为
 2     1     0     1     0     0     1     0     0    10
-4    -2     3     0     1     0     0     1     0    10
 1    -2     1     0     0     1     0     0     1    14
-2     1    -1     0     0     0     0     0     0     0

CHOOSE AND CHANGE:
 1   0.5     0   0.5     0     0   0.5     0     0     5
 0     0     3     2     1     0     2     1     0    30
 0  -2.5     1  -0.5     0     1  -0.5     0     1     9
 0     2    -1     1     0     0     1     0     0    10

CHOOSE AND CHANGE:
 1   0.5     0   0.5     0     0   0.5     0     0     5
 0   7.5     0   3.5     1    -3   3.5     1    -3     3
 0  -2.5     1  -0.5     0     1  -0.5     0     1     9
 0  -0.5     0   0.5     0     1   0.5     0     1    19

CHOOSE AND CHANGE:
 1     0     00.266667-0.0666667   0.20.266667-0.0666667   0.2   4.8
 0     1     00.4666670.133333  -0.40.4666670.133333  -0.4   0.4
 0     0     10.6666670.333333     00.6666670.333333     0    10
 0     0     00.7333330.0666667   0.80.7333330.0666667   0.8  19.2

Optimal Solution:
x1 = 4.8
x2 = 0.4
x3 = 10
x4 = 0
x5 = 0
x6 = 0
```

8 整数规划

整数规划，在线性规划的基础之上，要求每一个数字都是整数的规划问题。

8.1 分支限界算法

- **问题描述**：一共 n 个变量，目标是在一组约束条件下最小化一个线性目标函数。所有变量的数值都是大于等于 0，并且所有的约束都是小于等于表达式，并且每一个都是整数。
- **整数规划**：

$$\begin{aligned} \text{Minimize} \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\ \text{Subject to} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq b_m \\ & x_1, x_2, \ldots, x_n \in \mathbb{N} \end{aligned}$$

- **算法输入**：
  - $c_1, c_2, c_3, \ldots, c_n$ 是目标函数的系数
  - a_{ij} 是约束矩阵里的系数
  - $b_1, b_2, b_3, \ldots, b_n$ 是约束条件的右侧系数
  - $x_1, x_2, x_3, \ldots, x_n$ 是基本变量
- **算法分析**：
  - **时间复杂度**：最坏情况下为 $O(2^n)$
- **C++ 实现**

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <limits>
using namespace std;
// 表示线性规划问题的结构体
struct LinearProgram {
```

```
    vector<vector<int>> coefficients;  // 系数矩阵
    vector<double> constants;          // 右侧常数
    vector<int> objectiveFunction;     // 目标函数系数
};
// 表示节点的结构体
struct Node {
    vector<int> solution;  // 当前节点的解
    int value;             // 目标函数值
    int level;             // 节点的层级
};

// 用于比较节点的优先级队列
struct CompareNode {
    bool operator()(const Node& n1, const Node& n2) {
        return n1.value < n2.value;  // 按目标函数值升序排列
    }
};

// 检查当前解是否满足整数要求
bool isIntegerSolution(const vector<int>& solution) {
    for (int value : solution) {  // 使用 int 类型
        if (fabs(value - round(value)) >
        ↪  numeric_limits<double>::epsilon()) {
            return false;  // 不是整数解
        }
    }
    return true;  // 是整数解
}

// 分支限界算法
int branchAndBound(const LinearProgram& lp) {
    int n = lp.coefficients[0].size();  // 变量个数
    priority_queue<Node, vector<Node>, CompareNode> pq;  // 优先级队
    ↪  列
    Node rootNode;
    rootNode.solution.resize(n, 0);
    rootNode.value = 0;
```

```
    rootNode.level = 0;
    pq.push(rootNode);
    int round = 1;
    while (!pq.empty()) {
        cout << "Round " << round << ":\n";
        Node currentNode = pq.top();
        pq.pop();
        cout << "Current- Level: " << currentNode.level << ",
        ↪ Solution: [";
        for (int i = 0; i < n; ++i) {
            cout << currentNode.solution[i];
            if (i < n - 1) {
                cout << ", ";
            }
        }
        cout << "], Value: " << currentNode.value << "\n";
        if (currentNode.level == n) {
            cout << "Found a BEST solution: [";
            for (int i = 0; i < n; ++i) {
                cout << currentNode.solution[i];
                if (i < n - 1) {
                    cout << ", ";
                }
            }
            cout << "], Value: " << currentNode.value << "\n";
            return currentNode.value;  // 返回当前节点的目标函数值
        } else {
            for (int i = 0; i <= 1; ++i) {  // 分支，每个变量可以取整
            ↪ 数值或者不取整数值
                Node childNode = currentNode;
                childNode.solution[currentNode.level] = i;
                childNode.level++;
                // 计算子节点的目标函数值
                childNode.value = 0;
                for (int j = 0; j < n; ++j) {
                    childNode.value += childNode.solution[j] *
                    ↪ lp.objectiveFunction[j];
```

```
78                  }
79                  // 计算约束条件是否满足
80                  bool constraintsSatisfied = true;
81                  for (int j = 0; j < lp.coefficients.size(); ++j) {
82                      double constraintValue = 0;
83                      for (int k = 0; k < n; ++k) {
84                          constraintValue += childNode.solution[k] *
                            ↪  lp.coefficients[j][k];
85                      }
86                      if (constraintValue > lp.constants[j]) {
87                          constraintsSatisfied = false;
88                          break;
89                      }
90                  }
91                  if (constraintsSatisfied) {
92                      cout << "Look Child - LP: " << childNode.level
                        ↪  << ", Solution: [";
93                      for (int i = 0; i < n; ++i) {
94                          cout << childNode.solution[i];
95                          if (i < n - 1) {
96                              cout << ", ";
97                          }
98                      }
99                      cout << "], Value: " << childNode.value << "\n";
100                     // 如果是整数解，更新最优解
101                     if (isIntegerSolution(childNode.solution) &&
                        ↪  childNode.value < pq.top().value) {
102                         pq.push(childNode);
103                     } else if
                        ↪  (!isIntegerSolution(childNode.solution)) {
104                         pq.push(childNode);
105                     }
106                 }
107             }
108         }
109         round++;
110     }
```

```
    return -1;  // 未找到最优解
}

int main() {
    LinearProgram lp;
    lp.coefficients = {
            {-1, 1},   // 系数矩阵
            {2, 3}
    };
    lp.constants = {1, 11};
    lp.objectiveFunction = {-3, -5};
    int result = branchAndBound(lp);
    cout << "Optimal value: " << result << endl;
    return 0;
}
```

代码中以下面式子为例子，给出了详细的过程.

```
min z = -3x-5y
s.t.
    -x + y <= 1
    2x +3y <= 11
```

结果为

```
Round 1:
Current- Level: 0, Solution: [0, 0], Value: 0
Look Child - LP: 1, Solution: [0, 0], Value: 0
Look Child - LP: 1, Solution: [1, 0], Value: -3
Round 2:
Current- Level: 1, Solution: [1, 0], Value: -3
Look Child - LP: 2, Solution: [1, 0], Value: -3
Look Child - LP: 2, Solution: [1, 1], Value: -8
Round 3:
Current- Level: 2, Solution: [1, 1], Value: -8
Found a BEST solution: [1, 1], Value: -8
Optimal value: -8
```

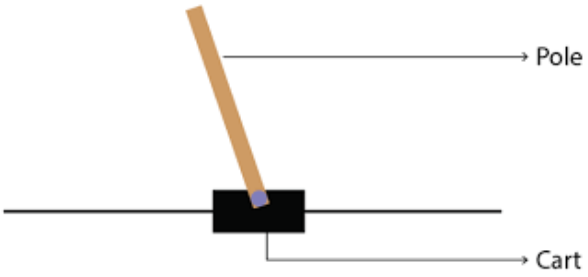

图 2: CartPole-v1 环境

9 强化学习

强化学习是指智能体 (Agent) 在一定的环境下，通过采取一系列的活动并通过活动的反馈学习优化自身的算法。而深度强化学习是将深度学习应用在强化学习里，我实现了深度强化学习算法 DQN, 之后使用 DQN 在 OpenAi 提供的 Gym 环境下进行试验。

9.1 DQN

- **算法思路**：Deep Q-Network 是一种深度强化学习算法，他是 Q-Learning 的深度强化学习实现，通过深度神经网络来近似 Q 函数，从而学习到在不同状态下选择不同动作的最优策略。
- **经验回放**：经验回放是指我们探索过的每一条经验都会被放在缓冲池存起来，之后可以反复使用。采用随机的策略去抽取，打破了数据的相关性，提高算法的训练效率和稳定性。
- **目标网络**：目标网络的主要作用是生成目标 Q 值，这种目标 Q 值用于计算 Q-learning 的损失函数。在传统 Q-learning 中，Q 值的更新直接依赖于当前估计的 Q 值，而 DQN 使用目标网络来生成目标 Q 值，减缓目标的变化，有助于更稳定地学习。
- **Gym 环境**：OpenAI 下的 Gym 环境是一个能够提供各种的模拟环境，包括 Atari 游戏，棋盘游戏，2D 以及 3D 模拟的工具包，供我们训练自己的智能体。自己使用 CartPole-v1 环境。
- **Python 实现**：自己使用 PyTroch 实现了 DQN，网络架构为很简单的三个全连接层，激活函数使用 SELU 激活函数，学习率是 1/10000，下面是结果示意图。

```
1 import torch
2 from torch import nn
3 from torch.nn import functional
4 from torch.utils.data import TensorDataset, DataLoader
```

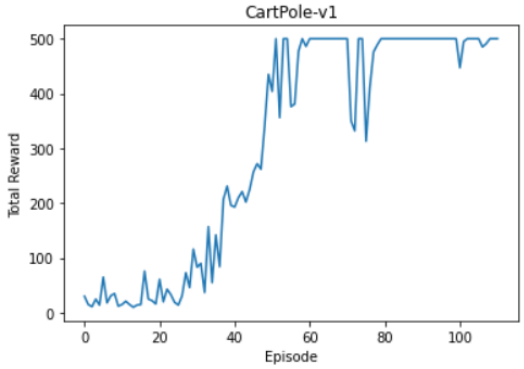

图 3: 结果图

```
import gym
from collections import deque
import random
class Model(nn.Module):
    def __init__(self, input_features, output_values):
        super(Model, self).__init__()
        self.fc1 = nn.Linear(in_features=input_features, out_features=32)
        self.fc2 = nn.Linear(in_features=32, out_features=32)
        self.fc3 = nn.Linear(in_features=32, out_features=output_values)

    def forward(self, x):
        x = functional.selu(self.fc1(x))
        x = functional.selu(self.fc2(x))
        x = self.fc3(x)
        return x

# Parameters
use_cuda = True
episode_limit = 100
target_update_delay = 2  # update target net every target_update_delay
↪  episodes
test_delay = 10
learning_rate = 1e-4
epsilon = 1  # initial epsilon
min_epsilon = 0.1
epsilon_decay = 0.9 / 2.5e3
gamma = 0.99
```

```
memory_len = 10000

env = gym.make('CartPole-v1')
n_features = len(env.observation_space.high)
n_actions = env.action_space.n

memory = deque(maxlen=memory_len)
# each memory entry is in form: (state, action, env_reward, next_state)
device = torch.device("cuda" if use_cuda and torch.cuda.is_available()
    else "cpu")
criterion = nn.MSELoss()
policy_net = Model(n_features, n_actions).to(device)
target_net = Model(n_features, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

def get_states_tensor(sample, states_idx):
    sample_len = len(sample)
    states_tensor = torch.empty((sample_len, n_features),
        dtype=torch.float32, requires_grad=False)

    features_range = range(n_features)
    for i in range(sample_len):
        for j in features_range:
            states_tensor[i, j] = sample[i][states_idx][j].item()

    return states_tensor
def normalize_state(state):
    state[0] /= 2.5
    state[1] /= 2.5
    state[2] /= 0.3
    state[3] /= 0.3

def state_reward(state, env_reward):
    return env_reward - (abs(state[0]) + abs(state[2])) / 2.5


```

```
def get_action(state, e=min_epsilon):
    if random.random() < e:
        # explore
        action = random.randrange(0, n_actions)
    else:
        state = torch.tensor(state, dtype=torch.float32, device=device)
        action = policy_net(state).argmax().item()

    return action

def fit(model, inputs, labels):
    inputs = inputs.to(device)
    labels = labels.to(device)
    train_ds = TensorDataset(inputs, labels)
    train_dl = DataLoader(train_ds, batch_size=5)

    optimizer = torch.optim.Adam(params=model.parameters(),
    ↪  lr=learning_rate)
    model.train()
    total_loss = 0.0

    for x, y in train_dl:
        out = model(x)
        loss = criterion(out, y)
        total_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    model.eval()

    return total_loss / len(inputs)

def optimize_model(train_batch_size=100):
    train_batch_size = min(train_batch_size, len(memory))
    train_sample = random.sample(memory, train_batch_size)

    state = get_states_tensor(train_sample, 0)
```

```
    next_state = get_states_tensor(train_sample, 3)

    q_estimates = policy_net(state.to(device)).detach()
    next_state_q_estimates = target_net(next_state.to(device)).detach()

    for i in range(len(train_sample)):
        q_estimates[i][train_sample[i][1]] = (state_reward(next_state[i],
        ↪  train_sample[i][2]) +
                                              gamma *
                                              ↪  next_state_q_estimates[i].max())

    fit(policy_net, state, q_estimates)

def train_one_episode():
    global epsilon
    current_state = env.reset()
    normalize_state(current_state)
    done = False
    score = 0
    reward = 0
    while not done:
        action = get_action(current_state, epsilon)
        next_state, env_reward, done, _ = env.step(action)
        normalize_state(next_state)
        memory.append((current_state, action, env_reward, next_state))
        current_state = next_state
        score += env_reward
        reward += state_reward(next_state, env_reward)
        optimize_model(100)
        epsilon -= epsilon_decay

    return score, reward

def test():
    state = env.reset()
    normalize_state(state)
    done = False
```

```
137     score = 0
138     reward = 0
139     while not done:
140         action = get_action(state)
141         state, env_reward, done, _ = env.step(action)
142         normalize_state(state)
143         score += env_reward
144         reward += state_reward(state, env_reward)
145 
146     return score, reward
147 
148 def main():
149     best_test_reward = 0
150     res_score = []
151 
152     for i in range(episode_limit):
153         score, reward = train_one_episode()
154         print(f'Episode {i + 1}: score: {score} - reward: {reward}')
155         res_score.append(score)
156         if i % target_update_delay == 0:
157             target_net.load_state_dict(policy_net.state_dict())
158             target_net.eval()
159 
160         if (i + 1) % test_delay == 0:
161             test_score, test_reward = test()
162             print(f'Test Episode {i + 1}: test score: {test_score} - test
                ↪  reward: {test_reward}')
163             if test_reward > best_test_reward:
164                 print('New best test reward. Saving model')
165                 best_test_reward = test_reward
166                 torch.save(policy_net.state_dict(), 'policy_net.pth')
167 
168     if episode_limit % test_delay != 0:
169         test_score, test_reward = test()
170         print(f'Test Episode {episode_limit}: test score: {test_score} -
            ↪  test reward: {test_reward}')
171         if test_reward > best_test_reward:
```

```
172             print('New best test reward. Saving model')
173             best_test_reward = test_reward
174             torch.save(policy_net.state_dict(), 'policy_net.pth')
175 
176     print(f'best test reward: {best_test_reward}')
177 
178 if __name__ == '__main__':
179     main()
```