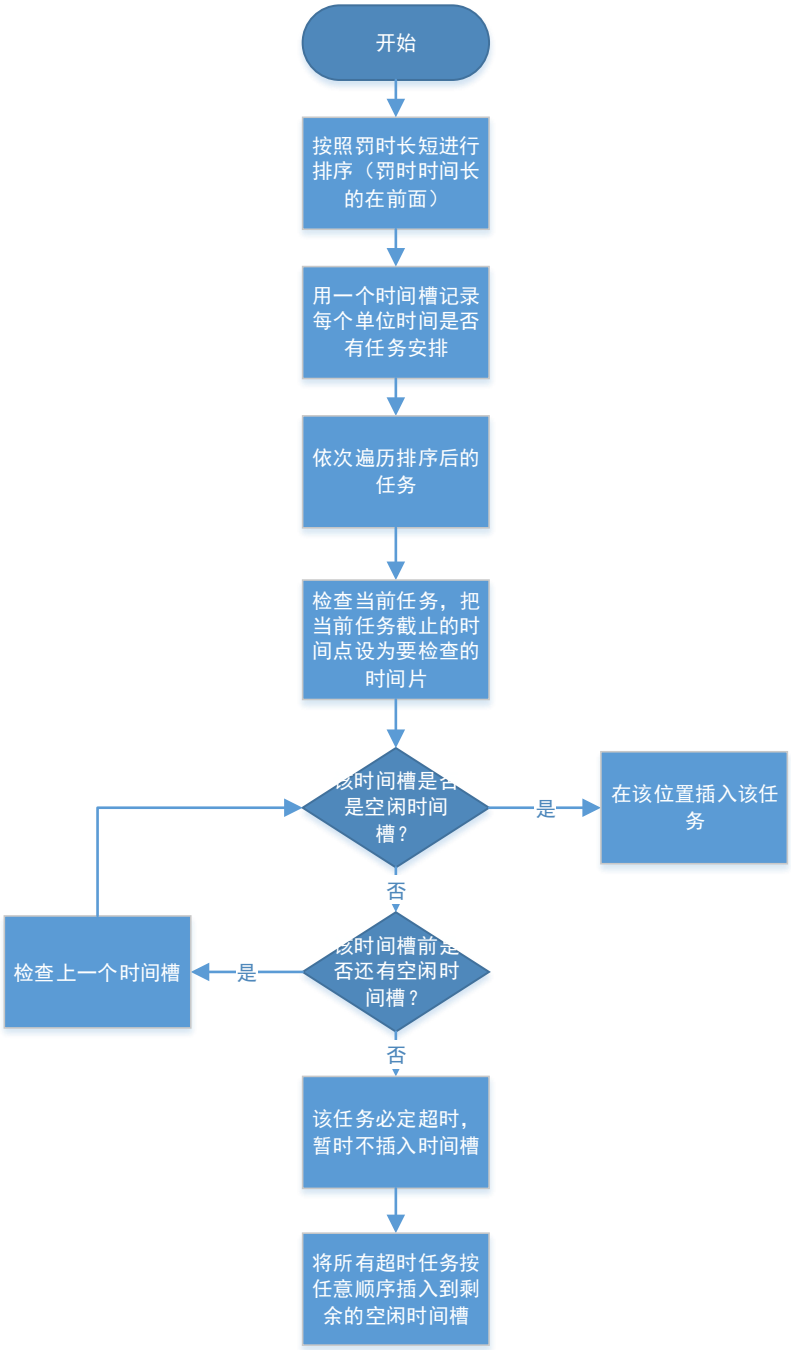
实验四 贪心算法实现最佳任务调度 实验报告

姓名：陈矛　　学号：SA22225116

一、实验原理

1. 算法流程图：



2. 活动选择问题：

对几个互相竞争的活动进行调度，它们都要求以独占的方式使用某一公共资源。而在同一时间内只有一个活动能使用这一资源。 假设有一个需要使用某一资源的 n 个活动组成的集合 S={a1,a2,a3,...,an}。 每个活动 ai 都有一个要求使用该资源的起始时间 si 和一个结束时间 fi,且 si <fi。 如果选择了活动 i， 则它在

半开时间区间[si, fi)内占用资源。 若区间[si, fi)与区间[sj, fj)不相交,则称活动 i 与活动 j 是兼容的。活动选择问题就是要选择出一个由互不兼容的问题组成的最大子集合。

3. 贪心策略

动态规划是贪心算法的基础。贪心算法即通过做一系列的选择来给出某一问题的最优解。 对算法中的每一个决策点， 做一个当时最佳的选择。

1) 贪心算法的使用条件： 贪心选择性质和最优子结构是两个关键的特点。
   如果我们能够证明问题具有这些性质， 那么就可以设计出它的一个贪心算法。

   a) 贪心选择性质： 一个全局最优解可以通过局部最优（贪心） 选择来达到。
   b) 最优子结构： 对一个问题来说， 如果它的一个最优解包含了其子问题的最优解， 则称该问题具有最优子结构。

2) 贪心算法的基本思路：
   a) 建立对问题精确描述的数学模型， 包括定义最优解的模型；
   b) 将问题分解为一系列子问题， 同时定义子问题的最优解结构；
   c) 应用贪心原则确定每个子问题的局部最优解， 并根据最优解的模型， 用子问题的局部最优解堆叠出全局最优解。

4. 时间复杂度和空间复杂度

1) 该算法的时间复杂度为 O(nlogn)。根据后续的代码，可以看出，该算法总共可以分为两个流程：一是对任务进行排序，二是依次检查排序后的任务并放到对应的时间片位置。排序使用 Java 自带的排序函数，默认使用快速排序，时间复杂度为 O(nlogn)。依次检查任务时，时间复杂度为 O(n)。所以最后的总时间复杂度为 O(nlogn)。
2) 该算法的空间复杂度为 O(n)。需要建立一个和任务个数 n 等长的时间片数组，用来记录每个任务被存放在哪个位置，该操作所需的空间复杂度为 O(n)。

二、运行结果及分析

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

图 16-7 单处理器上带期限和惩罚的单位时间任务调度问题的一个实例

根据该任务序列，实现任务调度问题。

1. 实现这个问题的贪心算法， 并写出流程图或者伪代码。

算法流程图：

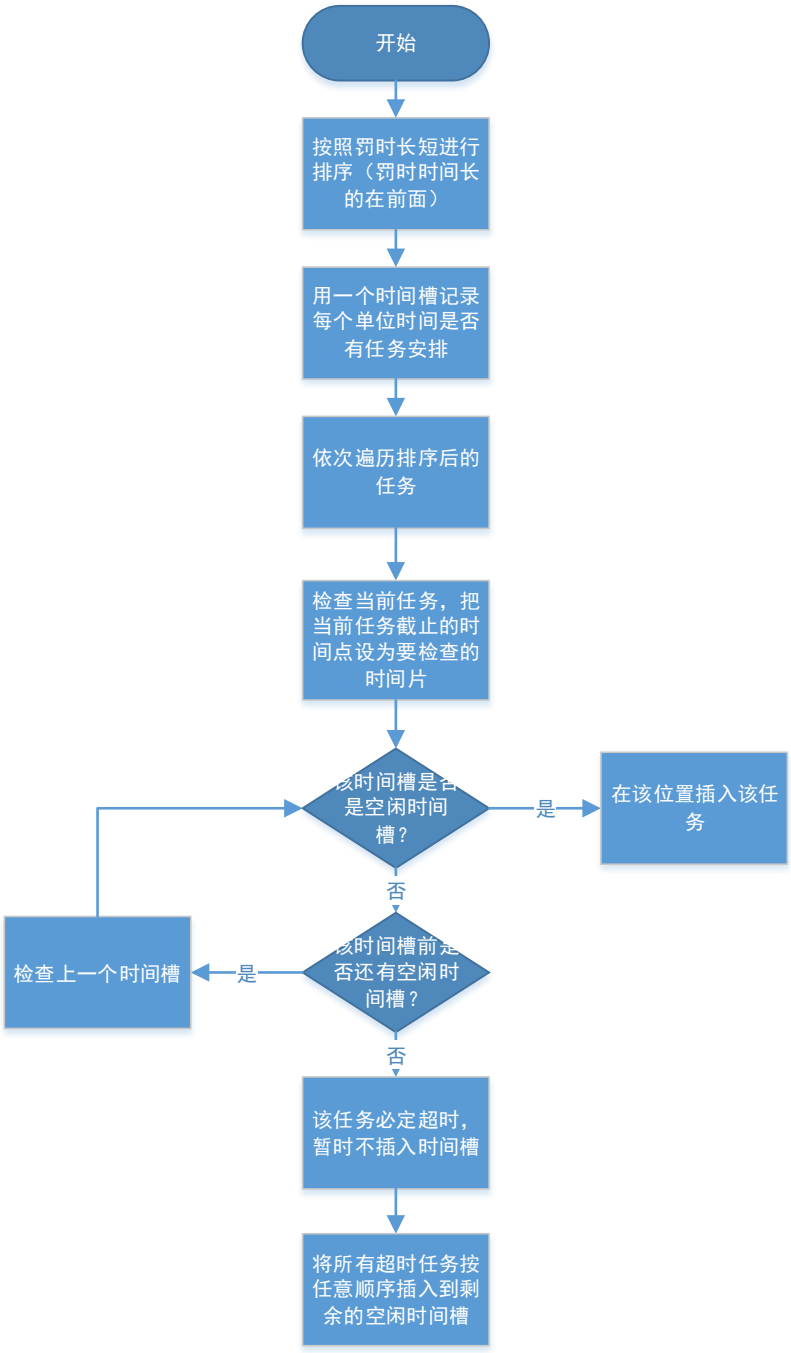


图 算法流程图

关键代码：

```
public class TaskDispatch {

    private static void greedyTaskDispatch(Task[] tasks) {
        int n = tasks.length;
        Arrays.sort(tasks); // 根据惩罚从大到小排序
        System.out.println("Task 按惩罚从大到小排序后：");
        for(Task t : tasks) {
```

```
            System.out.println(t.toString());
        }

        int[] route = new int[n]; // 记录任务的最终调度顺序，route[i]表示第(i+1)天调度的任务id。天数从1开始，而数组下标从0开始。
        Arrays.fill(route, -1);

        int punishment = 0; // 记录总的惩罚值
        for(int i = 0; i < n; ++i) { // 处理第i个任务
            for(int j = tasks[i].deadline - 1; j >= 0; --j) { // deadline的范围是[1,n]，故此处需修正下标
                // j为第i个任务的deadline，应将第i个任务尽可能在靠近deadline的天完成。
                // 这样卡点完成的好处是：让后续有更紧急deadline要求的任务更可能按时完成。
                if(route[j] == -1) { // 若第j天未安排任务
                    route[j] = tasks[i].id;
                    break;
                }
                if(j == 0) { // 第1天~第deadline天都已安排了任务，故当前任务必超时
                    punishment += tasks[i].weight;
                }
            }
        }

        System.out.println("总的惩罚为: " + punishment);
        System.out.println("任务执行顺序为: ");
        for(int i = 0; i < n; ++i) {
            System.out.print("第" + (i+1) + "天执行的任务为: ");
            if(route[i] != -1) {
                System.out.println(" " + tasks[route[i]]);
            }
            else {
                System.out.println(" 任选一个已超时任务执行");
            }
        }

    }

    public static void main(String[] args) {
        int n = 7; // 任务数量
        int[] deadline = new int[]{4,2,4,3,1,4,6}; // 任务期限，范围是闭区间[1,n]
```

```
        int[] weight = new int[]{70,60,50,40,30,20,10}; // 任务未按时完成的惩罚

        Task[] tasks = new Task[n];
        for(int i = 0; i < n; ++i) {
            tasks[i] = new Task(i, deadline[i], weight[i]);
        }
        greedyTaskDispatch(tasks);
    }

}
```

贪心思路：

贪心考虑的是局部的最优解，而不是全局的最优解，为了达到最小的误时惩罚，按照贪心策略，肯定是**先去按时完成或提前完成惩罚较大的任务**。若一个时间片上已经安排了按时完成的、有着较大惩罚的任务，那么，这个有着较小惩罚的任务，则需要提前完成（提前完成有着很多种方法，按照贪心策略，应该是放在紧邻的上一个时间片完成，若这个时间片也有任务，继续往前移，直到有空闲的时间片可以执行这个任务，否则，这个任务会带来惩罚）。

算法实现流程：

1) 根据罚时的长短进行排序，将罚时时间长的放在前面。
2) 开一个数组作为时间槽，记录每个单位时间是否有任务安排。
3) 根据罚时的时间长短判断哪个优先 （罚时长的优先，在步骤 1 中已经完成了排序，依次遍历即可），尽量将任务安排在截至时间完成，否则把任务往前一个时间片放，以此类推。
4) 若在截至时间前都有任务安排，先不要将该任务加入到时间槽中（可能产生后效性），可以将这个任务舍去，同时，将其增加到罚时中。
5) 等到对全部的任务过滤一遍后，可以将罚时的任务随意地插入到时间槽中（既然都有罚时了，那么在截止时间后的任何一个时间片，完成该任务都一样）。

算法运行结果（其中包括排序后的 Task 顺序、总的惩罚、任务执行的顺序）：

```
Task按惩罚从大到小排序后：
Task{id=0, deadline=4, weight=70}
Task{id=1, deadline=2, weight=60}
Task{id=2, deadline=4, weight=50}
Task{id=3, deadline=3, weight=40}
Task{id=4, deadline=1, weight=30}
Task{id=5, deadline=4, weight=20}
Task{id=6, deadline=6, weight=10}
总的惩罚为：50
任务执行顺序为：
第1天执行的任务为：  Task{id=3, deadline=3, weight=40}
第2天执行的任务为：  Task{id=1, deadline=2, weight=60}
第3天执行的任务为：  Task{id=2, deadline=4, weight=50}
第4天执行的任务为：  Task{id=0, deadline=4, weight=70}
第5天执行的任务为：  任选一个已超时任务执行
第6天执行的任务为：  Task{id=6, deadline=6, weight=10}
第7天执行的任务为：  任选一个已超时任务执行

Process finished with exit code 0
```

得到这样结果的原因（以该示例为例）：

1) 首先将所有的 Task 按照 weight 降序排序，这样子做之后，可以保证优先给超时代价更高的任务安排合适的运行时间。
2) 排序完之后，依次扫描任务。第一次检查 Task0 的 deadline 为 4，按照贪心算法的思想，会将该任务安排到第四个时间槽执行。此时任务序列为{-1,-1,-1,task0,-1,-1,-1}（-1 表示尚未安排任务）。
3) 同理，检查完 Task1 之后，任务序列为：{-1,task1,-1,task0,-1,-1,-1}。
4) 在检查 Task2 时，发现该任务的 deadline 为 4，但第四个时间槽已经有任务了，于是按照该算法，向前移动一个时间槽，将该任务安排到第三个时间槽上。此时任务序列为{-1,task1,task2,task0,-1,-1,-1}。
5) 同理，检查 task3 后，任务序列为{task3,task1,task2,task0,-1,-1,-1}。
6) 当检查 task4 时，发现在 deadline 之前已经没有空闲的时间槽了，所以该任务必定超时，暂时不处理它，并将它的超时代价记录下来。同理，task5 也一定会超时。
7) 检查 task6 时，按照之前的规律将其加入时间片，任务序列为{task3,task1,task2,task0,-1,task6,-1}。
8) 至此，所有任务检查完毕。会发现安排任务的时间片里面还空闲了两个块，正好对应两个超时任务，按任意顺序插入其中即可。
9) 至此，算法执行结束，所有任务得到调度，总的惩罚为 50。

2. 将每个 Wi 替换为 max{W1,W2……Wn}-Wi 运行算法、 比较并分析结果。

替换所有 Wi 后，算法整体思路不变，只需要改变 Wi 数组即可，所以在执行算法之前，加入代码段：

```
int maxWeight = weight[0];
for(int w : weight) {
    maxWeight = w > maxWeight ? w: maxWeight;
}
for(int i = 0; i < n; ++i) {
    weight[i] = maxWeight - weight[i];
}
```

完成对 Wi 的修改即可。

算法的流程和第一问中完全一致，故不再赘述。

修改 Wi 后的代码运行结果如下：

```
Task按惩罚从大到小排序后:
Task{id=6, deadline=6, weight=60}
Task{id=5, deadline=4, weight=50}
Task{id=4, deadline=1, weight=40}
Task{id=3, deadline=3, weight=30}
Task{id=2, deadline=4, weight=20}
Task{id=1, deadline=2, weight=10}
Task{id=0, deadline=4, weight=0}
总的惩罚为: 10
任务执行顺序为:
第1天执行的任务为:  Task{id=2, deadline=4, weight=20}
第2天执行的任务为:  Task{id=4, deadline=1, weight=40}
第3天执行的任务为:  Task{id=3, deadline=3, weight=30}
第4天执行的任务为:  Task{id=1, deadline=2, weight=10}
第5天执行的任务为:  任选一个已超时任务执行
第6天执行的任务为:  Task{id=0, deadline=4, weight=0}
第7天执行的任务为:  任选一个已超时任务执行

Process finished with exit code 0
```

三、实验总结

实验里面有很多容易出错的地方，比如在进行任务排序时，需要注意罚时长短的顺序；在安排任务时，需要注意罚时的限制，需要尽量安排在截至时间内完成，否则需要考虑往前推进；在处理罚时任务时，需要注意不要产生后效性，需要舍去过早的任务。

关于自己的代码部分，在写代码的过程中有遇到过一些 bug，其中几个关键的地方包括：安排任务时需要使用循环进行遍历，并且需要使用判断语句进行判断是否能在截止时间完成，如果不能，需要推进时间。还有罚时任务处理时需要进行舍去，需要使用判断语句。

做完这次实验，有很多的收获: 通过实验，对贪心算法在任务调度问题上的应用有了更深入的了解。不仅如此，还了解了罚时对任务调度的影响，学习到了代码实现贪心算法的方法。总之，完成这个算法实验让我更好地理解贪心算法的思想以及在实际问题中的应用。

实验日期：2022 年 12 月 7 日

附录

实验源代码：

1. Task.java

```java
package SolutionForLab4;

/**
 * @author ChenMao
 * @create 2022-12-07 20:41
 */

public class Task implements Comparable<Task> {
    public int id;
    public int deadline;
    public int weight;

    public Task(int id, int deadline, int weight) {
        this.id = id;
        this.deadline = deadline;
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Task{" +
                "id=" + id +
                ", deadline=" + deadline +
                ", weight=" + weight +
                '}';
    }

    @Override
    public int compareTo(Task newTask) { // 按惩罚由大到小排序，调用 Arrays.sort 时会用到
        return Integer.compare(newTask.weight, this.weight);
    }

}
```

2. TaskDispatch.java

```java
package SolutionForLab4;

/**
```

```java
 * @author ChenMao
 * @create 2022-12-07 22:54
 */
import java.util.Arrays;


// 在单处理器上具有期限和惩罚的单位时间任务调度问题。
public class TaskDispatch {

    private static void greedyTaskDispatch(Task[] tasks) {
        int n = tasks.length;
        Arrays.sort(tasks); // 根据惩罚从大到小排序
        System.out.println("Task 按惩罚从大到小排序后：");
        for(Task t : tasks) {
            System.out.println(t.toString());
        }

        int[] route = new int[n]; // 记录任务的最终调度顺序，route[i]表示第(i+1)天调度的任务 id。天数从 1 开始，而数组下标从 0 开始。
        Arrays.fill(route, -1);

        int punishment = 0; // 记录总的惩罚值
        for(int i = 0; i < n; ++i) { // 处理第 i 个任务
            for(int j = tasks[i].deadline - 1; j >= 0; --j) { // deadline 的范围是[1,n]，故此处需修正下标
                // j 为第 i 个任务的 deadline，应将第 i 个任务尽可能在靠近 deadline 的天完成。
                // 这样卡点完成的好处是：让后续有更紧急 deadline 要求的任务更可能按时完成。
                if(route[j] == -1) { // 若第 j 天未安排任务
                    route[j] = tasks[i].id;
                    break;
                }
                if(j == 0) { // 第 1 天~第 deadline 天都已安排了任务，故当前任务必超时
                    punishment += tasks[i].weight;
                }
            }
        }

        System.out.println("总的惩罚为：" + punishment);
        System.out.println("任务执行顺序为：");
        for(int i = 0; i < n; ++i) {
            System.out.print("第" + (i+1) + "天执行的任务为：");
```

```
            if(route[i] != -1) {
                System.out.println(" " + tasks[route[i]]);
            }
            else {
                System.out.println(" 任选一个已超时任务执行");
            }
        }

    }

    public static void main(String[] args) {
        int n = 7; // 任务数量
        int[] deadline = new int[]{4,2,4,3,1,4,6}; // 任务期限，范围
是闭区间[1,n]
        int[] weight = new int[]{70,60,50,40,30,20,10}; // 任务未按
时完成的惩罚

        // 将 Wi 替换为 max{W1,W2……Wn} - Wi
        int maxWeight = weight[0];
        for(int w : weight) {
            maxWeight = w > maxWeight ? w: maxWeight;
        }
        for(int i = 0; i < n; ++i) {
            weight[i] = maxWeight - weight[i];
        }

        Task[] tasks = new Task[n];
        for(int i = 0; i < n; ++i) {
            tasks[i] = new Task(i, deadline[i], weight[i]);
        }
        greedyTaskDispatch(tasks);
    }

}
```