2018

Job Shop 程序设计
作业概要设计

2017211305/2017211306 班 32 组

于海鑫/田静悦

Job Shop 大作业概要设计

1. 用户界面设计

1. 命令行模式

命令行模式使用方式如下：

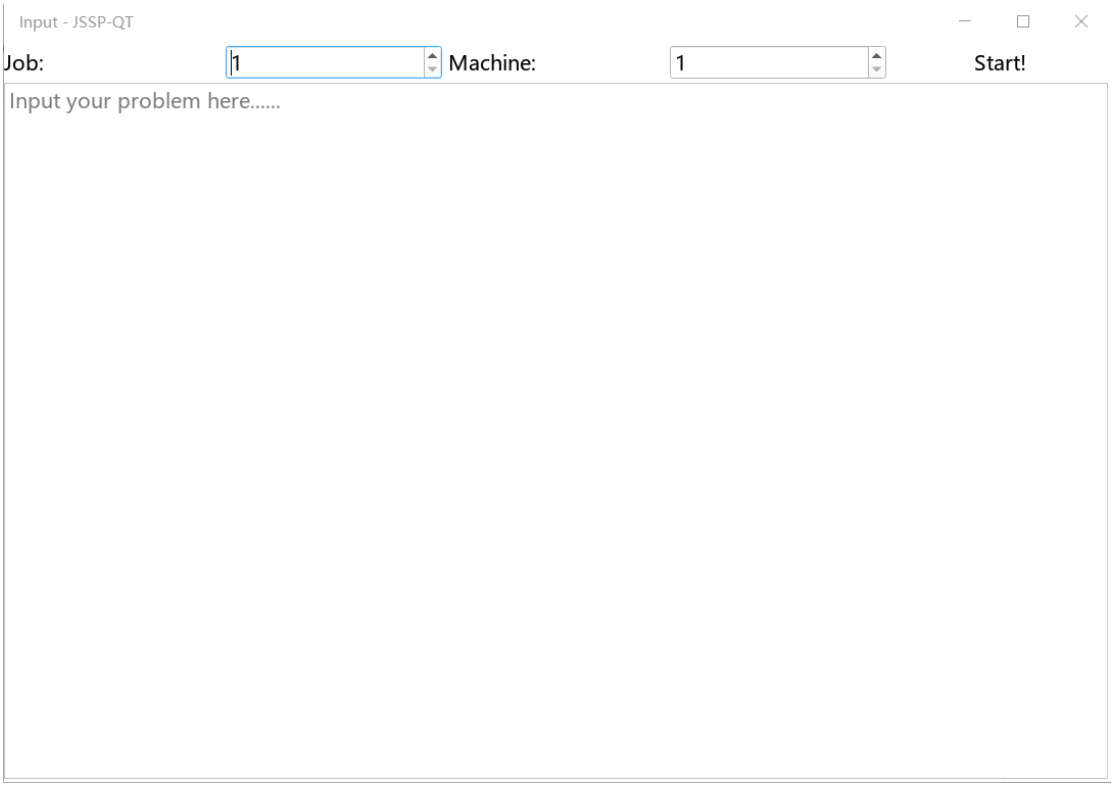
```
JobShop.exe < <输入文件>
```

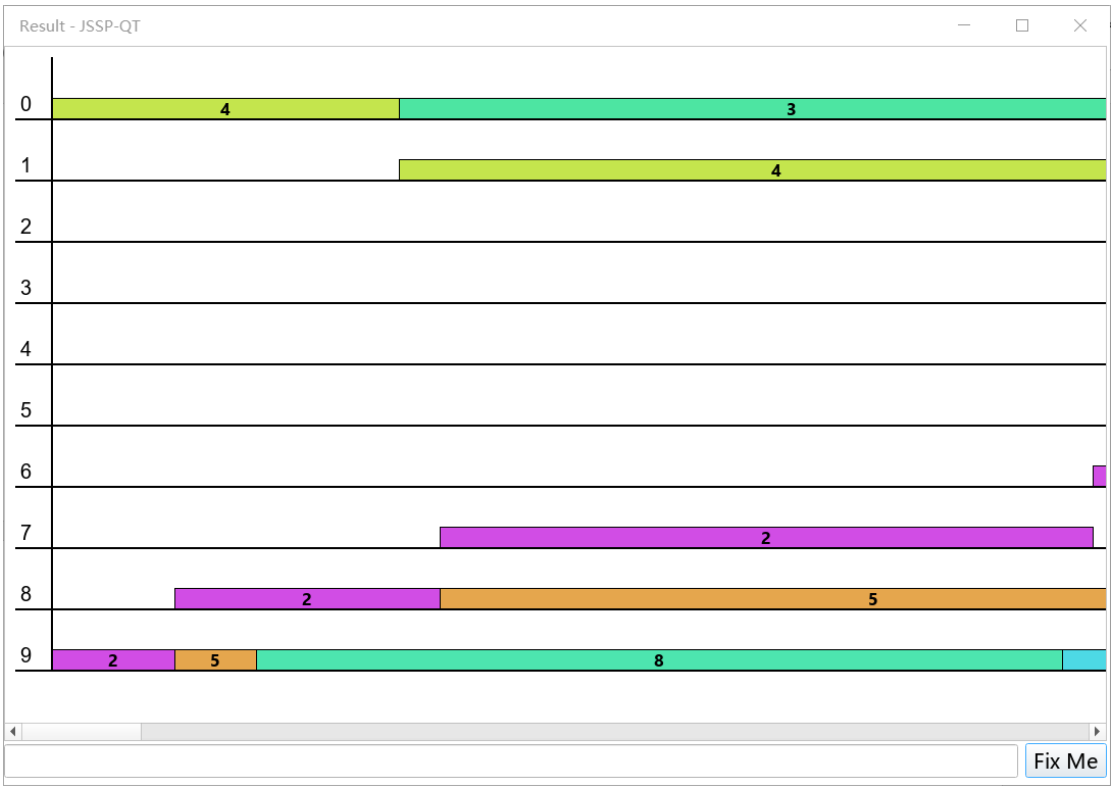
如果没有重定向输入文件，则默认从标准输入流中读取样例信息。

输出时会在命令行窗口显示每一个最新找到的解，当程序运行结束时，会输出当前最优解的排序方式，并输出程序的运行时间（以秒为单位）。同时会将最优解的排序方式以及 makespan 以写入的方式存储在 result.txt 中。

2. 图形界面模式

图形界面共两个窗口，效果如下：





3. 高层数据结构定义

1. 全局变量定义

```
job_t job[MAXJOBS];    // 用于记录样例信息以及生成结果

int job_size;          // 用于记录样例中工件的数目

int machine_size;      // 用于记录样例中机器的数目

int terminate_flag;    // 用于确认程序是否应当停止运行

#define MAXJOB 30          // 最大工件规模

#define MAXMACHINE 30  // 最大机器规模
```

2. 模块常量/变量定义

```
#define TRY_COUNT 10              // 算法回溯深度

int best_makespan = INFINITAS;// 最优的 makespan，INFINITAS 为 0x7fffffff
```

3. 主要结构体定义

```
typedef struct JOB {

 int            estime[MAXMACHINE];          /**< Earlist starting time of this job on

each machine. Which is simply the sum of this job's processing times on the machine

before [order[machine]] in this jobs prescribed ordering. */

 int            mhtime[MAXMACHINE];          /**< Minimum halting time of this job

after [machine num]. Which is simply the sum of this job's processing times on the

machine after [order[machine]] in this jobs prescribed ordering.*/

 int            magic[MAXMACHINE];           /**< The number generated and managed

by the God in the computer. Every one who changed the name of this feild will be seen as

an evil and will be cursed by the God. */

 int            order[MAXMACHINE];           /**< Required machine order for the job.

*/

 int            process_time[MAXMACHINE];/**< Process time of each machine. */

 int            step[MAXMACHINE];            /**< Solution step indexed by machine. */

 int            next[MAXMACHINE];            /**< Next job on machine [i]. */

 int            prev[MAXMACHINE];            /**< Previous job blah blah. */

 int            start[MAXMACHINE];           /**< Start time of this job on each

machine. */
```

4. 系统模块划分

1. 模块介绍

1.1 algorithm/io.cpp

处理输入输出

1.2 algorithm/bottle.cpp

移动瓶颈算法的实现部分

1.3 algorithm/eval.cpp

计算解的 makespan

1.4 algorithm/onemachine.cpp

基于 J. Carlier 的 “The One Machine Problem

“ 实现的基于 Schrage 算法的分支定界法

1.5 gantt/*.cpp

生成甘特图

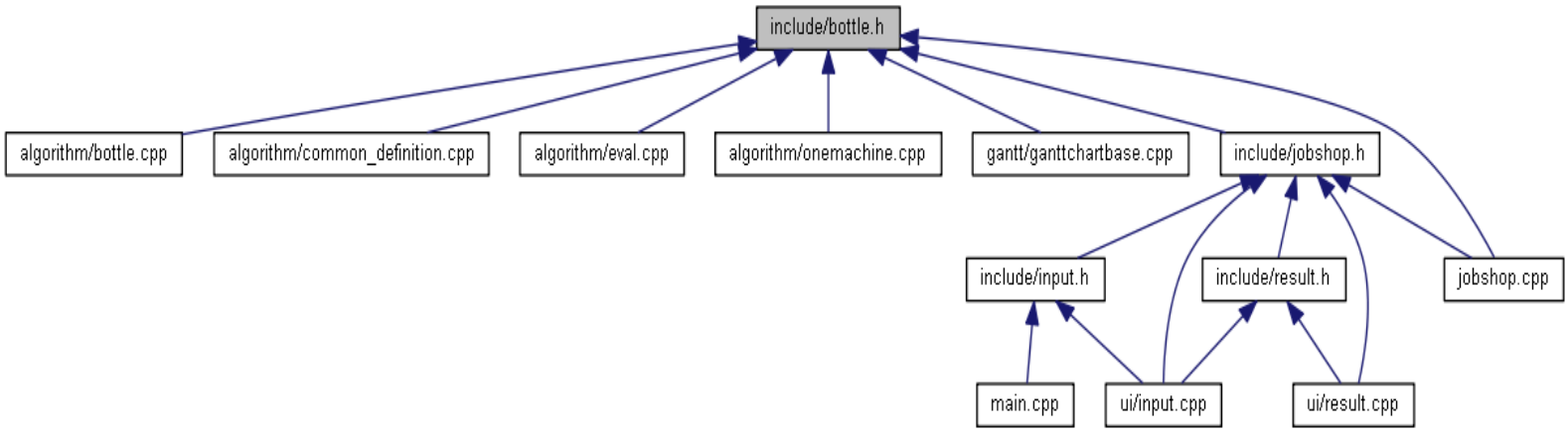
1.6 ui/*.cpp

绘制窗口

1.7 main.cpp

为程序提供入口点

2. 模块关系图



3. 模块主要函数说明

3.1bottle.c

1) int run_bottle_neck(void)

移动瓶颈法的驱动函数

2) static void shifting_bottle_neck(sequence_t *seq, mo_t *mo, int *times,int *final_makespan)

移动瓶颈法的主要实现

3.2eval.c

1) int eval(sequence_t* sequence)

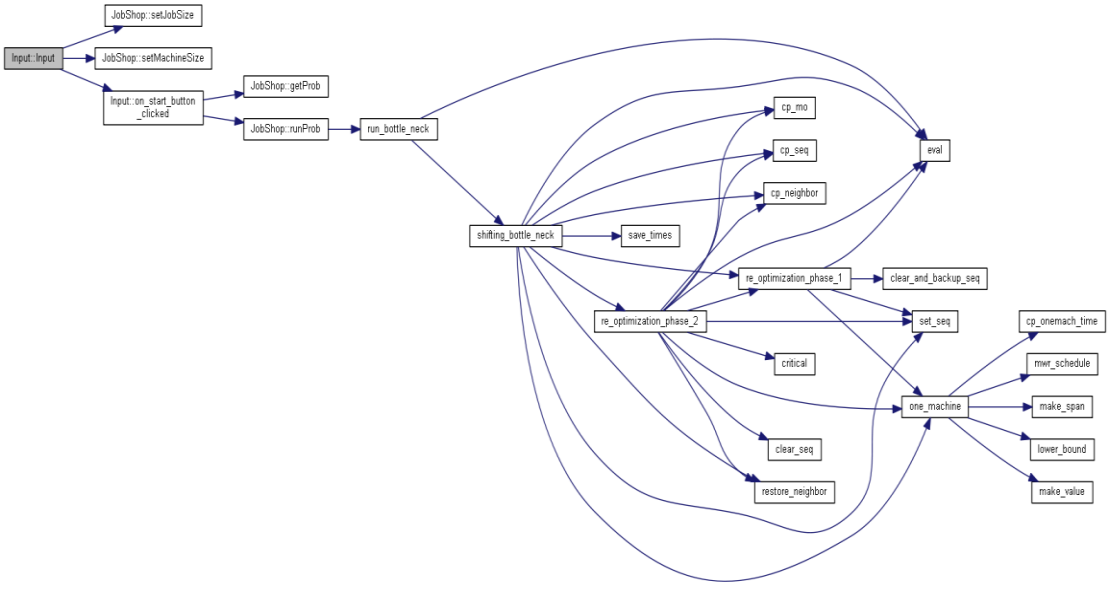
计算给定的序列的 makespan

3.3onemachine.c

1) int one_machine(onemach_times_t one, int *bestorder)

使用分支定界法为单个机器进行排序

4. 函数调用关系图



4 算法概述

1. 算法概述

该大作业中使用的算法是 J.Adams 等人于 1988 年提出的移动瓶颈法，该算法可表述如下：

- **记 M 为全部机器构成的集合**
- **给定已经将操作排好序的机器集合 M^0 （也就是给定属于 M^0 的机器的析取弧的方向）**
- **迭代：**
    - **选出没有排序的机器中的“瓶颈机器”，记为 k**
    - **基于 M^0 的排序，对 k 上的操作进行排列**
    - **对 M^0 中的机器进行重新排序**

尽管原始的移动瓶颈法得到的结果已经能在较短的时间里生成尚可的解法，但其结果仍和最优解相去甚远。因此我们在基本的移动瓶颈法上引入回溯机制。也就是说我们在每一个迭代过程中不仅仅处理一个瓶颈，而是进而处理多个瓶颈，找出最优的结果来进行下一次迭代。实践表明该方法可以提高解的质量，不过回溯层数的选取需要较多的实验才能得出。

2. 瓶颈机器的选取

选取所谓瓶颈机器的基本思想很简单，就是找到在未排序的机器中，运行时间最长的那一个即可。在实践中，这个部分可以在计算其余过程时“顺便”进行。

3. 瓶颈机器上的排序

在瓶颈机器上的排序，其实就是解决单一机器排序问题。该部分 J.Carlier 已经做了较为详细的研究。因此本次大作业中的排序算法就是基于 J. Carlier 于 1982 年发表的论文“The one-machine sequencing problem” 所描述的算法进行的。其本质是应用了 Scharge 算法的分支定界法。在此不再赘述。

4. 重排

所谓重排，也是一个不断应用解决单一机器排序问题的算法的过程。该过程可以表述如下：

- **给定已经将操作排好序的机器集合** M^0
- **对全部** $k \in M^0$ **进行如下操作**
    - **清除其序列**
    - **将其余机器视为约束条件，求解单一机器排序问题**

此处我们认为，在进行重新排列时，应将结束时间最长的机器放在前面进行处理，使得重新排序时获得的优势最大化。故我们在此引入了一个排序过程，实践证实该改进可以以可以忽略的实践代价小幅度地提升解的质量。同时我们还会进行多次重排，进一

步提高解的质量。更进一步地，J.Adams 等人的论文中提到，如果在重排之后，移去最后几（记为 α ）个非关键机器上的序列并将之重新使用上述算法，还能更进一步得到更好的结果。Adams 给出的 α 值为 $|M|^{0.5}$，我们编写代码时亦使用此值。

5. 计算 makespan

本算法表示解序列的方式是基于析取图的表示法，因此直接建图计算 makespan 是可行的。但是根据我们之前使用人工蜂群算法以及遗传算法求解该问题的经验，建图过程是整个算法的瓶颈部分。究其原因，是内存引用的过于频繁。在本作业中，因为辅助变量的存在，我们可以使用动态规划的方法来快速求出 makespan。其状态转移方程如下：$makespan[i] = MAX\{makespan[i-1], job[i].estime[job[i].order[machine\_size - 1]] + job[i].process\_time[job[i].order[machine\_size - 1]]\}$

其中 $0 < i < jobsize, makespan[0] = 0$ 。

6. 解的输出

在求解 makespan 时，我们计算的 estime 其实就是各个操作的开始时间。因此当我们得到一个不错的 makespan 时，我们将解序列中的 estime 复制到 start_time 内。在输出时，引入中间变量的数组：

```
typedef struct PAIR_ASSISTANT_TYPE {
```

```
    int start_time;          /**< Start time of this node. */
    int job_num;       /**< Serial number of the job of this node. */
    int mach_num;          /**< Serial nubmer of the machine of this node. */
    int proc_time;          /**< Process time (a.k.a duration) of this node.*/
    int step;            /**< Serial number of the order of this node in the job. */
} pair_ass_t;
```

对其进行排序。即可得到输出。

5 创新点

1. 算法创新

原始的 Shifting Bottleneck 算法生成的结果并不是十分的优秀，因此我们在初始的 SB 算法上引入了回溯机制。使得该算法在执行时可以进行一定深度下的搜索。这给予了我们获取更优解的可能性。在实践中，可以发现这一机制切实地提升了解的质量。

2. 实现创新

在实现算法时，我们基于个人对于 C/C++ 运行时的理解。对程序的结构以及语句的执行顺序进行了一些小的调整，包括但不限于：

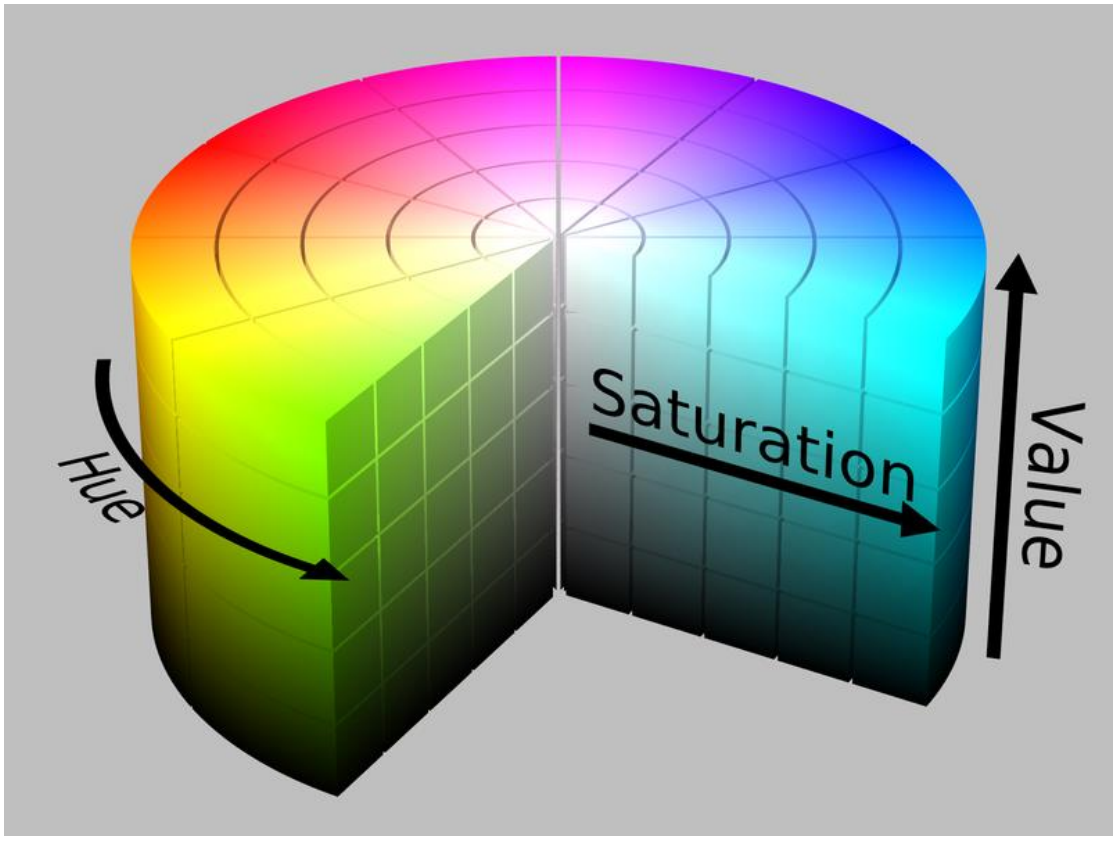
1) 在安排结构体内位域的位次时，将频繁使用的放在前面
2) 内部函数全部使用 static inline 修饰，减少函数调用开支。

同时，我们在使用 GCC 为低版本的 Windows 生成程序时，使用了如下编译选项进行编译，启用较为激进的优化策略保证执行效率：

```
CFLAGS = -Wall -Ofast -ffast-math -funroll-loops -malign-double -minline-all-stringops \
    -ftree-parallelize-loops=4 -flto -fuse-linker-plugin
```

3．颜色生成策略创新

如果仅仅简单地在 RGB 颜色空间随机选取颜色作为甘特图内的颜色的话。我们得到的结果可能不会十分理想，或者，非常难看。因此，在随机生成颜色时，我们没有使用通常的 RGB 颜色空间，而是在所谓的 HSV（色相-饱和度-明度） 颜色空间内随机生成点。



上图就是所谓的 HSV 颜色空间。在指定特定的饱和度以及明度之后，我们生成的颜色已经可以保障不会影响阅读了。但是，为了保证用户能看的赏心悦目，我们使用古老的黄金分割比例作为随机的基准，这样即可得到一系列较为赏心悦目的颜色。

当然，该生成策略也有其局限性，那就是生成的颜色有时会过于接近，如何有效地解决该问题还有待日后的深入研究。