Cost Constrained Fixed Job Scheduling

Qiwei Huang[1], Errol Lloyd[2]

[1] UTStarcom Inc.
33 Wood Ave. South, Iselin, NJ 08830,U.S.A
qhuang@utstar.com
[2] Dept. of Computer and Information Sciences
University of Delaware, Newark, DE 19716,U.S.A.
elloyd@cis.udel.edu

Abstract. In this paper, we study the problem of cost constrained fixed job scheduling (*CCFJS*). In this problem, there are a number of processors, each of which belongs to one of several classes. The unit time processing cost for a processor varies with the class to which the processor belongs. There are *N* jobs, each of which must be processed from a given start time to a given finish time without preemption. A job can be processed by any processor, and the cost of that processing is the product of the processing time and the processor's unit time processing cost. The problem is to find a feasible scheduling of the jobs such that the total processing cost is within a given cost bound. This problem (*CCFJS*) arises in several applications, including off-line multimedia gateway call routing. We show that *CCFJS* can be solved by a network flow based algorithm when there are only two classes of processors. For more than two classes of processors, we prove that *CCFJS* is not only *NP-Complete*, but also that there is no constant ratio approximation algorithm. Finally, we present an approximation algorithm, derive its worst-case performance ratio (non constant), and show that it has a constant approximation ratio in several special cases.

1 Introduction

Fixed job scheduling (sometimes called *interval scheduling*) has been studied extensively. In fixed job scheduling, we need to process without preemption a given set of jobs on several processors, such that a job starts at a given time and finishes at a given time. Many variations of *fixed job scheduling* have been considered in the literature (cf. Fischetti, Martello & Toth [1,2], Kolen & Kroon [3,4,5,6], Kroon, Sen, & Deng [7], and Jansen [8]), and the computational complexity of those variants has been established. In particular, Kroon, Sen, & Deng [7] studied the optimal cost chromatic partition problem (*OCCP*), one variation of *fixed job scheduling* with processor-dependent processing cost. In that problem, a sufficient number of processors are available. A job can be processed by any one processor during a fixed time interval, and if job *j* is carried out by processor *p*, then the associated processing cost is k_p. The objective is to find a feasible non-preemptive schedule to achieve the minimum total processing cost.

In this paper, we study cost constrained fixed job scheduling (*CCFJS)*, which is similar to [7]. In *CCFJS*, there are a number of processors, each of which belongs to

one of several classes. The unit time processing cost for a processor varies with the class to which the processor belongs. Each job requires processing by one and only one processor without preemption. The cost of processing a job is the unit time processing cost of the processor times the job's processing time. The problem is to find a feasible scheduling of the jobs within a given cost bound, or equivalently, to find a feasible scheduling of minimum cost. The difference between *CCFJS* and *OCCP* is that, in *OCCP*, the processing cost of a job depends only on the processor: if a number of jobs are processed by the same processor, they have the same processing costs; In *CCFJS*, the processing cost of a job depends not only on the processor, but also on the processing time (the processing cost is equal to the product of the unit time processing cost of that processor and the processing time).

CCFJS arises in off-line multimedia gateway call routing. Multimedia gateways interconnect different media networks (circuit-switched PSTN, packet-switched IP, ATM, wireless). A multimedia gateway routes each incoming call to one of its media networks. Media networks differ in the unit time media cost for calls routed to the network, and each media network has a bandwidth capacity on the number of simultaneous calls. The media cost of a call is equal to the call duration times the unit time media cost of the network to which the call is routed. The goal of call routing is to minimize the media cost (or equivalently within its cost bound) taken over all calls.

In section 2, we formally define *CCFJS* and provide relevant terminology. In section 3, we show that *CCFJS* can be solved by a network flow based algorithm (i.e. in polynomial time) when there are only two classes of processors. In section 4, we show that for more than two classes of processors, *CCFJS* is not only *NP-Complete*, but also that there is no constant ratio approximation algorithm. In section 5, we present an approximation algorithm, derive its worst-case performance ratio (non constant), and show that it has a constant approximation ratio in several special cases.

2 Problem Descriptions and Terminology

In this section, we formally define *CCFJS* as a decision problem. The optimization version of the problem should be clear.

Instance of *CCFJS*: A cost bound $C_B > 0$; Jobs $J_1, ...J_N$, and for each job J_i, a start time s_i and a finish time f_i ($0 \le s_i < f_i$); K classes of processors, and for each class $j = 1, ...K$, the number B_j of processors, and the unit time processing cost C_j for processors in this class. Let $B = \sum_{i=1}^{K} B_i$ be the total number of processors. We assume that $0 < C_1 < C_2 < ... < C_K$.

Question: Does there exist a feasible schedule for the N jobs, such that the cost $C_T = \sum_{i=1}^{N} C_{j_i}(f_i - s_i) \le C_B$? Here, C_{j_i} is the unit time processing cost of the processor on which job J_i is processed.

Relative to the specification of *CCFJS*, a feasible schedule is an assignment of each job to a processor such that each job must be processed by one processor from its

given start time to its given finish time without preemption, and each processor can process at most one job at a time.

Throughout this paper, we make use of the following terminology: Jobs J_i and $J_j (i \neq j)$ are *compatible* if the time intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. Job J_i is *active* at time t if $t \in [s_i, f_i)$. *Rank*(t) is the number of *active* jobs at time t. Further, *Rank*(J_i) is the maximum number of active jobs (including J_i itself) at any point of time $t \in [s_i, f_i)$. To avoid trivial infeasible instances, we assume throughout the paper that:

$$\max\left(Rank(J_i)(i = 1, 2, ...N)\right) \leq B \tag{2.1}$$

A *null job* has zero processing time with start time equal to finish time. Adding *null jobs* to an instance of *CCFJS* doesn't affect the feasibility of the problem since *null jobs* introduce no cost. A feasible schedule has the following fundamental property:

Partition Property: n of the total N jobs ($n<N$) can be processed by P processors ($P<B$) if and only if the remaining $N-n$ jobs can be processed by $B-P$ processors.

3 A Polynomial Time Algorithm for Two Classes of Processors

In this section, we show that when there are two classes of processors (i.e. K=2), *CCFJS* can be solved in polynomial time. The algorithm we give has two steps: (1) Build a flow network based on each job's start time and finish time along with the total number of processors; (2) Apply a minimum cost network flow algorithm [9] to obtain a minimum total cost for the constructed flow network. It will follow that if the minimum total cost is greater than C_B, then *CCFJS* has no feasible solution; otherwise *CCFJS* has a feasible solution.

3.1 Construct a Flow Network

The algorithm that we give constructs a flow network in the form of a layered directed acyclic graph (*DAG*). Each layer has exactly B vertices and each edge connects two vertices from two adjacent layers. Each vertex represents a job, and will be assigned a weight equal to the job's processing time. A path represents a sequence of jobs that can be processed by the same processor. Clearly, any two jobs on the same path must be compatible. Further, if any P vertex-disjoint paths ($0 < P < B$) are removed from the flow network, then in the remaining flow network, there will exist $B-P$ vertex-disjoint paths, such that each remaining vertex belongs to exactly one of the $B-P$ vertex-disjoint paths (i.e. the **Partition Property** holds**)**. In section 3.2, we will let P be equal to B_2, and the jobs on the resulting B_2 vertex-disjoint paths will be assigned to processors of class 2. Likewise, the jobs on the remaining B_1 vertex-disjoint paths will be assigned to the processors of class 1.

3.1.1 Building a Layered DAG

The algorithm ***Build_Layered_DAG*** described below has two inputs: the total number of processors B and an array of N jobs stored in $J[1..N]$. The output is a

layered DAG in which each layer contains exactly B vertices, and each edge is directed from a vertex in one layer to a vertex in the next higher layer.

Algorithm Build_Layered_DAG*(Input J[1..N], B; Output G=(V,E))*
Sort and store the 2N values of J[k].start_time and J[k].finish_time (k=1..N) into ascending order in A[1..2N];
V ← {s, t}; L ← 1; F ← 0;
weight(s) ←0; weight(t) ← 0; Layer[0] ←{s};
for i ← 1 to N+1 do Layer[i] ←NULL;
for i ← 1 to 2N do
j←A[i].index;
if A[i].type = start_time
Layer[L] ← Layer[L] ∪ vertex(J[j]); // Add job J[j] into Layer L
weight(vertex(J[j]))←J[j].proc_time; // Set job J[j]'s weight to its length
F ← 0;
else / A[i].type = finish_time */*
if (F =0)
F ← 1;
//Duplicate vertices of layer L into layer L+1
Layer[L+1] ← Layer[L];
//Set duplicated vertices' weight to zero
weight(v) ← 0 for each v ∈ Layer[L+1];
Add an adequate number of vertices with weight zero (null jobs) into layer L, such that laye L has exactly B number of vertices;
L←L+1; //Advance the current layer L to L+1
//Remove job J[j] from the current layer L
Layer[L] ← Layer[L] - vertex(J[j]);
Layer[L] ←{t};
for i ← 1 to L-1 do V←V ∪ Layer[i];
for every vertex v ∈ Layer[i]
if v has a duplicated vertex u ∈ Layer[i+1] then E ← E ∪ (v,u);
else for each non-duplicated vertex u ∈ Layer[i+1], E ← E ∪ (v,u);
Assign capacity one and weight zero to each edge;

The data structures used in the algorithm are: Each element of *J[1..N]* has three fields: *J[k].start_time* is job J_k's start time, *J[k].finish_time* is job J_k's finish time and *J[k].proc_time* is job J_k's processing time. Clearly, *J[k].start_time+ J[k].proc_time= J[k].finish_time.* Array $A[1..2N]$ stores the sorted start and finish times of all the jobs. Each element of A has three fields: $A[i]$.*type* (*start_time* or *finish_time*) indicates whether it is a start time or a finish time; $A[i]$.*value* is either the start or the finish time depending on the $A[i]$.*type*; $A[i]$.*index* is the job's index in $J[1..N]$. Each element of array *Layer*[*1..N+2*] is used to store the vertices of a given layer.

Build_Layered_DAG works as follows: The algorithm first sorts the *2N* time values into ascending order, and stores them in array *A*[1..2*N*]. If the finish time of a job is the same as the start time of other jobs, the finishing times are placed before the starting times in *A*[1..2*N*]. Then, the algorithm processes *A[i]* with index *i* increasing from 1 to 2*N*. If *A*[*i*].*type* is *start_time*, then a vertex representing job *J[A[i].index]* is added to layer *L*, and the weight of the vertex is set to *J[A[i].index].proc_time*. If *A*[*i*].*type* is *finish_time*, then each vertex of layer *L* is duplicated and placed in a new layer (*L*+1). The weight of each duplicated vertex in this new layer is set to zero, and the vertex of job *J[A[i].index]* is removed from the new layer. Note that if there are contiguous elements of *A*[*i*].*type* equal to *finish_time*, then the algorithm keeps on removing the vertex of job *J[A[i].index]* from the new layer until encountering the first *A*[*i*].*type* equal to *start_time*. (i.e. a new layer is constructed only for the first of a series of *finish_times*). Note that the number of vertices in the new layer is at most *B* from the assumption (2.1). If the old layer has fewer than *B* vertices, the algorithm adds enough vertices with weight zero into the old layer to ensure that it has exactly *B* vertices (vertices with weight zero represent *null jobs*).

After building the final layer, the algorithm adds edges: for each vertex *v* in layer *k* (not the final layer), if *v* has a duplicated vertex *u* in layer *k*+1, add edge (*v*,*u*); otherwise add edge (*v*,*u)* for each non-duplicated vertex *u* in the layer *k*+1. The running time of ***Build_Layered_DAG*** is $O(N \log N + NB^2)$.

3.1.2 An Example

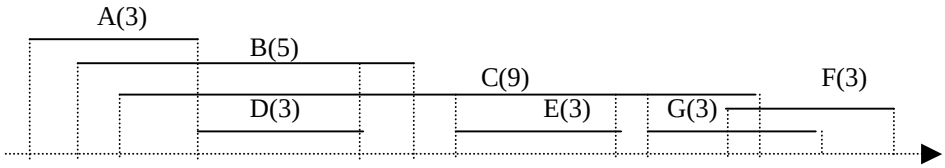

Fig. 1. Job Sequence Input for *Build_Layered_DAG*, *B*=3

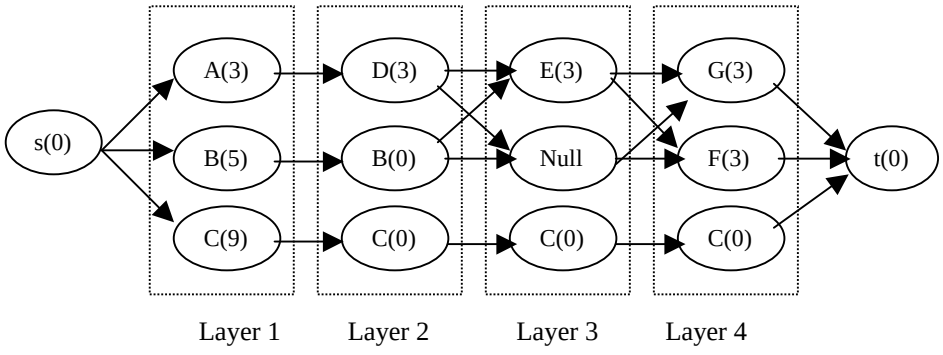

Fig. 2. Output Layered DAG from Fig. 1

Figure 1 shows seven jobs with start times and finish times in sorted order. Each job's processing time is shown in parentheses. Figure 2 is the layered DAG produced by ***Build_Layered_DAG***. Each vertex's weight is indicated in the parentheses.

3.1.3 Properties of the Layered DAG

In this section we prove the following theorem about the properties of the layered DAG. These properties are clearly illustrated in the example above.

Theorem (3.1) The graph $G = (V, E)$ generated by ***Build_Layered_DAG*** has the following properties: 1. $G = (V, E)$ is a DAG in which each layer (except the first and the last) has exactly *B* vertices, and any two vertices (except for the *null jobs* vertices) in that layer are not compatible. 2. Each vertex in a layer can have at most one duplicated vertex in the next layer. 3. Each edge connects two vertices from two adjacent layers. 4. Each vertex *u* is located on at least one path from *s* to *t* (an *s*-*t* path). 5. There exist exactly *B* vertex disjoint *s*-*t* paths in the graph, such that each vertex (except *s* and *t*) belongs to exactly one of those *B* paths. Note that the selection of those *B* paths may not be unique. 6. If any *P* vertex-disjoint paths ($0 < P < B$) are removed (except *s* and *t*) from $G = (V, E)$, then the remaining flow network has the above 1-5 properties with *B* replaced by $B - P$. 7. A set of *m* jobs can be processed by *P* processors subject to the **Partition Property** if and only if there exist *P* ($0 < P < B$) vertex-disjoint *s*-*t* paths in $G = (V, E)$ containing all non-duplicated and duplicated vertices of those *m* jobs. These *P* vertex-disjoint paths may not be unique.

The first 4 properties and property 6 follow easily from the algorithm. Property 5 can be proved by using induction on the number of layers constructed. In property 7, the "if" part is straightforward and the "only if" part can be proved by using induction on the number of processors *P*. The details of the proofs are omitted here due to space limitation.

3.2 The Algorithm and Its Correctness

Having constructed a flow network, we use a minimum cost network flow algorithm [9] to find B_2 vertex-disjoint paths with minimum total weight. Clearly, all vertices not on those B_2 vertex-disjoint paths can fit into $B - B_2 = B_1$ vertex-disjoint paths. In ***Two_Class_Scheduling*** described below, input *J[1..N]* is the same as the input for ***Build_Layered_DAG***, B_1 and B_2 are the numbers of processors of the two processor classes, and C_1 and C_2 are the unit time processing costs of those classes. We assume $C_1 < C_2$. The algorithm returns the minimum cost C_T^*. Note that $C_T^* \le C_B$ (C_B is the cost bound) decides the feasibility of the scheduling.

Algorithm Two_Class_Scheduling*(Input J[1..N],* B_1, B_2, C_1, C_2 *; Output* C_T^**)*

Build_Layered_DAG*(Input J[1..N],* $B_1 + B_2$ *; Output G'=(V',E'));*

Convert G'=(V',E') into an edge capacitated flow network G=(V,E) using the standard techniques (i.e. split each vertex $u \in V' - \{s,t\}$ *into two vertices u' and u" and add an directed edge (u',u") with direction the same as* $s \to t$ *direction, capacity equal to one and weight equal to vertex u's weight).*

Compute a minimum cost flow in G=(V,E) with flow value of B_2 *[9];*

Assign the jobs that are on the minimum cost flow to class 2 processors, and let L_2^* *be the total processing time of such jobs; Assign the jobs that are not on the minimum cost flow to class 1 processors and let* L_1^* *be the total processing time of such jobs.*

Calculate the total cost $C_T^* \leftarrow L_1^* * C_1 + L_2^* * C_2$.

The following theorem establishes the correctness of the algorithm:

Theorem (3.2): ***Two_Class_Scheduling*** correctly computes the minimum cost and the feasibility of *CCFJS* when there are only two classes of processors.

Proof: For given *N* jobs, let *L* be the total processing time. Given a scheduling, let L_1 be the total processing time on class 1 processors and L_2 be the total processing time on class 2 processors. Thus $L = L_1 + L_2$. Since $C_1 < C_2$, the cost $L_1 * C_1 + L_2 * C_2$ of processing all of the jobs is minimized if L_2 is minimized. From property 7 of theorem (3.1), L_2 is minimized if and only if all the jobs contributing to L_2 are on the minimum cost flow with flow value B_2. Thus, the theorem is established. ■

For the flow network *G=(V,E)* where all capacities are one, the running time of the best minimum cost flow algorithm is $O(|VE| + |V|^2 \log |V|)$ ([9]), which is also the running time of ***Two_Class_Scheduling***.

4 Complexity for More Than Two Classes of Processors

In this section, we show that *CCFJS* is not only *NP-Complete*, but also that there is no constant ratio approximation algorithm for *CCFJS* when the number of classes of processors is more than two.

Theorem (3.3): If $P \neq NP$ and $r \geq 1$(r is a constant), then there is no polynomial time approximation algorithm with ratio bound *r* for *CCFJS* when the number of classes of processors is at least 3.

Proof: The proof is by contradiction. Suppose that for some $r \geq 1$, there is a polynomial time approximation algorithm ***A*** for *CCFJS* with ratio bound *r*, i.e. $C_A / C_T^* \leq r$ where C_A is the cost returned by the algorithm ***A*** and C_T^* is the optimal solution. We will show how to use ***A*** to solve instances of *Numerical 3-Dimensional*

Matching (*N3DM*) in polynomial time. Since *N3DM* problem is *NP-Complete* [10], our theorem follows. Recall the definition of *N3DM* [10]:

INSTANCE of *N3DM*: Integers t, d and a_i, b_i, c_i for $i = 1, 2, ...t$, satisfying the following relations: $\sum_{i=1}^{t}(a_i + b_i + c_i) = td$ and $0 < a_i, b_i, c_i < d$ for $i = 1, 2, ...t$.

QUESTION: Are there permutations ρ and σ of $\{1, 2, ..., t\}$, such that: $a_i + b_{\rho(i)} + c_{\sigma(i)} = d \quad (i = 1, ..., t)$?

Consider a particular instance of *N3DM*. We construct an instance of *CCFJS* instance (inspired in part from [7]) as follows. Define

$$U = 49dt^4r^2 \tag{3.4}$$

$$V = U - 7dt^2r = 49dt^4r^2 - 7dt^2r \tag{3.5}$$

$$W = U + 7dt^2r + 3d = 49dt^4r^2 + 7dt^2r + 3d \tag{3.6}$$

$$Z = W + U + d = 98dt^4r^2 + 7dt^2r + 4d \tag{3.7}$$

Define K=3, $C_1 = \left(14dt^3 + 14dt^2 - 5dt/r + 5d/r\right)/Z$, $C_2 = 1/r$ and $C_3 = 7t^2$. It can be easily verified that $0 < C_1 < C_2 < C_3$. Define $C_B = 49dt^4$, $B_1 = t$, $B_2 = t^2 - t$ and $B_3 = t^2$. In this instance of *CCJFS*, the total number of processors is $B = B_1 + B_2 + B_3 = 2t^2$. Next, we choose $t^2 + 2t$ distinct rational numbers (see figure 3) E_i, F_j and $X_{i,j}$ with $i, j = 1, 2, ..., t$ such that:

$$U < F_j < U + d < E_i < U + 2d < X_{i,j} < U + 3d \tag{3.8}$$

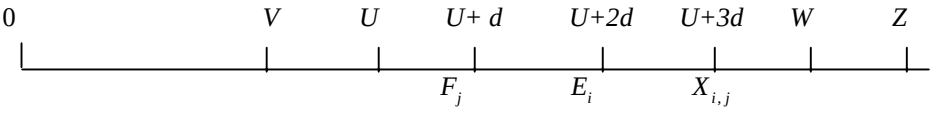

Fig. 3. Job instance construction relationship

Then, we define $N = 6t^2 + t$ jobs. We will identify these jobs by their start time and finish time pairs, rather then by separate names. In that context, these jobs are as follows:

$[0, E_i)$ $(i = 1, 2, ..., t)$

$(t-1)$ times $[V, F_j)$ $(j = 1, 2, ..., t)$

$[X_{i,j}, W + a_i + b_j)$ $(i, j = 1, 2, ..., t)$

$(t-1)$ times $[U, E_i)$ $(i = 1, 2, ..., t)$

$[X_{i,j}, U + 3d)$ $(i, j = 1, 2, ...t)$

$[E_i, X_{i,j})$ $(i, j = 1, 2, ..., t)$

$[F_j, X_{i,j})$ $(i, j = 1, 2, ..., t)$

$[W + d - c_k, Z)$ $(k = 1, 2, ..., t)$

$[U, F_j)$ $(j = 1, 2, ..., t)$

This completes the construction of the instance of *CCFJS*. Clearly, this construction requires polynomial time, and it is easy to verify that the instance satisfies (2.1). Assume algorithm ***A*** is applied to the above instance of *CCFJS*. We show that $C_A \le r * C_B$ if and only if the instance of *N3DM* has a solution, thus algorithm ***A*** can solve the instance of *N3DM* in polynomial time.

To show that the instance of *N3DM* has a solution when $C_A \le r * C_B$, we prove the following lemmas:

Lemma (3.9): Jobs $[0, E_i)$ $(i = 1, 2, ..., t)$ can only be assigned to class 1 processors.

Proof: Suppose a job $[0, E_i)$ is assigned to a non class 1 processor. Thus the unit time processing cost for this job is at least 1/*r*. From (3.8) and (3.4), the processing time of this job is $E_i > U + d > U = 49dt^4 r^2$. Thus $C_A > 49dt^4 r = r * C_B$, a contradiction. ■

Lemma (3.10): Jobs $(t-1)$ times $[V, F_j)$ $(j = 1, 2, ..., t)$ can only be assigned to class 2 processors.

Proof: Jobs $(t-1)$ times $[V, F_j)$ $(j = 1, 2, ..., t)$ can't be assigned to class 1 processors due to lemma (3.9). The rest of proof is similar to the proofs of lemma (3.9). ■

Lemma (3.11): Jobs $(t-1)$ times $[U, E_i)$ $(i = 1, 2, ..., t)$ and $[U, F_j)$ for $j = 1, 2, ..., t$ can only be assigned to class 3 processors.

Proof: Immediate result of lemmas (3.9) and (3.10). ■

Lemmas (3.12): Jobs $[W + d - c_k, Z)$ $(k = 1, 2, ..., t)$ can only be assigned to class 1 processors.

Proof: Similar to the proofs of lemma (3.9). ■

Lemma (3.13): Jobs $[X_{i,j}, W + a_i + b_j)$ $(i, j = 1, 2, ..., t)$ can't be assigned to class 3 processors, and there is no idle time for each of the class 1 and class 2 processors during $[U + 3d, W)$.

Proof: Similar to the proofs of lemma (3.9). ■

Lemma (3.14): Jobs $[X_{i,j}, U + 3d)$ for $i, j = 1, 2, ...t$ can only be assigned to class 3 processors.

Proof: Immediate result of lemma (3.13) ■

Lemma (3.15): There is no idle time for each of the $2t^2$ processors during time $[U, U + 3d)$.

Proof: Obvious from the definition of the jobs. ■

After proving the above lemmas, now we show that there is no idle time for each of the class 1 processors during time interval $[W, W + d]$ and the instance of *N3DM* has a solution. From lemmas (3.9), (3.12), (3.13) and (3.15), it follows that jobs $[0, E_i)$, $[E_i, X_{i,j})$, $[X_{i,j}, W + a_i + b_j)$ and $[W + d - c_k, Z)$ $(i, k = 1, 2, ..., t)$ must be

assigned to class 1 processors. Each *i* and *k* occur exactly once. From lemmas (3.10), (3.13) and (3.15), it follows that jobs $(t-1)$ times $[V, F_j)$, $[F_j, X_{i,j})$, $[X_{i,j}, W + a_i + b_j)$ must be assigned to class 2 processors, where each j $(j = 1, 2, ..., t)$ occurs exactly $(t-1)$ times. Thus from lemma (3.13), for the jobs $[X_{i,j}, W + a_i + b_j)$ assigned to class 1 processors, each j $(j = 1, 2, ..., t)$ occurs exactly once. From lemmas (3.11), (3.14) and (3.15), it follows that jobs $(t-1)$ times $[U, E_i)$, $[E_i, X_{i,j})$ and $[X_{i,j}, U + 3d)$ must be assigned to class 3 processors, where each i $(i = 1, 2, ..., t)$ occurs $(t-1)$ times, and that jobs $[U, F_j)$ and $[F_j, X_{i,j})$ $(j = 1, 2, ..., t)$ must be assigned to class 3 processors. Finally, from the fact that $\sum_{i=1}^{t} (a_i + b_i + c_i) = td$ and the conclusions that each *i*, *j* and *k* occurs exactly once for jobs $[X_{i,j}, W + a_i + b_j)$ and $[W + d - c_k, Z)$ assigned to class 1 processors, it follows that there is no idle time for each of the class 1 processors during time interval $[W, W + d]$. Thus $W + a_i + b_j = W + d - c_k$. If we define $\rho(i) = j$ and $\sigma(i) = k$, then $a_i + b_{\rho(i)} + c_{\sigma(i)} = d$ for $i = 1, 2, ..., t$ and the instance of *N3DM* has a solution.

Now suppose the instance of *N3DM* has a solution, the jobs assignment can follow the above proof (see figure 4). The total processing cost

$$C_A = tZ * \left(14dt^3 + 14dt^2 - 5dt / r + 5d / r\right) / Z + (t^2 - t) * (W + 2d - V) / r + t^2 * 3d * 7t^2$$

$$= 49dt^4 = C_B \le r * C_B$$

Thus algorithm ***A*** can solve *N3DM* in polynomial time, which contradicts the assumption that $P \neq NP$. ■

From theorem (3.3), we can easily prove the following corollary by letting *r*=1 and $C_A = C_T$:

Corollary (3.16) *CCFJS* is *NP-Complete*.

5 An Approximation Algorithm

In this section, we present an approximation algorithm based on network flows, derive its worst-case performance ratio (non constant), and show that it has a constant approximation ratio in some special cases.

Class 1 processors: $B_1 = t,\ C_1 = \left(14dt^3 + 14dt^2 - 5dt/r + 5d/r\right)/Z$

0	E_1	X_{11} $W + a_1 + b_1$	$W + d - c_3$	Z
0	E_2	X_{23} $W + a_2 + b_3$	$W + d - c_1$	Z
0	E_3	X_{32} $W + a_3 + b_2$	$W + d - c_2$	Z

Class 2 processors: $B_2 = t^2 - t,\ C_2 = 1/r$

0	V	F_1 X_{21} $W + a_2 + b_1$		Z
0	V	F_1 X_{31} $W + a_3 + b_1$		Z
0	V	F_2 X_{12} $W + a_1 + b_2$		Z
0	V	F_2 X_{22} $W + a_2 + b_2$		Z
0	V	F_3 X_{13} $W + a_1 + b_3$		Z
0	V	F_3 X_{33} $W + a_3 + b_3$		Z

Class 3 processors: $B_3 = t^2,\ C_3 = 7t^2$

0	U	E_1 X_{12}	$U + 3d$	Z
0	U	E_1 X_{13}	$U + 3d$	Z
0	U	E_2 X_{21}	$U + 3d$	Z
0	U	E_2 X_{22}	$U + 3d$	Z
0	U	E_3 X_{31}	$U + 3d$	Z
0	U	E_3 X_{33}	$U + 3d$	Z
0	U	F_1 X_{11}	$U + 3d$	Z
0	U	F_2 X_{32}	$U + 3d$	Z
0	U	F_3 X_{23}	$U + 3d$	Z

Fig. 4. An *CCFJS* Instance when t=3

5.1 The Algorithm *Approximate_Cost*

Recall in section 3 for the two classes of processors case, the minimum cost flow is computed with flow value equal to the number of processors of the most expensive class of processors. The jobs that are on the minimum cost flow are assigned to the processors of the expensive class, and the remaining jobs are assigned to the processors of the cheap class. Adapting this method to three or more classes of processors, in the algorithm ***Approximate_Cost***, we consider partitioning the classes of processors into an expensive set and a cheap set. We then compute the minimum

cost flow with flow value equal to the sum of the number of processorss of the expensive set of processors. Each job that is on that minimum cost flow will be assigned to one of the processors in the expensive set, and each job that is not on that minimum cost flow will be assigned to one of the processors in the cheap set. In order to assign each job in the two sets to a particular class of processor, we use a greedy approach: specifically, we compute the minimum cost flow with flow value equal to the number of processors of the most expensive class and assign the jobs that are on that minimum cost flow to processors of that most expensive class. We then remove those jobs and the processors of that class and iterate until every job is assigned to a processor of a particular class. Since we don't know in advance how to partition the processors into an expensive set and a cheap set, we perform the above computation for each possible partition (for *K* classes of processors, there are *K*-1 partitions), and retain the partition and the associated assignment of jobs to the processors that yield the smallest cost.

5.2 Algorithm Complexity and Performance Ratio

Similarly to section 3.2, the running time of ***Approximate_Cost*** is $O(K^2 |VE| + K^2 |V|^2 \log |V|)$. Note that we are not partitioning the expensive set and the cheap set recursively (instead, we use the greedy approach described above). Recursive partitioning will lead to exponential complexity in terms of *K*. Before we analyze the performance ratio of the algorithm, we first provide the following theorem and corollary (recall that we assume $0 < C_1 < C_2 < ... < C_K$):

Theorem (3.17) In *Approximate_Cost*, for $i = 1, ..., K$, let X_i be the total processing time assigned to class *i* processors, let y_i be the minimum total processing time assigned to a single class *i* processor, and let Y_i be the maximum total processing time assigned to a single class *i* processor. Then

$$Y_1 \geq X_1 / B_1 \geq y_1 \geq Y_2 \geq X_2 / B_2 \geq y_2 ... \geq Y_K \geq X_K / B_K \geq y_K \qquad (3.18)$$

Corollary (3.19) Let $X = \sum_{i=1}^{K} X_i$ be the total processing time, and *approx_cost* be the cost returned. Then $approx\_cost = \sum_{i=1}^{K} C_i X_i \leq X \sum_{i=1}^{K} C_i B_i / \sum_{i=1}^{K} B_i$ (3.20)

In ***Approximate_Cost***, a minimum cost network flow algorithm is applied between each pair $(i, i+1)(i < K)$ classes of processors, such that the flow value includes B_{i+1} without B_i. Thus theorem (3.17) can be proved by using minimum cost network flow property. Corollary (3.19) can be proved by showing that when $X_1 / B_1 = X_2 / B_2 = ... = X_K / B_K$, *approx_cost* reaches its upper bound $X \sum_{i=1}^{K} C_i B_i / \sum_{i=1}^{K} B_i$.

Consider a particular partition in ***Approximate_Cost*** where the cheap set includes processors from class 1 to class j (j=1…K-1) and the expensive set includes processors from class j+1 to K. After the minimum cost flow is computed, let $X_{1..j}$ be the total processing times and c_{j_1} be the cost of jobs that are assigned to the cheap set, and let $X_{j+1..K}$ be the total processing times and c_{j_2} be the cost of jobs that are assigned to the expensive set. Let c_j be the final cost as calculated in that partition. Analogous to corollary (3.19), we have:

$$\begin{aligned} &c_{j\_1} \le X_{1..j} \sum_{i=1}^{j} C_i B_i / \sum_{i=1}^{j} B_i, \quad c_{j\_2} \le X_{j+1..K} \sum_{i=j+1}^{K} C_i B_i / \sum_{i=j+1}^{K} B_i, \\ &c_j = c_{j\_1} + c_{j\_2} \le X_{1..j} \sum_{i=1}^{j} C_i B_i / \sum_{i=1}^{j} B_i + X_{j+1..K} \sum_{i=j+1}^{K} C_i B_i / \sum_{i=j+1}^{K} B_i \end{aligned} \tag{3.21}$$

Let c_{opt} be the optimal processing cost. Since $0 < C_1 < ... < C_K$, the optimal value c_{opt} will not decrease if $C_2, ...C_j$ decreases to C_1 and $C_{j+2}, ...C_K$ decrease to C_{j+1}. Thus,

$$c_{opt} \ge X_{1..j} C_1 + X_{j+1..K} C_{j+1} \tag{3.22}$$

$$\frac{c_j}{c_{opt}} \le \frac{X_{1..j} \dfrac{\sum_{i=1}^{j} C_i B_i}{\sum_{i=1}^{j} B_i} + X_{j+1..K} \dfrac{\sum_{i=j+1}^{K} C_i B_i}{\sum_{i=j+1}^{K} B_i}}{X_{1..j} C_1 + X_{j+1..K} C_{j+1}} \le \max\left(\frac{\sum_{i=1}^{j} C_i B_i}{C_1 \sum_{i=1}^{j} B_i}, \frac{\sum_{i=j+1}^{K} C_i B_i}{C_{j+1} \sum_{i=j+1}^{K} B_i} \right) \tag{3.23}$$

Since the algorithm returns the smallest cost for each j,

$$\rho = \min\{c_j \mid j = 1, 2, ..., K-1\} / c_{opt} \tag{3.24}$$

Finally from (3.23) and (3.24),

$$\rho \le \min\left(\max\left(\frac{\sum_{i=1}^{j} C_i B_i}{C_1 \sum_{i=1}^{j} B_i}, \frac{\sum_{i=j+1}^{K} C_i B_i}{C_{j+1} \sum_{i=j+1}^{K} B_i} \right), j = 1, 2, ..., K-1 \right) \tag{3.25}$$

Example $K = 3, C_1 = 3, B_1 = 100, C_2 = 6, B_2 = 50, C_3 = 12, B_3 = 25$.

By applying (3.25), we have $\rho \le 4/3$. In practical situations (for example in the off-line multimedia gateway call routing), the more expensive the processors, the less

the number of processors. In this example, $C_1B_1 = C_2B_2 = C_3B_3$. In general, if K>2 is a constant and $C_1B_1 = C_2B_2 = ... = C_KB_K$, then from (3.25), $\rho \le K/2$, i.e. ρ is bounded by a constant.

Example $K = 4, C_1 = 1, B_1 = 100, C_2 = 2, B_2 = 100, C_3 = 4, B_3 = 100, C_4 = 8, B_4 = 100$

By applying (3.25), we have $\rho \le 3/2$. In this example, $B_1 = B_2 = B_3 = B_4$, $C_{i+1}/C_i = 2$ $(i = 1, 2, 3)$. In general, if K>2 is a constant, $B_1 = B_2 = ... = B_K$, and $C_{i+1}/C_i = q$ $(i = 1, ..., K-1)$ is a constant, then from (3.25), $\rho \le 2\sum_{j=0}^{K/2-1} q^j / K$, i.e. ρ is also bounded by a constant.

6 Summary

In this paper, we have studied the problem of *CCFJS* and we present a complete classification of its computational complexity. We show that *CCFJS* is polynomial solvable when there are only two classes of processors. We prove that the general *CCFJS* is *NP-Complete* and that there is no constant ratio approximation algorithm. We further present an approximation algorithm and analyze its worse case performance ratio.

References

[1] Matteo Fischetti, Silvano Martello, Paolo Toth, *The Fixed Job Schedule Problem with Spread-Time Constraints*, Operations Research. 35(6), 849-858, 1987.

[2] Matteo Fischetti, Silvano Martello, Paolo Toth, *The Fixed Job Schedule Problem with Working-Time Constraints*, Operations Research. 37(3), 395-403, 1989.

[3] Antoon W.J. Kolen, Leo G. Kroon, *On the Computational Complexity of (Maximum) Class Scheduling*, European Journal of Operational Research, 54, 23-38, 1991.

[4] Antoon W.J. Kolen, Leo G. Kroon, *License Class Design: Complexity and Algorithms*, European Journal of Operational Research, 63, 432-444, 1992

[5] Antoon W.J. Kolen, Leo G. Kroon, *On the Computational Complexity of (Maximum) Shift Class Scheduling*, European Journal of Operational Research, 64, 138-151, 1993.

[6] Antoon W.J. Kolen, Leo G. Kroon, *An Analysis of Shift Class Design Problems*, European Journal of Operational Research, 79, 417-430, 1994.

[7] Leo G. Kroon, Arunabha Sen, Haiyong Deng, Asim Roy, *The optimal cost chromatic partition problem for trees and interval graphs*, Graph Theoretical Concepts in Computer Science, LNCS, vol. 1197, Springer-Verlag, New York/Berlin, 1996.

[8] Klaus Jansen, *Approximation Results for the Optimal Cost Chromatic Partition Problem*, Journal of Algorithms, 34, 54-89, 2000.

[9] Ravindra K.Ahuja, Thomas L.Magnanti, James B.Orlin, *Network Flows.* Prentice Hall, 1993

[10] Michael R.Garey, David S.Johnson, *Computer and Intractability, A Guide to the Theory of NP-Completeness*. Twenty-second printing, 2000.