Problema de Escalonamento de Funcionário Preguiçoso

Edith Zaida Sonco-Mamani

16 de dezembro de 2009

1 Introdução

Foi introduzido uma nova classe de problemas de programação de tarefas, introduzido por Arkin [1] , neste caso tomamos um novo olhar sobre o problema a partir de ponto de vista dos trabalhadores que executam as tarefas. Na verdade, é natural esperar que alguns trabalhadores podem não ter a motivação para executar seu trabalho em seus níveis máximos de eficiência em quanto a resolução de tarefas, talvez seja porque eles não têm participação nos lucros da empresa, ou porque simplesmente são preguiçosos.
O exemplo a seguir ilustra a situação de um "típico trabalhador de escritório ", que pode ser uma pequena peça de uma grande burocracia:

Exemplo São as 3:00 p.m., e Dilbert vai para casa às 5:00 p.m. Dilbert tem duas tarefas que foram dadas a ele: um requer 10 min, o outro requer uma hora. Se existe uma tarefa em seu "in-box ", Dilbert deve trabalhar sobre ela, ou tem risco de ser demitido. No entanto, se ele tem várias tarefas, Dilbert tem a liberdade de escolher qual fazer primeiro. Ele também sabe que às 3:15, outra tarefa vai aparecer - uma reunião pessoal de 45 minutos. Se Dilbert começa primeiro a tarefa de 10 minutos, ele estará livre para participar da reunião de pessoal às 3:15 e depois trabalhar sobre a tarefa de uma hora de 4:00 até 5:00. Por outro lado, se Dilbert forma parte da tarefa de uma hora às 3:15, ele pode ser dispensado da reunião. Depois de terminar o trabalho de 10-minutos de 4:10, ele terá 50 min para mexer os polegares, passar ferro a sua gravata, ou desfrutar de outras trivialidades irracionais. Naturalmente, Dilbert prefere esta segunda opção.
Neste problema, o funcionario(trabalhador-maquina) quer fazer coisas tão pouco (ou fácil) como possível, o que é o objetivo de reverter o problema de escalonamento de taferas clássica. Claro que há um pressuposto básico, o chamado requisito ocupado, que o funcionario deve continuar trabalhando enquanto tenha tarefas para executar, caso contrário o problema se tornaria trivial e a estratégia ideal para o funcionario seria apenas ficar ocioso, sem fazer nada.

Tais um problema de programação de taferas do funcionário preguiçoso é indicado como CD-LBSP. Mais precisamente, existe um conjunto de tarefas J_1,. . . , J_n. Onde a tarefa J_i, para i=1 até n, chega no tempo a_i ,o tempo de processamento da tarefa é t_i. Cada tarefa está associada a um peso. Existe um prazo comum D para todas as tarefas. A tarefa J_i pode ser executado apenas

se o seu tempo de partida é entre a_i e $D - t_i$. Em outras palavras, uma tarefa só pode ser iniciada durante ou após a sua hora de liberação, e uma vez iniciado, não pode ser interrompido e devem ser concluídos no prazo comum D. Em qualquer momento uma tarefa é chamada *executável* se ela pode ser executada. Neste trabalho, nós nos concentramos no problema LBSP(Lazy Bureaucrat Scheduling Problem) e nas quatro funções objectivo:

- min-time-spent: Minimizar a quantidade total de tempo gasto em trabalhar.
- min-weighted-sum: minimizar a soma ponderada das tarefas concluídas.
- min-makespan: Minimizar o tempo máximo de conclusão das tarefas.
- min-number-jobs: Minimizar o número total de tarefas concluídas.

2 Cronograma

Apresentamos um cronograma das atividades a serem realizadas no seguinte projeto:

Cronograma de Atividades		
Data	Etapa	Descrição
13 de Novembro	Etapa 1	O modelo
		LBP : Sem preempção
		Resultados
		Algoritmos para casos especiais.
20 de Novembro	Etapa 2	LBP : Com preempção.
		Minimizando o tempo total de tarefas
		Minimizando a soma ponderada das tarefas concluídas
		Minimizando o makespan
30 de Novembro	Etapa 3	Lemas.
		Algoritmo SJF
07 de Dezembro	Etapa 4	PTAS para minimizar o makespan.
		PTAS para minimizar o tempo gasto
		Tempo de liberação comum: NP
15 de Dezembro	Projeto final	

2.1 O Modelo

Consideramos um conjunto de tarefas 1,...,n com tempo de processamento J_1,...,J_n respectivamente. A tarefa i chega no tempo a_i e tem o tempo limite(deadline) d_i. Assumimos que todo J_i, a_i e d_i são valores não negativos e enteiros. As tarefas têm prazos rígidos, o que significa que cada tarefa só pode ser executada durante o intervalo permitido $I_i = [a_i, d_i]$. É indicado c_i=d_i-t_i como o tempo crítico do que a tarefa i, a tarefa i deve ser iniciada no momento c_i, de outro jeito,

não vai ter alguma chance de ser completada a tempo.
As tarefas são executadas em um único processador, o funcionario (Preguiçoso). O funcionario executa apenas uma tarefa cada vez.

2.1.1 Busy requirement.

O funcionario escolhe um subconjunto de tarefas a executar. Desde que seu objetivo é minimizar o seu esforço, ele prefere permanecer ocioso o tempo todo e deixar todos as tarefas não executadas. No entanto, este cenário é proibido por lo que nós chamamos requisitos gulosos, que exige que o funcionario trabalhe em uma tarefa executável, se existe alguma tarefa executável. Uma tarefa é "executável"se ela chegou, o prazo ainda não passou, e ela ainda não está totalmente processada. No caso com preempção, pode haver outras restrições que governam se uma tarefa é ou não é uma tarefa executável: Vamos falar sobre isso depois.

2.1.2 Funções Objetivo:

Em problemas de escalonamento tradicional, se é imposible completar o conjunto de todas as tarefas até os seus prazos, tipicamente tenta-se a otimizar de acordo com alguns objetivos, como por exemplo, maximizar a soma ponderada das tarefas feitas no tempo certo, minimizar o atraso máximo das tarefas, ou minimizar o número de tarefas atrasadas. Para o problema de LBP , consideramos quatro diferentes funções objetivo, que surgem naturalmente de considerar o objetivo de ser funcionario ineficiente:

1. minimizar a quantidade total de tempo gasto trabalhando. Este objetivo, naturalmente, apela a um "funcionario preguiçoso ".

2. minimizar a soma ponderada das tarefas concluídas. Aqui, nós geralmente assumimos que o peso da tarefa i é o seu comprimento, t_i, no entanto, outros pesos também são de interesse. Este objetivo pede para o funcionário "malicioso ", cujo objetivo é minimizar as taxas que a empresa recolhe sobre a base de seu trabalho, supondo que a taxa (na proporção da duração da tarefa, ou uma taxa fixa por tarefa), é recolhido apenas para as tarefas que são realmente concluídas.

3. Minimizar o makespan, o tempo máximo de conclusão das tarefas. Este objetivo apela a um burocata impaciente, cujo objetivo é ir para casa o mais cedo possível, após a conclusão da última tarefa, ele é capaz de completar. Ele se preocupa com o número de horas passadas no escritório. Note-se que, em contraste com os problemas de planejamento standard, o makespan no LBP muda, é uma função das tarefas que passaram os seus prazos e não pode mais ser executadas.(observe a seguinte tabela)

4. Minimizar o número total de trabalhos completados.

Tabela 1: Resumo de resultados sem interrupções; ver secção 3. Arkin [1]

Instância	Objetivo	Complexidade
Tarefas de comprimento 1:$t_1 = t_2 = ... = t_n = 1$	1	Tempo polinomial
Tarefas de intervalos curtos:$\forall i d_i - a_i < 2t_i$	1-3	Tempo pseudo-polinomial.
Ratios $R = O(1) e \Delta = O(1)$	1-3	Tempo pseudo-polinomial.
Mesmo tempo de chegada:$a_1 = a_2 = ... = a_n$	1-3	Tempo pseudo-polinomial. (Débilmente)NP-completos Difícil para aproximar
Tarefas de tamanho de ratio	1-2	Fortemente NP-completos Difícil para aproximar

2.1.3 Parâmetros adicionais do modelo

Tal como acontece com a maioria dos problemas de planejamento, debem ser definidos parâmetros adicionais do modelo. Por exemplo, deve-se explicitamente permitir ou proibir preempção nas tarefas. Se uma tarefa é preemptiva, é interrompido e pode ser retomado mais tarde, sem nenhum custo adicional. Se preempção é proibida, então uma vez que uma tarefa é iniciado, ele deve ser completada sem interrupções.

Também é preciso especificar se o planejamento ocorre on-line ou off-line. Um algoritmo de planejamento é considerada off-line, se todas as tarefas são conhecidas para o programador em primeiro lugar, e é on-line se as tarefas são conhecidas por o programador apenas que elas chegam. Neste trabalho, vamos limitar em programação off-line; (ver tabela 2).

Tabela 2: Resumo de resultados com interrupções; ver secção 4. Arkin [1]

Preempção	Objetivo	Complexidade
I	1-3	Tempo polinomial
II	1-3	(Débilmente)NP-completos, mesmo quando $a_1 = a_2 = ... = a_n$.
III	1-3	(Débilmente)NP-completos, mesmo quando $a_1 = a_2 = ... = a_n$. Difícil para aproximar

2.2 Resultados da literatura

Neste trabalho, é apresentado o problema do Funcionário Preguiçoso (LBP) e são desenvolvidos algoritmos e resultados sólidos para várias versões do LBP. A partir desses resultados, podemos tirar algumas características gerais dessa nova classe de problemas de escalonamento e descrever: (1) situações em que os algoritmos de escalonamento tradicionais estendem ao LBP e (2) situações em que estes algoritmos não se aplicam mais.

2.2.1 Sem preempção

É provado que o LBP é NP-completo, como é frequentemente no caso de problemas de escalonamento tradicional. Assim, nos concentramos em casos especiais para estudar algoritmos exatos. Quando todas as tarefas têm o tamanho de uma unidade de tempo, escalonamento óptimos podem ser encontrados em tempo polinomial. Os seguintes três casos têm algoritmos pseudo-polinomiais: (1), quando cada intervalo I_i da tarefa i é menor que o dobro do tempo de processamento da tarefa i; (2) quando as taxas do comprimento do intervalo para o comprimento da tarefa e a tarefa mayor para tarefa menor são simultaneamente delimitadas, e (3) quando todas as tarefas chegam ao sistema, ao mesmo tempo. Estes últimos problemas de escalonamento são resolvidos usando programação dinâmica tanto para Funcionário Preguiçoso e métricas tradicionais. Assim, nessas definições, as métricas do Funcionário Preguiçoso e métricas tradicionais são resolvidas usando técnicas similares.

Do ponto de vista de aproximação, no entanto, o padrão e as métricas do Funcionário Preguiçoso se comportar de maneira diferente. Métricas padrão geralmente permitem algoritmos de tempo polinomial tendo boas taxas de aproximação. entretanto é mostrados que as métricas do Funcionário Preguiçoso são difíceis de serem aproximados. Esta dificultade deriva mais do requerimento de ficar ocupado e menos da métrica particular em questão, que é o requerimento de ficar ocupado aparece para tornar o problema muito mais difícil.(Ironicamente, mesmo em problemas de otimização padrão, a gestão, muitas vezes impõe essa exigência, porque intuitivamente parece desejável.)

2.2.2 Preempção

A exigência de ficar ocupado determina que o trabalhador deve ficar ocupado enquanto o trabalho está no sistema. Se o modelo permite a preempção é preciso especificar em que condições uma tarefa pode ser interrompida ou reiniciada. Distinguimos três versões das regras de interrupção, as quais são listadas da mais permissiva a mais restritiva. Em particular, os trabalhadores são obrigados a executar as seguintes tarefas: (I) toda tarefa que tem chegado e está antes do seu *deadline*,
(II) qualquer tarefa que tem chegado e ainda existe tempo para concluí-lo antes do seu *deadline*, ou
(III) qualquer tarefa que tem chegado, mas com a restrição de que se ela for iniciada, deve ser concluída.
Consideramos que todas as três métricas e todas as três versões de interrupção. Mostramos que para as três métricas, versão I é polinomial solucionável, e a versão III é NP-completo. Muitos dos resultados difíceis para sem preempção transitam para a versão III.

Os principais resultados desta literatura são para a versão II. É mostrado que o problema geral é NP-completo. Então, os resultados estão concentramos em minimizar o makespan em dois casos

especiais complementares:
(1) Todos as tarefas têm um tempo de chegada comum e prazos arbitrários. (2) Todos as tarefas têm um prazo comum(deadline) e tempos arbitrários de chegada. É mostrado que o primeiro problema é NP-completo, enquanto que o segundo problema pode ser resolvido em tempo polinomial.

Estes últimos resultados ilustram uma característica curiosa da LBP. Um pode se converter um caso especial para o outro, invertendo a direção do tempo. No LBP, ao contrário de muitas definições de escalonamento, esta inversão do tempo muda a complexidade do problema (ver tabela. 3).

Tabela 3: Ilustração que a reversão de tempo altera a complexidade, veja Secção 4.3.. Arkin [1]

Instância	Preemção	Objetivo	Complexidade
Prazos arbitrários Idênticos tempos de chegada $a_1 = a_2 = ... = a_n$	II	3	(Débilmente)NP-completo
Arbitrarios tempos de chegada Idênticos *deadlines* $d_1 = d_2 = ... = d_n = D$	II	1-3	Tempo polinomial .

3 LBP: Sem preempção

Nesta seção, assumimos que nenhuma tarefa pode ser interrompida: se uma tarefa é iniciada, então é realizada sem interrupção até sua conclusão. É mostrado que o problema do funcionario preguiçoso(LBP) sem interrupção é fortemente NP-completo e não é approximable dentro de qualquer fator. Estes fortes resultados distingue o nosso problema de métricas tradicionais de planejamento, que podem ser aproximadas em tempo polinomial. Vamos mostrar, que vários casos especiais do problema têm pseudo-algoritmos de tempo polinomial, utilizando aplicações de programação dinâmica.

3.1 Resultados difíceis

Começamos por descrever o relacionamento entre as diferentes funções objetivo no caso de planejamento sem interrupção. O problema de otimizar o trabalho total (função objetivo 1) é um caso especial do problema de minimizar a soma ponderada das tarefas concluídas (função objetivo 2), porque cada tarefa que é executada deve ser concluída. (Os pesos tornam-se os comprimentos das tarefas). Além disso, se todos as tarefas têm a mesma hora de chegada, dicimos que é o tempo zero, então os dois objetivos, minimizando o trabalho total e minimizando o makespan (ir para casa mais cedo) são equivalentes (funções objetivo 1 e 3), desde que menhum escalonamento viável tenha uma lacuna. O primeiro teorema se aplica, portanto, a todas as três primeiras funções objetivo:

Teorema 3.1 *O problema do funcionário preguiçoso sem interrupção é (fracamente) NP-completo*

para as funções objetivo (1)-(3), e não é approximable dentro de qualquer fator fixo, mesmo quando os tempos de chegada sejam todos iguais.

Proof Para a prova do Teorema 1 é usado uma redução do problema de SUBSET SUM [4]: Dado um conjunto de inteiros $S = x_1, x_2, ..., x_n$. e um inteiro objetivo T, deve de existir um subconjunto $S\prime \subseteq S$ tal que $\Sigma_{x_i \epsilon S\prime} x_i = T$ ∎

É construido uma instância do LBP com $n+1$ tarefas, cada um com tempo de *release* de zero ($a_i = 0$ para todo i). Para $i = 1, ..., n$, a tarefa i tem tempo de processamento $t_i = x_i$ e *deadline* d_i= T. A tarefa $n+1$ tem tempo de processamento $t_{n+1} = 1 + \Sigma_{xi \epsilon S\prime} x_i$ e *deadline* $d_{n+1} = T + t_{n+1} - 1$, assim, a tarefa $n+1$ pode ser iniciado no momento $T-1$ ou mais cedo. Como a tarefa $n+1$ é muito longa, o funcionário quer evitar executá-la, mas pode fazê-lo se e somente se, ele seleciona um subconjunto de tarefas desde $1, ..., n$ para executar cuja soma de comprimentos seja exatamente T. Em resumo, a maior tarefa $n+1$ é executada se e somente se o problema do subconjunto for resolvido exatamente e executando a tarefa maior leva a um escalonamento cujo makespan (isto é, o trabalho total executado) não está dentro de qualquer elemento fixo da solução ótima.
Mostramos agora que o LBP sem preempção é fortemente NP-completo. Como mostraremos na Seção 3.2, o LBP do Teorema 1, quando todos os tempos de chegada são iguais, tem um algoritmo de tempo pseudo-polinomial. No entanto, se os tempos de chegada e os *deadlines* são inteiros arbitrários, o problema torna-se fortemente NP-completo. A redução se aplica a todas as primeiras três funções objetivo.

Teorema 3.2 *O problema do funcionário preguiçoso sem interrupção é fortemente NP-completo para as funções objetivo (1)-(3), e não é approximable dentro de qualquer fator fixo.*

Proof Claramente o problema está em NP, desde que nemhuma solução pode ser representada por uma lista ordenada de tarefas,dadas seus tempos de chegada. Para mostrar a dificultada, vamos usar uma redução do problema de 3-partição [4]:Dado um conjunto $S = x_1, ..., x_{3m}$ de $3m$ positivos inteiros e um número inteiro positivo limite B tal que $B/4 < x_i < B/2$, para $i = 1, ..., 3m$ e $\Sigma_{x_i} = mB$, existe uma partição de S em m conjuntos disjuntos $S1, ..., S_m$, tal que para $i = 1, ..., m$, $\Sigma_{x_j \epsilon S_i} x_j = B$? (Note que, pela suposição de que $B/4 < x_i < B/2$, cada conjunto S_i deve conter exatamente três elementos.). ∎

A função objetivo (1) é um caso especial da função objetivo (2), porque sem preempção, qualquer tarefa que é iniciada deve ser concluída. Além disso, as instâncias difíceis serão projetados de modo que não existam lacunas, garantindo que a solução ideal para a função objetivo (1) também é a solução ideal para a função objetivo (3).
É construida uma instância do LBP contendo três classes de trabalhos:

- Tarefas elemento- é definio tarefa "elemento "correspondente a cada elemento $x_i \epsilon S$, com tempo de chegada 0, *deadline* $d_i = (m-1) + mB$, e tempo de processamento x_i.

- Tarefas de unidade - é definido $m-1$ tarefas de uma unidade de tempo, cada um de comprimento 1. A tarefa da unidade i (para $i = 1, ..., m-1$) tem tempo de chegada $i(B+1)-1$ e prazo $i(B+1)$. Note que para essas tarefas de compimento de unidade temos $d_j - a_j = 1$; assim, estas tarefas devem ser tratados imediatamente após a sua chegada, ou não.

- Tarefa grande - É definida uma tarefa "grande "de comprimento $L > (m-1)+mB$, tempo de chegada 0, e *deadline* de $L + (m-2) + mB$. Note que, a fim de concluir este trabalho, ele deve ser iniciado no tempo de $(m-2) + mB$ ou antes.

Tal como na prova do Teorema 3.1, o funcionário preguiçoso quer evitar de executar uma tarefa longa, mas pode fazê-lo, se e somente se todos os outras tarefas estão realmente executadas. Caso contrário, haverá um momento em que o trabalho grande será a única tarefa no sistema eo funcionário preguiçoso será obrigado a executá-la. Assim, as tarefas de unidade devem ser feitas imediatamente após a sua chegada, e as tarefas elemento devem caber nos intervalos entre as tarefas de unidade. Cada intervalo entre tarefas consecutivas de unidade se o comprimento é exatamente B.(observe figura 1). Em resumo, a tarefa grande não é processada se e somente se todas as tarefas elemento e de unidade podem ser processadas antes de seus prazos, o que acontece se e somente se a instância correspondente de 3-partição é uma instância "sim."

Note que desde que L pode ser tão grande como nós queremos, isso também implica que nenhum algoritmo de aproximação de tempo polinomial com qualquer aproximação fixada limite pode existir, a menos que $P = NP$.

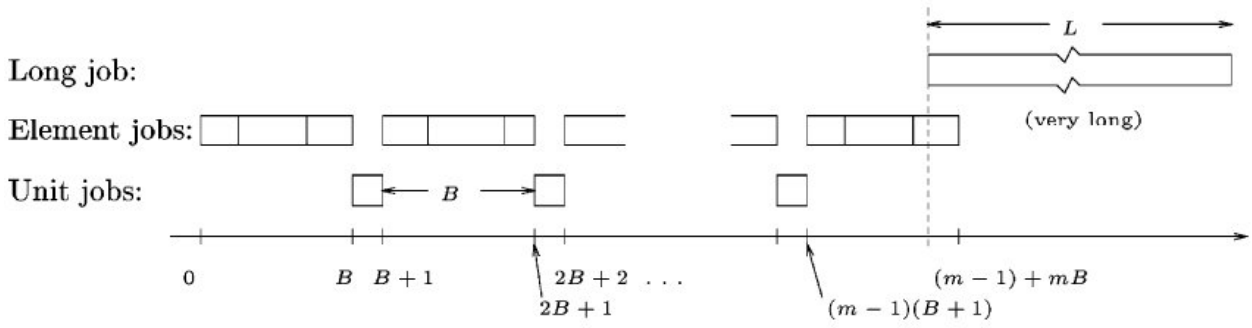

Figura 1: A prova da dificultade do LBP sem preempção e tempos de chegada arbitrários

3.2 Algoritmos para casos especiais

3.2.1 Tarefas de comprimento de unidade

Considere o caso especial do LBP em que todas as tarefas têm tempos de processamento de unidade. (todas as entradas são assumidas como inteiros.) A política do escalonamento última data de vencimento primeiro (LDD) seleciona a tarefa no sistema com o último prazo. Note que esta política fica em no preempção para tarefas de comprimento de uma unidade, uma vez que todas as tarefas têm tempos de chegada inteiros.

Teorema 3.3 *Considere a política de escalonamento último deadline primeiro (LDF - latest deadline first) quando as tarefas têm comprimento de uma unidade e todas as entradas são inteiras. O LDD minimiza a quantidade de trabalho executado.*

Proof Suponha por contradição que nenhum escalonamento ideal é LDD. É usado um argumento de câmbio. Considere um escalonamento ótimo (não-ideal LDD) que tem o menor número de pares de tarefas executadas em ordem não-LDD.
O escalonamento deve ter duas tarefas vizinhas i, j tais que $i < j$ no escalonamento, mas $D_i < D_j$, e j está no sistema quando i inicia sua execução. Considere o primeiro par de tais trabalhos. Há dois casos:
(1) O novo escalonamento com i e j comutada, é viável. Ele não executa nenhum trabalho mas que o escalonamento ótimo, e por isso é também ideal.
(2) O escalonamento com i e j comutada não é viável. Isso acontece se a data límite de i já passou. Se nenhuma tarefa está no sistema para substituir i, então, obter um melhor escalonamento que o escalonamento ótimo e chegamos a uma contradição. Caso contrário, nós substituímos i com a outra tarefa e repetimos o processo de mudança. ∎

Obtemos um escalonamento executando não mais trabalho do que uma programação ideal, mas com o menor número de pares de tarefas em ordem não-LDD, uma contradição.

3.2.2 Janelas estreitas

Considere agora a versão em que as tarefas são grandes em comparação com os intervalos, ou seja, os intervalos são "estreitos ". Seja R um comprimento limite da razão do comprimento da janela para o comprimento da tarefa, ou seja, para cada tarefa i, d_i-$a_i < R.t_i$. É mostrado que um algoritmo pseudo-polinomial existe para o caso das janelas suficientemente estreitas, ou seja, quando R ≤ 2.

Lema 3.4 *Suponha que, para cada tarefa i, d_i-$a_i < 2.t_i$. Então, se a tarefa i pode ser escalonada antes da tarefa j, então a tarefa j não pode ser escalonada antes da tarefa i.*

Proof Reescrevemos a hipótese: para cada i, $d_i - t_i < t_i + a_i$. O fato que a tarefa i pode ser agendada antes da tarefa j é equivalente a afirmar que $a_i + t_i \leq d_j - t_j$, desde o mais cedo que a

tarefa i pode ser concluído é o tempo $a_i + t_i$ e o mais tarde que a tarefa j pode ser iniciado é em tempo de $d_j - t_j$. Combinando essas desigualdades, obtemos :

$$a_j + t_j > d_j - t_j \geq a_i + t_i > d_i - t_i,$$

o que implica que a tarefa j não pode ser escalonada antes da tarefa i. ∎

Corolário 3.5 *Partindo do pressuposto de que $d_i - a_i < 2.t_i$ para cada i, a ordenação de qualquer subconjunto de tarefasem un escalonamento é unicamente determinada.*

Teorema 3.6 *Suponha que para cada tarefa i, d_i-$a_i < 2.t_i$. Seja $K = max_i d_i$. Então, o LBP pode ser resolvido no tempo $O(nKmax(n, K))$.*
Observe que, para $R > 2$ não conhecemos nenhum algoritmo eficiente sem condições adicionais. Deixe W ser un limite da relação da janela mais longa para a janela menor, e deixe Δ ser um limite da relação da tarefa maior para a tarefa menor. Note que o limite de R e Δ implica um limite em W, e limite em R e W significa um limite em Δ. No entanto, um limite em somente Δ não é suficiente para um algoritmo de tempo pseudo-polinomial.

Proof Vamos a usar programação dinâmica para encontrar o caminho mais curto em um grafo acíclico orientado (DAG). Existem $O(nK^2)$ estados nos quais o sistema pode entrar. Seja (i, j, τ) o estado do sistema cuando o processador começa a executar a j-ava unidade de trabalho para a tarefa i no tempo τ. Assim $i = 1, ..., n, j = 1, ..., t_i$ e $\tau = 0, ..., K$.Transições de estado para estado são definidas de acordo com as seguintes regras:
(1) Não preempção: Uma vez que uma terefa é iniciada, ela deve ser completada sem interrupções.
(2) Quando uma tarefa é concluída no tempo τ, um outra tarefa deve começar imediatamente, se existir no sistema. (Pelo Lema 3.4, sabemos que esta tarefa ainda não foi executada.) Caso contrário, o sistema está ocioso e inicia executando uma tarefa tão logo um chega.
(3) O estado (i, t_i, τ) é um estado final, se e somente se, quando a tarefa i é concluída no tempo τ, nenhuma tarefa pode ser executada posteriormente.
(4) O estado inicial tem transição para as tarefas $(i, 1, 0)$ que chegam primeiro.

O objetivo do programa dinâmico é para encontrar o comprimento de um caminho mais curto a partir de um estado inicial a um estado final. Dependendo de como vamos atribuir pesos para as arestas, podemos forçar nosso algoritmo para minimizar todas as três métricas de Secção 2. Para completar a análise do tempo, nota que só nK do NK^2 estados têm mais que centralidade constante, e esses estados têm cada uma centralidade delimitada por n. ∎

Para $R > 2$ sabemos de nenhum algoritmo eficiente, sem condições adicionais. Deixe W ser um límite na taxa da janela maior para janela menor, e seja Δ um limite na taxa da tarefa mais comprida para o menor tarefa. Note-se que os limites de R e Δ implicam um limite em W, e os

limites em R e W significa um salto em Δ. No entanto, um limite em Δ sozinho não é suficiente para um algoritmo de tempo pseudo-polinomial.

Teorema 3.7 *Mesmo com um limite na razão Δ, o LBP sem interrupção é fortemente NP-completo. Não pode ser aproximada dentro de um fator de Δ-ϵ, para qualquer $\epsilon > 0$, a menos que $P = NP$.*

Proof Modificar a redução de 3-partição do Teorema 3.2. alterando todas as fixas tarefas unidade para ter comprimento $B/3$, e ajustar os tempo de chegada e os prazos em conformidade. ∎

Em vez de uma tarefa muito longa como na prova do Teorema 3.2, criamos uma seqüência de tarefas de duração limitada, que servem a mesma finalidade. Uma unidade antes do prazo das tarefas elemento (ver Teorema 3.2) uma seqüência de tarefas maiores $l_1, ..., l_m$ chegam. Cada tarefa l_i inteiramente completa sua janela e portanto, só podem ser executados diretamente quando ela chega. A tarefa l_{i+1} chega no prazo da tarefa l_i. Além disso, uma seqüência de tarefas curtas $s_1, ..., s_m$ chega, onde cada tarefa curta s_i é também inteiramente completa sua janela e só pode ser executada quando ela chega. Tarefas curtas s_i sobrepõe l_i e l_{i+1}; chega uma unidade antes do prazo da tarefa l_i. Tarefas s_i tem comprimento $B/4$ e tarefas l_i têm um comprimento $\Delta.B/4$. Assim, se todas as tarefas que compõem o problema 3-partição podem ser executadas, tarefas $l_1, ..., l_m$ serão evitados pela execução das tarefas $s_1, ..., s_m$. Caso contrário, as tarefas $l_1, ..., l_m$ devem ser executadas. O índice de m pode ser adaptado a qualquer ε.
Limites de ambos Δ e R são suficientes para produzir um algoritmo pseudo-polinomial.

Teorema 3.8 *Seja $K = max_i d_i$. Dado o limite em R e Δ, o problema do funcionario preguiçoso sem preempção pode ser resolvido em $O(K.n^{4Rlg\Delta})$ para funções objetivo (1)-(3).*

Proof Modificação do algoritmo de programação dinâmica do Teorema 3.6 para situações mais complejas. Um conjunto de tarefas potencialmente disponíveis para trabalhar em um escalonamento determinado no tempo τ são as tarefas j que ainda não foram executadas. para o qual $d_j - t_j \geq \tau$. Nosso estado de espaço irá codificar o complemento deste conjunto para cada tempo τ, especificamente, o conjunto de tarefas que foram executados antes mas poderiam ter sido executado no tempo τ. ∎

3.2.3 Tarefas com tempo de liberação comum.

Na próxima versão do problema todas as tarefas são liberadas no tempo zero, ou seja, $a_i = 0$ para todo i. Este problema pode ser resolvido em tempo polinomial por programação dinâmica. A programação dinâmica funciona por causa do seguinte resultado estrutural: Existe um planejamento ótimo que executa as tarefas Earliest Due Date (EDD).
Na verdade, este problema é um caso especial do problema geral seguinte: minimizar a soma ponderada das tarefas não cumplidas em seus prazos.

Teorema 3.9 *O LBP pode ser resolvido em tempo pseudo-polinomial quando todas as tarefas têm um tempo de liberação comum.*

4 LBP: Com preempção

Nesta seção consideramos o problema do funcionário preguiçoso em que as tarefas podem ser interrompidas: uma tarefa em progresso pode ser anulada, enquanto outra tarefa é processada, e então possivelmente pode ser retomada mais tarde. É importante distinguir entre as diferentes restrições que especificam quais tarefas estão disponíveis para serem processadas. Consideramos três escolhas naturais de tais restrições:

Restrição I: A fim de trabalhar na tarefa i no tempo τ, só é necessário que o tempo atual τ encontra-se dentro do intervalo da tarefa I_i: $a_i \leq \tau \leq d_i$.

Restrição II: A fim de trabalhar na tarefa i no tempo τ, não só é necessário que o tempo atual τ encontra-se dentro do intervalo da tarefa I_i, mas também que a tarefa tenha uma chance de ser concluída, por exemplo, se ela é processada sem interrupção até sua conclusão.
Esta condição é equivalente a exigir que $\tau \leq c\prime_i$, onde $c\prime_i = d_i - t_i + y_i$ é o tempo crítico ajustado da tarefa i: $c\prime_i$ é o último momento para iniciar a tarefa i a fim de cumprir seu prazo d_i, dado que uma quantidade y_i da tarefa já foi concluída.

Restrição III: A fim de trabalhar na tarefa i, é necessário que $\tau \epsilon I_i$. Além disso, é necessério que toda tarefa que é iniciada seja evetualmente concluída.

Dividimos esta seção em subseções, onde cada subseção considera uma das três funções objetivo da Seção 2.1, na qual os objetivos são minimizar o tempo total de trabalho, a soma dos pesos de tarefas concluídas, ou o makespan do escalonamento. Por cada métrica, vemos que as limitações em preempção pode afetar drasticamente a complexidade do problema.
A restrição III faz com que o LBP com preempção seja bastante semelhante ao LBP sem preempção. Na verdade, se todas as tarefas chegam ao mesmo tempo ($a_i = 0$ para todo i), então as três funções objetivo são equivalentes, e que o problema é difícil:

Teorema 4.1 *O LBP com preempção, sob restrição III (um deve completar qualquer tarefa que é iniciada), é (fracamente) NP-completo e difícil de ser aproximado para as três funções objectivo.*

Proof Vamos a usar a mesma redução como a redução dada na prova do Teorema 3.1. Note que qualquer escalonamento para uma instância dada pela redução, no qual todas as tarefas processadas devem ser eventualmente completadas, pode ser transformado em um escalonamento equivalente sem preempções. Isso torna o problema de encontrar um escalonamento ótimo sem preempções equivalente ao problema de encontrar um escalonamento ideal no caso interrompível sob a Restrição III. ∎
Note que não podemos usar uma prova semelhante ao do Teorema 3.2 para mostrar que esse problema é fortemente NP-completo, desde que a interrumpção pode levar a melhores escalonamentos nessa instância.

4.1 Minimizando o tempo total de trabalho

Teorema 4.2 *O LBP com preempção, sob restrição I (um pode trabalhar em qualquer tarefa no intervalo da tarefa) e objetivo (1) (minimizar o tempo total de trabalho), tem solução polinomial.*

Proof O algoritmo de escalonamento de tarefas, segundo a última data de vencimento (LDD), no qual em todos os momentos do sistema, a tarefa com o último prazo é processado, com os laços quebrados arbitrariamente. Um argumento de intercâmbio mostra que esse é ótimo. Suponha que existe um escalonamento ótimo que não é LDD. Considere a primeira vez na qual uma programação ótima difere do LDD, e deixe OPT ser um ótimo escalonamento em que neste momento é o mais tarde possível. Vamos deixar OPT executar um pedaço da tarefa i, p_i e LDD executa um pedaço de trabalho j, p_j. Sabemos que a $d_i < d_j$. Queremos mostrar que podemos substituir a primeira unidade de p_i por uma unidade de p_j, contrariando a escolha da OPT, e, assim, provar a afirmação. Se em OPT, a tarefa j não é completamente processada, então esta troca é possível, e estamos a fazer. Por outro lado, se todas as tarefas j são processadas em OPT, tal troca faz com que uma unidade da tarefa j depois seja removida, deixando uma diferença de uma unidade.
Se esta diferença não pode ser preenchida por qualquer outro pedaço de tarefa, teremos um escalonamento com menos trabalho que OPT, que é uma contradição. Por conseguinte, assumir que esse intervalo pode ser preenchido, possivelmente cause um intervalo de uma unidade mais tarde. Continue este processo, e em sua conclusão, ou um intervalo de uma unidade continuará contradizendo a otimalidade da OPT, ou não existirá nenhum intervalo, contrariando a escolha do OPT. ∎

Teorema 4.3 *O LBP com preempção, sob restrição II (um pode somente trabalhar em tarefas que podem ser completadas) e objetivo (1) (minimizar o tempo total de trabalho), é (fracamente) NP-completo*

Proof Se todos os tempos de chegada são os mesmos, então este problema é equivalente a um em que a função objetivo é minimizar o makespan, o qual é mostrado ser NP-completo no Teorema 4.7. ∎

4.2 Minimizando a soma dos pesos das tarefas concluídas

Teorema 4.4 *O LBP com preempção, sob restrição I (um pode trabalhar em qualquer tarefa no intervalo da tarefa) e objetivo (2) (minimizar a soma ponderada das tarefas concluídas), tem solução polinomial.*

Proof Sem perda de generalidade, suponha que as tarefas $1, ..., n$ estão indexadas, crescentemente em ordem a seus prazos limites.
Mostramos como decompor as tarefas em componentes separadas para ser tratadas de forma independente. Escalonamento das tarefas de acordo com o EDD(se um trabalho está sendo

executado e passa seu prazo limite, interrompe e executar a seguinte tarefa).Sempre que há um intervalo(potencialmente de tamanho zero), onde não tem tarefas no sistema, as tarefas são divididas em componentes separados que podem ser escalonadas de forma independente e seus pesos somados.
Agora vamos concentrar em um exemplo como um conjunto de tarefas(não tendo intervalos). Vamos modificar o escalonamento EDD interrompendo uma tarefa ε unidades de tempo antes que ela seja concluída. Então, passamos o resto das tarefas do escalonamento seguinte por ε unidades de tempo e continuamos o processo. No final do escalonamento, tem duas possibilidades. (1) O último trabalho é interrompido porque a seu prazo limite expirou, neste caso, obtemos um escalonamento em que nenhuma tarefa é completada; (2) a última tarefa é concluida e além disso todos as outras tarefas cujos prazos não passaram também são obrigadas a concluir
A prova é completada com as seguintes observações:
(1) Existe um escalonamento ótimo que completa de todas as suas tarefas no final, e
(2) O escalonamento acima executa o máximo valor de trabalho possível. (Em outras palavras, EDD ("minus ε ") permite que se execute a quantidade máxima de trabajo em tarefas de 1 a i sem concluir nenhuma delas.) ∎

Teorema 4.5 *O LBP com preempção, nos termos da restrição II (só se pode trabalhar em tarefas que podem ser concluídas) e objetivo (2) (minimizar a soma ponderada das tarefas concluídas), é (fracamente) NP-completo.*

Proof Considere o LBP nos termos da restrição II, onde o objetivo é minimizar o makespan. A prova do Teorema 4.7 terá instâncias difíceis onde todas as tarefas têm a mesma hora de chegada, e onde a melhor solução completa qualquer tarefa que se inicia. Assim, para esses casos, a métrica de minimizar o makespan é equivalente à métrica de minimizar a soma ponderada das tarefas concluídas, para os pesos proporcionais ao tempo de processamento. ∎

4.3 Minimizando o makespan: ir para casa mais cedo

Assumimos agora que a meta do funcionário é ir para casa o mais rapidamente possível. Começamos por notar que, se os tempos de chegada sço os mesmos ($a_i = 0$, para todo i), então o objetivo (3) (ir para casa o mais rapidamente possível) é equivalente ao objetivo (1) (minimizar o tempo total de trabalho), uma vez que, nos termos de qualquer das três restrições I-III, o funcionário estará ocupado sem parar até que ele possa ir para casa.

Observe que se os prazos limites são os mesmos ($d_i = D$, para todo i), então os objetivos (1) e (3) são completamente diferentes. Considere o seguinte exemplo. Job 1 chega a tempo $a_1 = 0$ e é de comprimento $t_1 = 2$, Job 2 chega ao tempo $a_2 = 0$ e é de comprimento $t_2 = 9$, Job 3 chega ao tempo $a_3 = 8$ e é de comprimento $t_3 = 2$, e todas as tarefas têm prazo $d_1 = d_2 = d_3 = 10$. Então, a fim de minimizar o tempo total de trabalho, o funcionário vai fazer as tarefas 1 e 3, um total de 4 unidades de trabalho, e vai para casa no momento 10. No entanto, a fim de ir para

casa o mais rapidamente possível, o funcionário vai fazer o trabalho 2, realizando 9 unidades de trabalho, e ir para casa em tempo de 9 hora (uma vez que não existe suficiente tempo para fazer ou a tarefa 1 ou a tarefa 3)

Teorema 4.6 *O LBP com preempção, sob restrição I (um pode trabalhar em qualquer tarefa no intervalo da tarefa) e objetivo (3) (ir para casa o mais rapidamente possível), tem solução polinomial.*

Proof O algoritmo é para escalonar, segundo a última data de vencimento (LDF). A prova é similar a prova do Teorema 4.2 ∎

Se em vez de restrição I impomos a restrição II, o problema torna-se difícil:

Teorema 4.7 *O LBP com preempção, sob restrição II (só se pode trabalhar em tarefas que podem ser concluídas) e objetivo (3) (ir para casa o mais rapidamente possível), é (fracamente) NP-completo, mesmo se todos os tempos de chegada são os mesmos.*

Proof É feita uma redução do SUBSET SUM. Consideremos uma instância de SUBSET SUM dada por um conjunto S de n inteiros positivos, x_1, x_2,. . . , X_n, e a soma objetivo T. É construida uma instância da versão necessária do LBP como se segue. Para cada inteiro x_i , temos a tarefa i, que chega a hora $a_i = 0$, tem comprimento de $t_i = x_i$, e é devido ao tempo $d_i = T + x_i$ - ε, onde ε é uma constante (basta usar $\varepsilon = 1/3n$). Além disso, temos uma longa tarefa $n+1$, com comprimento de $t_{n+1} > T$, que chega em um momento $a_{n+1} = 0$ e é devido ao tempo $d_{n+1} = T$-$2\varepsilon + t_{n+1}$. Afirmamos que é possível que o funcionário vai para casa pelo tempo T se e somente se existe um subconjunto de x_1,...,x_n que somados de exatamente T. Se existe um subconjunto de x_1,. . . , x_n cuja soma é exatamente T, então o funcionário pode executar o correspondente subconjunto de tarefas (de comprimento total igual a T) e ir para casa no tempo T; ele é capaz de evitar fazer qualquer um dos outros trabalhos, desde os seus tempos críticos queda em uma hora mais cedo (T-ε ou T-2ε), tornando inviável para eles começar no tempo T, pela nossa suposição. ∎

Se, por outro lado, o funcionário é capaz de ir para casa no momento T, então nós sabemos o seguinte:

1. O funcionário deve ter acabado de concluir uma tarefa no tempo T. Ele não pode tirar uma tarefa e ir para casa no meio de uma tarefa, desde que a tarefa deve ser completável no instante em que ela é iniciada (ou reiniciada) para trabalhar, e ela continua completável no momento em que ele gostaria de tirar e ir para casa.

2. O funcionário deve ficar ocupado o tempo inteiro de 0 até o tempo T. Ele não tem permissão para ficar ocioso durante qualquer periodo de tempo, pois ele poderia trabalhar em alguma tarefa disponível, por exemplo, a tarefa J_{n+1}.

3. Se o funcionário inicia uma tarefa, então ele deve terminá-lo. Primeiro, notamos que, se ele inicia a tarefa J_i e faz pelo menos ε dela, então ele deve terminá-la, desde que no tempo T menos que $x_i - \varepsilon$ continua a ser feito da tarefa, e não é devido, até o momento $T - \varepsilon + x_i$, tornando-se viável para retornar a tarefa no tempo T (então ele não pode ir para casa no tempo T). Em segundo lugar, devemos considerar a possibilidade de que ele pode realizar pequenas quantidades (menos de ε) de algumas tarefas sem terminá-las. No entanto, neste caso, a quantidade total que ele completa de apenas estas tarefas que começou é no máximo $n\varepsilon \leq (1/3)$. Esta é uma contradição, desde que seu tempo total de trabalho é constituído por este comprimento fracionário de tempo, mais a soma dos comprimentos integrais dos trabalhos que terminou, o que não pode adicionar ao inteiro T. Assim, para ele ir para casa exatamente no tempo T, ele deve ter concluído todas as tarefas que ele começou. Finalmente, note que ele não pode usar a tarefa J_{n+1} como enchimento e fazer parte dela antes de ir para casa no momento T, desde que, se ele começa-la e trabalha menos tempo 2ε sobre ela, então, pelo mesmo raciocínio como acima, ele será forçado a permanecer e concluí-la. Assim, ele não irá iniciá-la em tudo, desde que ele não pode termina-la antes do tempo T (lembre-se que $t_{n+1} > T$).

A conclução que o funcionário deve completar um conjunto de tarefas cuja soma de comprimentos seja exatamente T. Assim, temos reduzido SUBSET SUM para o nosso problema, mostrando que é (fracamente) NP-completo.

Note que o LBP que foi construído não tem dados inteiros. No entanto, nós podemos "alongar "o tempo para obter um problema equivalente em que todo dado é integral. Deixando $\epsilon = 1/3n$, multiplicamos todos os comprimentos das tarefas e datas de vencimento por $3n$ (ver fig.2).

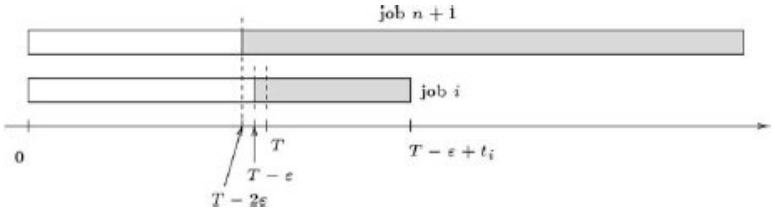

Figura 2: Prova de dificultade do LBP com preempção, supondo que todos os tempos de chegada são no tempo 0

Observação. Hepner e Stein [8] recentemente publicaram um algoritmo com tempo pseudopolinomial para este problema, resolvendo assim um problema aberto em uma versão anterior deste artigo [1]. Chegamos agora a um dos principais resultados do artigo. Vamos enfatizar este resultado, pois ele usa e faz uma análise de um algoritmo bastante sofisticado para mostrar que, em contraste com o caso de idêntico tempo de chegada, o LBP, com idêntico prazos limite tem solução polinomial. Especificamente, os problemas abordados nos Teoremas 16 e 17 são idênticos, exceto que o fluxo de tempo é revertido. Assim, podemos demonstrar que em contraste com a maioria dos problemas de programação clássica, no LBP, quando o tempo flui em uma direção,

o problema é NP-difícil, visto que quando o fluxo de tempo é revertido, o problema é solúvel em tempo polinomial. O restante desta seção é dedicada a provar o seguinte teorema:

Teorema 4.8 *O LBP com preempção, sob restrição II (só se pode trabalhar em tarefas que podem ser concluídas) e objetivo (3) (ir para casa o mais cedo possível), é solúvel em tempo polinomial se todas as tarefas têm os mesmo deadline ($d_i = D$, para todo i).*

Começamos com uma definição de um "interrupção forçada ". Existe uma interrupção forçada a partir do tempo τ se τ é o primeiro momento em que o trabalho total que chega por hora τ é menor que τ. Este (primeira) interrupção forçada termina no momento da chegada, $\tau\prime$, da próxima tarefa. Posteriormente, pode haver mais interrupções forçadas, cada um determinado por considerar o problema de escalonamento que se inicia no final, $\tau\prime$, da anterior interruplção forçada. Notamos que uma interruplção forçada pode ter comprimento zero.
Sob o objectivo de "ir para casa cedo ", podemos supor, sem perda de generalidade, que não existem interrupções forçadas, já que o nosso problema realmente começa apenas no tempo $\tau\prime$ que é onde a última interrupção forçada termina.(O funcionário certamente não está autorizado para ir a casa antes do termino de $\tau\prime$ da última interrupção forçada, já que mais tarefas llegam depois de $\tau\prime$, que pode ser processadas antes de seus *deadlines*.) Apesar de um escalonamento ideal pode conter interrupções que não são forçadas, o seguinte Lema implica que existe um ótimo escalonamento sem ter nemhuma interrupção não forçada.

Lema 4.9 *Considere o LBP do Teorema 4.8, e assuma que não existem interrupções forçadas. Se existe um escalonamento com makespan T, então existe um escalonamento com nenhuma interrupção, tendo também makespan T.*

Proof Considere a primeira interrupção no escalonamento, a qual começa no tempo g . Porque a interrupção não é forçada, existe alguma tarefa j que não está completada, e seu tempo crítico é no momento $g\prime \leq g$. Isto é porque deve existir uma tarefa que chegou antes de g que não está completada no escalonamento , e no tempo g não é mais viável para completá-la e, portanto, o seu momento crítico é antes de g. O intervalo e tempo entre $g\prime$ e T pode consistir de (1) interrupções, (2) trabalhar em tarefas concluídas, e (3) trabalhar em trabalhos que nunca serão concluídas. Considere um escalonamento avaliado em que, após o tempo $g\prime$, tarefas do tipo 3 são removidos, e as tarefas do tipo 2 são adiadas para o final do escalonamento. (Desde que uma tarefa de tipo 2 está concluída e todos os tarefas têm o mesmo prazo, sabemos que é possível movê-lo para mais tarde no escalonamento, sem passar seu tempo crítico. Pode não ser possível deslocar uma (parte da) tarefa de tipo 3 no final do escalonamento, desde que o seu tempo crítico pode ter passado.) No escalonamento, estender a tarefa j para preencher o espaço vazio. Note que não existe suficiente trabalho na tarefa j para preencher o espaço, desde que o momento crítico de $g\prime$ significa que a tarefa deve ser executada continuamente até que limite D, a fim de concluí-la. ∎

Lema 4.10 *Considere um LBP do Teorema 4.8 no qual não existem interrupções forçadas. Qualquer escalonamento viável pode ser reorganizados para que todas as tarefas concluídas sejam ordenadas por seus tempos de chegada e todas as tarefas incompletas sejam ordenadas por seus tempos de chegada.*

Proposição 4.11 *Se por cada interrupção no escalonamento S, existem suficientes tarefas longas para ser concluídas, a fim de preencher a interrupção, em seguida, um escalonamento viável terminando em T existe.*

Lema 4.12 *Existe um escalonamento viável terminando no tempo T se e só se existe um m para o qual $T(m, n) < \infty$*

5 Algoritmo de SJF

O algoritmo SJF funciona da seguinte forma. Sempre que a máquina está ociosa e existem algumas tarefas executáveis, escalonar a tarefa (entre as tarefas executáveis) com o menor tempo de processamento. Esfabhod et al . [2] mostrou que SJF é um algoritmo 2-aproximação com a função objetivo [*min-makespan*] e este limite é fechado. No mesmo artigo eles deixaram uma questão aberta que pede para a razão de pior caso do algoritmo SJF sob a função objectivo [*min-time-spent*]. Foi respondida essa questão em [3], eles mostraram que SJF tem a mesma proporção pior caso de 2. O exemplo em . [2], fica dedicada à função objetivo [*min-makespan*], e ainda se aplica à função objetivo [*min-time-spent*]. Para ser exaustivo, apresentamos um exemplo simples para mostrar o limite inferior de 2. Basta ter três tarefas, dois de tempo de processamento 1 e outro de $1 + \varepsilon$, onde $\varepsilon > 0$ é um número arbitrariamente pequeno. O prazo comum é de 2. O algoritmo SJF irá produzir uma programação de tempo de 2, enquanto o ideal é apenas $1 + \varepsilon$. Ela implica um limite inferior de 2 para o algoritmo SJF. No seguinte nós só precisamos de mostrar:

$$\frac{T_{SJF}(I)}{T*(I)} \leq 2$$

(1)

vale para qualquer instância I, onde $T*(I)$ e $T_{SJF}(I)$ são o tempo total gasto em um escalonamento ideal e na programação gerada pelo algoritmo SJF, respectivamente. Se isso não é verdade, ou seja, se existe alguma instância que viole a desigualdade acima, considere um mínimo de contra-exemplo I_s em termos do número de tarefas. A instância I_s tem duas propriedades:

1. $T_{SJF}(I_s)/T*(I_s) > 2$.
2. Para qualquer instância I onde $|I| < |I_s|$, por exemplo, o número de tarefas em I é menor que em I_s, então temos que $T_{SJF}(I)/T*(I) \leq 2$.

Desde $T_{SJF}(I_s) \leq D$, temos $T*(I_s) < D/2$. Deixe $\sigma*$ ser uma ótima programação e σ uma programação gerada pela SJF na instância I_s respectivamente. Considere o primeiro ponto de

tempo, t_1 em que a máquina torna-se ociosa na programação $\sigma*$. Claramente $t_1 < D/2$. Deixe t_2 ser o primeiro ponto de tempo em que a máquina torna-se ociosa na programação σ. Queremos mostrar que $t_1 = t_2$. Se isto não é mantido, poderia existir dois casos:

Caso 1: $t_1 > t_2$Considere as tarefas escalonadas em $\sigma*$, que chegam antes do tempo t_2. É evidente que o tempo total de processamento delas é maior do que t_2 (caso contrário, $t_1 \leq t_2$). Assim, deve existir a6lguma tarefa J que é iniciada antes do tempo t_2 em $\sigma*$, mas não está programada em σ. Seja p o tempo de processamento da tarefa J. Então $p > D - t_2 > D - t_1 > t_1$, pois J não pode caber na programação de σ dentro do intervalo $[t_2, D]$. Isso conflita com o fato de que $p \leq t_1$. Assim, este caso não pode acontecer.

Caso 2.$t_1 < t_2$. Neste caso, deve existir alguma tarefa J com tempo de processamento p que é iniciado antes do tempo t_1 em σ mas não está programada em $\sigma*$. Claramente $p + t_1 > D$. Isso mostra que $p > D/2$ e J é a maior tarefa programada em σ. Além disso, existe alguma tarefa $J' \in \sigma * -\sigma$ (isto é, $J' \in \sigma*$ mas $J' \notin \sigma$). No tempo t que J é iniciado em σ, não existem pequenas tarefas executáveis (com tempo de processamento de no máximo $D/2$). Assim, o tempo do lançamento do J' é maior do que t, pois, caso contrário J' seria programado em vez de trabalho J pela regra do algoritmo SJF. Daqui resulta que a máquina deve tornar-se inativo no momento $t < t_1$ na programção $\sigma*$. Esta é uma contradição com a definição de t_1. Portanto, temos provado que $t_1 = t_2 < D/2$. Claramente em σ qualquer tarefa programada por t_1 tem um tempo de processamento inferior a $D/2$, que também deve ser programada em $\sigma*$,e vice-versa. Isto implica que σ e $\sigma*$ agendam as mesmas tarefas pelo tempo t_1. Considere uma nova instância I removendo todas as trarefas que chegam antes do tempo t_1. Observe que $T_{SJF}(I)/T*(I) = (T_{SJF}(I_s) - t_1)/(T*(I_s) - t_1) > 2$. Isso implica que I é uma contra-exemplo com menos tarefas do que I_s. Isso conflita com o suposição de que I_s é contra-exemplo mínimo. Daí a desigualdade (1) vale para todas as instâncias.

Então é concluido que:

Teorema 5.1 *A razão de pior caso do algoritmo SJF é de 2 para CD-LBSP sob função objetivo [min-time-spent].*

6 Um PTAS para minimizar makespan

Esfahbod et al. [2] apresentou um algoritmo de 2-aproximação com a função objetivo [*min-makespan*]. A seguir, vamos mostrar que para qualquer inteiro fixo k, existe um algoritmo de aproximação com um limite de $(k + 1)/k$. É primeiro investigado a seguinte propriedade de um escalonamento ideal.

Lema 6.1 *Deve existir um escalonamento ideal que obedece a regra primeiro em chegar primeiro em servir(FCFS) para minimizar o makespan.*

Proof É fácil de demonstrar, com uma estratégia de intercâmbio. Começamos com umescalonamento ótimo que não obedece à regra $FCFS$. Deve existir duas tarefas adjacentes J_i e J_j com tempos de liberação $a_i < a_j$, enquanto a tarefa J_j está programada antes que a tarefa J_i num escalonamento ótimo. Trocando duas tarefas não perturbá o escalonamento restantes e a otimalidade permanece. Continue este processo até que qualquer duas tarefas adjacentes satisfassam a propriedade que a tarefa que chegar mais tarde é iniciada mais tarde. Finalmente, temos um escalonamento ótimo que obedece a regra $FCFS$. ∎

Para qualquer inteiro dado $k > 0$, distinguimos as tarefas como segue: Uma tarefa é chamada grande se seu tempo de processamento é tão comprido quanto $D/(k+1)$; de outro jeito a tarefa é chamada de pequena. Seja S um conjunto de tarefas pequenas e L um conjunto de tarefas compridas. Suponha que $|L| = m \leq n$. Claramente em qualquer escalonamento ao menos k tarefas compridas podem ser executadas. Considere todos os subconjuntos de L que consiste em até k grandes trabalhos. O número de subconjuntos,está limitado superiormente por $N_k = \binom{m}{k} + \binom{m}{k-1} + ... + \binom{m}{1} + \binom{m}{0} = O(m^{k+1})$. Denotado por L_i, para $i = 1, 2, ..., N_k$.

Algoritmo A_k

1. Organizar as tarefas em ordem não decrescente em ordem dos tempos de liberação. Escalone as tarefas ordenadas uma por uma, enquanto elas são executáveis (isto é, aplicar a regra FCFS para as tarefas) e obter um escalonamento σ_0

2. Para $i = 1, 2, ..., N_k$, escalone as tarefas em $L_i \bigcup S$ com FCFS. Denote o escalonamento por σ_i. Descartar o escalonamento σ_i se:

   - alguma tarefa $L_i \bigcup S$ não pode ser escalonado pelo prazo comum D com FCFS,ou
   - em algum ponto t onde a máquina está ociosa uma tarefa $J_j \notin L_i \bigcup S$ poderia ser iniciada, nomeadamente $p_j \leq D - t$.

3. Selecione o melhor (com o mínimo makespan) entre todos os escalonamentos restantes, incluindo σ_0.

A estratégia de descarte na Etapa 2, garante que o escalonamento é expulsado se e somente se não for viável (não respeitando o requisito de ocupado).

Teorema 6.2 *Para qualquer k dado, A_k é um algoritmo de aproximação com o taxa de pior caso, no máximo, $1 + 1/k$ sob a função objetivo [min-makespan].*

Proof Considere um escalonamento ótimo $\sigma*$ que satisfaz a regra FCFS, por exemplo, as tarefas executadas estão em ordem de suas chegadas. Se $\sigma*$ não executa todas as tarefas pequenas, então o ótimo makespan $C*$ é ao menos $D - D/(k+1) = kD/(k+1)$. Note que o makespan C_{A_k} dado por A_k é ao muito D. Assim $C_{A_k}/C* \leq 1 + 1/k$. Agora suponha que $\sigma*$ executa todas as tarefas

pequenas juntas com $k_1 \leq k$ tarefas grandes. Mais precisamente $\sigma*$ escalona essas tarefas com a regra FCFS. Obviamente deve existir um escalonamento viável idéntico para $\sigma*$ no passo 2. Em este caso coseguimos um escalonamento com o ALgoritmo A_k. A combinação dos dois casos, implica que a taxa do pior caso do algoritmo de A_k é $1 + 1/k$.
Finalmente é estimado o tempo de execução do Algoritmo A_k, $O(nlogn)$ para escalonar tarefas com FCFS. O tempo de compilação do algoritmo é determinado no passo 2. Esse tem un tempo de $O(n^{k+2}logn)$. Como k é um inteiro fixo, o algoritmo A_k é polinomimal. ∎

Para qualquer número pequeno fixo $\varepsilon > 0$, seja $k = \lceil 1/\varepsilon \rceil$, temos um esquema de aproximação com a taxa de pior caso, no máximo, $1 + \varepsilon$. Assim, chegamos à seguinte conclusão.

Corolário 6.3 *Existe um PTAS sob a função objetivo [min-makespan].*

7 Um PTAS para minimizar o tempo gasto

Em esta seção, vamos mostrar que para qualquer inteiro fixo k, existe um algoritmo de aproximação B_k com um límite de $1 + 1/k$ sob a função objetivo *[min-time-spent]*.
Suponha que existe m diferentes tempos de liberação $0 = T_1 < T_2 < ... < T_m$. Obviamente se $m = 1$ as funções objetivo *[min-time-spent]* e *min-makespan* são equivalentes. Se $m > 1$, as duas funções objetivo pdem fazer diferença desde que algum espaçõ ocioso pode ser introduzido no escalonamento. Para lidar com *[min-time-spent]* nossa idéia principal é analisar uma fase por fase o escalonamento. Vamos aplicar recursivamente o algoritmo A_k em cada fase e tentar descobrir a configuração de um escalonamento ideal.

Baseado no Algoritmo A_k definimos uma série de algoritmos A_k^i, para $i = 1, 2, ..., m$. O algoritmo A_k^i trabalha exatamente como o algoritmo A_k sob as seguintes codições:
(1) Somente os trabalhos que chegam ao tempo ou depois do tempo T_i são escalonados por A_k e (2) uma tarefa é grande se seu tempo de processamento é mais comprido que $(D - T_i)/(k + 1)$ e pequeno em caso contrário.
Seja $M(i, j)$ o makespan produzido pelo algoritmo A_k^i no escalonamento todas as tarefas chegam no intérvalo fechado $[T_i, T_j]$, enquanto o escalonamento correspondente é denotado por $S(i, j)$, para $0 \leq i \leq j \leq m$.. Por simplicidade denotamos $M(i, i)$ como $M(i)$. Seja $P(j)$ o tempo gasto aplicando o algoritmo B_k(definido recursivamente) para as tarefas com chegada no tempo T_j ou antes desse tempo, enquanto o correspondente escalonamento é denotado por σ_j para $j = 1, ..., m$.

Algoritmo B_j

1. $P(0) = 0, P(1) = M(1)$; Seja σ_1 o correspondente escalonamento.

2. Para $j = 2, ..., m$ faça:
   $P(j) = min\{P(j-1) + M(j), P(j-2) + M(j-1, j), P(1) + M(2, j), M(1, j)\}$; Suponha que

$P(j)$ é determinado por $P(h)+M(h+1,j)$ para algum $1 \leq h \leq j-1$. Seja M_h o makespan para o escalonamento σ_h, por exemplo o tempo de conclusão das tarefas que chegam antes T_{h+1}. Construímos σ_j como se segue:

- Se $M_h \leq T_{h+1}$, então σ_h não tem interseção com $S(h+1,j)$, o escalonamento de tarefas que chegam em $[T_{h+1}, T_j]$. Combinando as duas programações temos um escalonamento viável σ_j.
- $M_h > T_{h+1}$, mantemos o escalonamento σ_h e eliminamos o escalonamento $S(H+1,j)$ (sem mudar a ordem da tarefa e escalonar as tarefas o mais cedo possível) para o tempo M_h. Assim conseguimos um escalonamento viável.

3. O escalonamento final $\sigma = \sigma_m$

Antes de analizar o algoritmo , é necessário mostrar que, se $M_h > T_{h+1}$ para algum h, escalonar σ_j é viável. Em outras palavras , as tarefas escalonadas em $S(h+1,j)$ podem ser completadas no tempo D depois do atraso. Se não é verdade , então $M_h+M(h+1,j) \geq M_h+\sum_{J_i \in S(h+1,j)} p_i > D$. Seja T_l a hora final do último intervalo ocioso da programação σ_h (e não houver tempo ocioso, $T_l = T_1$). Obviamente $T_l + M(l,j) \leq D$, assim $P(h) + M(h+1,j) \geq P(l-1) + (M_h - T_l) + M(h+1,j) > P(l-1) + D - T_l \geq P(l-1) + M(l,j)$. Este conflicto com a suposição de que $P(j)$ é determinado por $P(h) + M(h+1,j)$. Assim σ é viável.
O algoritmo B_k usa o algoritmo A_k como subrutina. Para calcular $P(m)$ é preciso executar A_k por $O(m^2)$ vezes. O tempo de execução de B_k é $O(n^{k+4}logn)$. É polinomial para qualquer inteiro fixo $k > 0$.

Teorema 7.1 *Para qualquer k , a taxa de pior caso do algoritmo B_k é no máximo $1 + 1/k$ sob a função objetivo [min-time-spent].*

Proof Vamos provar este teorema por indução. Se existe somente um tempo de liberação (*realisetime*) T_1, por exemplo, $m = 1$, B_k trabalha exatamente como A_k e o makespan $M(1)$ é igual ao tempo gasto $P(1)$. Pelo teorema 5.1 conseguimos o limite de $1 + 1/k$.

Suponha que o teorema vale para l tempos distintos de liberação, ou seja, vale para $m = l$. É querido mostrar que ainda é verdadeira para $m = l + 1$. Considere um escalonamento ótimo $\sigma*$. Se não houver tempo ocioso antes de todas as tarefas estejam concluídas, conseguimos a conclução com os dois pontos seguintes:

- Em $\sigma*$, o tempo gasto é igual a do makespan.
- O tempo gasto pelo algoritmo B_k é ao máximo $M(1, l+1)$, no aqual é ao máximo $1 + 1/k$ vezes o makespan de $\sigma*$, pelo teorema 5.1.

Agora, vamos supor que existe de fato algum tempo ocioso antes de todas as suas tarefa(de $\sigma*$) são concluídas. Denotam o intervalo passado ocioso, $[x, y]$. Obviamente y deve ser o tempo de liberação de alguma tarefa (caso contrário, a tarefa que se inicia no momento y teria sido escalonado anteriormente, devido ao requerimento guloso). Seja $y = T_h, h \leq l + 1$. O escalonamento ótimo $\sigma*$ é dividido em duas partes σ_1^* e σ_2^*, onde σ_1^* é um escalonamento ótimo para essa tarefas que chegam antes de T_{l+1}, e σ_2^* é um escalonamento ótimo para as tarefas que cheguem no tempo ou depois de T_{l+1}. Denote por OPT_1 e OPT_2 os dois valores objetivo σ_1^* e σ_1^*, respectivamente. Então $OPT = OPT_1 + OPT_2$ é o valor objectivo de σ^*. Por indução, temos $P(h-1) \leq (1 + 1/k)OPT_1$. Note que não existe tempo ocioso em σ_2^*. Assim, $M(h, l+1) \leq (1 + 1/k)OPT_2$ pelo Teorema 5.1. Por outro lado, $P(l+1) \leq P(h-1) + M(h, l+1)$. Portanto, temos:

$$P(l+1) \leq (1 + 1/k)(OPT_1 + OPT_2) = (1 + 1/k)OPT$$

Este teorema fica provado. ∎

Agora temos :

Corolário 7.2 *Existe um PTAS sob a função objetivo [min-time-spent].*

Considerações finais: Este trabalho faz um estudo sobre o problema de escalonamento do burócrata preguiçoso, em resumo é uma tradução da documentação referida na referência, estos artigos fornecem sistemas de aproximação de tempo polinomial e NP-completo para as funções objectivo indicadas na seção 2.

Referências

[1] J.S.B. Mitchell S.S. Skiena. E.M. Arkin, M.A. Bender. The lazy bureaucrat scheduling problem: Proceedings of the sixth workshop on discrete algorithms (wads). *Springer-Verlag*, 1663:122–133, 1999.

[2] Sharifi A. Esfahbod B, Ghodsi M. Common-deadline lazy bureaucrat scheduling problems. in: Dehne fkha, sack j-r smid mhm (eds) algorithms and data structures. *Lecture Notes in Computer Science*, 2748:59–66, 2003.

[3] G. Zhang. L. Gai. On lazy bureaucrat scheduling with common deadlines. *Springer Science+Business Media*, J Comb Optim (2008) 15:191–199, 2007.

[4] D.S. Johnson. M.R. Garey. Computers and intractability: A guide to the theory of np-completeness. *W.H. Freeman, San Francisco*, 1979.