Universidade Federal de Santa Catarina
Centro Tecnológico
Curso de Ciências da Computação

DIEGO MARTINS MEDITSCH
EDUARDO ACHAR
FELIPE MACACARI PIEROTTI

Trabalho 1 - Sistemas Operacionais I:

Simulação de Algoritmos de Escalonamento

Florianópolis - Santa Catarina
2024

SUMÁRIO

1. **INTRODUÇÃO**
2. **PRIMEIRA INTUIÇÃO**
3. **DESENVOLVIMENTO DO ALGORITMO**
4. **OTIMIZAÇÃO**
5. **ANÁLISE DE COMPLEXIDADE**

INTRODUÇÃO

Para solucionar o problema proposto - simulação de algoritmo de escalonamento - nós passamos por algumas possíveis soluções, as quais serão detalhadas ao decorrer deste relatório. Em resumo, começamos buscando alguma forma de resolver o problema sem ter que percorrer todas as tarefas a serem selecionadas, de forma a minimizar o custo computacional da solução. No entanto, logo percebemos que isso seria impossível para o problema dado, e partimos para uma solução que percorre as tarefas disponíveis de maneira a comparar todas elas antes de tomar uma decisão sobre a ordem em que elas devem ser executadas.

No entanto, essa solução ficou bastante ineficiente, e acabamos por buscar maneiras de otimizar essa versão até chegar em um novo tipo de solução.

Nosso código final implementa uma solução eficiente para o problema de agendamento de tarefas com lucro máximo, sem que haja sobreposição de tarefas (intervalos). A estratégia utilizada é combinar uma abordagem gulosa (ordenar os processos pelo tempo de término) com programação dinâmica e busca binária para otimizar a escolha das tarefas.

PRIMEIRA INTUIÇÃO

A primeira ideia que tivemos para solução do problema foi a de tentar encontrar uma forma de não percorrer todas as tarefas, assim buscando atingir uma complexidade menor que $\mathbf{O(n)}$. No entanto, logo percebemos que isso não seria possível, pois mesmo que o melhor caminho seja tomado até a última tarefa, sempre pode haver uma tarefa de alto lucro no final da execução que só “caberia” no tempo de execução se um caminho menos lucrativo tivesse sido escolhido até lá. Em outras palavras, o único jeito de achar uma solução desse tipo seria prevendo o futuro.

Nossa ideia seguinte foi pensar em uma forma de resolver o problema que nos possibilitasse construir a solução do problema ao longo da execução do programa, de maneira que o conjunto de tarefas a serem realizadas fosse percorrido uma única vez, e cada pedaço do caminho fosse definido antes de partir para o próximo, uma clássica estratégia de algoritmos “gulosos” como o de Dijkstra. Com essa ideia em mente, começamos a desenvolver o código em si.

DESENVOLVIMENTO DO ALGORITMO

Tendo em mente a natureza do algoritmo a ser desenvolvido, pensamos na melhor maneira de guardar as informações de entrada e percorrer as tarefas disponíveis. Para percorrer as tarefas de maneira sequencial (e mais tarde garantir o funcionamento da busca binária), decidimos ordenar as tarefas pelo tempo de término delas. Criamos para isso um vetor capaz de guardar o tempo de início, fim e lucro de cada tarefa em triplas, e ordená-las por tempo de fim. Isso garante que mantenhamos a associação de cada tarefa com seus atributos mesmo modificando a ordem do vetor. Em seguida, criamos um vetor que guarda o lucro máximo até cada tarefa, e definimos que o lucro até a primeira tarefa é exatamente o lucro da mesma.

Feita a inicialização, agora basta percorrer o vetor de tarefas, buscando a última tarefa que não sobrepõe com a atual e somando o lucro da mesma com o lucro da tarefa atual. Para decidir qual tarefa escolher, aqui temos o passo “guloso” do algoritmo, onde decidimos se escolhemos essa tarefa ou não, escolhendo a opção que maximiza o lucro entre a tarefa atual somada à escolhida ou simplesmente mantém a tarefa atual, sem adicionar a nova.

Fazendo isso de maneira iterativa até termos percorrido todas as tarefas, garantimos que no final teremos o conjunto de tarefas com o maior lucro possível. O único “problema” é que o algoritmo ficou bastante ineficiente, já que para escolher a próxima tarefa que não sobrepunha com a atual, estávamos percorrendo todo o vetor de tarefas até encontrá-la (assim percorrendo todas as tarefas dentro de um loop que percorre por todas as tarefas, caracterizando um algoritmo de complexidade $\mathbf{O(n^2)}$.

OTIMIZAÇÃO

Partindo desse cenário, com o vetor de tarefas ordenado, fica evidente que é possível otimizar essa busca que era linear até o momento. Portanto, optamos por otimizar essa etapa de encontrar a próxima tarefa através da implementação de uma busca binária.

Além de contribuir na otimização do nosso algoritmo, a busca binária é de fácil implementação. Sua principal ideia é, dividir um vetor ordenado em 2 partes iguais, escolhendo entre a porção inferior e a superior, de acordo com o valor buscado, e fazer isso de forma recursiva para a porção escolhida, até que reste o número escolhido, ou o mais próximo dele, a depender da implementação.

ANÁLISE DE COMPLEXIDADE

Nota-se, que não passamos mais por todos os elementos da lista, ao buscar um valor utilizando a busca binária, na verdade, recursivamente dividimos a lista em dois, o que resulta em $\mathbf{\log_2 (n)}$ operações por busca.

Então, saímos de um cenário onde eram realizadas *n* buscas lineares pelo vetor, concluindo $\mathbf{O(n^2)}$ operações, e após a etapa de otimização no algoritmo, resultou-se em uma complexidade final de $\mathbf{O(n * \log_2(n))}$ operações.