Festival, um problema de planejamento de atrações com múltiplos palcos

Michael Douglas Gonçalves Nóbrega[1],
Wladimir Araújo Tavares[1]

[1]Universidade Federal do Ceará (UFC) – Campus Quixadá

dougnobrega@alu.ufc.br, wladimir@lia.ufc.br

Abstract. *In this article, we will present and review some methods to solve the 2018 Marathon Regional Festival problem that fits as a multi-stage festival attraction planning problem, including Complete Search and dynamic programming with top-down and bottom-up. All of the methods presented use the bitmask technique to represent a set easily and interestingly.*

Resumo. *Neste artigo, apresentaremos e analisaremos alguns métodos para solucionar o problema Festival da Regional da Maratona de 2018 que se encaixa como um problema de planejamento de atrações com múltiplos palcos, sendo eles Complete Search e programação dinâmica com as abordagens top down e bottom-up. Todos os métodos apresentados usam a técnica de bitmask para representar um conjunto de maneira fácil e interessante.*

1. Introdução

A Maratona de Programação é um evento promovido pela Sociedade Brasileira de Computação(SBC), desde do ano de 1996, com o objetivo de despertar o interesse dos alunos pela área da Computação. Ela estimula a difusão de conhecimento através da proposição de problemas desafiadores, que exigem a familiarização com estruturas de dados e algoritmos clássicos, bem como suas vantagens e limitações. Além disso, a competição colabora com o desenvolvimento e aperfeiçoamento dos alunos, ajudando na melhoria do ensino de algoritmos de maneira geral. Os problemas abordados nessa competição representam uma boa fonte de estudo para os alunos que querem aprofundar e praticar os conhecimentos adquiridos ao longo do curso.

Em setembro de 2018, a fase regional da Maratona de Programação apresentou 13 novos problemas. Neste artigo, analisaremos o problema chamado Festival, proposto nessa regional, que pode ser considerado um problema de seleção de atrações em festival com múltiplos palcos sujeito a algumas restrições. Em cada palco, temos uma seleção de atrações do festival. Cada atração j, pode ser descrita por 3 inteiros i_j, f_j e o_j representando, respectivamente, os horários de início, fim do show e o número de músicas apresentadas pelo cantor que já foram ouvidas pelo usuário em seu sistema de *streaming* favorito. Para melhorar a experiência dos usuários que estão no festival, deseja-se maximizar a quantidade de músicas conhecidas seguindo os seguintes critérios:

- Cada atração deve ser assistida por completo.
- Todos os palcos devem ser visitados, ou seja, pelo menos uma atração deve ser vista em cada palco.

- O tempo de deslocamento entre palcos pode ser ignorado
- Cada Festival tem no máximo 10 palcos e 1000 atrações

Note que a restrição, que obriga que pelo menos uma atração de cada palco deve assistida, introduz uma restrição interessante ao problema.

2. Modelagem do problema

Uma atração será modelada por uma 5-tupla $(palco, inicio, fim, musicas, prox)$ representando o palco em que a atração será realizada, o início da atração, fim da atração, número de músicas já ouvidas pelo usuário e posição do próxima atração válida, respectivamente. A próxima atração válida será uma atração que começa logo após o fim da atração considerada.

Inicialmente, o vetor v das atrações será ordenado pelo tempo de início da cada atração. A ordenação do vetor v permite que a próxima atração possa ser encontrada utilizando uma busca binária com a complexidade $\mathcal{O}(nlogn)$

O subproblema que queremos resolver pode ser formalizado da seguinte maneira: $M[i][S]$ representa a quantidade máxima de músicas escutadas pelo usuário considerando as atrações $i \dots n$ no vetor v considerando o conjunto de palcos S já visitados. Observe que a solução final é representado pelo conjunto $M[0][\emptyset]$

O conjunto de palcos S será representado por um *bitmask*. Esta técnica permite associar um subconjunto de um conjunto pequeno a um valor inteiro. Se um i-ésimo elemento está em S então o i-ésimo bit da representação inteira de S está ligado. Logo, o valor 0 representa o conjunto $\emptyset$ e $2^n - 1$ representa o conjunto com todos os n elementos. As operações sobre o conjunto são substituídas por operações de manipulação de bits[Halim 2013]. Por exemplo, checar se o i-ésimo elemento está em S será substituída por $S\&2^i \neq 0$ e adicionar o i-ésimo elemento no conjunto S será substituída por $S|2^i$.

Seja n o número de atrações e m o número de palcos. O número de subproblema que devem ser resolvidos será $\mathcal{O}(n2^m)$

2.1. Estrutura Recursiva

Para montar a estrutura recursiva precisamos ter em mente os casos bases e as transições. Por exemplo, $i = n$ quando isso acontecer significa que todas as atrações foram vistas, entretanto se todas as atrações tiverem sido vistas podemos ter chegado em dois pontos, o primeiro é quando todos os palcos foram vistos, quando isso acontecer vamos retornar 0, pois desta forma manteremos o resultado quando voltar da recursão, entretanto se $i = n$ e pelo menos um palco não foi visto temos que eliminar aquela resposta e uma forma fácil de eliminar a resposta é retornando $-\infty$.

$$M[i][S] = \begin{cases} 0, & i == n \text{ e } S = 2^n - 1 \\ -\infty & i == n \text{ e } S \neq 2^n - 1 \end{cases}$$

Quando $i \neq n$, temos duas escolhas para serem analisadas. Se a atração i será selecionada ou não. Se a atração i não for selecionada, basta calcular a solução $M[i + 1][S]$, caso contrário, precisamos encontrar $M[v[i].prox][S \cup v[i].palco] + v[i].musicas$

$$M[i][S] = max(M[i + 1][S], M[v[i].prox][S \cup v[i].palco] + v[i].musicas)$$

3. Complete Search

Uma abordagem valida, porém ineficiente seria utilizando Complete Search. O problema de utilizar essa solução é a quantidade de sobreposição de subproblemas que são ignoradas, ou seja, pra cada sequência de atrações tudo vai ser calculado quantas vezes necessário sem ter a otimização de se um subproblema já foi calculado não precisamos calcular de novo. A construção desse algoritmo segue a estrutura recursiva do problema.

Algorithm 1 SOLVE

Entrada: i, $bitmask$
Saída: Quantidade de músicas ouvidas da solução ótima

início
  se *i* = *n and bitmask* = $2^n - 1$ **então**
    retorna *0*
  senão se *i* = *n* **então**
    retorna $-\infty$
  retorna *max(SOLVE(*i + 1, $bitmask$*),SOLVE(* $v[i].prox$, $bitmask$ | *(1* << $v[i].palco$*)) +* $v[i].musicas$*)*

4. Algoritmo Top-Down com memorização

O algoritmo *top-down* pode ser construído diretamente através da estrutura recursiva do problema. Como o problema possui uma grande sobreposição de subproblemas utilizamos uma tabela de memorização de subproblemas já resolvidos para evitar o recálculo desse subproblema.

Algorithm 2 SOLVE

Entrada: i, $bitmask$
Saída: Quantidade de músicas ouvidas da solução ótima

início
  se *i* = *n and bitmask* = $2^n - 1$ **então**
    retorna *0*
  senão se *i* = *n* **então**
    retorna $-\infty$
  senão se *Memo*$[i][bitmask]$ *!=* -1 **então**
    retorna *Memo*$[i][bitmask]$
  retorna *Memo*$[i][bitmask]$ = *max(SOLVE(*i + 1, $bitmask$*),SOLVE(* $v[i].prox$, $bitmask$ | *(1* << $v[i].palco$*)) +* $v[i].musicas$*)*

5. Algoritmo Bottom-Up

A abordagem *bottom-up* segue do principio de calcular os subproblemas mais fáceis e depois calcular subproblemas mais difíceis até chegar no resultado que desejamos, dessa forma essa abordagem calcula todas as possibilidades gerando toda a tabela de memorização. A desvantagem dessa abordagem é que pode ser calculado subproblemas que não fazem parte da solução de determinado problema.

Note que o Algoritmo *Top-down* pode calcular apenas uma parte da tabela de memorização enquanto o bottom-up sempre calcula a tabela toda, podemos deduzir que mesmo o *bottom-up* tendo maior aproveitamento dos recursos do computador existem momentos onde o *top-down* vai ser mais rápido por ter que fazer menos processamento computacional.

Algorithm 3 SOLVE

Saída: Quantidade de músicas ouvidas da solução ótima

início
    $inicializar(dp, -\infty)$
    $i \leftarrow n - 1$
    $OK \leftarrow 2^n - 1$
    dp[n][ok] $\leftarrow 0$
    enquanto $i \geq 0$ **faça**
        $bitmask \leftarrow 0$
        enquanto $bitmask \leq OK$ **faça**
            $dp[i][bitmask] = max(dp[i+1][bitmask], dp[v[i].prox][bitmask \mid (1 << v[i].palco\ )\ ] + v[i].musicas)$
            $bitmask \leftarrow bitmask + 1$
        $i \leftarrow i - 1$
    retorna $dp[0][0]$

6. Resultados

Foi enviado as 3 soluções debatidas no Uri Online Judge[1] a qual duas foram aceitas e uma recebeu tempo limite excedido. A que recebeu tempo limite excedido utilizou-se da técnica *Complete Search* a qual testava todas as possibilidades possíveis, desta forma ignorando a sobreposição de subproblemas e como sempre recalculava os subproblemas não conseguiu ser executada em menos de 1 segundo para os casos de teste do site. Já as outras duas soluções que utilizavam programação dinâmica levaram menos de 0.030 segundos para entregar a resposta certa. O código *top-down* levou 0.024 segundos, enquanto o *bottom-up* levou 0.004 segundos. A solução da recursão levou mais tempo comparado ao *bottom-up* pela quantidade de saltos na memoria pela recursão e a quantidade de *cache miss* que torna o programa mais lento no geral, já o outro código não tem saltos na memoria e tem uma taxa de *cache hit* muito alta pela forma que acessamos o vetor na hora de calcular todos os subproblemas.

[1] Site: https://www.urionlinejudge.com.br

7. Conclusão

Neste artigo foi apresentado uma solução eficiente para o problema Festival, utilizando busca binaria pra pre-calcular as próximas posições validas e programação dinâmica com *bitmask* para calcular a solução de maneira eficiente e interessante, neste problema em especifico o bottom-up se sobressaiu do top-down tornando uma ótima escolha para esse paradigma.

Referências

Halim, S. (2013). *Competitive Programming 3: The New Lower Bound of Programming Contests*. Lulu, 3th edition.