UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ALGORITMOS E ESTRUTURAS DE DADOS III

Trabalho Prático 2

Aluna Ana Luiza de Avelar Cabral

Introdução

O Objetivo deste trabalho é, dado um jogo de cartas, descobrir se ele pode ser ganho e descobrir o melhor resultado possível no jogo.

O jogo consiste de:

- uma sequência de *S* cartas, sendo cada uma um valor positivo ou zero;
- um monte inicial (com um valor positivo ou zero);
- um limite global *X* (que é um valor positivo *X*);
- e um mínimo final *M* (valor positivo).

O jogo funciona da seguinte forma: as S cartas vão ser apresentadas em sequência, uma sequência em que a ordem não pode ser alterada. A cada carta apresentada deve-se fazer uma escolha: ou somá-la ao monte ou subtraí-la do monte.

Se, ao somar, o valor obtido ultrapassar o limite *X*, o jogador é eliminado na hora. Se, ao subtrair, o valor obtido for menor do que zero, o jogador também é eliminado. Se o jogador chegar até o final da sequência de cartas sem ser eliminado em suas opções locais, seu monte final é analisado: caso o valor do monte seja no mínimo M, o jogador ganha o jogo e leva o valor obtido para casa. Caso o valor do monte seja menor do que o valor M mínimo o jogador perde.

Assim, o objetivo do trabalho é descobrir:

- Se **existe** uma sequência de escolhas que pode ser feita na qual o jogador consegue chegar ao final da sequência de cartas sem ser eliminado;
- Caso exista essa sequência, qual o maior valor possível que pode ser atingido.
- Se esse maior valor possível ganha o jogo (é maior ou igual ao mínimo final *M*) ou não.

Para resolver o problema são solicitados dois algoritmos: um algoritmo do tipo força bruta e outro que seja implemente os paradigmas ou de Programação Dinâmica ou de Algoritmos Gulosos.

Modelagem e Solução Proposta

1. Modelagem

O paradigma Força Bruta que deveria ser obrigatoriamente implementado consiste em varrer o espaço de soluções analisando uma a uma e assim encontrar a melhor. O segundo algoritmo seria Guloso ou Programação Dinâmica.

Analisando o problema e casos de teste, não consegui encontrar uma estratégia gulosa que alcançasse a solução. A estratégia gulosa que pensei seria fazer a melhor escolha local, o que garantiria a solução global. Porém na escolha local (somar ou subtrair o próximo valor da sequência no monte) decisões gulosas locais não levam ao ótimo global.

Analisando o problema do ponto de vista da Programação Dinâmica, existem subproblemas aninhados, logo armazenar os valores já calculados traria mais eficiência ao problema. Podemos ver todos os caminhos possíveis como uma sequência de 0's e 1's. Sendo 0 representando uma soma e 1 representando uma subtração, temos:

(Toy 1)

Valor do monte: 5
Sequência: 5 - 3 - 7

Modelando cada bit, 0 ou 1, como o sinal de um elemento da sequência. Os casos possíveis são:

	5	3	7
	|	|	|
0: +	0	0	0
1: -	0	0	1
	0	1	0
V = 5	0	1	1
	1	0	0
	1	0	1
	1	1	0
	1	1	1

Assim, podemos ver que todas as sequências possíveis de serem feitas são os números binários de 0 a 111….111 consecutivamente, onde o último é composto do número de elementos da sequência em 1's. Por essa abordagem, podemos ver como a Programação Dinâmica resolve o problema:

O monte inicial é sempre o mesmo um mesmo caso de teste. Assim, toda vez que uma subsequência que começa no início já tiver sido calculada, ela pode ser reusada e não precisa ser calculada de novo até aquele ponto. Exemplo: calculamos 000. Então há o caso 001. Não é necessário recalcular 00 iniciais: como já foram calculados antes, os salvamos em uma estrutura de dados e consultamos, e a partir dele começamos o cálculo. Assim, prova-se que esse problema pode ser resolvido por Programação Dinâmica.

A resolução por força bruta consiste em percorrer todos os casos possíveis para achar a solução. Pela análise acima vemos que uma sequência de S elementos terá S bits no número binário, logo terá 2^S caminhos possíveis a serem percorridos. É fácil ver que esse número exponencial se eleva rapidamente:

$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
….
$2^{10} = 1024$
…..

Uma sequência com apenas 10 elementos já possui 1024 sequências diferentes a serem percorridas. No problema as sequências podem ter até 1000 elementos, o que geraria 2^{1000} sequências diferentes a serem visitadas. Podemos ver porque a solução por programação dinâmica é recomendada: para grandes valores na sequência o tempo para

percorrer o espaço de soluções cresce exponencialmente, dobra a cada elemento incluído na sequência, tornando dependendo do caso inviável.

2. Implementações realizadas

A) Força Bruta e Programação Dinâmica usando vetor de bits

Para resolver por força bruta a primeira questão é percorrer todo o espaço de soluções. Precisa-se pensar em uma forma automática de gerar todas as combinações possíveis e de visitá-las, sem esquecer de nenhuma.

A primeira ideia que me veio para gerar o espaço de soluções possíveis foi justamente o desenho acima, números binários. Ao percorrer os números de forma crescente, como binários, de 00000 até 111..111 (S 1's), vou estar gerando e percorrendo todas as soluções possíveis de serem construídas.

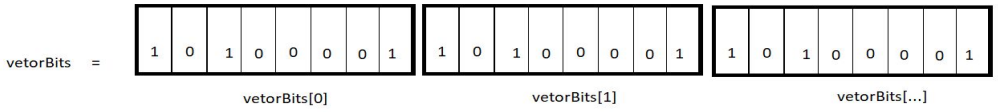
Então optei por esta modelagem: fazer um vetor de bits e ir somando 1 ao número, que ao somar ele muda para a próxima configuração e ao final vai ter gerado todos os caminhos possíveis.

Assim foi feita a modelagem: um bit de um campo representando o sinal de um elemento da sequência. O campo vai ser um valor numérico para que ao somar 1 nele mude automático a configuração dos bits interna.

Porém como a sequência tem até 1000 elementos conforme a especificação, preciso de um campo numérico com 1000 bits. Na linguagem C não existe campo assim. O maior campo tem 64 bits, long long int. Sendo assim, optei por representar então usando um vetor de campos de forma a completar os 1000 bits. E a cada adição arrumei o código para alterar os campos vizinhos como é feito na adição de decimais:

```
   0 8          0 9
+     1      +     1
-------      -------
   0 9          1 0
```

A ideia *inicial* que tive para a programação dinâmica era agrupar cada campo desses em valores salvos para a respectiva subsequência. Assim, quanto menor o campo melhor para fazer o reuso dos valores calculados. Sendo assim escolhi como campo para o vetor de bits o tipo unsigned char, que é o tipo com menos bits (8) o que melhora para agrupar e não é signed, não permitindo valores negativos, só 0 -> 255 o que ajuda nas somas número a número que precisarão propagar.



Para acessar os bits dentro de um campo usei uma máscara de bits e as operações AND e OR com essa máscara e o campo analisado.

A ideia para a programação dinâmica seria agrupar os valores gerados para cada campo, de 0 a 255 no valor de montante que eles geram, e salvá-los em uma estrutura de dados que os armazenaria. Mais tarde esse processo se mostrou, além de extremamente complexo e cheio de subcasos, um processo ineficiente. Isso porque ele precisa armazenar

para cada subsequência inicial, para cada uma das configurações de 0 a 255, de 0 a 255 posteriores para cada, visto que reusar o segundo campo só pode acontecer se o primeiro for igual. Isso gera uma matriz de n dimensões, altíssimo custo de espaço mesmo sendo alocado sob demanda e dinamicamente. Nos testes feitos com a programação dinâmica desta forma em alguns o tempo de execução da programação dinâmica era maior do que o força bruta. Isso é compreensível: são muitas estruturas de dados complexas e percorridas, como a alocação era dinâmica elas eram percorridas com ponteiros auxiliares e sem acesso indexado direto, o que ao invés de melhorar o custo piorava.

Além disso, houve o problema de só serem armazenados resultados de 8 em 8 elementos. Assim, um caso de teste com menos de 8 elementos não usava a programação dinâmica. Um caso com 50 elementos usava 50/8 = 6 vezes apenas. Não era o mais eficiente possível. Sendo assim mais para frente abandonei essa implementação para programação dinâmica, após os resultados dos testes abaixo, e mantive esta abordagem apenas para força bruta, para programação dinâmica fiz nova implementação.

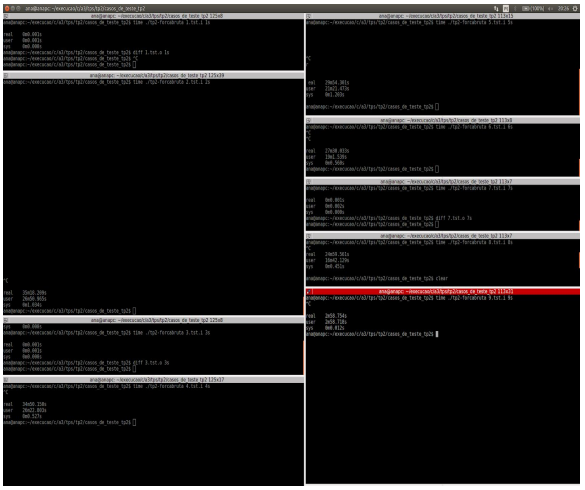


Fig. 1: Força Bruta com de vetor Bits

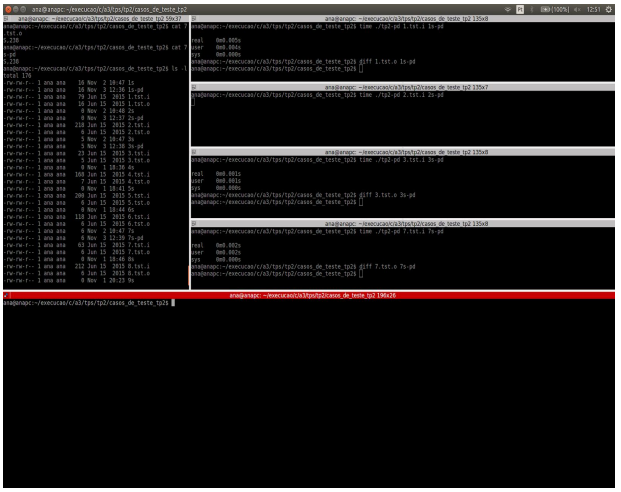


Fig. 2: Programação Dinâmica com vetor de Bits

B) Programação Dinâmica usando matrizes

A outra forma que pensei para armazenar os resultados da programação dinâmica foi usar matrizes alocadas dinamicamente se baseando não no vetor de bits mas na subestrutura ótima do problema.

Pensei em outra forma de gerar o espaço de soluções que traria melhor armazenamento e reuso dos valores mais efetivo.

- Inicialmente calculo 0 e 1. (V **+** S1) e (V **-** S1).
- Percorro
    - 0 e faço 0+0 = 00 e 0+1 = 01.
    - 1 e faço 1+0 = 01 e 1+1 = 11.
- Então percorro os últimos calculados, fazendo +0 e +1 em cada um. E faço isso até percorrer toda a sequência.

Isso baseia na propriedade: cada sequência é a sequência anterior +0 ou +1. Isso faz com que cada novo elemento na sequência para ser calculado apenas faça uma só soma, da subsequência anterior com o valor atual.

S_0	0	1					
S_1	00	01	10	11			
S_2	00 0	00 1	01 0	01 1	10 0	10 1	11 0	11 1

A Estrutura de Matriz se mostrou muito mais eficiente e menos complexa do que a Programação Dinâmica Partindo de um vetor de bits. Porém nos testes ela também não se mostrou eficiente.

A ineficiência vem da ausência de poda na estrutura. Apesar de cada resultado aproveitar o anterior, ela não descarta os caminhos descartados da sequência (caminhos em que há estouro abaixo de 0 ou acima de X ao somar/subtrair). Fazer a poda nesta estrutura é extremamente complexo, e seriam tantas condições que não compensaria o custo da análise a cada iteração, deixando-a sem poda e apenas propagando -1 para os subcaminhos que foram podados. Porém isso causa que mesmo com subestrutura ótima e cada subsequência com custo apenas 1, causa precisar percorrer $\Sigma\, 2^n$, para n de 1 a S. Esse somatório resulta em $(2^{(n-1)} - 4)$ visitas: de novo número exponencial de visitas, mas dessa vez devido à ausência de podas.

Na figura abaixo executei o mesmo caso usando o algoritmo Força Bruta e o de Programação Dinâmica baseado em matrizes. Vemos que o tempo é quase igual. A implementação da Programação Dinâmica desta forma é bem ineficiente.

```
ana@anapc:~/execucao/c/a3/tps/tp2/tp-novo/casos_de_teste_tp2$ time ./tp2-pd 10.tst.i 10s-pd

real    0m0.586s
user    0m0.388s
sys     0m0.196s
ana@anapc:~/execucao/c/a3/tps/tp2/tp-novo/casos_de_teste_tp2$ time ./tp2-forcabruta 10.tst.
i 10s-pd

real    0m5.331s
user    0m5.329s
sys     0m0.001s
ana@anapc:~/execucao/c/a3/tps/tp2/tp-novo/casos_de_teste_tp2$
```

Assim, busquei outra forma de implementar a Programação Dinâmica.

C) Programação Dinâmica usando árvores binárias

Finalmente o algoritmo usado foi a implementação via árvores binárias. Ele é simples como o algoritmo de matriz, se baseia na mesma propriedade do problema, e permite podas, que é o que torna o algoritmo eficiente.

A vantagem de ter feito outras implementações antes foi a modularização mais expressiva que comecei a fazer. No código de programação dinâmica via vetor de bits o código era muito amplo e pouco modularizado em arquivos e funções, o que me deu **muito trabalho** o tempo todo de consulta em partes anteriores do código. Sendo assim, após o código complexo, no código de matrizes já comecei a modularizar em funções e em arquivos até mesmo etapas básicas como a leitura de arquivos, de modo a viabilizar melhor desenvolvimento.

3. Implementações atuais

Como mostrado na seção acima, a implementação do algoritmo força bruta fiz usando o vetor de bits e o algoritmo de programação dinâmica eu usei árvores binárias. Seguem detalhes de cada um abaixo.

A) Algoritmo Força Bruta

O algoritmo força bruta usa um vetor de bits para gerar todas as configurações possíveis para a sequência (todos os cenários de +/- em cada elemento da sequência possíveis sem deixar de cobrir nenhum). Cada bit do vetor representa o sinal (+/-) que vai ser aplicado à um dado elemento da sequência: assim cada bit representa o sinal de um elemento da sequência como mostrado acima, é um mapeamento direto e 1 para 1.

Sabendo que há esse mapeamento, o algoritmo faz o seguinte essencialmente:

- Gera uma configuração para os bits (logo, para os sinais)
    - Para essa configuração, gera-se o montante final com tal configuração dos sinais, somando e subtraindo do monte os valores da sequência, de acordo com o que o sinal deve ter cada elemento da sequência de acordo com o vetor de bits.
        - Caso em algum momento ao adicionar estoure o limite X ou ao subtrair o monte fica abaixo de zero, aí abandona essa configuração visto que ela é inviável (Poda essencial mesmo no força bruta, definida no problema).
    - Se chegar em um montante final sem estourar, compara o valor do montante final obtido com o maior montante final obtido até agora, salvo em uma variável. Caso esse seja o maior montante final obtido até agora o montante final máximo passa a ter o seu valor.
- Gera a nova configuração do vetor de bits e recomeça a busca na nova configuração.

  OBS.: Gerar a nova configuração do vetor de bits é somar um ao campo atual, o que vai fazer uma nova configuração em um campo sozinho automaticamente e com o tempo somar um vai ter gerado todos os valores possíveis de configuração <u>para aquele campo</u>. Para que inclua todos os campos eu construí a função confereVizinho(...), que soma um quando muda de grandeza e propaga a soma, como em 09 + 1 = 10, 99 + 1 = 100 (muda os outros algarismos).

<u>Algumas observações do algoritmo:</u>

- O acesso ao sinal de um bit de um campo é feito com uma máscara de bits e operações AND / OR da máscara com o campo analisado.
- Percorrer uma configuração dada dos bits é feita com a variável j, que percorre os campos unsigned char do vetor, cada campo de uma configuração (Para S = 1000, serão 1000 bits / 8bits por campo = 125 campos de unsigned char, vetor de unsigned char de 125 posições).
- l é uma variável usada para condição de parada. Como é um vetor não valeria a pena checar se cada elemento dele tem a configuração do momento de parada. Então uma ideia foi somar todos eles, e a soma do valor de parada só pode ser obtida no próprio valor de parada. A variável l é usada para isto.

<u>Desafios e dificuldades desta implementação e algoritmo:</u>

- Cada valor de S precisa de uma quantidade de bits diferente e não posso deixar bits vazios em um campo porque cada bit a mais a ser percorrido dobra o tempo de execução do programa.
  Isso acabou gerando uma complexidade alta. Tive que pensar em vários subcasos como:
  - j está em um campo que deve percorrer todos os bits dele ou não?
  - Caso o campo não deva ser percorrido completo, quantos bits devo analisar?
    - Nos campos não completos devo começar a percorrer os bits da direita para a esquerda e não da esquerda para direita como fiz com os outros. Isso porque os números binários (e decimais) crescem para a esquerda: $...2^3.2^2.2^1.2^0$. Assim, nesses campos a forma de percorrer é diferente e o mapeamento de bit para elemento da sequência S também (por começarem da direita).
  - A configuração é composta *apenas* de um campo não completo? (casos com até 7 elementos na sequência analisada).
- Uso de unsigned char:
  - Um dos motivos da escolha por unsigned char foi por ter o mínimo de bits por campo, 8 (que mais tarde seria bom para programação dinâmica mas acabei desistindo dessa abordagem). Mas de qualquer forma a lógica seria a mesma porque seriam necessários até 1000 bits e há no máximo 64 bits por campo.
  - Outro motivo foi não representar valores negativos, o que facilita a adição em cascata no vetor necessária para gerar todas as configurações.

  Porém o uso de unsigned char traz várias peculiaridades como em um loop *for* normal: (255 + 1) = 0 e não igual a 256, o que traz loops infinitos no caso de *for* para menor do que 256 (sempre será). Loops foram um desafio nesse ponto.

<u>O que poderia melhorar:</u>

- Modularização em funções para tornar o código mais limpo. Mas em compensação nas implementações de Programação Dinâmica com outras estruturas de dados logo depois aqui modularizei já o máximo possível.

Explicado a abstração principal do algoritmo e dissertados os detalhes, tem como ver a aplicação no código que está todo comentado em seções para mostrar o início de cada um dos processamentos descritos aqui.

Análise Experimental e Análise de Complexidade de Tempo e Espaço

A máquina usada para testar o algoritmo foi um Intel Core i5 com 8GB de memória RAM usando sistema operacional Ubuntu 16.04. Para medir o tempo de execução do programa foi usada a função *time* da linha de comando. Para garantir que não houve vazamentos de memória foi usado o programa *valgrind* na execução. Por fim, para garantir que as saídas geradas estão iguais às saídas dos casos de teste foi usado o *diff* na linha de comando.

Foram feitos testes e com os toys, porém o código só passou em um tempo aceitável nos testes com menos de 50 elementos (testes 1, 3 e 7). Nos outros ele continua executando. Isto é esperado porque para processar 50 elementos na sequência seria necessário processar 2^{50} configurações diferentes. Se formos pensar, 2^{10} já são 1024

configurações a serem analisada, cada configuração com, no pior caso, n operações. Assim, **no pior caso o algoritmo é O(S * 2^S)**, 2^S configurações e cada uma com S operações para a complexidade de tempo. Assim processar 50 entradas realmente tem um custo alto, que na própria especificação do problema fala e é com base nisso que o jogo / cassino da especificação se mantém, nessa dificuldade da força bruta com relação ao tempo.

Para a complexidade espacial, o algoritmo tem um custo fixo para todos os programas O(1) para algumas variáveis além do custo variável. O custo de espaço variável vem com o vetor de bits O(S) e com o vetor sequência para armazenar os elementos da sequência O(S). Assim, O(1) + O(2S) = O(S). Assim, **o algoritmo tem complexidade assintótica de espaço linear com o número de elementos na sequência.**

Continuando agora com a Análise Experimental, nos casos que gerei observei que o algoritmo força bruta retorna resultados em tempo aceitáveis quando tem até 25 elementos na sequência. Em alguns casos, com 17 elementos já sentimos o tempo de processamento, e acima de 26 o tempo de processamento já fica bem ruim.

B) Algoritmo usando Programação Dinâmica

Para a solução do problema usando Programação Dinâmica eu usei as Estruturas de Dados árvores binárias.

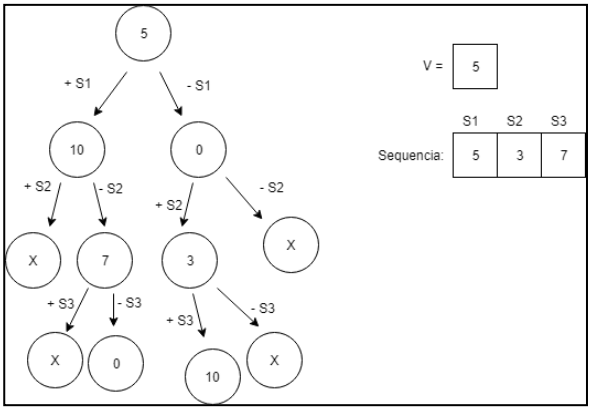


Fig.3: Toy 1 como árvore binária

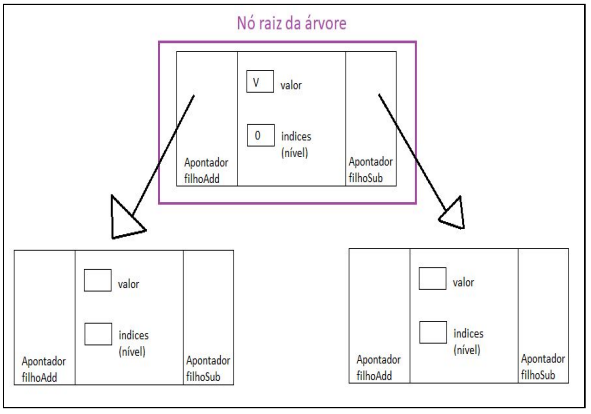


Fig.4: Implementação dos nós e nó raiz

Como é possível ver na figura 4, cada nó da árvore possui *valor*, o valor do monte de cartas naquele ponto, e *indices*, o índice s ou nível da árvore em que o nó está, uma variável usada para detectar se está num nó folha. Além disso, possui dois apontadores para outros nós: apontador filhoAdd e apontador filhoSub, que apontam para (valor + prox elemento da sequencia) e para (valor - proximo elemento da sequencia) respectivamente.

O primeiro nó da árvore, o nó raiz, é o nó no nível 0 que possui o valor inicial do monte de cartas.

<u>Pseudo-algoritmo:</u>

- Inicia programa abrindo arquivos da linha de comando.
    - Se algo está errado, imprime na tela e aborta o programa.
    - Se ok continua o programa.
- Le a quantidade de Instâncias no arquivo.

Para cada instância do arquivo:

- Lê as informações do caso de teste e as salva (lê e salva a sequência, S, M, X e V).
- Cria o nó raiz da árvore com o valor inicial V do monte e inicializa variáveis iniciais de um caso de teste.
- Função recursiva que irá percorrer os nós da árvore.
  Visita o nó raiz:
  - Caso este nó visitado seja uma folha:
    - Confere se o valor do monte nele é o maior até agora. Se for, salva como maior valor atingido até agora
    - Sai do nó retornando o endereço do próprio nó.
  - Caso este nó não seja um nó folha e sim qualquer outro nó:
    - Soma o próximo item da sequência ao montante atual e checa se a soma ultrapassa o limite X.
      - Caso não ultrapasse:
        - Cria um filhoAdd e inicializa esse nó.
        - Visita esse nó
        - Ao fim da visita desaloca esse nó.
      - Caso ultrapasse:
        - Não faz nada (filhoAdd continua tendo o valor inicializado para ele, NULL).
    - Subtrai o próximo item da sequência do montante atual e checa se a subtração fica abaixo do limite mínimo 0.
      - Caso não esteja abaixo do limite:
        - Cria um filhoSub e inicializa esse nó.
        - Visita esse nó
        - Ao fim da visita desaloca esse nó.
      - Caso esteja abaixo do limite:
        - Não faz nada (filhoSub continua tendo o valor inicializado para ele, NULL).
    - Retorna para quem chamou a função o endereço do próprio nó.
- Confere o valor máximo obtido nas folhas com M e com base na comparação imprime o resultado do caso de teste.
- Desaloca a memória do nó raiz e da sequência para começar nova instância.
- Fecha os arquivos e encerra o programa.

Como explicado, faço uma busca em profundidade recursiva para montar a árvore e aproveito e já confiro as folhas dela em busca de cada valor atingido, se este foi maior do que o máximo atual. Assim, faço a construção da árvore, busca nela e desalocação de memória no mesmo caminhar sobre ela. Para desalocar eu retorno o endereço do nó visitado ao sair dele para o ponto em que foi chamado, uma vez que não visitarei de novo.

Além disso, faço podas no espaço de solução toda vez que uma adição/subtração de um valor da sequência ultrapassa o limite: eu só visito caminhos possíveis, caminhos em que houve estouro não são visitados e são abandonados, o que garante a eficiência do algoritmo.

Análise de complexidade Temporal e Espacial do algoritmo de Programação Dinâmica

Análise Temporal

Se o algoritmo não tivesse podas, a complexidade temporal seria exponencial para visitar todos os nós:

- 1 nó no nível 0 (2^0)
- 2 nós no nível 1 (2^1)
- 4 nós no nível 2 (2^2)
- …
- 2^S nós no nível S (e S pode chegar a 1000).

Assim, no pior caso, o número de nós percorridos é $2^1 + 2^2 + ... + 2^S = \Sigma\, 2^n$, para n de 1 a S. Esse somatório resulta em $(2^{(S-1)} - 4)$ visitas em nó, logo $O(2^S)$, exponencial com o número de elementos da sequência no pior caso. Porém, na prática, as podas garantem a eficiência do algoritmo ao eliminar vários caminhos possíveis.

Análise Espacial

Durante toda a execução, gasta-se S para armazenar os elementos da sequência, e gasta-se um espaço fixo O(1) constante para as variáveis poucas do programa.

Além disso, variando ao longo da execução, gasta-se o espaço dos nós visitados ainda abertos. Ao sair de cada nó ele é desalocado. Assim, o pior caso de espaço aberto é visitar uma árvore com muitos elementos S (muito profunda) e com todos os elementos permitindo a adição sem estourar, o que causa ir armazenando recursivo as chamadas até chegar em uma folha para começar a desalocar a memória. Assim, o pior caso de espaço é a altura da árvore, que é S+1 (raiz + S).

Assim, O(S) fixo durante o programa e para o caso variável o pior caso é O(S+1), temos que a complexidade espacial é linear com o tamanho da sequência de entrada.

Análise Experimental

A máquina usada para testar o algoritmo foi um Intel Core i5 com 8GB de memória RAM usando sistema operacional Ubuntu 16.04.

Para medir o tempo de execução do programa foi usada a função *time* da linha de comando.

Para garantir que não houve vazamentos de memória foi usado o programa *valgrind* na execução.

Por fim, para garantir que as saídas geradas estão iguais às saídas dos casos de teste foi usado o *diff* na linha de comando.

Antes de colocar os resultados dos testes convém fazer algumas observações:

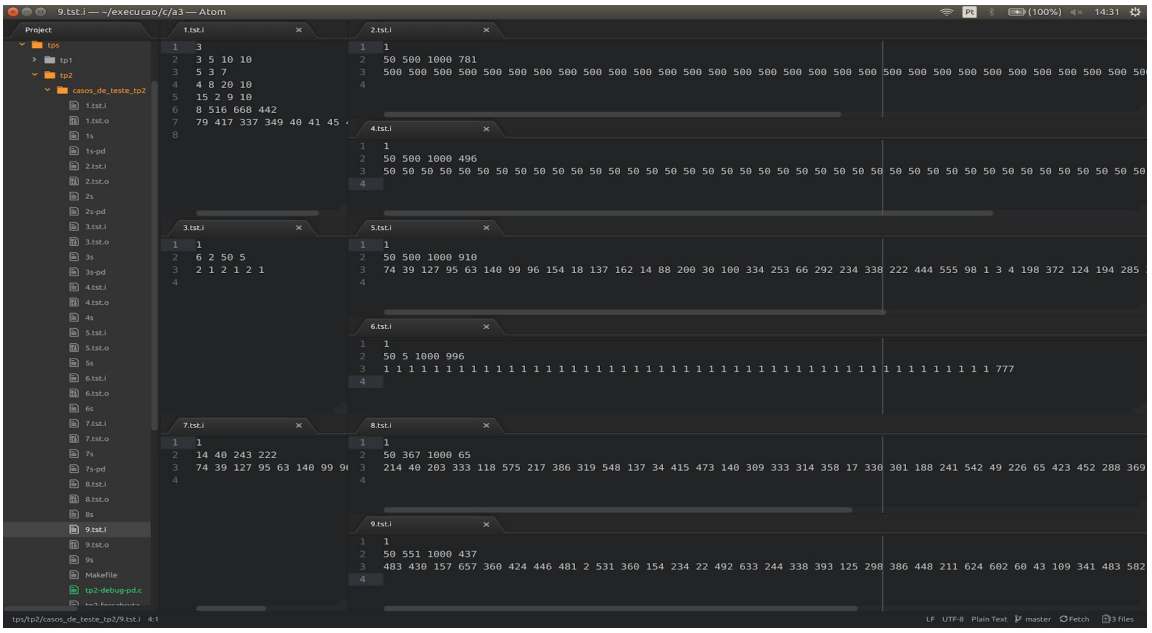
Aumento do tempo de execução ao usar valgrind

Reparei que, ao executar o caso de teste usando o **valgrind** e não usando o valgrind, o valgrind aumenta consideravelmente o tempo de execução dos programas. Assim, usei o valgrind para checar vazamento de memória mas medi o tempo sem ele porque ele piora bem o tempo.

Toy 2 demora um pouco mais mas passa

Ao executar o toy 2 parece que não irá executar porque demora um pouco, mas breve ele executa. O caso 2 requer mais processamento.

Problemas com os toys 4, 5 e 6 e debug



O código atual demora muito para processar os toys 4, 5 e 6, se é que termina de processar algum dia. Esperei 30 minutos para os três executando paralelamente e não consegui resultados, continuava executando.

Tentei fazer o debug gerando diversos testes. Observei que o processamento continua neles e fica caminhando nos nós próximos das folhas, o que é esperado. Eles também conseguem achar o valor máximo do monte final quando peço para imprimir na saída padrão os valores sendo processados, porém eles continuam processando.

Fui analisar qual a diferença entre os toys 2, 8 e 9, que também tem o mesmo número de vértices e o mesmo limite X do que os casos que não param de executar. Comparar a grandeza M não faz diferença nesse caso pois ela só é usada quando a visita aos nós acaba, o que não acontece nesses 3 casos de teste. Assim, a diferença está principalmente nos valores da sequência: 4, 5 e 6 tem valores menores do que 2, 8 e 9 dos elementos da sequência, o que faz que estoure (passe o limite X) menos, o que gera mais caminhos para percorrer. Com isto, faz sentido que 4, 5 e 6 demorem mais do que os outros casos de teste com 50 vértices. Porém eles não param em vários minutos de espera.

Outro teste que fiz é variar o número de vértices. No caso do teste 6, fui adicionando vértices para o mesmo valor de sequência de entrada e pude observar nas medições de tempo: a cada inserção de um novo elemento na sequência o tempo de execução dobra. Para o caso 6 o tempo de execução começa a se tornar perceptível, mesmo medindo com o *time*, a partir de 20 elementos na sequência. Como dobra a cada elemento, com 28 elementos o processamento demora 14 segundos. Seguindo estas contas, o processamento do caso 6 deveria levar 9,8 minutos. Porém aguardei 13 minutos e ele não processou.

Não acredito, sinceramente, que seja somente porque o processamento é naturalmente mais demorado para estes casos, acredito que há algum erro ou de lógica ou

de implementação, mas estou mais tendenciosa para acreditar em erro de implementação. Estava tendo problema com outros testes e quando fui procurar vi que estava acessando um índice do vetor de sequências inexistente apenas no caso da folha que eu não tinha pensado que seria uma exceção, e ao consertar funcionou.

Imagino que está acontecendo um de dois cenários: ou está havendo estouro de alguma variável mal dimensionada e causando valores inválidos (o que eu acredito menos porque fui bem fixa com essa parte e já conferi de novo também) ou estou acessando regiões não alocadas da memória em tempo de execução e o gcc não está avisando o acesso errado, e valores inválidos estão causando um loop.

Já conferi desalocação, retorno de funções, quase tudo e não achei o problema nesses casos de teste, só sei que ficam executando perto das folhas da árvore e às vezes sobem um pouco na árvore, que é o comportamento esperado de uma busca em profundidade mesmo.

Testei acesso indevido de memória tentando acessar de propósito valores inexistentes da sequência e o gcc não reclama de acesso indevido e dá um valor *default* para esses acessos indevidos. Tentei usar outro compilador e procurar uma IDE para tentar achar o erro mas o tempo acabou não dando devido às outras disciplinas.

Os outros toys: 1, 2, 3, 7, 8 e 9

Como vai ser visto ao executar os outros toys, eles executam e geram o resultado certo, e o fazem desalocando 100% da memória, todos. Além disso consomem pouca memória na execução pela estrutura do algoritmo que desaloca a memória ao acabar de usar o tempo todo.

Conclusão

Várias conclusões podem ser tiradas deste trabalho. A primeira que pude ver é que na busca por um melhor algoritmo senti falta de modularização no início e fui passando muito aperto por código poluído, o que me convenceu a modularizar o máximo possível o código, mesmo funções básicas como leitura e fechamento de arquivos. Modularizar aumenta a abstração o que facilita a visão global e o raciocínio, e permite não focar em outros detalhes desnecessários para o momento.

A questão do tempo de execução aumentar notavelmente ao se usar a ferramenta valgrind me impressionou também.

Um ponto muito importante deste trabalho foi mostrar a importância de “pequenas” otimizações quando se trata de um grande conjunto de dados: consegui ver o impacto ao trocar linhas de lugar e evitar processos na velocidade do processamento nitidamente.

Outro ponto foi o compilador gcc, que antes avisava acessos indevidos em tempo de execução e dava erros como seg fault de acesso, e agora não avisa mais e ainda retorna um valor *default* para o acesso, conforme teste que eu fiz exatamente para acessar campos indevidos e ver o que seria armazenado e como ficaria e execução. Erros de acesso indevido são ruins mas o compilador tratá-los automaticamente é muito ruim também, é melhor para qualquer programa que não execute nesses casos, como era antes.

Por fim, o trabalho mostra como a Programação Dinâmica viabiliza processamentos antes impossíveis em questão de tempo ao se aproveitar da subestrutura de subproblemas superpostos e do armazenamento dos valores e também das podas.