Projeto e Análise de Algoritmos
Trabalho Prático - Paradigmas

Raquel Yuri da Silveira Aoki (201566918)

[1]Universidade Federal de Minas Gerais (UFMG)
Departamento de Ciência da Computação

raquel.aoki@dcc.ufmg.br

1. Introdução

Este relatório é referente ao 1º Trabalho Prático da disciplina de Projeto e Análise de Algoritmos de 2015/2.

A proposta do trabalho, é considerando o ambiente de um grande estúdio de desenvolvimento de jogos chamado Unihard voltados para plataformas mobile, deve-se encontrar para um conjunto de fases do jogo o melhor conjunto de inimigos para cada fase de maneira automática.

Existe uma lista de inimigos disponíveis para colocar no jogo, e a cada um é associado a um *ranking* que quantifica a dificuldade em derrotá-los. Cada fase possui um nível de dificuldade, sendo que a soma dos *rankings* dos inimigos deve ser igual ao nível de dificuldade da fase. O melhor conjunto de inimigos para cada fase é definido como o conjunto cujo número de inimigos é o menor possível.

Nas próximas seções serão discutidas as abordagens utilizadas para resolver o problema proposto e análise do desempenho dessas soluções.

2. Modelagem

Como mostrado na Seção 1, o objetivo é obter para uma fase o conjunto com o menor número de inimigos. A resolução desse problema é similar ao *Problema do Troco*, uma variante do problema da mochila, que se resume em como obter um determinado valor de troco com o menor número possível de moedas, respeitando os valores das moedas disponíveis. No caso desse trabalho prático, o valor do troco é o nível de dificuldade da fase e as moedas seriam os inimigos disponíveis.

Note que pode existir mais de uma solução ótima. Por exemplo, para uma fase de nível de dificuldade 20, e com o conjunto de inimigos de *ranking* {5,10,15}, são soluções ótimas o conjunto de inimigos {5,15} e {10,10}. Além disso, na resolução do problema é considerado que um inimigo pode aparecer mais de uma vez no conjunto de solução ótimo e que o inimigo de *ranking*=1 sempre está presente.

As soluções apresentadas nas próximas seções, consideram que o arquivo de entrada com os inimigos está sempre ordenado em ordem crescente de *ranking*. O problema proposto será resolvido usando três algoritmos distintos de Paradigmas da Computação. São eles o Força Bruta (com bracktraking), um Algoritmo Guloso e Programação Dinâmica.

3. Soluções

A seguir são mostradas as abordagens utilizadas para resolver o problema.

3.1. Força Bruta

Algoritmos que utilizam a Força Bruta geram a testam todas as possíveis soluções. No caso do problema do jogo Assassin's Greed V, para cada fase ele testaria todas as combinações de inimigos cuja soma dos *rankings* sejam iguais ao nível da fase, e dessas combinações ele ficaria com a menor. Por testar todas as soluções possíveis, ele garantidamente encontra uma solução ótima.

Algumas "podas"podem ser realizadas de modo que a obtenção da solução ótima não seja prejudicada e o algoritmo fique mais rápido. Para realizar essas podas, foi utilizado o *backtracking*, que abandona uma solução parcial assim que é detectado que ela não é a ótima. No contexto do problema, considere que até certo ponto da enumeração das soluções, o menor conjunto encontrado tem *a* inimigos; se na próxima enumeração que o algoritmo fizer já tiverem sido selecionados *a* inimigos e ainda não tiver chegado ao final (a soma dos *rankings* dos inimigos selecionados é igual ao nível da fase ou a soma dos *rankings* supera o valor do nível da fase), essa enumeração é descartada e não é necessário chegar ao seu final, pois seu resultado já é pior que a menor solução encontrada até aquele ponto.

Note que essa poda não prejudica a obtenção da solução ótima do problema, pois somente enumerações com maior numero de inimigos que a menor já encontrada são descartadas.

Na implementação desse paradigma, foi feita uma busca em profundidade recursiva. Para o caso de um nível, o pseudocódigo ilustra a implementação utilizada:

```
//num_foes: n de inimigos da tentativa corrente
//menor_num_foes: solucao com menor n de inimigos ja encontrada
//solucao: variavel boleana
//menor_conjunto_inimigos: menor conjunto de inimigos ja encontrada
//conjunto_inimigos: conjunto de inimigos da solucao corrente

forcabruta(nivel, inimigos, num_foes, &menor_num_foes, &solucao,
&menor_conjunto_inimigos, conjunto_inimigos )

  for(int i=inimigos.size()-1; i >=0 ;i--)
    subnivel = nivel - inimigos[i];
    teste (...)
    if(teste)
        if(subnivel==0)
            if(solucao == false || num_foes+1 < menor_num_foes)
                menor_num_foes = num_foes+1;
                menor_conjunto_inimigos = conjunto_inimigos;
                menor_conjunto_inimigos[i]++;
                solucao=true;
        else //continua a procurar a solucao
            vector<int> novo_conjunto_inimigos = conjunto_inimigos;
            novo_conjunto_inimigos[i]+=1;
            forcabruta(subnivel,inimigos,num_foes+1,menor_num_foes,
                solucao, menor_conjunto_inimigos, novo_conjunto_inimigo
```

em que a variável *teste* é *true* se o subnível é maior ou igual a 0 e a quantidade de inimigos dessa solução parcial ainda é menor que o menor conjunto de inimigos já obtido. Para os demais níveis, repete-se esse mesmo processo.

3.1.1. Análise Complexidade

Complexidade de tempo: Para uma fase de nível D, considerando i a quantidade de inimigos e m a profundidade máxima da árvore, tem-se que a complexidade é exponencial no pior caso $O(i^m)$, sendo que m é no máximo D (todos os inimigos selecionados tem *ranking*=1). O melhor caso é quando a profundidade é igual a 1, com $O(i)$. Como o arquivo possui n níveis os quais deseja-se obter o menor subconjunto de inimigos, o pior caso fica $O(ni^m)$ e o melhor caso $O(ni)$.

Complexidade de espaço: Por não precisar manter todos os nós na memória, a complexidade é linear $O(im)$. Para todos os n níveis tem-se $O(nim)$, onde i é a quantidade de inimigos e m a profundidade máxima.

3.2. Algoritmo Guloso

Um algoritmo guloso é aquele que sempre faz a escolha que parece ser a melhor no momento em questão. Ou seja, faz uma escolha localmente ótima na esperança que leve a uma solução globalmente ótima. Além disso, é uma característica dos algoritmos gulosos nunca se arrepender, isto é, feita uma escolha ele nunca volta atrás.

Resolvendo o problema utilizando uma abordagem gulosa, considerando uma fase de nível de dificuldade D, queremos escolher um inimigo I_1 tal que $D - ranking(I_1)$ seja o mais próximo de 0 possível.

Considerando o nível de dificuldade de uma única fase, o pseudocógido abaixo ilustra a abordagem gulosa utilizada para encontrar o menor subconjunto de inimigos:

```
guloso()
    vector<int> inimigos_selecionados;
    int num_inimigos = 0;
    int j = length(foes);
    while(j >=0 & nivel != 0)
        if(nivel>=foes[j])
            aux = floor(nivel, foes[j]);
            num_inimigos = num_inimigos+aux;
            nivel = nivel - aux*foes[j];
            for(int k=1; k<=aux;k++)
                inimigos_selecionados.push_back(foes[j]);
        j--;
```

Essa mesma lógica pode ser usada para obter o menor conjunto de inimigos para todas as fases, acrescentando para isso somente um *for* externo, para variar o valor do nível de dificuldade das fases. Observe que nessa abordagem é essencial que os inimigos estejam ordenados em ordem crescente de *ranking* (no while ele percorre o vetor do final para o inicio, ou seja, o maior *ranking* para o menor).

3.2.1. Análise Complexidade

Complexidade de tempo: Considerando uma fase, tem-se que o algoritmo é linear e seu pior caso é O(i), onde *i* é a quantidade de inimigos disponíveis para colocar na fase. No melhor caso, é O(1), onde basta somente olhar o primeiro inimigo da lista disponível. A complexidade para todas as *n* fases no pior caso seria então O(ni) e no melhor caso O(n), onde *n* é a quantidade de fases.

Complexidade de espaço: Para cada fase é guardada na memória somente um vetor com os inimigos selecionados, que tem complexidade no pior caso O(D), sendo D o nível da fase (vetor de 1's de tamanho igual ao nível da fase) e no melhor caso O(1). Para todas as fases conjuntamente, o pior caso é igual à $O(max(D))$.

3.2.2. Otimalidade

Como dito na Seção 2, o problema de encontrar o menor conjunto de inimigos para uma dada fase é similar ao problema do Troco Mínimo. E utilizando uma estratégia gulosa para resolver problemas dessa classe, nem sempre a solução ótima pode ser encontrada. Para exemplificar, considere uma fase de nível igual a 20 e o seguinte conjunto de inimigos disponíveis I={2,3,10,15}. Neste caso, a solução ótima seria escolher $I_{sol-otima}$ ={10,10}, mas o algoritmo guloso retornaria I_{sol} ={15,3,2}.

Portanto, a otimalidade das soluções utilizando uma abordagem gulosa nesse tipo de problema depende dos *rankings* dos inimigos disponíveis. Em [5] é mostrado que a otimalidade só pode ser garantida na classe de problemas similares ao Problema do Troco Mínimo se as moedas fazem parte de um Sistema Canônico. No trabalho prático em questão, os *rankings* dos inimigos devem seguir esse sistema para que a otimalidade seja garantida.

3.3. Programação Dinâmica

Utilizando Programação Dinâmica tem-se que a resolução é divida em problema principal e os subproblemas. No trabalho proposto, considerando uma única fase do jogo, o problema principal é obter o menor conjunto de inimigos cuja soma dos *rankings* seja igual ao nível D da fase; e os subproblemas são semelhantes ao problema principal, que é descobrir o menor conjunto de inimigos para uma fase d, para todo $0 \leq d \leq D$.

3.3.1. Subestrutura ótima, sobreposição de subproblemas e otimalidade

Para compreender a abordagem utilizada, considere o pseudocógido abaixo para uma única fase:

```
prog_dinamica(nivel)
    vector<int> n_fases(nivel+ 1,10000); // vetor para memorizacao
    vector<int> inimigos(nivel+ 1); // vetor para guardar os inimigos
    n_fases[0] = 0;
    for(int i = 1; i < nivel + 1; i++)
        for(int j =0;j<foes.size(); j++)
```

```
if(i >= foes[j] && (1 + n_fases[i-foes[j]]) < n_fases[i])
  n_fases[i] = 1 + n_fases[i - foes[j]];
  inimigos[i] = j;
```

Note que o algoritmo memoriza a melhor solução para todos os níveis d abaixo do nível D que se deseja. Além disso, considerando I o conjunto com todos os inimigos, a equação de recorrência é:

$$\text{OPT(d)} = \begin{cases} 0 & , d = 0 \\ min\{OPT(d-i)\} + 1 & , \forall i \in Ie(d-i) \geq 0 \end{cases}$$

Observa-se que essa abordagem gera uma subestrutura ótima. Como prova, considere que I é uma solução ótima para a fase de nível D. Então I'=I-i, sendo i um dos inimigos da solução ótima, é uma solução ótima para o subproblema da fase de nível D-i. Agora suponha que I' não é ótimo para esse subproblema, e que existe uma outra solução $I^* \neq I'$. Se I^* é uma solução ótima, ela precisa conter um número menor de inimigos que I', e se combinarmos I^* com i, nos obtemos a solução ótima para o nível D, mas contradiz a suposição original que I é a solução ótima, portanto é um absurdo, provando que a subestrutura é ótima.

A sobreposição de subproblemas ocorre pois um mesmo subproblema pode ser "chamado"mais de uma vez. Por exemplo, considere um subproblema de nível 9 e com inimigos de *ranking* I={2,3,5}. Caso seja selecionado o inimigo de *ranking*=5, temos agora que calcular o novo subproblema de nível 4; mas caso sejam selecionados os inimigos de ranking *ranking*=3 e depois o de *ranking*=2, caímos novamente no subproblema de nível 4, mostrando a sobreposição.

A solução ótima desse problema é construída em cima dos pressupostos da subestrutura ser ótima e da sobreposição de subproblemas. Como essas duas características foram verificadas válidas, pode-se dizer que esse algoritmo sempre nos leva a solução ótima.

3.3.2. Análise da Complexidade

Complexidade de tempo Para uma fase, o pior caso e o melhor caso é O(iN), onde *i* é a quantidade de inimigos disponíveis e N é o nível da fase. Para todas as *n* fases, a complexidade é O(n i max(N)).

Complexidade de espaço Para uma fase, o espaço requerido é O(N) para a memorização das soluções, e para todas as fases é O(max{N}),pois o algoritmo considerado não guarda a memorização de um nível para o outro.

4. Análise Experimental

Considere a complexidade de tempo dos algoritmos no pior caso:

1. Força Bruta: $O(n * i^m)$;
2. Algoritmo Guloso: $O(i * n)$;
3. Programação Dinâmica: $O(n * i * max(N))$,

em que *i* é a quantidade de inimigos, *n* é a quantidade de fases, *N* é o valor máximo do nível de uma fase e *m* é a profundidade máxima da árvore (pior caso igual a N).

Dada as complexidades, o desempenho dos programas serão comparados de três formas: quanto ao aumento do número de inimigos, o aumento do número de fases e o aumento do valor máximo da fase.

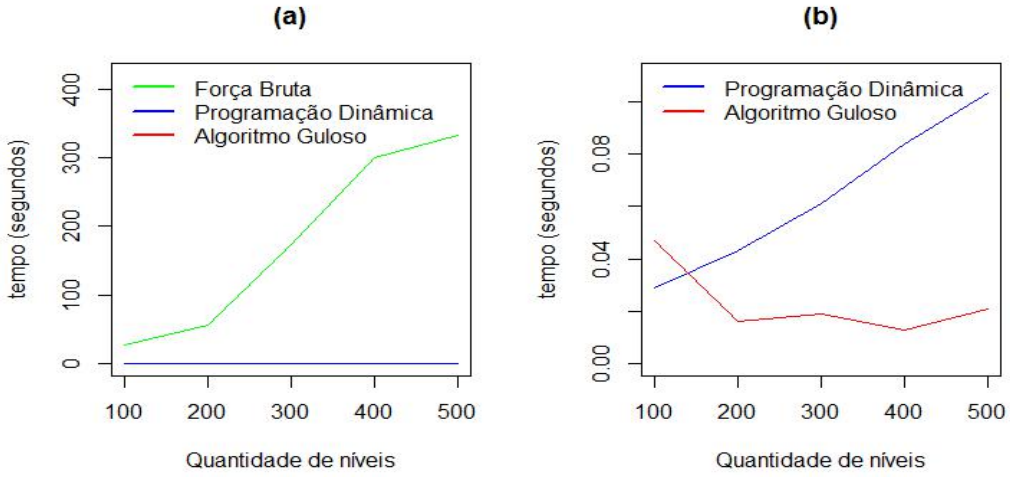

Figura 1. Análise Experimental - Comparação dos algoritmos com o aumento das fases.

Na Figura 1, é mostrada a comparação do tempo dos algoritmos com o aumento da quantidade de fases. Observa-se nitidamente que o tempo demandado pelo algoritmo Força Bruta quando n aumenta tem um crescimento muito maior que o Programação Dinâmica e o Guloso (Figura 1(a)). Nesse exemplo optou-se por escolher um caso em que o Força Bruta não explodia para ter uma melhor comparação com os demais. Na Figura 1(b) tem-se os mesmos dados da Figura 1(a), mas sem o algoritmo Força Bruta. Embora a diferença seja pequena em comparação com o Força Bruta, observa-se que o aumento de tempo do algoritmo com Programação Dinâmica é maior que o Guloso quando a quantidade de fases cresce. Os parâmetros utilizado foram $max(N) = 500$, $i = 15$ e *ranking* máximo do inimigo 100.

A Figura 2 (a) e (b) compara o tempo gasto pelos algoritmos quando a quantidade de inimigos aumenta. Em (a), é possível ver que o crescimento do algoritmo Força Bruta é exponencial e muito maior que os outros algoritmos, que não demandam nem 0.01s para rodar os mesmos arquivos de entrada. Os parâmetros utilizados foram $max(N) = 300$, $n = 50$ e *ranking* máximo do inimigo 50. Na Figura 2(a), tem-se uma comparação somente entre o algoritmo Guloso e o de Programação Dinâmica. Nesse caso, os parâmetros utilizados foram $max(N) = 30000$, $n = 10000$ e *ranking* máximo do inimigo 30000, e dessa forma, foi possível observar a diferença de tempo gasta entre as duas abordagens. A utilização de um alto valor de $max(N)$ evidência a diferença entre o desempenho do Algoritmo Guloso (que não depende desse valor) e do Programação Dinâmica (que depende desse valor).

A partir da Figura 2(c) é possível observar como o valor máximo de N interfere na complexidade do algoritmo que usa Força Bruta mesmo para entradas relativamente pequenas, que no caso foram $i = 10$, $max(i) = 50$ e $n = 1$. Enquanto as outras abordagens levam no máximo 0.015 segundos para computar o output, o força bruta para

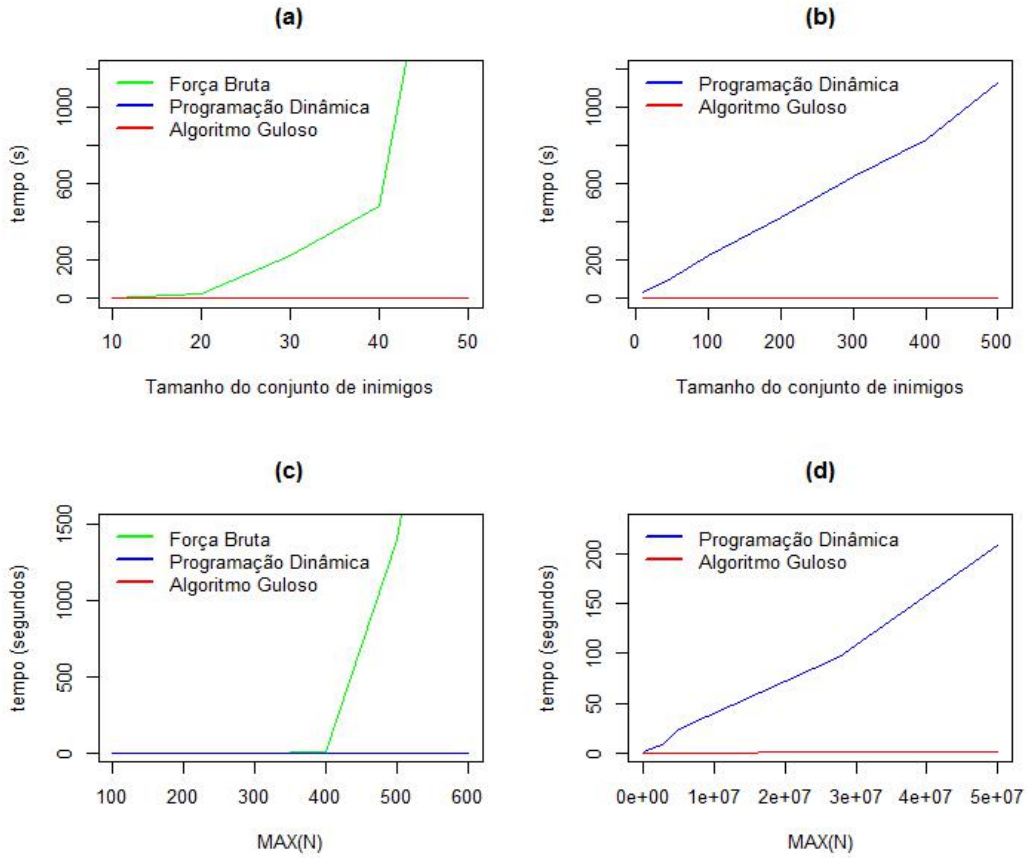

Figura 2. Análise Experimental - Comparação dos algoritmos com o aumento do número de inimigos e com o aumento do valor máximo de N.

$max(N) = 500$ leva 23 minutos e para $max(N) = 600$ ele leva mais de 70 minutos, deixando claro sua complexidade exponencial. Na Figura 2(d) é feita uma comparação somente entre o algoritmo guloso e o de programação dinâmica utilizando números maiores, no caso, $i = 50$, $max(i) = 50$, $n = 5$ e com os valores de $max(N)$ variando de 50000 a 50000000. Nessa figura nota-se que o algoritmo de programação dinâmica, embora menos que o de força bruta, também sofre bastante influência do valor $max(N)$, enquanto o tempo do guloso tem um tempo praticamente constante.

Os resultados obtidos na análise experimental estão de acordo com as complexidades obtidas. Isto é, o algoritmo Força Bruta, que tem complexidade exponencial é o mais lento; depois o algoritmo Programação Dinâmica, cuja complexidade depende ainda do valor do nível máximo; e o mais rápido é o Algoritmo Guloso, que infelizmente não gera sempre a solução ótima.

A análise experimental das Figuras 1, 2(b),2(c) e 2(d) foram feitas no computador "jumento"da sala 3017, com 3.9GB de memória RAM, processador Intel(R) Core(TM)2 Duo CPU E7200 @2.53 GHz, com o sistema operacional Ubuntu 10.04. A análise presente na Figura 2(a) foi feita no notebook Samsung, com 3.7GB de memória RAM, processador Inter(R) Core(TM) i3-3110M CPU @2.400 GHz*4, com o sistema operacional Ubuntu 14.04 LTS. Os conjuntos de níveis e inimigos dentro dos limites estabelecidos

foram selecionados aleatoriamente no software livre R.

5. Uso e compilação

Conforme descrito no enunciado do trabalho prático, os programas podem ser compilados através de *bash compile_α.sh* e executado com *bash execute_α.sh arquivo1.txt arquivo2.txt*, em que α pode ser *fb*, *ag* ou *pd* e arquivo1.txt tem o conjunto de inimigos disponíveis e arquivo2.txt os níveis das fases.

O programa foi testado no computador "jumento"disponível do laboratório da pós graduação na sala 3017. O compilador utilizado foi o g++ (no computador testado o gcc não estava disponível).

Os programas fazem isso das bibliotecas include iostream, fstream, vector, iterator e algorithm.

6. Conclusão

Nesse relatório do Trabalho Prático 1 foram apresentadas três soluções distintas para o problema proposto, que é obter o menor conjunto de inimigos para várias fases, tais que a soma dos *rankings* dos inimigos do conjunto seja igual o nível de dificuldade da fase. Na Seção 2 foi discutida a modelagem desse problema, que é semelhante ao Problema do Troco Mínimo (Coin Change). Na Seção 3 foram descritas as soluções apresentadas, mostrando que somente a abordagem via Força Bruta e Programação Dinâmica levam a soluções ótimas. Na Seção 4 foi feita a análise experimental, que confirmou a complexidade calculada para os algoritmos desenvolvidos e na Seção 5 tem uma breve descrição de como fazer o uso dos algoritmos.

De maneira geral, pode-se dizer que o esse trabalho proposto proporcionou uma boa experiência em como os paradigmas da computação podem ser usados para resolver problemas reais. Além disso, esse trabalho pôde mostrar que nem sempre a abordagem mais simplista (no caso, a gulosa) nos leva a solução ótima do problema.

Comparando as três soluções obtidas, tem-se que o Algoritmo Guloso é o mais rápido, porém como já dito, nem sempre gera as soluções ótimas; o algoritmo Força Bruta encontra a solução ótima, mas pode levar um tempo infinito para concluir essa tarefa; a melhor opção se torna o Programação Dinâmica, que embora possa levar um tempo razoável para certas entradas, resolve o problema de maneira ótima e bem mais rápido que o Força Bruta.

Referências

[1] Sobre o problema do Troco Mínimo - 1
https://en.wikipedia.org/wiki/Change-making_problem/#Simple_dynamic_programming

[2] Sobre o problema do Troco Mínimo - 2
http://crbonilha.com/pt/o-problema-do-troco/

[3] Sobre algoritmos gulosos
http://marathoncode.blogspot.com.br/2012/05/algoritmos-gulosos.html

[4] Sobre programação dinâmica na resolução do problema do Troco Mínimo
http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/

[5] CAI, J. 2009, *Canonical Coin Systems for Change-Making Problems*