O caminho dos castelos

Lucas Dimer Justo

Engenharia de Software — PUCRS

20 de maio de 2019

Resumo

Este relatório apresenta o processo de desenvolvimento de um algoritmo para resolver o segundo problema proposto na disciplina Algoritmos e Estruturas de Dados II do 3º semestre do curso de Engenharia de Software. O conde Sibério, governante da cidadezinha de Fragoletto no século 13, usou as riquezas da cidade para montar um poderoso exército com o objetivo de dominar o maior número de castelos possíveis na região. Dito isto, o problema consiste em ajudá-lo em sua campanha, dizendo qual exatamente é o maior número de castelos dominável, e por qual caminho dominá-los, seguindo um conjunto especifico de regras.

Introdução

O conde Sibério, governante da cidadezinha de Fragoletto, usou as riquezas da mesma para montar um poderoso exército e sair conquistando castelos pela região. Fomos contratados por ele, para ajudá-lo em sua campanha, dizendo qual o maior número de castelos conquistáveis e qual caminho seguir para conquistar este número de castelos seguindo as regras de combate medieval que seguem abaixo:

1. O conde Sibério ataca os castelos de um em um, seguindo pelas estradas até o castelo seguinte;
2. Ele nunca divide seu exército, pois não é esperto ter dois exércitos mais fracos em vez de um mais forte;
3. O conde só ataca castelos quando tem soldados suficientes para conquistar o castelo e mantê-lo ocupado (mantê-lo ocupado significa deixar 50 soldados de guarnição);
4. Atacar um castelo é dureza e se ele está ocupado por n soldados inimigos o exército atacante perderá 2n soldados;
5. Ao mover- se para atacar outro castelo o conde deixa para trás uma guarnição de 50 soldados;

Com os pergaminhos em mãos e as estratégias bem compreendidas, devemos analisar vários casos distintos, e sobre eles dizer, qual o maior número de castelos conquistáveis com

os soldados disponíveis. Claro que como todo grande poderoso, o conde também quer saber como as respostas foram obtidas e o tempo que isso custou.

Decifrando os pergaminhos

Para termos como ajudar na campanha do conde, precisávamos de informações privilegiadas sobre a região e seus castelos. Então, o conde providenciou isso enviando espiões para analisar a área. Eles nos retornaram com pergaminhos que vem no seguinte formato.

1. Na primeira linha há 3 números, o primeiro referente a quantos soldados temos disponíveis no castelo do conde (castelo 0), o segundo número referente a quantos castelos (X) há na região e o terceiro referente a quantas estradas (Y).
2. Em todas as X seguintes linhas vêm 2 números, o número do castelo (identificação) e quantos soldados protegem tal castelo.
3. Em todas as Y seguintes linhas (que vem após as linhas X) também vêm 2 números, sinalizando pela identificação dos castelos, quais deles tem estradas para outros quais.

Exemplo de pergaminho:

1020 5 8

1 102

2 103

3 20

4 500

5 50

0 1

0 2

1 3

1 4

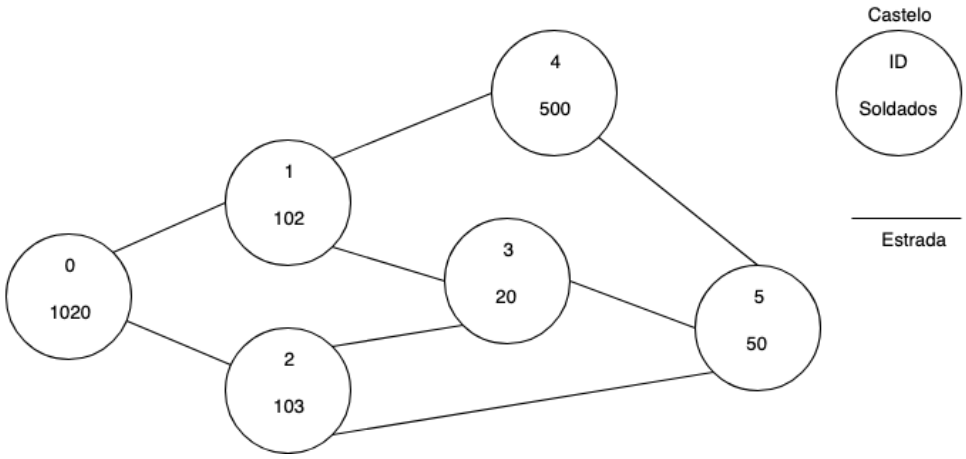
2 3

2 5

3 5

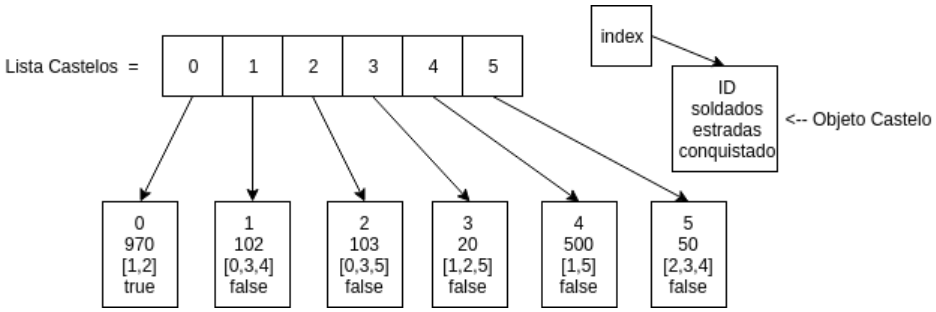
4 5

Resolvemos representar este emaranhado de informações na forma de um grafo, com os castelos sendo nodos e as estradas suas arestas. Este pergaminho do exemplo gera o seguinte grafo:



Transição do pergaminho para estrutura

Para representar o grafo em uma linguagem de programação (neste caso Java), foi criada uma lista (baseada em vetor) de objetos Castelo. E como mostrado na imagem abaixo, a lista (que é basicamente uma estrutura de índices que guardam referências para um objeto) se aproveita do fato do castelo ter um ID representado por número inteiro (mesmo tipo do índice), para guardar os castelos nos índices respectivos aos seus IDs, assim tornando as operações de add, remove e get O(1).



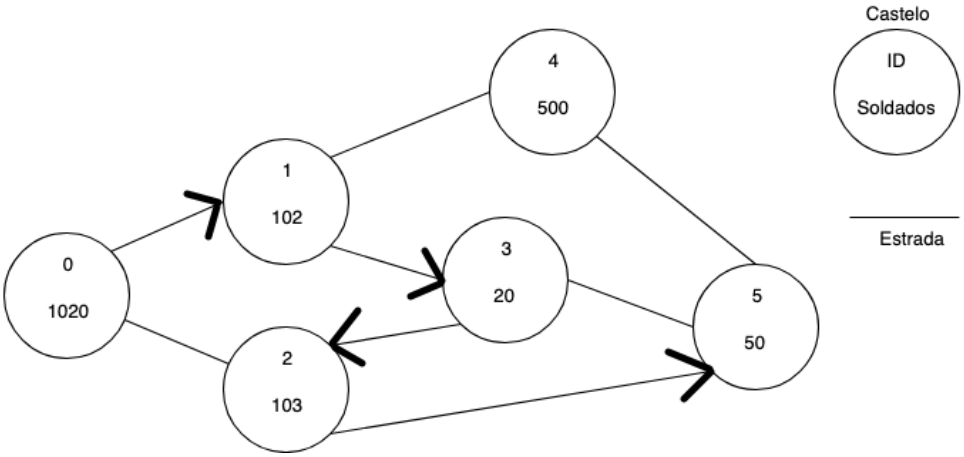
Mas como transformamos os números descritos nos pergaminhos, nesta estrutura organizada sobre a qual o computador pode guardar dados e processar algoritmos? Para entender melhor como isso acontece, além das imagens acima e da sessão "introdução", há este pseudo código que traduz o que é feito pelo código fonte em Java, que faz este processo de transição de pergaminho para estrutura (grafo):

```
Lista castelos = []

constroiGrafo(nomeDoPergaminho):

   in = Scanner(nomeDoPergaminho)

   //os disponiveis tem -50 porque precisamos deixar 50 soldados de
guarnição no castelo 0.

   soldadosDisponiveis = in.next()-50

   //construtor Castelo = (id, soldados, conquistado)

   fragoletto = new Castelo(0,soldadosDisponiveis,true)

   castelos.add(fragoletto)

   NCastelos = in.next()

   NEstradas = in.next()

   Enquanto(NCastelos+NEstradas > 0):

      Se(NCastelos > 0):

         atual = new Castelo (in.next(),in.next(),false)

         castelos.add(atual)

         NCastelos--

      Se(NEstradas > 0):

         ID1 = in.next()

         ID2 = in.next()

         castelos.get(ID1).estradas.add(ID2)

         castelos.get(ID2).estradas.add(ID1)

          NEstradas--
```

Conquistando os castelos

Agora que temos tudo definido e bem construído, precisamos desenvolver um algoritmo que consiga, baseado nesta estrutura, dizer qual o maior número possível de castelos conquistáveis e qual o primeiro caminho encontrado para alcançar este número. A solução desenvolvida foi: para cada nodo do grafo (tendo como início o castelo 0, castelo de Fragoletto) ver quais nodos podem ser conquistados a partir do atual (seguindo as regras descritas na introdução deste relatório) e assim por diante recursivamente, sempre checando e anotando qual o maior número de castelos conquistados, e o caminho que levou a esta conquista, a cada chamada recursiva.

Um caminho com mais castelos conquistados, encontrado no exemplo das sessões "Decifrando os pergaminhos" e "Transição do pergaminho para estrutura", usando o algoritmo descrito no parágrafo anterior, seria o da imagem abaixo:



1. Começamos no castelo 0, com uma lista de estradas = [1,2]. Para cada estrada se o castelo ao qual ela leva não está conquistado, mas é conquistável. Vamos para ele recursivamente repetir o processo e marcamos o castelo como conquistado.
2. A partir do passo 1, chegamos ao castelo 1, com lista de estradas = [0,3,4]. A estrada que leva ao castelo 0, nos levaria para um castelo já conquistado (o que não pode acontecer), então vamos checar a próxima estrada (3), não está conquistado, mas é conquistável. Marca o 3 como conquistado e chama recursivamente para ele.
3. A partir do passo 2, chegamos ao castelo 3, com lista de estradas = [1,2,5]. A estrada que leva ao castelo 1, nos levaria para um castelo já conquistado, então vamos checar a próxima (2), o castelo 2 não está conquistado, mas é conquistável. Marca o 2 como conquistado e chama recursivamente para ele.
4. A partir do passo 3, chegamos ao castelo 2, com lista de estradas = [0,3,5]. As estradas que levam aos castelos 0 e 3, nos levariam a castelos já conquistados, então vamos checar a próxima (5), o castelo 5 não está conquistado, mas é conquistável. Marca o 5 como conquistado e chama recursivamente para ele.
5. A partir do passo 4, chegamos ao castelo 5, com lista de estradas = [2,3,4]. As estradas que levam aos castelos 2 e 3, nos levariam a castelos já conquistados, então vamos checar a próxima (4), o castelo 4 não pode ser conquistado porque não há soldados o suficiente para isso. Então como já checamos todas as estradas do castelo 5 para esta situação, checamos se os 4 castelos conquistados até aqui, foram o maior número de castelos já conquistados até agora e, caso seja, guardamos a lista que representa este caminho ([0,1,3,2,5]) como a lista que levou ao caminho com mais castelos conquistados. Este processo de armazenar os castelos conquistados em lista e checar se foi o maior caminho até agora aconteceu em todos os passos (só foi ocultado para não complicar muito a explicação).

O que aconteceria depois destes 5 passos que foram usados de exemplo para chegar na resposta do caso do pergaminho de exemplo? A saída da recursão do castelo 5, onde ele seria marcado como não conquistado novamente e retirado da lista de conquistados, assim voltando

ao castelo 2 e continuando a checagem de estradas por lá, para prosseguir o algoritmo vendo se haveriam mais testes ou uma outra saída de recursão voltando para o castelo 3 e assim por diante. Deste jeito (checando todos os casos possíveis), garantimos que, mesmo que o resultado demore ele com certeza virá correto. A resposta verdadeira deste caso de exemplo do pergaminho é realmente essa, 4 castelos pelo caminho [0,1,3,2,5], mas caso continuássemos a acompanhar o algoritmo, como o computador fará, encontraríamos mais 2 caminhos que levam a 4 castelos ([0,1,3,5,2] e [0,2,5,3,1]), que são desconsiderados porque já foi encontrado um caminho com a mesma quantidade de castelos, que só seria alterado caso encontrássemos um caminho que conquiste 5 castelos. Também vale lembrar que este algoritmo pode entrar com recursão no mesmo castelo mais de uma vez, como por exemplo, quando ele entra no castelo 5, ao acabar a recursão do castelo 5 volta para o castelo 2, ao acabar a recursão do castelo 2 volta para o castelo 3, e do castelo 3 entra novamente no castelo 5 para desta vez conquistar o castelo 2 e formar o caminho [0,1,3,5,2]. Assim, constatamos que neste exemplo, entramos no castelo 5 por dois castelos diferentes (3 e 2).

Toda essa explicação do algoritmo foi implementada em Java e testada em 13 casos diferentes. O pseudo código que simplifica a implementação em Java segue abaixo:

```
soldados = castelos.get(0).soldados
caminho = []
maxCastelos = 0
calculaMaxCastelos(Castelo C, caminhoAtual=[0]):
   para cada vizinho NÃO conquistado V de C:
      soldadosNecessarios = (v.soldados*2)+50
       Se(soldadosNecessarios<soldados):
          v.conquistado=true
          caminhoAtual.add(v.id)
          soldados-=soldadosNecessarios
          Se(caminhoAtual.size()>maxCastelos):
              acheiCaminho(caminhoAtual)
              maxCastelos=caminhoAtual.size()
          calculaMaxCastelos(v,caminhoAtual)
   //saindo da recursão
   c.conquistado=false
   caminhoAtual.remove(c.id)
   soldados+=(c.soldados*2)+50
```

```
//este é o método acheiCaminho(caminhoAtual) que atribui o maior caminho encontrado até então, ao caminhoResposta.

acheiCaminho(caminhoAtual)

   caminho.clear

   para cada int id em caminhoAtual:

      caminho.add(id)
```

Resultados

Como dito anteriormente, após implementar o algoritmo em Java, testamos ele em 13 casos de teste diferentes. E os resultados foram estes da tabela abaixo:

nome do caso	maxCastelos	caminho	tempo(ms)	tempo(s) (piso)
meu	4	[0, 1, 3, 2, 5]	0ms	0s
enunciado	1	[0, 5]	0ms	0s
caso30	7	[0, 1, 21, 15, 23, 24, 26, 4]	13ms	0s
caso33	9	[0, 8, 1, 18, 11, 26, 28, 13, 32, 15]	52ms	0s
caso36	10	[0, 2, 22, 17, 25, 10, 35, 13, 36, 27, 31]	98ms	0s
caso39	10	[0, 13, 6, 8, 12, 19, 20, 11, 4, 32, 24]	297ms	0s
caso42	11	[0, 15, 14, 10, 11, 20, 38, 33, 34, 8, 30, 28]	773ms	0s
caso45	12	[0, 15, 14, 20, 22, 23, 13, 5, 3, 24, 21, 39, 33]	1140ms	1s
caso48	13	[0, 4, 3, 2, 41, 36, 24, 8, 46, 32, 26, 45, 40, 12]	7384ms	7s
caso51	14	[0, 17, 8, 6, 33, 11, 51, 1, 31, 15, 38, 20, 18, 5, 50]	17793ms	17s
caso54	15	[0, 8, 17, 18, 19, 1, 50, 4, 13, 40, 32, 39, 29, 30, 5, 3]	65375ms	65s
caso57	16	[0, 3, 57, 53, 49, 12, 20, 30, 2, 34, 27, 1, 6, 42, 43, 50, 56]	344309ms	344s
caso60	16	[0, 54, 20, 16, 24, 34, 50, 38, 51, 23, 22, 40, 30, 18, 53, 15, 9]	97177ms	97s

A partir dos resultados, podemos constatar que, o algoritmo se comporta de maneira confiável por testar todos os casos. Mas em contrapartida, apresenta demora para obter os resultados pelo mesmo motivo. Notamos também, ao observar os casos 57 e 60, que o aumento na demora não se dá somente quando o número de nodos e arestas crescem (por que o caso60 tem mais nodos e mais arestas que o caso57) e mesmo assim o 57 demora consideravelmente mais. Isso nos leva a pensar em novas variáveis, como por exemplo, os números de soldados (tantos os soldados disponíveis para nós, quanto os soldados inimigos) e o jeito que as estradas

estão distribuídas. Pois se há muitos soldados disponíveis e poucos inimigos, vamos avançando mais no grafo, assim como se houver mais estradas chegando e saindo de castelos com poucos inimigos (por que são fáceis de conquistar). Quanto mais longe chegarmos no grafo, mais testamos e mais teremos que testar.

Conclusão

Com o desenvolvimento de uma solução, para um problema no qual encontrei dificuldades, aprendi que nem sempre podemos desenvolver o algoritmo mais otimizado no tempo que temos. E as vezes é melhor entregar uma solução que resolve o problema do conde Sibério mesmo sendo lenta, e assim manter a cabeça em cima dos ombros, do que não entregar uma solução e perder a cabeça. Também aprendi que, quando os problemas vão evoluindo e consequentemente ficando mais difíceis, explicações com trechos de códigos bem comentados deixam de ser o suficiente para a compreensão do nosso algoritmo, e então precisamos achar novas formas de explicar, como por exemplo, imagens seguidas de uma sequência de passos que simulam o algoritmo como se fosse um teste de mesa.