Analisando jogos de civilizações antigas

Abraão de Morais Duarte, Giovani de Souza Fetzner

Universidade do Vale do Rio dos Sinos (UNISINOS)
São Leopoldo – RS – Brazil

duarteabraao@edu.unisinos.br, giovanisf@edu.unisinos.br

Resumo. *Este artigo aborda o desenvolvimento de algoritmos para resolver o problema da análise de jogos de civilizações antigas. O problema basicamente consiste em determinar, dentro de um conjunto C com placas que possuem dois valores, A e B, uma forma de organizar as placas de modo que a soma de todos os valores de A seja igual à soma de todos os valores de B. O presente artigo compreende duas abordagens para o problema: heurísticas aleatorizadas e força bruta. Além disso, este estudo inclui uma análise da complexidade das soluções propostas.*

1. Introdução

O problema da análise de jogos de civilizações antigas propõe um interessante desafio de equilíbrio matemático. Nesses jogos, dois jogadores registram suas pontuações em placas divididas ao meio, cada lado correspondendo a um dos participantes. O objetivo é reorganizar as placas de modo que as somas das pontuações dos jogadores sejam iguais. Para isso, é permitido inverter as placas ou, se necessário, descartar uma delas. No entanto, a solução deve ser maximizada, devemos buscar a maior soma possível, e, em casos onde múltiplas opções de descarte resultem na mesma soma, deve-se priorizar a exclusão da placa de menor valor. Esse problema combina elementos lógicos e matemáticos, como comparações e análise combinatória, tornando-se um caso interessante para a resolução através de algoritmos.

Neste artigo, propomos algumas abordagens para resolver o desafio dos jogos de civilizações antigas, com uma análise baseada na complexidade assintótica e na qualidade das soluções em casos onde heurísticas são aplicadas.

2. Primeira solução

Para uma solução inicial, optamos por aplicar heurísticas aleatorizadas, visto que uma solução exata exigiria a avaliação de todas as possíveis combinações de placas, incluindo suas somas e a consideração de descartar uma delas. Tal abordagem resultaria, provavelmente, em um algoritmo de complexidade exponencial. Além disso, desenvolver uma solução mais otimizada com tempo de execução polinomial necessitaria de uma análise mais aprofundada e, possivelmente, poderia se revelar inexistente.

O algoritmo desta primeira solução inicia com a entrada de um array de Placas, sendo Placa uma classe com atributos A e B, sendo A e B números inteiros. O primeiro procedimento realizado é a soma geral de todos os elementos do array e a verificação se este valor é ímpar. O valor da soma geral ser ímpar implica que é impossível ter um combinação de placas que tenha a soma de todos os valores de A igual a soma de todos os valores de B, já que a soma de dois valores iguais, que é o objetivo de desafio, deve gerar um valor par.

Em seguida após a verificação da soma o algoritmo entra em um laço de repetição onde ele executará $n \cdot k$, sendo n o tamanho do campeonato e k uma constante definida pelo desenvolvedor. Quanto maior for a constante k mais laços o algoritmo irá executar, sendo assim mais chances de chegar perto da solução ótima.

Dentro do laço, é gerado um número aleatório que determina a quantidade de placas a serem invertidas. Com base nesse número, é criada de forma aleatória uma lista contendo os índices das placas que serão alteradas. Em seguida, o algoritmo executa os laços de repetição necessários para inverter as placas conforme a lista gerada. Após as inversões, o algoritmo soma todos os valores de A e de B e realiza verificações para garantir que as somas sejam iguais. Caso as somas coincidam, a variável solução é atualizada.

Esse processo de soma e verificação é realizado duas vezes: uma vez com a exclusão de uma placa aleatória e outra sem qualquer descarte. Como mencionado anteriormente, caso a soma geral seja ímpar, o algoritmo não executará os laços e verificações sem o descarte de placas. Ele se limitará a executar apenas os laços que envolvem o descarte, garantindo assim que não haja processamento desnecessário.

Este algoritmo possui, no máximo, dois laços de repetição aninhados: um laço externo, que executa $k \cdot n$ iterações, e outros internos, que executam n vezes, sendo n o número de placas. Assim, ao calcular a complexidade, baseado em uma análise assintótica da primeira solução, obtém-se $O(kn^2)$.

3. Segunda solução

O algoritmo da segunda solução começa com a mesma entrada que a primeira e realiza as mesmas verificações iniciais. Em seguida, adota a abordagem de força bruta que explora todas as combinações possíveis das placas utilizando recursão.

Inicialmente, o método gera combinações no formato original das placas, sem realizar inversões. Caso nenhuma solução válida seja encontrada, o algoritmo retrocede para a última chamada recursiva e tenta novamente, desta vez invertendo a configuração. Com isso, as primeiras placas a serem invertidas correspondem às mais distantes da raiz da árvore de estados.

A função “encontrarCombinações”, apresentada no pseudocódigo a seguir, é a responsável por realizar as chamadas recursivas que percorrem todas as combinações possíveis de placas, resultando em uma complexidade de $O(2^n)$. Conforme mencionado, o algoritmo é executado em dois casos com base na verificação inicial: o primeiro, sem descartes, e o segundo, com descartes. O caso com descartes é processado em todas as situações, enquanto o caso sem descartes ocorre apenas quando a soma geral é par.

O processo de descarte é implementado dentro de um laço que, a cada iteração, considera o descarte de uma placa e chama novamente a função recursiva para $n - 1$ placas. Caso nenhuma solução seja encontrada, a placa descartada é substituída por outra, garantindo que apenas um descarte seja mantido por vez. Isso resulta em uma complexidade de $O(m \cdot 2^{n-1})$ para a segunda solução, sendo $m = n$ para o pior caso.

3.1. Pseudocódigo

Segue abaixo (Figura 1) o pseudocódigo do segundo algoritmo desenvolvido.

```
1.  função segundaSolucao(campeonato[ ] : Placa){
2.    int somaGeral ← getSomaGeralAB(campeonato)
3.
4.    se (somaGeral % 2 == 0) então {
5.      int resultado ← encontraCombinações(campeonato, 0, 0, 0)
6.      se (resultado > 0) então
7.        retorna "solução com todas as placas"
8.      retorna "solução descartando a menor placa"
9.    }
10.   Lista placasDescartadas ← [ ]
11.   enquanto (tamanho(placasDescartadas) < tamanho(campeonato)) faça {
12.     Placa menorPlaca ← encontrarMenorPlaca(campeonato, placasDescartadas)
13.     placasDescartadas.adiciona(menorPlaca)
14.     Placa[] restante ← campeonato sem menorPlaca
15.     int resultado ← encontraCombinações(restante, 0, 0, 0)
16.       se (resultado > 0) então
17.         retorna "solução descartando menorPlaca"
18.    }
19.    retorna "impossivel"
20. }
```

Figura 1: Pseudocódigo da segunda solução

4. Resultados

Esta seção apresenta os resultados da análise de complexidade das duas soluções propostas, além da avaliação de assertividade do algoritmo da primeira solução, que utiliza heurísticas aleatorizadas. A análise da complexidade de cada solução baseia-se na comparação do aumento do número de operações realizadas por cada algoritmo em função do crescimento do tamanho da entrada de dados. A Figura 2 ilustra graficamente o crescimento de cada algoritmo, permitindo uma comparação visual.

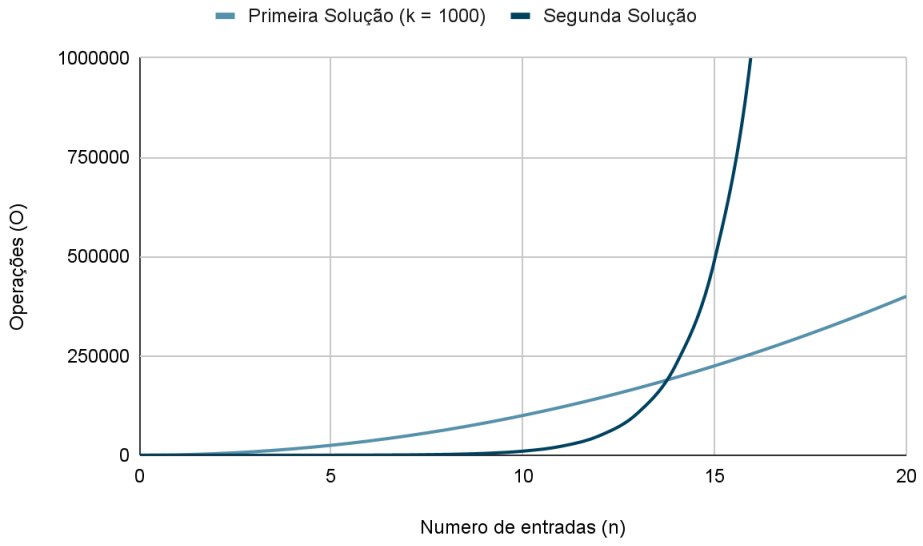

Figura 2: Comparação entre comportamento dos algoritmos propostos.

A partir do gráfico, observa-se que a primeira solução, devido a uma constante $k = 1000$, realiza um número significativamente maior de operações para entradas menores, em comparação à segunda solução. No entanto, à medida que o tamanho da entrada aumenta, a segunda solução, com complexidade exponencial, acaba ultrapassando a primeira em número de operações. Isso torna a primeira solução mais eficiente para entradas maiores ou iguais a 14. Contudo, por ser um algoritmo aleatorizado, é importante considerar também a qualidade das soluções obtidas.

Considerando que existem três possíveis respostas para cada instância do problema — impossível, algum valor sem descarte e algum valor com descarte —, a solução ótima é sempre o maior valor, caso não seja impossível. O algoritmo apresenta uma assertividade de 100% nos casos classificados como impossíveis, pois, independente do valor de k, ele não encontrará uma solução. No entanto, para os casos em que há solução, o algoritmo frequentemente qualifica erroneamente como impossível diversas situações quando k é um valor pequeno, na casa das dezenas. Esse tipo de erro, ao afirmar que não há solução quando ela existe, é bastante significativo, mostrando uma má qualidade na resposta.

Porém, à medida que aumentamos a constante k para valores na ordem de milhares, o algoritmo passa a fornecer respostas para praticamente todos os casos. Embora nem sempre sejam soluções ótimas, os resultados ficam muito próximos da solução ideal, com uma taxa de erro inferior a 1% de diferença do valor esperado.

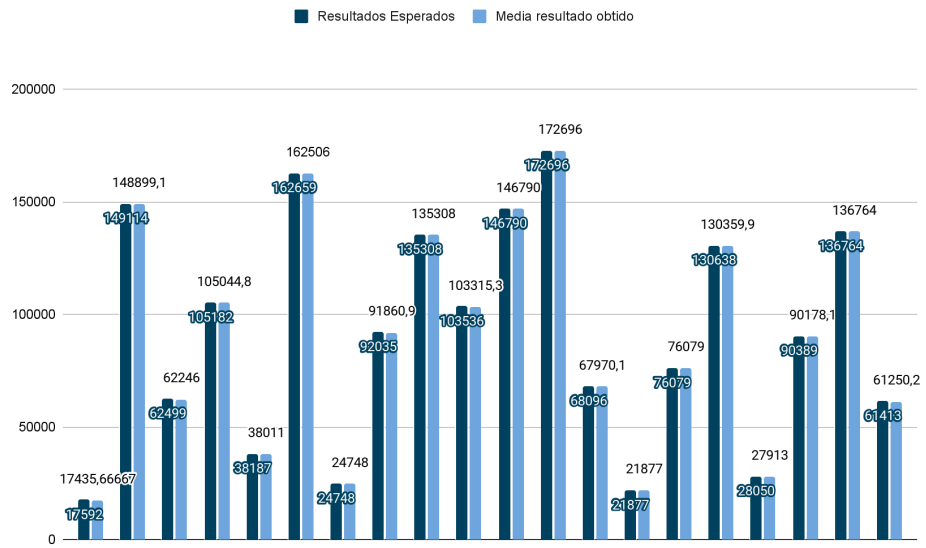

Figura 3: Gráfico de resultados esperados vs resultados obtidos numa bateria de 10 testes com k = 500.

5. Conclusão

Conclui-se, portanto, que a segunda solução oferece uma resolução ideal para o problema proposto. No entanto, sua complexidade exponencial a torna inviável para instâncias maiores do problema. Por outro lado, a primeira solução, apesar de não garantir sempre a solução ótima, apresenta uma complexidade polinomial e fornece respostas muito próximas do ideal, sendo mais adequada para entradas maiores.