Arizona Jones

Pedro Vanzella[1]

[1]Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681 - Porto Alegre / RS / Brasil

pedro@pedrovanzella.com

Resumo. *Uma solução para o problema do Arizona Jones no Templo de Tel Dor é proposta.*

1. Introdução

O problema a ser resolvido consiste em encontrar, dentro de um conjunto de números, a maior seqüência crescente de números que, em base 6, têm somente um dígito de diferença entre um e outro.

Por exemplo, no conjunto $782\ 229\ 446\ 527\ 874\ 19\ 829\ 70\ 830\ 992\ 430\ 649$, a maior seqüência é $649\ 829\ 830$.

Para isto, os números são organizados em um grafo e o maior caminho é encontrado. Como veremos na seção 4.7, isto é uma operação que roda em tempo linear, graças a algumas propriedades interessantes deste grafo.

2. Estruturas de Dados

Vamos representar o conjunto de números como um grafo. Isto nos permitirá ligar os nodos que formam seqüências válidas e encontrar a maior seqüência, procurando pelo maior caminho.

Para representar internamente o grafo, utilizamos três dicionários: um dicionário para nodos, um para arestas e um para o peso dos nodos. Este último será explicado em detalhe nas seções 4.7 e 4.5.

O dicionário de nodos tem uma propriedade interessante: seu índice é o número em base 10, como foi lido do arquivo; seu conteúdo é a representação em base 6 do mesmo número, como um array de dígitos, para facilitar a comparação. Esta representação é guardada no array de modo a evitar o recálculo da conversão de base, trocando processamento por memória.

Este grafo é dirigido e acíclico. Isto fica óbvio quando pensamos nos parâmetros do problema: se, para criar uma aresta, precisamos ligar um nodo a alguém maior que ele, não podemos ter ciclos (i.e. A não pode ser maior que B ao mesmo tempo que B é maior que A). Como veremos na seção 4.7, isto é muito bom para nós.

Podemos ver na Figura 3 um grafo gerado de acordo com o exemplo dado na seção 1

782
229
446
527
874
19
829
70
830
992
430
649

Figura 1. Arquivo de Entrada

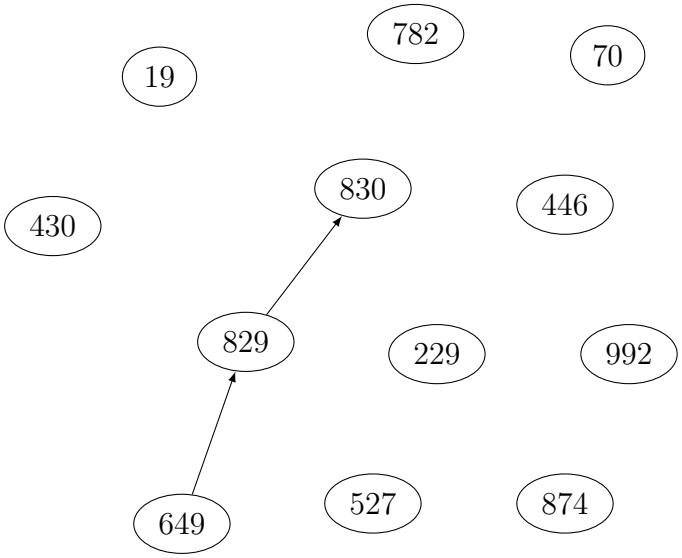

Figura 2. Exemplo de grafo gerado

Figura 3. Entrada e grafo

3. Entrada

Como entrada temos arquivos que listam os números, em base 10, um por linha. Por exemplo, o arquivo da Figura 3

Uma seqüência mais complicada, como a seguinte, gera um grafo muito mais complicado, como o da Figura 4

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

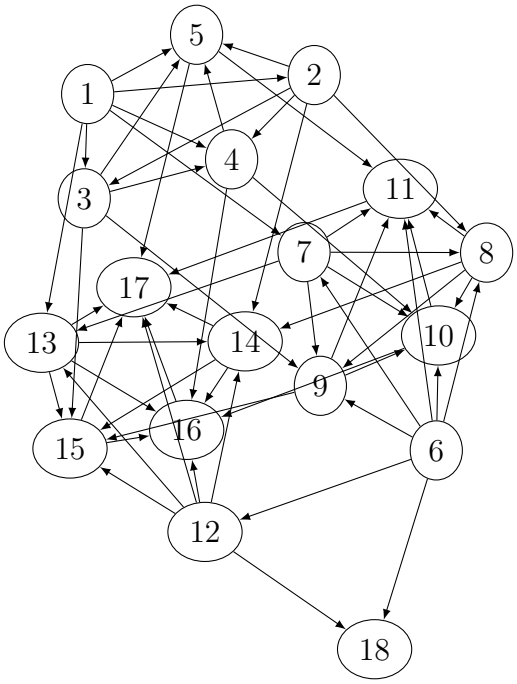

Figura 4. Exemplo de grafo mais complexo

4. Algoritmos

Para resolver o problema, dividiu-se o programa em diferentes funções, para facilitar a leitura e explicação.

4.1. Criar Nodos

Esta função é trivial: lê-se cada linha do arquivo e adiciona-se ela ao dicionário de nodos

```
criar_nodos(arquivo):
    para cada linha l em arquivo:
        nodos[l] = []
        pesos[l] = 0
```

Onde [] é um *array* vazio. Este array, nesta implementação, conterá a representação em base 6 do seu índice correspondente. Isto é feito para memorizar a conversão, já que este valor poderá ser testado várias vezes pelo algoritmo descrito na sessão 4.3.

Veja que também inicializamos uma posição no dicionário de pesos. Este peso será atualizado pelo algoritmo descrito na seção 4.5.

Dentre as vantagens de utilizar-se um dicionário para armazenar os nodos, está o fato de que não precisamos nos preocupar com a existência ou não de um elemento ao inserí-lo - o dicionário cuida disso, internamente, e não teremos elementos repetidos. Além disso, o acesso a um elemento é $O(1)$, ou algo muito próximo disso. Isto faz com que o custo deste algoritmo seja $O(n)$, onde n é o número de linhas no arquivo de entrada.

4.2. Criar Arestas

O algoritmo de criar arestas itera pelos nodos, conectando aqueles que podem formar uma seqüência válida para o problema.

```
criar_arestas():
    para todo u em nodos:
        para todo v maior que u em nodos:
            se comparar(u, v):
                arestas[u,v] = Verdadeiro
```

Há uma chamada para o algoritmo comparar(), que é explicado na seção 4.3.

Não é necessário testar todos os nodos contra todos, no entanto. Como somente seqüências crescentes podem ser geradas, filtramos v para ser maior que u. Ainda assim, este algoritmo roda em $O(n^2)$, sendo n o número de nodos no grafo.

O conteúdo do dicionário de arestas não é importante. O valor dele pode, portanto, ser qualquer um. Aqui setamos para Verdadeiro por conveniência durante os testes de existência de aresta.

Note também que, em arestas[u,v], u,v deve ser a concatenação de u, uma vírgula e v, formando uma string como 123,124. Algumas linguagens podem permitir outros formatos, mas este é mais garantido para a maioria delas.

4.3. Comparar

Esta função compara dois índices, u e v, retornando um booleano que indica se u pode ser ligado a v através de uma aresta.

```
comparar(u, v):
    se v < u:
        retorna Falso

    bsu = base6(u)
    bsv = base6(v)

    digitos_diferentes = 0

    se len(bsv) - len(bsu) > 1:
        retorna Falso

    bsu = pad(bsu, lenght(bsv))

    para i de lenght(bsv) a 0:
        se bsu[i] != bsv[i]:
            digitos_diferentes += 1
        se digitos_diferentes > 1
            retorna Falso

    retorna Verdadeiro
```

Aqui vemos uma chamada para a função base6(), que será explicada na seção 4.4.

Há também uma chamada para pad(). Esta função apenas adiciona zeros à esquerda de seu primeiro argumento até que ele fique com o tamanho de seu segundo argu-

mento. Isto nos permite testar números que têm uma quantidade de dígitos diferentes.

Como um número pode diferir de outro em um dígito quando representados em base 6, temos duas opções: todos os dígitos são iguais, mas um dígito novo é adicionado à esquerda do menor (*e.g.* 231_6 e 1231_6) ou a quantidade de dígitos é a mesma, mas apenas um dígito é diferente entre ambos os números (*e.g.* 231_6 e 232_6).

O algoritmo também verifica se v é maior que u. Este é um teste de sanidade apenas, já que o algoritmo criar_arestas() (seção 4.2) só chama o algoritmo comparar() para vs maiores que u.

Outra verificação é feita antes de se testar todos os dígitos, comparando os tamanhos dos dois *arrays* de números em base 6. Se a diferença é maior que 1, há garantidamente mais que um dígito diferente também, e estes nodos não podem ser conectados. Esta verificação é interessante porque pode rodar em tempo constante, dependendo de como a linguagem representa *arrays*. Em uma linguagem que não guarde o tamanho do *array* como propriedade do mesmo, não há vantagens, no entanto.

Após estas verificações, itera-se pelo maior *array* (que é garantidamente bsv, já que v é maior que u), comparando dígito a dígito, do final para o início, com bsu. Caso mais que um dígito diferente seja encontrado, a verificação pára e retorna-se Falso, para indicar que os nodos não podem ser conectados por uma aresta.

Esta função é $O(u)$, mas como u é tão pequeno, comparado, pro exemplo, ao número de nodos no grafo, podemos desconsiderar seu custo nos cálculos posteriores, como se fosse $O(1)$.

4.4. Converter para Base 6

A conversão para base 6 em si não é importante - ela pode ser realizada por uma função de alguma biblioteca, ou manualmente com matemática simples. O que é importante aqui é a memorização do resultado.

```
base6(num):
    se nodos[num] == []:
        nodos[num] = Conversor.ConverteBase(6, num)
    retorna nodos[num]
```

Vemos aqui uma chamada para uma biblioteca arbitrária que faz a conversão e retorna um *array* de inteiros, dígito a dígito. Mas isto somente é feito caso nodos[num] seja um *array* vazio ([]). Ele não ser vazio significa que base6() já foi chamada nele, e seu conteúdo é a representação em base 6 de sua chave.

4.5. Calcular Tamanho do Caminho

O dicionário de pesos tem como chaves os nomes dos nodos, e como valores o tamanho do maior caminho que vai até ele. Para calcular este valor, olha-se todos os vizinhos que chegam em um nodo u. O valor do nodo u é o valor do seu maior vizinho, mais um. Caso ninguém chegue em u, seu valor é zero. Isto será crucial para encontrarmos o maior caminho na seção 4.7.

```
calcular_tamanho_caminho(u):
    para todo v em vizinhos_que_chegam(u):
```

```
        se pesos[v] == 0:
            pesos[v] = calcular_tamanho_caminho(v) + 1
        se pesos[v] > pesos[u]:
            pesos[u] = pesos[v] + 1

    retorna pesos[u]
```

O algoritmo vizinhos_que_chegam() é explicado na seção 4.6.

4.6. Vizinhos que Chegam

Este algoritmo tem como função encontrar todos os nodos que têm uma aresta que os conectam em u.

```
vizinhos_que_chegam(u):
    lista = []
    para todo v em nodos:
        se existe_aresta(v, u):
            lista.add(v)

    retorna lista
```

Observe que existe_aresta() é chamado com os parâmetros invertidos, isto é, se existe uma aresta de v para u. Uma lista de nodos é montada e retornada, contendo todos os nodos que possuem uma aresta para u.

4.7. Achar Maior Caminho

A idéia desde algoritmo é simples: já que já temos o tamanho do maior caminho que passa em cada nodo (dado pelo algoritmo descrito na seção 4.5, podemos usar isto para encontrar o maior caminho, nodo a nodo.

```
achar_maior_caminho():
    maior_peso = encontrar_maior_peso(pesos)
    maior_caminho = []
    maior_caminho.add(maior_peso)

    vizinhos = vizinhos_que_chegam(maior_peso)

    enquanto lenght(vizinhos) > 0:
        maior_peso = encontrar_maior_peso(vizinhos)
        maior_caminho.add(maior_peso)
        vizinhos = vizinhos_que_chegam(maior_peso)

    retorna maior_caminho
```

Primeiro pegamos o nodo de maior peso (o algoritmo encontrar_maior_peso() é descrito na seção 4.8) e adicionamos ele a uma lista vazia. Para os fins deste algoritmo, vamos considerar que List.add() adiciona um elemento ao início da lista. Caso ele adicione ao fim, basta inverter a ordem dos elementos ao fim do algoritmo.

Em seguida, escolhe-se dentre os nodos que chegam neste que acabamos de encontrar o de maior peso. Isto é, pode haver vários caminhos que passam por um nodo, mas estamos escolhendo aquele com mais nodos. Adiciona-se este nodo ao início da lista e repete-se o processo para ele, até que não haja nodos com arestas apontando para ele.

4.8. Encontrar Maior Peso

Este algoritmo é muito simples: itera-se por uma lista de nodos, achando o de maior peso entre eles. Os detalhes de implementação podem variar um pouco de linguagem para linguagem. Por exemplo, uma linguagem que permita filtrar dicionários pode tornar este algoritmo mais fácil de se implementar; uma linguagem mais fortemente tipada pode apresentar algum outro obstáculo.

```
encontrar_maior_peso(pesos):
    mais_pesado = pesos.primeiro()

    para todo p em pesos:
        se pesos[p] > pesos[mais_pesado]:
            mais_pesado = pesos[p]

    retorna mais_pesado
```

De qualquer forma, o único requisito é que este algoritmo receba uma lista de apenas alguns nodos (que são filtrados pelo algoritmo descrito na seção 4.7 utilizando o algoritmo da seção 4.6), e retornar apenas um nodo - o de maior peso.

4.9. Chamada Principal

Agora que temos o algoritmo bem dividido, podemos chamá-lo em poucas etapas:

```
main():
    criar_nodos(arquivo)
    criar_arestas()
    para todo u em nodos:
    calcular_tamanho_caminho(u)

    maior_caminho = achar_maior_caminho()
    imprime(maior_caminho, lenght(maior_caminho))
```

Primeiro cria-se os nodos a partir de um arquivo, como visto na seção 4.1. Este arquivo deve ser passado para o programa de alguma maneira, como pelos parâmetros de linha de comando.

Em seguida, criamos as arestas. O algoritmo descrito na seção 4.2 utiliza-se dos nodos já criados para isto.

O próximo passo é iterar por todos os nodos, calculando o tamanho do maior caminho até aquele nodo, como visto na seção 4.5.

Uma vez feito isso, podemos facilmente encontrar o maior caminho. O algorimto da seção 4.7 retorna uma lista. Guardamos esta para podermos imprimí-la, junto de seu tamanho, dando a resposta do problema.

5. Resultados

Na Tabela 1 podemos ver as saídas do algoritmo, junto do seu tempo de execução. A primeira e a terceira coluna são especialmente importantes para a validação do resultado, listando respectivamente o arquivo de entrada e a maior seqüência encontrada nele.

Como vimos na seção 4.2, temos um algoritmo que roda em $O(n^2)$, enquanto outros rodam em tempo linear ou melhor. Isto dá uma complexidade total de $O(n^2)$ para a solução.

Podemos ver na Figura 5 um gráfico relacionando tamanho da entrada com tempo de execução. De fato, isto corrobora o resultado de $O(n^2)$ - o gráfico se assemelha a uma parábola.

É importante notar que, enquanto o gargalo se encontra no algoritmo criar_arestas(), descrito na seção 4.2, este apenas está preparando o terreno para o problema ser resolvido. A solução do problema, dada na seção 4.7, é linear. Ainda assim, sem este primeiro passo, não seria possível resolver o problema. Este é um bom candidato para otimizações futuras.

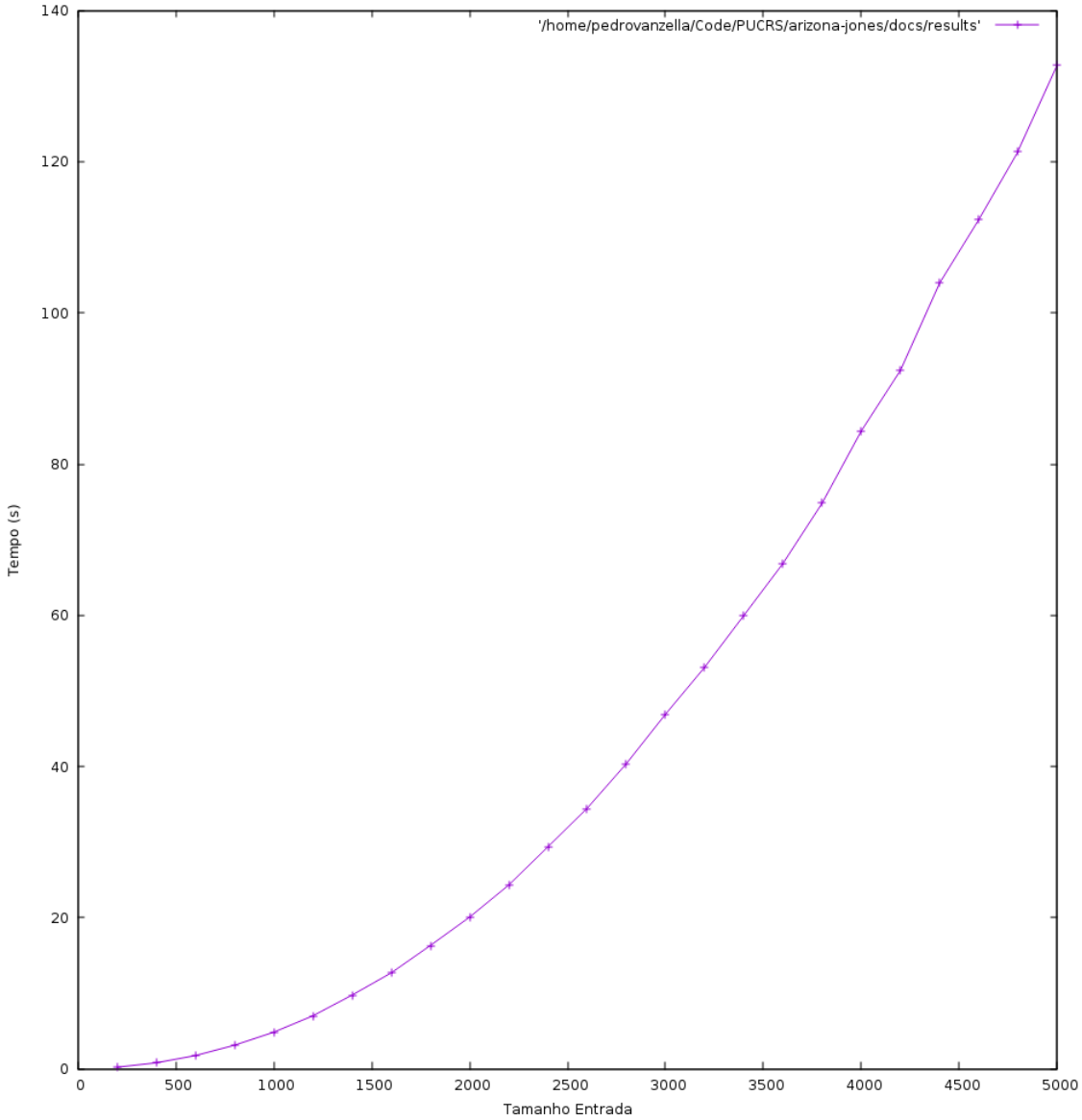

Figura 5. Gráfico de Resultados

Tabela 1. Resultados

Entrada	Tempo (s)	Caminho	Tamanho
teste0200b	0.23	7268 7484	2
teste0400b	0.80	10478 10490 11138 11144 11147	5
teste0600b	1.76	4981 5053 5071 7663 7735 7771	6
teste0800b	3.14	1134 1135 1136 1139 1283 6467 14243	7
teste1000b	4.86	582 1014 1015 1033 1249 16801 16804 16840	8
teste1200b	7.02	6057 13833 13905 13977 13989	5
teste1400b	9.76	2682 5274 5275 6355 14131 14134 14206 14242	8
teste1600b	12.71	6487 14263 15343 15361 15505 15506 15507	7
teste1800b	16.33	17676 17694 18990 18993 18994 18995	6
teste2000b	20.08	14018 14019 14037 14039	4
teste2200b	24.32	10765 11629 14221 14233 14234 14246	6
teste2400b	29.39	17971 18079 19375 19376 19377 19395	6
teste2600b	34.45	6262 14038	2
teste2800b	40.35	1514 5402 5438 13214 13244 13245 13246 14110 14182 14218	10
teste3000b	46.90	2593 5185 5186 6266 6272 6275 6299 7595 7775	9
teste3200b	53.11	5659 5875 7171 7387 7459 7471 7472 7508 15284 15296 15299	11
teste3400b	59.87	5833 6049 7345 7357 7501 7717 7729 7735 15511 15515	10
teste3600b	66.86	648 660 661 697 1129 1237 1255 1291 3883 3886 19438	11
teste3800b	74.84	2592 5184 6480 6486 7566 7640 7748 7766 7772 15548 15550 15551	13
teste4000b	84.38	2809 3025 3241 3673 7561 7705 7711 15487 15493 15505 15506 15507 15543	13
teste4200b	92.41	13176 13212 13428 13429 13861 15157 15175 15319 15331 15334 15335	11
teste4400b	103.96	2605 3253 3685 7573 7579 7591 7592 7700 15476 15478 15550	11
teste4600b	112.35	4105 5401 6049 6265 6271 6277 6295 6367 6475 14251 14255	11
teste4800b	121.35	72 288 324 336 372 1020 7500 15276 15277 15295 15299	11
teste5000b	132.73	1512 1944 7128 7344 7380 7488 7491 15267 15285 15291 15297 15299	12