

## Problemas intratáveis

---

Queremos examinar ainda uma outra dimensão relativa à noção básica de algoritmo. Suponha que temos um problema a ser resolvido,  $P$ , e que já encontramos um algoritmo,  $A$ , para resolver  $P$ . Suponha que já codificamos e compilamos um programa para o algoritmo  $A$ , resultando num código objeto  $R$ .

Então, dada uma instância válida de dados de entrada,  $E$ , podemos executar  $R$  sobre  $E$  e obter a resposta esperada. Correto? Bem, nem sempre. Na prática, o algoritmo  $A$ , e o seu código objeto  $R$ , podem ser *muito ineficientes*, em termos de tempo de execução ou uso de memória. Assim, embora correto, o código  $R$  pode levar dias, ou anos, para produzir a resposta correta. Ou pode precisar de milhares de giga bytes de memória. Se este for o caso, embora correto, o algoritmo  $A$  é imprestável.

Qual a alternativa?

Sempre poderemos criar outro algoritmo,  $B$ , que também resolve corretamente o problema original,  $P$ , e que seja eficiente no uso de tempo e de memória.

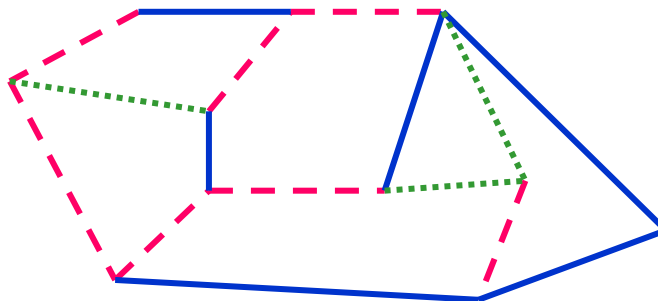
Surpresa: *nem sempre* isso é possível! Existem problemas para os quais *não existem* algoritmos *eficientes*. Significa dizer que há (grandes) instâncias desses problemas que não poderão ser resolvidas por programas reais, executando em computadores reais. Problemas deste tipo são ditos *intratáveis*.

O que é pior, já foi mostrado que muitos problemas de grande interesse prático são intratáveis. Porém, note que embora não consigamos resolver certas “grandes” instâncias desses problemas, para instâncias de tamanho razoável o algoritmo pode muito bem obter a resposta em tempo adequado, e com uso de memória tolerável. Também não estamos dizendo que todas as grandes instâncias são difíceis de resolver. Apenas que algumas delas são intratáveis.

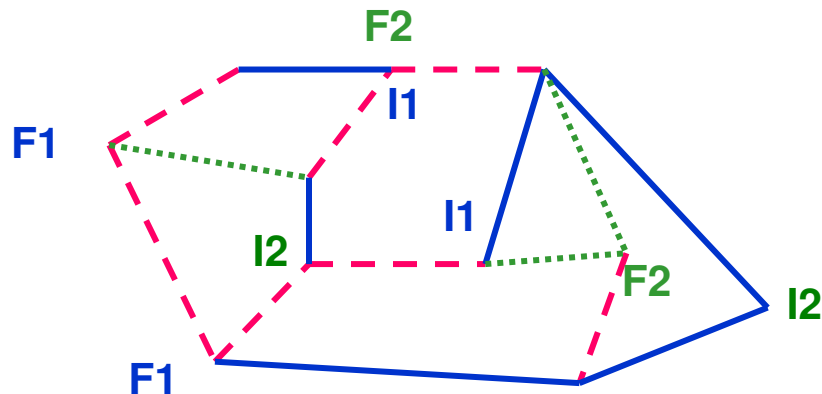
## Exemplo de problema intratável

---

Como um exemplo prático, considere o seguinte problema. Temos um mapa rodoviário, como o mostrado abaixo:



onde linhas de tipos diferentes indicam cores diferentes. Temos dois jogadores,  $J_1$  e  $J_2$ . Para cada um deles, estão marcadas no mapa algumas posições iniciais e finais, como mostrado na figura a seguir.



onde I1 e I2 assinalam as posições iniciais de J1 e J2, respectivamente, e F1 e F2 indicam as posições finais desses mesmos jogadores. Os jogadores se movem alternadamente, e o objetivo é alcançar uma das posições finais, sujeito às seguintes regras:

1. Cada jogador executa um movimento por vez.
2. J1 inicia e, na sequência, ambos se alternam.
3. Num único movimento, um jogador pode percorrer qualquer caminho, desde que os trechos individuais sejam da mesma cor e o caminho não passe por intersecções já ocupadas (pelo outro jogador, ou por si próprio).
4. Um movimento deve sempre partir de uma posição ocupada pelo jogador e não pode terminar em uma posição já ocupada por algum deles.
5. Vence o jogador que primeiro atingir uma posição final.

No caso indicado pelo mapa anterior, não é difícil determinar que J1 *sempre ganha*. Isto é, J1 tem uma estratégia de jogo que sempre o leva a ganhar, não importa quão bons sejam os movimentos de J2. Então J1 tem uma *estratégia vencedora*.

O problema que queremos resolver é o seguinte: dado o mapa, junto com as posições iniciais e finais, determine se J1 tem uma estratégia vencedora. Ou seja, queremos *um algoritmo* que, tomando como entrada o mapa e as posições, determine se J1 pode sempre ganhar.

De fato, esse algoritmo é até bem simples. Observe que podemos explorar, de forma sistemática, todas as possíveis seqüências de movimentos, começando com J1. Isto porque existe apenas um número finito de possibilidades, já que o mapa é finito. Com um pouco mais de detalhes, podemos associar um dentre três rótulos a cada intersecção do mapa: 1, se está ocupada por J1; 2 se está ocupada por J2; ou 0 se está livre. Como o número de intersecções é finito, o número de possíveis rotulações é também finito. E um movimento nada mais é que passar de uma configuração de rótulos para outra.

Portanto, o algoritmo que explora todas as possibilidades determinaria se J1 sempre consegue ganhar. Há um porém: o número de diferentes configurações é gigantesco, e o algoritmo teria que explorar elas todas, ou boa parte delas. Corre o risco de ser um algoritmo imprestável. De fato, esse é o caso, pode-se provar.

Para ser um pouco mais preciso, vamos considerar algumas maneiras de se medir o uso de recursos por algoritmos.

## Medidas de complexidade

Precisamos de um parâmetro numérico,  $N$ , que mede o “tamanho” dos dados de entrada. Em geral, podemos usar o número de bits necessários para se representar os dados de entrada.

No caso do exemplo anterior, usaremos o número de intersecções como o parâmetro que designa o tamanho dos dados de entrada. Portanto, o tamanho da particular instância mostrada no exemplo anterior é 10. Poderíamos também utilizar como medida de complexidade o número de ligações. Nesse caso, não é muito importante qual dessas opções adotemos.

Isto posto, precisamos de uma função  $f(N)$  que indique o número de passos básicos que o algoritmo deve executar ao tratar uma instância de tamanho  $N$ . Por passos básicos poderíamos entender as instruções básicas na linguagem de programação na qual o algoritmo foi codificado. Aqui há uma sutileza que vale a pena mencionar. Para um mesmo valor de  $N$ , o problema pode ter mais de uma instância válida. No caso do exemplo, certamente há mais de um mapa com 10 intersecções. Então, se numa instância de tamanho  $N$  o algoritmo executa  $M1$  passos, e em outra de mesmo tamanho ele executa em  $M2$  passos, quanto valeria  $f(N)$ ? Resolvemos esse problema dizendo que  $f(N)$  indica o número de passos do algoritmo numa *situação de pior caso*, dentre *todas* as instâncias de entrada de tamanho  $N$  do problema. Ou seja, o algoritmo nunca vai executar mais de  $f(N)$  instruções quando iniciado em uma *instância qualquer* de tamanho  $N$ .

E no caso do exemplo, qual seria a função  $f(N)$ ? E agora vem a surpresa:  $f(N) = 2^N$  (aqui o símbolo  $^$  indica potenciação). Então, para o caso do exemplo,  $f(10) = 2^{10} = 1024$ . O algoritmo precisaria de cerca de 1024 passos para terminar em algumas instâncias de tamanho 10.

Mas onde está a surpresa? Suponha que o algoritmo está executando num computador que é capaz de executar um milhão de instruções básicas por segundo. Tomando  $f(N) = 2^N$ , a tabela abaixo indica quanto tempo o algoritmo precisaria para resolver o problema para vários tamanhos de instâncias, considerando sempre o pior caso para cada valor de  $N$ :

$n$	10	20	50	60	100	
$f(n)=2^n$	1 ms	1 s	35.7 anos	3.000 anos	+ de 400 trilhões de anos	

Então, para mapas com até 20 cidades, o tempo de resposta é bem razoável: no máximo um segundo. Porém para mapas com 50 intersecções, um aumento não muito grande sobre as 20 anteriores, o algoritmo necessitaria de mais de 35 *anos* para terminar. Totalmente impraticável. Mapas com 100 intersecções poderiam chegar ao absurdo indicado na última coluna da tabela. Com certeza este algoritmo é impraticável.

Bem, mas esse computador não é muito rápido. Consideremos, então, um computador 10 mil vezes mais rápido. Os novos tempos de execução estão indicados na tabela abaixo:

Algoritmos: ... adianta executá-los?

n	50	60	100
$f(n)=2^n$	1,29 dias	3 anos	+ 40 bilhões de séculos

Então, para mapas com até 50 cidades, o tempo e resposta passou a ser razoável. Porém, alguns mapas com 100 intersecções – e casos reais podem ter ainda muito mais – continuam simplesmente fora do alcance.

Em geral, dizemos que um algoritmo *eficiente* é aquele cuja função de medida de complexidade é um polinômio. Tal como  $f(N) = N$ , ou  $f(N) = N^2$ . Algoritmos cuja função de medida de complexidade é exponencial, ou pior, são *ineficientes*. Portanto, o algoritmo simplório, que testa todas as possibilidades, é um algoritmo ineficiente para resolver o problema do mapa rodoviário.

Na tabela abaixo podemos observar a diferença enorme entre algoritmos eficientes e ineficientes, em relação ao tamanho da instância que conseguem resolver:

N \ f(N)	10	20	50	100	200
$N^2$	0,1 ms	0,4ms	2,5ms	10ms	40s
$N^5$	0,1s	3,2s	5,2m	2,8h	3,7dias
$2^N$	1ms	1s	35,7anos	400 trilhões de séculos	núm. de séculos tem 45 dígitos
$N^N$	2,8s	3,3 trilhões de anos	núm. de séculos tem 70 dígitos	núm. de séculos tem 185 dígitos	núm. de séculos tem 445 dígitos

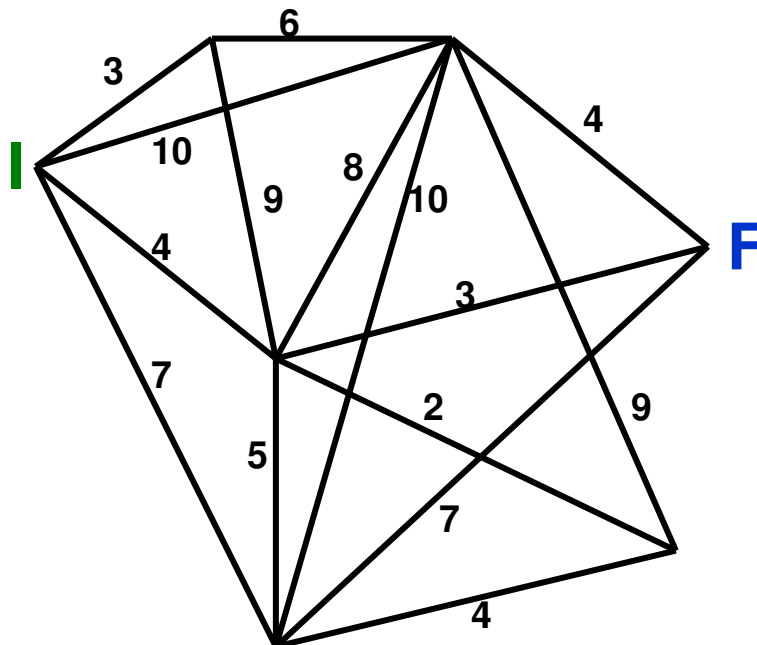
A pior surpresa vem agora: *nenhum outro* algoritmo que corretamente resolve o problema do mapa rodoviário é mais eficiente que o algoritmo simplório que testa todas as possibilidades! Esse problema simplesmente *não admite* algoritmo eficiente!

## Um exemplo prático importante

O problema anterior é sabidamente intratável. Existem muitos outros problemas interessantes e de importância prática que são dessa mesma natureza. Dado um problema  $P$ , como saber se é do tipo tratável ou não? Ou seja, como saber se existe um algoritmo eficiente para resolvê-lo? Note que só estamos buscando determinar se algum algoritmo existe. Não queremos, a priori, descobrir um tal algoritmo, se existir.

Ao colocar essa questão, deparamo-nos com um dos grandes pontos de desconhecimento em ciências da computação: simplesmente *ainda não sabemos* se muitos problemas de grande interesse prático são tratáveis ou não! Trata-se de uma das maiores questões em aberto em ciências da computação. A situação é ainda mais curiosa. Vamos chamar de NP essa classe de problemas, para os quais não sabemos se são tratáveis ou não – a nomenclatura é histórica. Sabemos que existem alguns problemas centrais na classe NP que são muito especiais, no sentido de que, se descobrirmos um algoritmo eficiente para resolver *qualquer um deles*, então poderemos obter algoritmos eficientes para *resolver todos* os problemas da classe NP! Essa é a grande questão: será que algum desses problemas especiais em NP tem um algoritmo eficiente para resolvê-lo? Bastaria concentrar os esforços de pesquisa em tentar obter algoritmos eficientes para um desses problemas especiais da classe NP. No entanto, mesmo após anos de pesquisa, envolvendo as mentes mais brilhantes em ciências da computação, ainda não conseguimos uma resposta definitiva para essa questão.

Vamos ilustrar com um dos principais problemas em NP, de grande interesse prático. Trata-se do “problema do caixeiro viajante”. Como entrada, recebemos um mapa de cidades conectadas por rodovias, como na figura abaixo:



Assumimos que as rodovias são todas bidirecionais. Os números que aparecem perto de cada rodovia, na figura, indicam o custo de se trafegar pela rodovia, entre as duas cidades

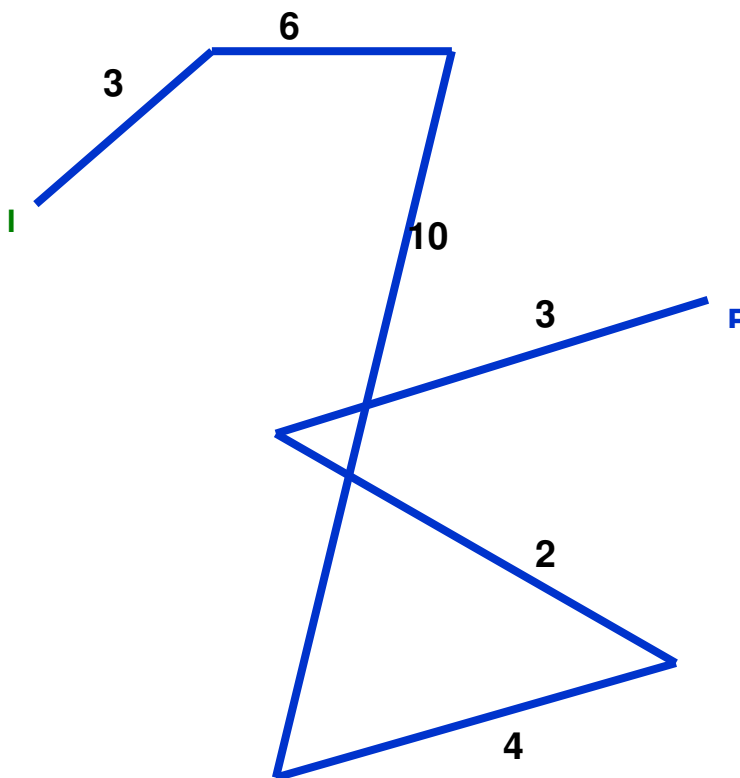
Algoritmos: ... adianta executá-los?

correspondentes. O custo é o mesmo em qualquer das direções de tráfego. Também estão assinaladas na figura uma posição inicial, I, e uma posição final, F. O segundo dado de entrada é um número inteiro positivo, K (não assinalado na figura). O problema é determinar se existe um percurso no mapa, partindo da posição inicial e terminando na posição final, e que satisfaça as duas condições a seguir:

1. O percurso deve passar por cada cidade exatamente uma vez, não importando em que ordem visite as cidades.
2. O custo total do percurso deve ser, no máximo, K. O custo do percurso é obtido somando-se os custos dos trechos individuais entre cidades por onde o caminho passa.

Não é difícil aceitar que uma grande gama de situações pode ser modelada como uma instância do problema do caixeiro viajante. Por exemplo, no lugar de cidades poderíamos ter torres de telecomunicações, as rodovias seriam os enlaces possíveis entre duas torres, com os custos de se utilizar cada enlace. O problema seria determinar se poderíamos realizar um *broadcast* para todas as cidades, sem exceder um certo custo máximo.

Queremos um algoritmo *eficiente* para resolver este problema. No caso da figura anterior, se K fosse 29, o caminho desejado existiria, como pode ser visto abaixo:



O custo do percurso seria 28. Portanto, abaixo do valor dado, K. Nesse caso, o algoritmo teria que responder positivamente. Note que não estamos exigindo que o algoritmo determine um caminho possível, se ele existir. Estamos pedindo, tão somente, uma resposta positiva ou negativa, indicando a existência ou não de um caminho que satisfaça as

condições do problema. Já se o valor de  $K$  fosse 25, a resposta teria que ser negativa, pois não existe caminho com custo total tão baixo.

É fácil ver que podemos criar um algoritmo para resolver este problema. Basta iniciar uma busca sistemática e exaustiva na cidade marcada como  $I$ . A cada nova iteração, tentamos todas as possibilidades de ir para cidades próximas, e que ainda não foram visitadas e mantendo o custo total parcial até agora sempre menor ou igual a  $K$ . Se encontrarmos a cidade final após um percurso por todas as demais cidades, responderíamos “sim”, caso contrário responderíamos “não”. Certamente, esse algoritmo resolve o problema. Porém seu custo, em termos de instruções básicas, seria muito alto. Pelo menos tão alto quanto  $2^M$ , onde  $M$  é o número de rodovias – embora não demonstremos esse fato nessas notas introdutórias. O algoritmo, portanto, seria ineficiente e imprestável na prática, onde mapas grandes são comuns.

E, até hoje, ninguém conseguiu desenvolver um algoritmo *garantidamente eficiente* para resolver este problema. Pior, não sabemos se um tal algoritmo existe ou não. Suspeita-se que não existe. Mas, se existir, a situação seria ainda mais curiosa: conseguiríamos obter algoritmos eficientes para uma vasta gama de problemas de grande interesse prático para os quais ainda não se conhece nenhum tal algoritmo! Se esse for o caso, há realmente algo importante em ciências da computação que ainda desconhecemos por completo.

## Enquanto isso ...

---

Não significa dizer que problemas como o do caixeiro viajante, e outros problemas em NP, não estejam sendo resolvidos todos os dias. Em alguns casos o número de cidades e rodovias pode ser pequeno o suficiente para que o algoritmo simples possa ser utilizado. Outras possibilidades seriam:

1. Uso de heurísticas. Podemos desenvolver algoritmos heurísticos para determinar um caminho de “custo barato” da cidade  $I$  até a cidade  $F$ , e passando por todas as demais cidades exatamente uma vez. Uma heurística é um processo de tentativas e tomadas de decisão que, *esperamos*, leve a uma *boa resposta*. Mas *não temos garantias* de que a resposta será a melhor possível. No entanto, muitas vezes, heurísticas sofisticadas e eficientes podem se constituir em alternativas atraentes para resolver esse tipo de problemas.  
Algoritmos evolutivos, cuja estratégia se espelha no processo de evolução natural, é uma classe de heurísticas que produz bons resultados.
2. Uso de algoritmo aproximativos. Um algoritmo aproximativo obteria, *garantidamente*, o custo do melhor caminho com um erro fixo. Por exemplo, o algoritmo poderia garantir que obtém o custo de um caminho com uma diferença de 5%, no máximo, em relação ao caminho ótimo. Portanto, algoritmos aproximativos eficientes, em vários casos, poderiam ser uma boa alternativa para atacar problemas desse tipo.
3. Finalmente, há outras tecnologias interessantes, hoje em desenvolvimento. Entre elas, estão computação quântica e computação molecular. Se desenvolvidas, podem

Algoritmos: ... adianta executá-los?

desencadear uma revolução em ciências da computação. Até lá, temos muito trabalho interessante pela frente.