



UNIVERSIDADE
AbERTA
www.univ-ab.pt

PROBLEMAS E COMPLEXIDADE

Este recurso é um complemento ao manual da unidade curricular de Introdução à Inteligência Artificial, de modo a permitir um maior formalismo no tratamento da teoria de conjuntos por um lado, e na teoria da complexidade dos problemas por outro. O texto tem por base em duas referências: J. Ferreira, “Elementos de Lógica Matemática e Teoria dos Conjuntos”, reedição dos capítulos iniciais de “Lições de Análise Real”, Departamento de Matemática / Instituto Superior Técnico, 2001; e M. Garey; D. Johnson, “Computers and Intractability: A Guide to the Theory of NP-Completeness”, W.H. Freeman, San Francisco, 1979.

José Coelho

Teoria dos Conjuntos

Este texto baseia-se em J. Ferreira, “Elementos de Lógica Matemática e Teoria dos Conjuntos”, reedição dos capítulos iniciais de “Lições de Análise Real”, Departamento de Matemática / Instituto Superior Técnico, 2001, muito embora existam pequenas diferenças de notação.

Chama-se a *designação* a uma identificação de um objecto matemático, seja número, conjunto, função, etc. Uma *proposição* é uma afirmação que pode ser verdadeira ou falsa, mas nunca uma coisa e outra, sendo duas proposições (p e q) equivalentes se tiverem o mesmo valor lógico ($p \Leftrightarrow q$). A conjunção de duas proposições p e q , designa-se por $p \wedge q$, é uma proposição que é verdadeira se e só se ambas as proposições forem verdadeiras, sendo a disjunção $p \vee q$ também uma proposição que é verdadeira se e só se alguma das proposições for verdadeira. Dada uma proposição p , a sua negação é designada por $\neg p$, sendo uma proposição verdadeira se e só se p for falsa, pelo que verifica-se $\neg(\neg p) \Leftrightarrow p$. Qualquer que sejam as proposições p e q , verificam-se as primeiras leis De Morgan: $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$; $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$. Uma implicação entre duas proposições p e q , representa-se por $p \Rightarrow q$ sendo uma proposição verdadeira se e só se, caso p seja verdadeira, então q também é verdadeira.

Sobre *designações* e *proposições*, podem existir *variáveis*, sendo as quais normalmente representadas por letras minúsculas. As *variáveis* têm de ser substituídas por objectos do respectivo domínio (por exemplo um número real, ou conjunto) de forma a obter *designações* ou *proposições*. Uma *designação* ou *proposição* que tenha variáveis, chama-se *expressão designatória* ou *expressão proposicional* respectivamente. Com *expressões proposicionais*, pode-se obter *proposições* utilizando o quantificador universal ou existencial: $\forall_x p_x$ significa que para qualquer que seja o valor da variável x , a proposição p_x é verdadeira; $\exists_x p_x$ significa que existe um valor da variável x , para a qual a proposição p_x é verdadeira, referindo-se para valores no domínio de x naturalmente. As segundas leis De Morgan permitem-nos efectuar a negação de proposições com quantificadores: $\neg \forall_x p_x \Leftrightarrow \exists_x \neg p_x$; $\neg \exists_x p_x \Leftrightarrow \forall_x \neg p_x$.

Os conjuntos são uma colecção de elementos distintos, sendo por exemplo $N=\{1,2,3,\dots\}$ o conjunto de números naturais. Representa-se por $x \in X$ e $x \notin X$ às proposições de x pertence a X e x não pertence a X respectivamente. Normalmente os conjuntos são definidos por expressões proposicionais, sendo os elementos do conjunto os que satisfazem a expressão proposicional: $X = \{x: p_x\}$. O número de elementos de um conjunto A , designa-se por $\#A = \sum_{x \in A} 1$. Um conjunto A está contido em B , ou A é subconjunto de B , se e só se: $\forall x, x \in A \Rightarrow x \in B$, escrevendo-se abreviadamente $A \subset B$ sendo a negação de $A \not\subset B$. Naturalmente que $A \subset B \wedge B \subset A \Rightarrow A = B$. Um conjunto sem elemento nenhum é um conjunto vazio e representa-se por: $\emptyset = \{ \}$. Dados dois conjuntos A e B , pode-se definir a intersecção e união dos conjuntos: $A \cap B = \{x : x \in A \wedge x \in B\}$, $A \cup B = \{x : x \in A \vee x \in B\}$. Caso $A \cap B = \emptyset$ diz-se que A e B são conjuntos disjuntos. A diferença de conjuntos é definida por $A \setminus B = \{x : x \in A \wedge x \notin B\}$, sendo formada pelos elementos de A que não pertencem a B .

Um par ordenado (a,b) são duas designações ordenadas, sendo (a,b,c) um terno ordenado, e (a_1, a_2, \dots, a_n) uma sequência. Duas sequências são iguais se e só se: $(a_1, a_2, \dots, a_n) = (b_1, b_2, \dots, b_n) \Rightarrow \forall i \in \{1, \dots, n\} a_i = b_i$. Um produto cartesiano de n conjuntos, é definido por: $A_1 \times \dots \times A_n = \{(x_1, \dots, x_n) : x_1 \in A_1 \wedge \dots \wedge x_n \in A_n\}$. Quando se tem $A_1 = A_2 = \dots = A_n = A$, chama-se de n^{a} potência de A e simplifica-se para: $A_1 \times \dots \times A_n = A^n$. Um subconjunto R de $A_1 \times \dots \times A_n$, é chamado uma *relação* sobre A_1, \dots, A_n , sendo no caso de $n=2$ chamada de *relação binária*. Se R é uma relação binária, muitas vezes pode-se abreviar $(a,b) \in R \Leftrightarrow a R b$, como é o caso, por exemplo, da desigualdade, que é uma relação binária em \mathbb{R}^2 : " \leq " = $\{(x, y) \in \mathbb{R}^2 : x \leq y\}$. Dada uma relação binária R tem-se que $R^{-1} = \{(x, y) : (y, x) \in R\}$ é a relação inversa de R , sendo R chamada de *simétrica* se e só se $\forall x, y (x, y) \in R \Rightarrow (y, x) \in R$, *anti-simétrica*, se e só se $\forall x \neq y (x, y) \in R \Rightarrow (y, x) \notin R$, de *reflexiva* se e só se $\forall x \in A (x, x) \in R$, de *irreflexiva* se e só se $\forall x \in A (x, x) \notin R$, e *transitiva* se e só se $\forall x, y, z (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$.

Uma relação $F \subset A \times B$ é uma função se e só se $(x, y) \in F \wedge (x, z) \in F \Rightarrow y = z$. Para funções, utiliza-se a seguinte abreviatura: $(x, y) \in F \Leftrightarrow F_x = y$. Uma função diz-se *sobrejectiva* se e só se $\forall y \in B \exists x \in A F_x = y$, e *injectiva* se e só se $\forall x, y \in A x \neq y \Rightarrow F_x \neq F_y$, sendo *bijectiva* se for simultaneamente *sobrejectiva* e *injectiva*.

Problemas e Complexidade

A teoria da complexidade dos problemas, estuda problemas de optimização, oferecendo uma classificação dos problemas relativamente à sua complexidade. O conhecimento da classe a que pertence um problema, é muito importante pois permite saber se valerá a pena procurar um algoritmo eficiente para obter a solução óptima, ou se por outro lado, mais vale investir numa *heurística* e ficarmo-nos por uma solução sub óptima. Esta secção é baseada no livro de M. Garey; D. Johnson, “Computers and Intractability: A Guide to the Theory of NP-Completeness”, W.H. Freeman, San Francisco, 1979.

Noções Básicas

De forma informal, pode dizer-se que um problema fica definido com a caracterização de uma instância do problema, e uma pergunta sobre a instância, ou seja, o que se pretende saber. Um problema diz-se de *decisão* se a pergunta é do tipo sim ou não, de *procura* se for pedida uma solução caso exista, e de *optimização* se for pedida a melhor solução segundo uma função de avaliação, sendo de *minimização* ou *maximização* de acordo com o objectivo pedido de minimizar ou maximizar a função. Um problema de *decisão* pode ser facilmente modelado como um problema de *procura*, basta que no caso do valor a retornar ser sim, devolver uma solução óptima “sim”, devolvendo o conjunto vazio no caso do não, tal como um problema de *optimização* fazendo o conjunto de soluções igual às soluções que minimizam ou maximizam o valor da função.

Para analisar a complexidade dos problemas, estuda-se a relação do tempo que um algoritmo leva a resolver uma instância do problema, com a dimensão da instância. Basicamente há dois tipos de algoritmos, os que têm o tempo de execução limitável por um polinómio dependente do tamanho da instância, e dos que não são limitáveis por um

polinómio. Ao primeiro tipo de algoritmos chamam-se *eficientes*, e aos segundo tipo de algoritmos *não eficientes*. Tal distinção é justificada porque um algoritmo não limitado por um polinómio tem normalmente uma evolução exponencial, pelo que acaba em instâncias de dimensões reais por disparar em termos de tempo, e ser sempre inferior a um algoritmo limitado por um polinómio.

Duas questões se levantam, como medir a dimensão de uma instância e como medir o tempo de processamento de um algoritmo. Estas questões justificam-se uma vez que ao implementar um algoritmo pode-se utilizar codificações de uma instância mais ou menos eficientes, bem como implementar um algoritmo numa linguagem mais rápida, ou correr o mesmo código num computador mais rápido. Assim sendo, poder-se-á pensar que o estudo da relação entre o tamanho da instância e o tempo de processamento está dependente de muitas condições específicas e não pode ser geral. Tal no entanto não verdade, uma vez que as diferenças referidas nunca podem ser exponenciais, quanto muito uma determinada linguagem (ou um computador mais rápido) multiplica por um factor o tempo de processamento, tal como o tamanho de uma instância apenas pode variar por um factor, devido à diferença entre uma boa e uma má codificação.

Para poder saber se um algoritmo é eficiente ou não, e poder distinguir entre dois algoritmos eficientes de forma a escolher o mais rápido, podia-se pensar num esquema de testar os diversos algoritmos num conjunto de instâncias do problema e analisar os tempos de resolução dos diversos algoritmos. Os que tivessem um comportamento exponencial, seriam não eficientes, e os eficientes seriam classificados de acordo com o tempo de processamento. Tal operação é no entanto muito custosa e fica dependente do conjunto de instâncias escolhidas. A solução acaba por ser simples como veremos de seguida, dado não interessar o calculo de tempos exactos uma vez que tais tempos estão dependentes da máquina e compilador, basta contabilizar o número máximo de comandos básicos que um algoritmo necessita, dependendo do tamanho da instância.

Complexidade temporal de um algoritmo

Sendo p_N uma expressão designatória com uma variável N , o tamanho da instância, que é um máximo do número de operações básicas necessárias a um determinado algoritmo,

$O(p_N)$ é a complexidade temporal desse algoritmo. Dadas duas expressões designatórias p e q , diz-se que $O(p_N) < O(q_N) \Leftrightarrow \lim_{N \rightarrow \infty} \frac{p_N}{q_N} = 0$, pelo que pode-se concluir diversas regras de simplificação, sendo g e k reais positivos quaisquer: $O(p_N) + O(q_N) = O(p_N + q_N) = \max\{O(p_N), O(q_N)\}$; $O(k \cdot p_N) = O(p_N)$; $O(N^g) < O(N^{g+k})$; $O(N^g) < O((1+k)^N)$; $O(\log N) < O(N)$. Pode-se agora definir mais rigorosamente, que um algoritmo é eficiente se existir um g positivo tal que a complexidade temporal do algoritmo seja inferior a $O(N^g)$, sendo não eficiente caso contrário.

Por exemplo, caso de existam dois ciclos, um dentro do outro, em que são percorridos todos os elementos da instância com N elementos, e dentro do ciclo são executadas duas operações, o número de passos será $2 \cdot N \cdot N$, sendo a complexidade temporal $O(N^2)$. Notar que neste exemplo, se pretendermos percorrer todos os pares de elementos mas não interessar a ordem, a variável do ciclo interior basta começar no elemento seguinte ao da variável anterior, reduzindo o número de passos para $2(N(N-1)/2) = N(N-1)$, mas a complexidade temporal permanece igual. Suponha-mos que temos disponível outro algoritmo para o mesmo problema que necessita de dois ciclos separados, com 4 operações em cada ciclo e 10 operações fora de ambos os ciclos. O número de passos será $10 + 4 \cdot N + 4 \cdot N = 10 + 8 \cdot N$, sendo a complexidade temporal de $O(N)$, e portanto melhor que o primeiro algoritmo. Vejamos um último caso, em que para cada elemento o algoritmo considera marca-lo com uma de 3 cores, repetindo o processo para os restantes elementos, após o qual troca a cor e repete o processo para os restantes elementos, e finalmente experimenta a última cor repetindo o processo novamente. Este tipo de algoritmo, normalmente implementado de forma recursiva, requer um número de passos da ordem de $k \cdot 3^N$, o que dará uma complexidade temporal de $O(3^N)$, sendo não só pior que os restantes algoritmos, como também não é eficiente dado não estar limitado por um polinómio.

Teoria da complexidade

Um problema de *decisão* Π é definido por um par (D_Π, Y_Π) com o conjunto de instâncias e as instâncias com a resposta *sim* respectivamente, em que $Y_\Pi \subset D_\Pi$. Um algoritmo M , diz-se que resolve o problema Π se para cada instância $I \in D_\Pi$ retornar *sim* se $I \in Y_\Pi$ e *não* caso contrário. Um exemplo de um problema, é o *PARTITION*, que pode ser definido como:

$$\Pi_{\text{PARTITION}} = (D_\Pi, Y_\Pi)$$

$$D_\Pi = \left\{ (A = \{1, \dots, n\}, f: A \rightarrow \mathbb{Z}^+) \right\}$$

$$Y_\Pi = \left\{ (A, f) \in D_\Pi : \exists A' \subset A \sum_{i \in A'} f(i) = \sum_{i \in A/A'} f(i) \right\}$$

Não há no entanto necessidade de todo este formalismo, basta definir a instância e a pergunta. Neste caso a instância é um conjunto A de elementos e uma função $f: A \rightarrow \mathbb{Z}^+$, sendo a pergunta se existe uma divisão dos elementos de A para a qual a soma dos valores da função f sejam iguais.

Consideramos dois tipos de máquinas, as determinísticas e as não determinísticas. A primeira executa um conjunto de instruções sequencialmente e deterministicamente, enquanto que a segunda tem um módulo “adivinhador” que coloca um palpite para a solução e pode realizar um número não limitado de sequencias de instruções em paralelo. Não existem naturalmente máquinas não determinísticas, o seu interesse é como veremos teórico.

Caso exista um algoritmo M que resolva o problema Π numa máquina determinística em tempo polinomial, então $\Pi \in P$. Caso exista um algoritmo M que resolva o problema Π numa máquina não determinística em tempo polinomial, então $\Pi \in NP$. Como o algoritmo de uma máquina determinística pode ser utilizado numa máquina não determinística ignorando a adivinhação, resulta que $P \subset NP$. O tamanho da instância pode ser medido por uma qualquer regra razoável, uma vez que a diferença para outra medida pode ser limitada por um polinómio, sendo insignificante para efeitos de determinar se o tempo é polinomial ou não.

Facilmente se pode provar que um dado problema pertence a P , basta para tal que se apresente um algoritmo polinomial, muito embora por vezes esse algoritmo possa ser complicado. Para provar que um problema pertence a NP é preciso fornecer apenas um algoritmo que utilize a adivinhação (a solução) e a verifique em tempo polinomial. O mais difícil será provar que um dado problema não pertence a P , ou seja, que não existe um algoritmo que resolva o problema em tempo polinomial. A dificuldade é de tal ordem que até hoje ninguém o fez para qualquer problema em NP .

Uma transformação de um problema $\Pi = (D_{\Pi}, Y_{\Pi})$ para um problema $\Pi' = (D_{\Pi'}, Y_{\Pi'})$, é uma função polinomial $f: D_{\Pi} \rightarrow D_{\Pi'}$, em que $\forall I \in D_{\Pi} I \in Y_{\Pi} \Leftrightarrow f(I) \in Y_{\Pi'}$, representando-se $\Pi \propto \Pi'$.

Naturalmente que se $\Pi' \in P \Rightarrow \Pi \in P$, e o mesmo para NP , pelo que pode-se provar que um problema pertence a P ou NP desta forma.

Um problema Π é *NP-completo* se existir uma transformação de qualquer problema de NP para Π , ou seja: $\Pi \in NP-completo \Leftrightarrow \forall \Pi' \in NP \Pi' \propto \Pi$. Esta definição significa que não existem problemas mais complexos em NP que os que pertencerem a *NP-completo*. Dada a dificuldade de provar que (não) existe um algoritmo polinomial para certos problemas em NP , esta definição permite agrupar um bom conjunto de problemas que apenas pertencerá a P caso $P=NP$. O primeiro problema a provar-se pertencer a esta classe foi o *SAT*, que é estudado neste texto no capítulo 4. Naturalmente que $(\Pi \in NP-completo \wedge \Pi \propto \Pi') \Rightarrow \Pi' \in NP-completo$, sendo esta a técnica de prova utilizada, transformar um problema *NP-completo* para o problema em causa de forma a provar que é *NP-completo*.

Um problema Π é numérico se o valor do maior número na instância não poder ser limitado por um polinómio no tamanho da instância. Sobre um problema $\Pi = (D_{\Pi}, Y_{\Pi})$ numérico pode-se definir um sub problema $\Pi_p = (D_{\Pi_p} = \{I \in D_{\Pi} : \text{Max}[I] \leq p(\text{Tam}[I])\}, D_{\Pi_p} \cap Y_{\Pi})$, sendo p um polinómio, $\text{Max}[I]$ o maior número na instância e $\text{Tam}[I]$ o tamanho da instância. Π_p está limitado pelos valores numéricos, não sendo portanto um problema numérico. Se $\Pi \in NP-completo \wedge \Pi_p \in P$, então existe um algoritmo pseudo polinomial para Π , dado que pode ser resolvido em tempo polinomial caso sejam restringidos os valores

numéricos. Nesse caso diz-se que pertence a *NP-completo* no *sentido normal*, sendo no *sentido forte* se $\Pi_p \in NP-completo$. Naturalmente os problemas não numéricos pertencem a *NP-completo* no *sentido forte* dado que pode-se limitar os valores numéricos que o problema fica igual e portanto *NP-completo*. O problema *PARTITION* referido anteriormente é *NP-completo* no *sentido normal*, enquanto que o *SAT* é *NP-completo* no *sentido forte*.

Um problema de *procura* Π é definido por um par $(D_\Pi, \{S_\Pi[I]: I \in D_\Pi\})$ com o conjunto de instâncias e para cada instância um conjunto de soluções. Um algoritmo M , diz-se que resolve o problema Π se para cada instância $I \in D_\Pi$ retornar uma solução $s \in S_\Pi[I]$, ou retornar *não* caso o conjunto seja vazio. Um exemplo de um problema, é o *TSP*, que pode ser definido como:

$$\prod_{TSP} = (D_\Pi, \{S_\Pi[I]: I \in D_\Pi\})$$

$$D_\Pi = \{(C = \{1, \dots, n\}, d: C^2 \rightarrow \mathbb{Z}^+)\}$$

$$S_\Pi[I] = \left\{ \langle c_1, \dots, c_n \rangle \in \text{Permutações}(C) : \neg \exists_{(c'_1, \dots, c'_n) \in \text{Permutações}(C)} d_{c'_1 c'_1} + \sum_{i=1}^n d_{c'_i c'_{i+1}} < d_{c_n c_1} + \sum_{i=1}^n d_{c_i c_{i+1}} \right\}$$

Naturalmente, um problema de *decisão* $\Pi = (D_\Pi, Y_\Pi)$ pode ser formulado como um problema de *procura* $\Pi = (D_\Pi, \{S_\Pi[I] = \{\text{"sim"}: I \in Y_\Pi\}: I \in D_\Pi\})$. Diz-se que um problema Π' é *Turing reduzível* a Π , escrevendo-se $\Pi' \propto_T \Pi$, se existir um algoritmo M' numa máquina determinística, para resolver o problema Π' , que faz uso de um algoritmo M para resolver o problema Π , em que M' é polinomial se M for polinomial. Naturalmente que se $\Pi' \propto \Pi$ então $\Pi' \propto_T \Pi$ dado que uma transformação é um caso particular em que em que o algoritmo M é chamado apenas uma vez.

Seja um problema Π qualquer, pertença ou não a *NP*, diz-se que é *NP-difícil* se existir uma *redução de Turing* de um problema *NP-completo*: $\Pi \in NP-difícil \Leftrightarrow \exists \Pi' \in NP-completo \Pi' \propto_T \Pi$, sendo estes problemas naturalmente pelo menos tão complicados como um problema *NP-completo*. Um problema é *NP-simples* se existir uma *redução de Turing* para um problema *NP*: $\Pi \in NP-simples \Leftrightarrow \exists \Pi' \in NP \Pi \propto_T \Pi'$, sendo estes problemas não mais complicados que um

problema *NP-completo*. A um problema simultaneamente *NP-difícil* e *NP-simples*, chama-se *NP-equivalente*, dado que não é mais simples nem mais complicado que um problema *NP-completo*.

Um problema de *optimização* Π é definido por um triplo $(D_\Pi, \{S_\Pi[I]: I \in D_\Pi\}, m_\Pi: D_\Pi \times S_\Pi[I] \rightarrow \mathbb{R})$ com o conjunto de instâncias, para cada instância um conjunto de soluções, e uma função valor. Ao menor valor de todas as soluções de uma instância — assumindo-se um problema de minimização — chama-se de valor óptimo: $OPT_\Pi(I)$. Diz-se que um algoritmo M é de *aproximação*, se para cada instância $I \in D_\Pi$ devolver uma solução $s \in S_\Pi[I]$, sendo o valor da solução $m_\Pi(I, s)$ abreviado para $M(I)$. Caso se garanta que para todas as instâncias $M(I) = OPT(I)$, então o algoritmo é de *optimização*.

Ainda não foi encontrado nenhum problema que pertença simultaneamente a P e *NP-completo*, pelo que caso saiba que o problema é *NP-completo*, deixa de ter sentido a procura de um algoritmo eficiente, a não ser que se esteja a tentar resolver um problema em aberto com já largas décadas. Aconselha-se neste caso será a construção de *heurísticas*, largando a questão da optimalidade, alargando o conjunto de soluções de interesse às soluções quase óptimas.