MAC 5747 GEOMETRIA COMPUTACIONAL

ANÁLISE DE ALGORITMOS

1. Introdução

```
inic LOAD   2
     ADD    =1
     STORE  2
     LOAD   *2
     ADD    =3
     STORE  1
     LOAD   3
     SUB    *1
     STORE  3
     LOAD   2
     ADD    =1
     STORE  2
     JUMP   inic
```

Analisar um algoritmo é mensurar e descrever o seu desempenho. Ao enterdermos o desempenho de um algoritmo teremos uma idéia de como ele se comportará na prática (pelo menos gostariamos que isto seja verdade). A análise de algoritmo também nos fornece uma maneira de compararmos diferentes soluções algorítmicas para um mesmo problema e identificar quais são mais eficientes e mais apropriados para os recursos computacionais que temos disponíveis.

Há muitas maneiras de compararmos algoritmos: eficiência, flexibilidade, complexidade, robustez, facilidade de uso, etc. Muitas dessas maneiras são difíceis de quantificar. Estaremos tipicamente interessados em medir o *tempo* que um algoritmo gasta para ser executado e a quantidade de *espaço* (quantidade de memória) necessária para executá-lo. Assim, o *custo* de um algoritmo englobará estes dois aspectos, ou seja, tempo e espaço. Na realidade, estaremos mais interessados em medir o tempo gasto por um algoritmo, já que para os algoritmos que estudaremos o espaço não será um fator critíco, tendo em vista que a maioria dos algoritmos que veremos nessa disciplina necessitam espaço proporcional à entrada do problema (número de pontos, retas, ...). (Isto é semelhante ao que é feito na disciplina Algoritmos em Grafos, quando não se discute o espaço gasto pelo, digamos, Algoritmo de Ford e Fulkerson, já que o espaço requirido pelo algoritmo é claramente proporcional ao tamanho do grafo (i.e. numero de vértices + número de arestas).)

Espaço e tempo não são recursos independentes. São comuns as situações onde é possível conservarmos um recurso às custas de gastarmos mais do outro, o que é frequentemente chamado de *tradeoff* espaço/tempo.

A análisa de um algoritmo consistirá em medirmos o número de operações *elementares* ou *primitivas* feitas por um algoritmo. Para tanto devemos, antes de mais nada, especificar um *modelo de computação*, o que será feito logo a seguir, na Seção 2. Na Seção 3 discutiremos um pouco sobre as possíveis medidade de complexidade. Análise assintótica será apresentada na Seção 4. Veremos algo sobre a análise de algoritmos recursivos na Seção 5. Complexidade computacional será o tópico da Seção 6. A Seção 7 tratará de Robustez e da chamada *hipótese de posição geal.* Finalmente, na Seção 8, pdem ser encontrados alguns exercícios.

Date: 18 de fevereiro de 2000.

Observação. Os tópicos que veremos (brevemente) a seguir foram extraído principalmente dos Capítulos 1 e 2 de Laszlo [5] e do Capítulo 1 de Preparata e Shamos [7] e podem ser encontrados em qualquer livro de Análise de Algoritmos. A referência clássica em análise de algoritmos é Aho, Hopcroft e Ullman [1]. Cormen, Leiserson e Rivest [3] é um livro enciclopédico sobre o assunto.

2. Modelo de computação

Um modelo de computaçãp especifica as operações *primitivas* ou *elementares* que podem ser executadas por um algoritmo e o *custo* dessas operações. Por exemplo, em alguns modelos multiplicar é 'mais caro' do que somar.

A escolha de um modelo de computação envolve um compromisso entre realidade (prática) e tratabilidade matemática. Assim, devemos escolher um esquema que capture as características do dispositivo computacional usado e ainda seja suficientemente simples para que permita uma análise completa.

Qual o tipo de modelo mais apropriado para algoritmos geométricos? Para tentarmos reposnder a esta pergunta deveremos examinar cuidadosamente a natureza do problema que esta sendo considerado.

O modelo de computação que é mais comumente usado em geometria computacional é a Máquina de Acesso Aleatório real (real-RAM — *real Random Access Machine*) com custos uniformes (cf. Capítulo 1, Seção 4 de Preparata e Shamos [7] e Capítulo 1 de Aho, Hopcroft e Ullman [1]).

Os registradores da máquina real-RAM podem armazenar números reais (que podem ter tamanho arbitrários — o que não é muito realista) e operações como $+$, $-$, $\times$, $\div$, $\surd$ são exemplos de operações *primitivas*, isto é, estão disponíveis a custo unitário. (Pretendemos especificar as operações primitivas que estarão sendo empregadas em cada algoritmo).

Uma vez escolhido o modelo de computação podemos agora prosseguir e selecionar as medidas de complexidade. Ao analisarmos a complexidade de um algoritmos estaremos principalmente interessados em estimar a sua complexidade de tempo (i.e., número de operações primitivas feitas por uma máquina real-RAM) e sua complexidade de espaço. No que se refere a complexidade de tempo, dependendo do tipo de problema sob consideração, serão estimados, por exemplo, o tempo gasto pelo algoritmo em pré-processamento, o tempo gasto para responder às consultas e o tempo gasto em atualizações. Ao expressarmos a complexidade dos algoritmos, como é usual, ignoraremos fatores constantes e consideraremos apenas o seu comportamento assintótico. Como também é usual, as complexidades de tempo e de espaço dos algoritmos serão expressas como funções do tamanho da entrada.

3. Medidas de complexidade

A escolha de um modelo de computação simplifica a análise de um algoritmo. Para simplificar ainda mais a análise expressaremos (como é usual) as complexidades de tempo e de espaço dos algoritmos como funções do tamanho da entrada (número de pontos, retas, etc.). O tamanho da entrada caracteriza o espaço (mínimo) usado pelo algoritmo para armazenarmos os dados do problema. A medida do tamanho da entrada dependerá do problema de uma maneira 'natural.' Se a entrada do problema consiste de uma seqüência de números (como em ordenação), o tamanho da entrada será a quantidade n de elementos na seqüência.

As medidas de complexidade usadas mais comumente são de pior caso, do caso médio e amortizada.

3.1. **Análise do pior caso.** O tempo de execução no *pior caso* descreve o maior tempo de execução para qualquer instância de um determinado tamanho. Por exemplo, busca seqüência faz n consultas, no pior caso, para determinar se um dado elemento pertence a um seqüência de n elementos (isto ocorre quando o elemento não pertence à seqüência). Assim, a função de complexidade de tempo T da busca seqüencial no pior caso é $T(n) = n$.

3.2. **Análise do caso médio.** O tempo de execução no *caso médio* tem como objetivo descrever o tempo médio gasto pelo algoritmo sobre todas as entradas de cada tamanho. O tempo médio gasto pelo algoritmo de busca seqüêncial é $(n+1)/2$, suponde-se que é igualmente provavel que procuremos cada um dos n elementos da seqüência e que a busca seja um sucesso (i.e., que encontremos o elemento). Se admitirmos que a busca possa terminar em fracasso (i.e., que **não** encontre o elemento) então a tempo médio gasto será algo entre $(n + 1)/2$ e n, dependendo da probabilidade de fracasso.

3.3. **Análise amortizada.** As operações executadas por um algoritmo são muitas vezes interdependentes. O custo de uma operação pode depender do custo das operações que já foram executadas anteriormente. Como exemplo, considere um algoritmo que executa operações de `empilha` e `desempilha` sobre uma pilha. Se a operação `empilha` é executada, no pior caso, k vezes ao longo da execução de um algoritmo, então o número de operações `desempilha` efetuadas por este mesmo algoritmo não poderá ser maior do que k. Análise amortizada mede o tempo médio gasto em cada operação, no pior caso. Mais sobre este tipo de análise pode ser encontrada no Capítulo 18 de Cormen, Leiserson e Rivest [3].

4. Análise assintótica

Até agora introduzimos duas maneiras de simplificarmos a análise de um algoritmo, a saber, adotar um modelo de computação e expressar o tempo de execução (e espaço necessário) como uma função do tamanho da entrada. Uma terceira simplicação se concentra em *eficiência assintótica*: como o tempo cresce a medida que o tamanho da instância vai para infinito. Se a complexidade de tempo ou espaço (e.g., no pior caso) de um algoritmo é expressa por uma função $f(n)$, estaremos interessados na sua taxa de crescimento e capturaremos isto através do crescimento de uma função mais simples $T(n)$.

Por exemplo, suponha que $f(n) = an^2 + bn + c$ para constantes a, b, e c. A medida que o tamanho da entrada cresce (isto é, n cresce), os termos de menor ordem se transformam em valores insignificantes e podem, portanto, ser ignorados — podemos tratar $f(n)$ como an^2. Ademais, mesmo o coeficiente a do termo lider é relativamente insignificante a medidade que o tamanho do problema cresce, assim podemos trocar $f(n)$ pela função $T(n) = n^2$ que é mais simples. Resumindo, ao expressarmos o tempo de execução de um algoritmo nós ignoraremos os fatores constantes e ficaremos apenas com o 'termo dominante.'

Notação assintótica nos fornece uma linguagem conveniente para expressarmos a taxa de crescimento de uma função de complexidade de tenpo ou de espaço em função do tamanho do problema.

4.1. **Notação** O**.** A notação O (lê-se Ó-grande) é usada para descrever limites superiores para a taxa de crescimento de uma função. Se $f(n)$ é uma função, $O(f(n))$ denota a classe de funções cuja taxa de crescimento não é maior que $f(n)$. Uma função $g(n)$ pertence a $O(f(n))$ se $g(n)$ não é maior que uma constante vezes $f(n)$ para n suficientemente grande. Mais formalmente, $g(n) \in O(f(n))$ (ou, $g(n) = O(f(n))$) se para algum número real $c > 0$ existe um inteiro $n_0 \geq 1$ tal que $g(n) \leq f(n)$ para todo $n \geq 0$.

A medida de qualidade de um algoritmo em geometria computacional tem sido tradicionalmente tempo (espaço) assintótico gasto, no pior caso. Assim um algoritmo de complexidade de tempo $O(n)$ (linear) é melhor que um de complexidade de tempo $O(n \log n)$ que é melhor do que um de complexidade $O(n \log^2 n)$, que é melhor do que $O(n^2)$, ...

4.2. **Notação** o**.** A notação o (lê-se ó-pequeno) é usada para descrever limite superior *estrito*. Dada uma função $f(n)$, $o(f(n))$ é a classe de funções cuja taxa de crescimento é estritamente menor do que $f(n)$. Mais precisamente, uma função $g(n)$ pertence a $o(f(n)$ se para *toda* constante $c > 0$ existe um inteiro $n_o \geq 1$ tal que $g(n) \leq cf(n)$ para todo $n \geq n_o$. Não é difícil perceber que $o(f(n)) \subset O(f(n))$ para toda função f.

A notação o nos fornece uma maneira de nos concentrarmos no termo dominante de uma função. Por exemplo, para enfatizar o termo lider de $f(n) = an \log n + bn + c$, nós reescreveriamos a função como $f(n) = an \log n + o(n \log n)$. Isto indica que os termos de menor ordem são dominados por $n \log n$ e podem ser desprezados.

4.3. **Notação** Ω**.** A notação Ω (lê-se omega) é usada para descrever limites inferiores para a taxa de crescimento de uma função. Dada uma função $f(n)$, $\Omega(f(n))$ denota a classe de funções cuja taxa de crescimento não é menor do que $f(n)$. A definição de $\Omega(f(n))$ é simétrica à definição de $O(f(n))$. Uma função $g(n)$ pertence a $\Omega(f(n))$ se para algum número real $c > 0$ existe um inteiro $n_0 \geq 1$ tal que $g(n) \geq cf(n)$ para todo $n \geq n_0$.

4.4. **Notaçào** Θ**.** As notações O e Ω podem ser usadas para definar a classe das funções cuja taxa de crescimento está entre dois limites. Dada uma função $f(n)$, $\Theta(f(n))$ denota $O(f(n)) \cap \Omega(f(n))$.

5. Análise de algoritmos recursivos

Muitos algoritmos têm a seguinte estrutura *recursiva*: para resolver um determinado problema eles fazem uma ou mais chamadas recursivas para resolver o mesmo problema em instâncias menores. Esses algoritmos tipicamente seguem o seguinte *método de divisão-e-conquista*: quebram o problema em diversos subproblemas que são similares ao problema original mas de tamanhos menores; resolvem recursivamente esses subproblemas menores; e, finalmente, combinam as soluções desses problemas menores para obter uma solução do problema para a instância original. (Veja Capítulo 1, Seção 3 de Cormen, Leiserson e Rivest [3].)

Já vimos esse método antes durante o curso. Um exemplo bem-conhecido de algoritmo que usa este método é o `MergeSort`.

Mais especificamente, algoritmos do tipo divisão-e-conquista têm três fases em cada nível da recursão:

Dividir: O problema é dividido em subproblemas de tamanho menor.

Conquistar: Os subproblemas são resolvidos recursivamente. Entretanto, se algum subproblema é suficientemente pequeno ele é resolvido diretamente.

Combinar: As soluções dos subproblemas são combinadas para obtermos uma solução do problema original.

Algoritmos baseados nesse método podem ser analisados através de *relações de recorrência* (ou simplemente recorrências). Um recorrência é uma função que tem argumentos inteiros e é definida em termos dela mesma.

Por exemplo, o tempo de execução $T(n)$ do `MergeSort`, no pior caso, satisfaz a seguinte relação de recorrência:

$$T(n) \leq \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + an & \text{se } n > 1 \\ b & \text{caso contrário } (n = 1) \end{cases}$$

Durante esta disciplina veremos várias vezes a relação de recorrência acima.

Exercício. Mostre que $T(n) = O(n \log n)$.

Observação. Será mais comum escrevermos $T(n) = O(n \log n)$, ao invés de $T(n) \in O(n \log n)$, apesar desta última maneira ser mais precisa.

6. Complexidade computacional

Por que alguns problemas parecem ser (computacionalmente) mais difíceis do que outros? Como podemos medir ou comprovar este diferente (e aparentemente intrínseco) grau de dificuldade?

Quais problemas podem ser resolvidos por um algoritmo? Quais não podem? Por quê?

Quais problemas podem ser resolvidos por um algoritmo eficiente? Quais não podem? Por quê?

Quanto mais 'poderoso' deve ser um algoritmo para que façamos com que um problema incomputável passe a ser computável?

Em complexidade computacional estamos interessados em questões deste tipo.

Se sabemos como é o desempenho de um algoritmo podemos compará-lo com o desempenho de um outro algoritmo para o mesmo problema. Entretanto, suponha que estamos interessados em saber se *existe* um algoritmo mais eficiente para resolver um problema ou se o algoritmo que temos em mãos é *ótimo*. Uma maneira de decidirmos se um algoritmo é ótimo é compará-lo com um segundo algoritmo que sabemos (de alguma maneira) ser ótimo, mas isto só funciona se o último algoritmo for sabidamente ótimo.

Como podemos determinar se um algoritmo é ótimo? Mais geralmente, como podemos determinar quanto tempo (esforço computacional) é necessário para que um algoritmo resolva um determinado problema.

A *complexidade de um problema* expressa a quantidade de recursos computacionais que são ambos, necessários e suficientes, para resolver o problema.

6.1. **Cota (limite) superior.** Qualquer algoritmo que resolve um problema determina uma cota superior para o problema, ou seja, se um algoritmo tem complexidade de tempo $O(f(n))$, então tempo $O(f(n))$ é suficiente para resolver o problema. Se não existe um algoritmo mais eficiente então o dado algoritmo é ótimo e a cota superior para o problema é apertada (*tight*).

6.2. **Cota (limite) inferior.** Encontrar um bom limite inferior para um problema é, via de regra, uma tarefa muito difícil. Uma cota inferior de $\Omega(f(n))$, no pior caso, para um problema significa que existe uma constante $c > 0$ tal que todo algoritmo que resolve o problema requer pelo menos $cf(n)$ passos para solucionar *alguma* instância de tamanho n. Dizemos que a complexidade de um problema é $\Theta(f(n))$ se, além do problema ter cota inferior $\Omega(f(n))$, existe um algoritmo de complexidade de tempo $O(f(n))$ para o problema (neste caso dizemos que o algoritmo é ótimo).

Existem pelo menos dois métodos de provar-se a cota inferior de um problema. O primeiro, e mais difícil, é através de *árvores de decisão*, onde, de uma maneira grosseira, uma árvore representa todas as possíveis computações de um algoritmo (em um determinado modelo) sobre instâncias de um determinando tamanho. As arestas que levam um nó da árvore aos seus filhos correspondem a instruções de desvio (decisões) feitas pelo algoritmo e cada folha da árvore corresponde uma possível resposta do algoritmo. Finalmente, é demonstrado que esta árvore de decisão tem altura $\Omega(f(n))$. (Veja, por exemplo, o Capítulo 9 de Cormen, Leiserson e Rivest [3] onde mostra-se que toda árvore de decisão que ordena n elementos tem altura $\Omega(n \log n)$. Veja também o Capítulo 1 Seção 4 de Preparata e Shamos [7].)

Outro método para mostrar a dificuldade intrínsica (cota inferior) de um problema é através de uma *transformação* deste problema em um outro problema cuja dificuldade computacional intrínsica é conhecida. Mais especificamente, suponha que temos dois problemas, Problema A e Problema B, tal que o problema A pode ser resolvido da seguinte maneira:

1. uma instância (ou entrada) do Problema A é convertida em uma instância 'apropriada' do Problema B;
2. Problema B é resolvido;
3. a solução do Problema B é transformada em uma solução do Problema A.

Nesse caso dizemos que o Problema A foi *transformado* no Problema B. Se os Passos 1 e 3 acima puderem ser feito 'rapidamente', então uma cota inferior para o Problema A implicará um conta inferior para o Problema B.

Algumas cotas inferiores que nos serão particularmente uteis são a do Problema da Unicidade de Elementos (*Element uniqueness*, cf. Capitulo 6 de Preparata e Shamos [7]) e da ordenação.

Problema 1 (Unicidade de Elementos). *Dado n números, decidir se existem dois números entre esses que são iguais.*

Teorema 2 (Dobkin e Lipton [4]). *Qualquer algoritmo (no modelo de árvore de decisão algébrica) que determina se os elementos de uma seqüência de n números são distintos requer, no pior caso, $\Omega(n \log n)$ comparações.* ∎

Uma prova do Teorema 2 abaixo pode ser encontrada na página 192 de Preparata e Shamos [7].

Teorema 3 (Ben-Or [2]). *Qualquer algoritmo (no modelo de árvore de decisão) que ordena uma seqüência de n números requer, no pior caso, $\Omega(n \log n)$ comparações.* ∎

Uma prova do Teorema 3 pode ser vista em Cormen, Leiserson e Rivest [3]. Usaremos estas cotas inferiores algumas vezes durante esta disciplina.

7. Robustez e hipótese de posição geral

Um algoritmo é robusto se ele sempre funciona para todo tipo de entrada, mesmo quando a instância seja 'degenerada'. (Isto é irônico, o algoritmo não funciona para uma determinada instância e chamamos a pobre da instância de *degenerada* ... Acho que é melhor chamarmos este tipo de instância de *especial*, ou caso *especial*. Esta é uma instância ou tipo de instância que deverá ser tratada separadamente.)

Uma possível fonte de erro é simplesmente o algoritmo estar incorreto. Agora, mesmo que o algoritmo esteja correto, pode ser que erros ocorram devido a problemas de precisão (arrendodamentos e truncamentos). Uma maneira de contornar isto é dar um certificado de garantia junto com algoritmo: "este algoritmo funciona para entradas coms tais e tais característica." Com isto estaremos restringindo o conjunto de possíveis instâncias do algoritmo. Uma outra possível saída é utilizarmos aritmética racional com precisão 'ilimitada' ou coisa do gênero, acredito que está foi a saída empregada no projeto LEDA (cf. Mehlhorn e Näher [6]).

A literatura em geometria computacional frequentemente usa a chamada *hipótese de posição geral* (*general position assumption*) que diz que serão consideradas somente as instâncias onde nenhum predicado geomérico requerido pelo algoritmo assume o valor zero (um predicado geométrico é calculado através do sinal de uma expressão). Por exemplo, para alguns algoritmos, a maioria que veremos, a hipótese de posição geral sobre as instâncias diz que não existem três pontos colineares ou quatro pontos cocirculares em um conjunto de pontos dados.

É claro que não desejamos excluir instância alguma e que não desejamos fazer esta hipótese, *entretanto*, muitas vezes durante o andamento desta disciplina, será conveniente utilizarmos a hipótese de posição geral para não nos perdermos em detalhes (que são, até certo ponto, menos importantes) e acabemos nos esquecendo do 'coração' do algoritmo que está sendo considerado.

8. Exercícios

1. Use indução matemática para mostrar que
$$1 + 2 + \cdots + n = n(n + 1)/2.$$
2. Use indução matemática para mostrar que uma árvore binária com n nós internos deve ter $n + 1$ folhas.
3. Resolva a recorrência abaixo onde $n = 2^k$ para algum inteiro positivo k e a são constates positivas.
$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + a & \text{se } n > 1 \\ b & \text{caso contrário } (n = 1) \end{cases}$$
4. Resolva a recorrência
$$T(n) = \begin{cases} T(n-1) + a & \text{se } n > 0 \\ b & \text{caso contrário } (n = 0) \end{cases}$$
onde a e b são constantes positivas.
5. Resolva a recorrência abaixo onde $n = 2^k$ para algum inteiro positivo k e a e b são constantes positivas.
$$T(n) = \begin{cases} T(n/2) + a & \text{se } n > 1 \\ b & \text{caso contrário } (n = 1) \end{cases}$$
6. Resolva a recorrência abaixo onde $n = 2^k$ para algum inteiro positivo k.
$$T(n) = \begin{cases} 2T(n/2) + 3n + 2 & \text{se } n > 1 \\ 4 & \text{caso contrário } (n = 1) \end{cases}$$
7. (Exercício 2.7.9 de [5]) Mostre que se $f_1(n) = O(g(n))$ e $f_2(n) = O(g(n))$ então $(f_1+f_2)(n) = O(g(n))$, onde $(f_1+f_2)(n) = f_1(n)+f_2(n)$.
8. (Exercício 2.7.10 de [5]) Mostre que se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ então $(f_1 \cdot f_2)(n) = O((g_1 \cdot g_2)(n))$, onde $(f_1 \cdot f_2)(n) = f_1(n)f_2(n)$.
9. (Exercício 2.7.11 de [5]) Mostre que se $f_1(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f_1)(n) = O(h(n))$.
10. (Exercício 2.7.12 de [5]) Mostre que se $\log_b(n) = \Theta(\log_c n)$ para constantes $b, c > 1$.
11. M(Exercício 2.7.13 de [5]) ostre que $a_m n^m + a_{m-1} n^{m-1} + \cdots + a_0 = O(n^m)$ para constantes arbitrárias $a_m, a_{m-1}, \ldots, a_0$.
12. (Exercício 2.7.14 de [5]) Mostre que $\log n = O(n^k)$ para todo $k > 0$.
13. (Exercício 2.7.15 de [5]) Mostre que $n^k = O(2^n)$ para todo $k > 0$.
14. (Exercício 2.7.16 de [5]) Mostre que se $f(n) = O(g(n))$, então $g(n) = \Omega(f(n))$.

Referências

[1] A.V. Aho, J.E. Hopcroft e J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[2] M. Ben-Or, Lower bounds for algebraic computation trees, *Proc. 15th ACM Annual Symp. on Theory of Comput.*, 1983, pp. 80–86.

[3] T.H. Cormen, C.E. Leiserson e R.L. Rivest, *Introduction to algorithms*, The MIT Electrical Engineering and Computer Scienece Series, The MIT Press, MacGraw-Hill Book Company, 1990, QA758 C811i.

[4] D. Dobkin e R. Lipton, On the complexity os computations under varying set of primitives, *Journal of Computer and System Sciences* **18** (1979), 86–91.

[5] M.J. Laszlo, *Computational geometry and computer graphics in C++*, Prentice Hall, Upper Saddle River, NJ, 1996.

[6] K. Mehlhorn e St. Näher, *The LEDA platform of combinatorial and geometric computing*, Cambridge University Press, 1999, `http://www.mpi-sb.mpg.de/ mehlhorn/LEDAbook.html`.

[7] F.P. Preparata e M.I. Shamos, *Computational geometry: An introduction*, Texts and Monographs in Computer Science, Springer-Verlag, New York, 1985, QA758 P927c.