Brute-Force Search

Peter J. Haas

INFO 150
Fall Semester 2018
Implications

Algorithm Complexity
Hard Problems and NP-Completeness
Algorithm Complexity

Review of number sequences

- A sequence is a function that assigns a number $a_n$ to each positive integer $n$ (or, to change notation, assigns a number $f(n)$ to each positive integer $n$)

- We saw different kinds
  - Constant: $a_n = 3$ for all $n$ [$a_n = a_{n-1}$ with $a_1 = 3$]
  - Linear: $a_n = 3n + 2$ [$a_n = a_{n-1} + 3$ with $a_1 = 5$]
  - Quadratic: $a_n = n^2$ [$a_n = a_{n-1} + 2n - 1$ with $a_1 = 1$]
  - Exponential: $a_n = 2^n$ [$a_n = 2a_{n-1}$ with $a_1 = 2$]
  - Factorial: $a_n = n!$ [$a_n = na_{n-1}$ with $a_1 = 1$]

Definition

The time complexity of an algorithm is the number of steps $f(n)$ that an algorithm takes on the “worst” input of size $n$. The function $f$ is called the (worst case) complexity function for the algorithm.

Examples

- Sort a list of $n$ names using heapsort: $f(n) \approx n \log n$
- Determine whether $P(x_1, x_2, \ldots, x_n)$ is a tautology, using truth tables: $f(n) = 2^n$ predicate evaluations
Is there a Fast Algorithm for My Problem?

Example: Tautology checking with $n$ variables

- Brute force via truth tables: $f(n) = 2^n$
- Can sometimes get a short proof: $p \lor (q \lor \neg q) \equiv p \lor t \equiv t$
- In general, can we find a short (polynomial-time) proof for a given tautology, i.e., that has $n^k$ steps for some $k$? Nobody knows!
- For an arbitrary tautology, does there exist a short proof? Nobody knows!

Example: Sorting $n$ items

- Brute force: Check all $n!$ possible orderings and choose the right one
- Quicksort Algorithm: $f(n) \approx \log(n!) \approx n \log n$

Example: Finding a path from point $A$ to point $B$ in a network

- Brute force: Check all possible network paths (exponential)
- A* Algorithm: Can be polynomial for some problems
Is there a Fast Algorithm for My Problem? (Continued)

Example: Can I send 5 gallons of water from point \( s \) to point \( t \) in a pipe network

- Brute force: Check all possible fluid assignments to pipes (exponential)
- Network Flow Algorithm: Is polynomial for all problems

![Pipe Network Diagram]

**Matching \( n \) dogs with \( n \) people**

- Each person has rank ordering of dogs
- Each dog has rank ordering of people
- Avoid any (dog, person) pair where each prefers the other to their assigned match
- Brute force: check all \( n! \) possible matchings
- Stable Marriage Algorithm: \( f(n) \approx n^2 \) (used for medical residencies)
Hard Problems and NP-Completeness

Some problems are hard, and cannot be solved faster than brute force

Some of these are called **NP-complete problems**
- If you can solve one in polynomial time, then essentially all brute force search problems can be solved in polynomial time
- It is unlikely that such a polynomial-time algorithm exists
- Proving that it doesn’t exist (assuming that it doesn’t) is the greatest unsolved problem in theoretical computer science

Some examples of NP-complete problems
- **Tautology**: Determining for a given compound proposition
- **Traveling salesperson**: Determine the shortest path to visit all nodes in a network and return to the starting point
  - $n!$ possible orders in which to visit $n$ points
- **Subset sum**: Given a set of positive integers, and a target number, is there a subset of the integers that add up exactly to the target?
  - NP-complete in general, easy for small target numbers
Why Should I care About NP-Completeness?

If you are a programmer:

▶ You should know which sorts of problems have solutions and which are hard (NP-complete)
▶ Identifying a problem as NP-complete avoids wasted effort and triggers search for practical workarounds
  ▶ Look for feasible special cases
  ▶ Find approximations to the exact optimal answer