Mat 3770
Algorithmic Paradigms

Spring 2014

Algorithmic Paradigms

Not all problems are solvable by computer

Example: the Halting Problem
Whether or not an arbitrary program halts
on all possible inputs is **not solvable**.

- Problems which are not solvable by computer programs are called **unsolvable**.
- The class of **solvable** problems can be partitioned into two general classes or sets: P and NP.

P and NP Problems

- **P** is the set of **Polynomial Time**, or **tractable**, problems.
- All problems that can be solved by a **deterministic Turing Machine** in polynomial time are in P.

  (**TM**s are the **theoretical** equivalent of "conventional" computers.)
- **NP** is the set of **Non–deterministic Polynomial Time** problems — these are problems for which we can guess the answer (the non–determinism) and then verify it in Polynomial Time... And we always guess the right answer!

P and NP Problems, Continued

- All problems that can be solved in polynomial time by a non-deterministic TM (NTM) are $\in$ NP.
- NTMs are similar to parallel machines with as many processors as we may need—every time there is a choice(s), start a new processor and pursue *all* paths. This is not very realistic.
- It is not known if P = NP.
  It is generally *believed*, but it has not been *proven* that
  $P \neq NP$

NP–Complete Problems

- Another class of problems: **NP–Complete**, the "hardest" problems in **NP**.
- These problems have the property that if **any one** of the NP–Complete problems can be solved in polynomial time, then **every** problem in NP has a polynomial time solution.
- If this is ever proven, one result would be that P = NP.

  I.e., all the problems that have been shown to be in NP can actually be solved in Polynomial Time.
- However, there are several thousand problems which have been shown to be in the class NP-Complete, so the likelihood is fairly slim that anyone will be able to prove any NP–Complete problem is in P.

NP–Complete Versus NP–Hard Problems

A problem may be stated in two different ways:

1. **Decision version**: Does there **exist** a subgraph H of a graph G, which has a specific property and has size k or bigger?
2. **Optimization version**: What is the largest subgraph of G with a specific property?

- The **Optimization** version of any problem is called **NP–Hard** if the **Decision** version of the problem is **NP–Complete**.
- Most optimization problems in physical (VLSI chip) design are NP–Hard.
- If a problem is known to be NP–Complete or NP–Hard, it is **unlikely, although not proven**, that a polynomial time algorithm does NOT exist for the problem.

Algorithmic Choices

- Because the problems in physical design are so complex, however, we still need the help of computers to find feasible (workable) solutions, which are not necessarily optimal.
- This is true of many problems, not just VLSI design.
- We have four choices:
  1. Exponential (or worse) Algorithms (optimal)
  2. Special Case Algorithms
  3. Approximation Algorithms
  4. Heuristic Algorithms

I. Exponential Algorithms

- If the input size is sufficiently small, it *may* be feasible to use algorithms with exponential time complexity.
- If the solution to a certain problem is critical to the chip performance, it may be practical to spend extra resources to solve the problem optimally. (E.g., using Integer Programming).
- One possibility is to solve small subproblems optimally, then use another algorithm to combine them into an overall solution.

II. Special Case Algorithms

- It may be possible to simplify a general problem by applying some restrictions to the problem.
- Layout problems are easier for simplified VLSI design styles such as standard cell, and this allows the use of special case algorithms.

III. Approximation Algorithms

These algorithms produce results with a **guarantee** they will never be worse than a lower bound determined by the **performance ratio** of the algorithm:

$$\frac{\textit{approximate solution}}{\textit{optimal solution}}$$

IV. Heuristic Algorithms

These algorithms produce (feasible) solutions, but do **not** guarantee the optimality of them.

To be effective, a heuristic algorithm must:

1. have low time and space complexity
2. produce an optimal or near–optimal solution in most *realistic* problem instances, and
3. have good average case complexities

An Example

The *channel routing* problem (determining what path the interconnect wire should take) in general is NP–Complete, but in many cases linear algorithms have been developed that come within 1 or 2 tracks of optimal.

Thus, for all practical purposes, the channel routing problem is considered solved.

ALGORITHMIC PARADIGMS

Classes of approaches to problem solving

1. **Exhaustive Search** (most naive)

- inspect the entire Solution Space by considering every possible solution and evaluating the cost of each of them
- will (eventually) produce an **optimal** solution
- very slow (not just hours or days, but a lifetime or more!)

Search: Branch and Bound

- a general and usually inefficient method for solving optimization problems
- the configurations of solutions can be stored in a tree structure where the traversal of one (downward) path in the tree produces a (possibly infeasible) solution
- aim: avoid searching the entire space by stopping at nodes (**pruning**) when it is ascertained an optimal solution cannot be represented by the current path
- generally, it is hard to claim anything about the running time of B & B algorithms

ALTERNATIVES

2. **Greedy Approach**

- do what looks good first: at each step, among the available choices, pick one that results in a **locally** optimal solution
- typically simpler than other classes of algorithms
- does not always produce **globally** optimal solutions
- even if they do produce optimal solutions, that can be difficult to prove

3. Dynamic Programming

- partitions problem into sub–problems, solves the sub–problems (perhaps recursively), then combines these solutions into a solution for the original problem
- applied when sub–problems are **not** independent of each other
- each sub–problem is solved once and the solution is saved for other sub–problems

Dynamic Programming, Continued

- effective when a given sub–problem may arise from more than one partial set of choices
- first, the structure of an optimal solution is recursively defined
- finally, the value of an optimal solution is computed in a bottom–up fashion

4. Hierarchical Approach

(aka **Divide and Conquer**)

- recursively partitions problems into independent sub–problems of approximately equal size (balanced)
- partitioning is usually top–down while the solution is constructed bottom–up (typically)

5. Mathematical Programming

(e.g., 0–1, linear, and integer programming)

- a set of **constraints** on the solution are expressed as a collection of inequalities
- the **objective function** is a minimization (or max) problem subject to the set of constraints

6. Simulated Annealing & Other Probabilistic Algorithms

- A technique to solve general optimization problems
- especially useful when the solution space of the problem is not well understood
- originated from observing crystal formation of physical materials (temperature, random movement)
- SA examines the configurations (feasible solutions) of the problem in sequence

- SA evaluates these feasible solutions as they are encountered and moves from one solution to another
- at "high temperature" — lots of random movement
- as "temperature" is lowered, less movement occurs, and we approach a minimum solution
- Stochastic Evolution, similar to SA, has produced even better results

7. Genetic Algorithms

(based on population genetics)

- **population**: a solution sub–space
- **evolve**: be modified
- **crossover**: merge previous solutions
- **mutation**: modify previous solution
- **fitness value**: a measure of the competence or quality of a solution

- at each stage of a GA, a population of solutions is stored and allowed to evolve through successive generations
- to create a new generation, new solutions are formed by crossovers and / or mutations
- the solutions selected for the next generation are probabilistically selected based on a fitness value