6. Intractability and approximation

1. Undecidability and intractability
2. Complexity classes of problems
3. Approximation algorithms
4. State-space search
5. Randomized and evolutionary algorithms

Inquiry

• How hard is optimization?
• How hard can really hard problems be?
• How helpful is it to mathematically characterize some problems as hard?
• Are some parts of these problems easier?
• How do you address hard problems that involve thinking and decisions?
Inquiry

• How helpful is it to mathematically characterize some problems as hard, or intractable?

• Are the problems solvable in reasonable time the same problems whose solutions can be checked in reasonable time?

• Are approximate solutions a practical way to address hard computational problems?

Inquiry

1. Is this an algorithm? (Does it halt for all \( n \)?)

\textbf{Mystery}(n)

\begin{verbatim}
while (n > 1)
    if n is even
        n \leftarrow n / 2
    else
        n \leftarrow 3n + 1
\end{verbatim}

Return true

2. What is involved in deciding whether to shoot at time \( t \), in a game of basketball?
6. Intractability and approximation

Topic objective

Describe the notion of *intractability* mathematically, including complexity classes, and ways to overcome this obstacle.

1. Undecidability and intractability

- Do you ever have problems that are *unsolvable*?
- *Problems* that are so hard to solve that it’s not worth the time?
Subtopic objective

6.1 Describe an intractable problem**

Decidable and tractable problems

- Some functions are \textit{computable}, others not
- Problems that are uncomputable \textit{predicates} are called \textit{undecidable}
- Decidable problems that take too much time to be worth solving are called \textit{intractable}
- Recognizing undecidable and intractable problems can save unbounded effort
Problems about programs

• Does program \( P \) halt on input \( x \)?
• Does program \( P \) halt on all inputs?
• Does program \( P \) halt on any inputs?
• Does program \( P \) halt in fewer than \( n \) steps on any inputs?
• What is output of program \( P \) on input \( x \)?
• \textit{The characterization of these problems is surprising; they are all undecidable}

Notation

• \( \phi_P \) is the function computed by program \( P \)
• \( #(P) \in \mathbb{N} \) is a natural number that encodes source of \( P \)
• \( f(x) \downarrow \) means that function \( f \) is defined for \( x \)
• \( f(x) \uparrow \) means that \( f \) is is \textit{partial} and is undefined for \( x \)
• Programs that sometimes loop forever compute partial functions
Undecidability of the HALT problem

- Let $\text{HALT}(P, x) =$
  
  \[
  \begin{cases} 
  \text{true} & \text{if program } P \text{ halts on input } x \ (\phi_P(x) \downarrow) \\
  \text{false} & \text{otherwise}
  \end{cases}
  \]

- If $\text{HALT}$ is decidable by some program $H$, then a program $S$ can be constructed, with input $x$, that halts iff $H$ determines that the program with code $x$ loops forever on input $x$

- Consider how $S$ behaves on input $\#(S)$:
  - $S$ halts ($\phi_S(\#(S)) \downarrow$) if $S$ hangs on $\#(S)$
  - $S$ hangs ($\phi_S(\#(S)) \uparrow$) if $S$ halts on input $\#(S)$

- Since $S$ hangs iff $S$ halts, $S$, hence $H$, cannot exist

Lower time bounds for problems

- For problems solved by repeated comparisons, algorithm may descend a decision tree

- Examples: Linear search and binary search make decisions at each comparison

- Minimum height of decision tree gives information-theoretic lower bound on running time for the best algorithm that solves the problem

- Example: Sorting by comparisons is $\Omega(n \ lg \ n)$
Some intractable problems

- Satisfiability, 3SAT
- Hamiltonian cycle, traveling salesperson
- Set partition
- Independent sets
- Stable marriage
- Channel assignment

Satisfiability (SAT)

- Given a formula $\phi$ in propositional logic (logic with $\neg$, $\wedge$, $\vee$, $\Rightarrow$, no predicates, no quantifiers), does a set of variable assignments exist that satisfies $\phi$ (makes $\phi$ true)?

- Examples: (a) $p \wedge q \wedge r$
- (b) $(p \wedge q \vee \neg q)$
- (c) $p \wedge \neg(q \vee \neg q)$
Satisfiability as a decision problem

- Let $SAT = \{ \phi \mid (\exists \rho) \phi_{\rho} = \text{true} \}$ where $\phi$ is a formula in logic and $\rho$ is a set of truth assignments
- Example: $\rho = \{ (p, \text{true}), (q, \text{true}), (r, \text{false}) \}$
- Then $SAT$ is the set of satisfiable formulas
- Thus $SAT$ is a language
- Equivalently, $SAT$ is the decision problem of telling whether a given formula is satisfiable (whether a formula is in the language $SAT$)

Algorithm for SAT

- Let $\phi$ be a sentence in propositional logic using Boolean variables $x_1, x_2, \ldots, x_n$
- $SAT(\phi) = \begin{cases} \text{true} & \text{if } nvars(\phi) = 0 \land \text{eval}(\phi) = \text{true} \\ \text{false} & \text{if } nvars(\phi) = 0 \land \text{eval}(\phi) = \text{false} \\ \land_{i \leq n} SAT(\phi[x_i / \text{true}]) \lor \land_{i \leq n} SAT(\phi[x_i / \text{false}]) & \text{otherwise} \end{cases}$
- Strategy: try each branch of decision tree, or each row of truth table
3-Satisfiability (3SAT)

- **Conjunctive Normal Form (CNF):** In propositional logic, a CNF formula is the conjunction of a set of clauses consisting of the disjunction of literals,

- **Example:** \((p \lor q) \land (q \lor r \lor s)\) is in CNF

- **3-CNFS:** formulas in which every clause contains exactly three literals

- **3SAT:** Satisfiability problem for 3-CNF formulas

HALT and SAT are about computing procedures

- **SAT** input is logic circuit or a spec for a table lookup; **HALT** input is a program

- Other intractable and undecidable problems about computation can be reduced to **SAT** and **HALT**

- The bad news: computation about computation has limits
What separates intractability from undecidability

- **Intractable** problems are known or believed to require running times of $\Theta(2^n)$ for input of size $n$
- **Undecidable** problems have no solutions that terminate in finite time

**Intractable problems**

- Some problems have no known *polynomial time* ($O(n^k)$) solutions for any constant $k$
- These are considered *intractable* because for sufficient $n$ they may take “forever” in practice
- Exponential-time examples: Hanoi, password guessing, understanding English
- Others are called *NP-complete* problems: solutions are checkable, but not known to be obtainable, in polynomial time
Hamiltonian-cycle problem

- **Problem:** Given a graph \( G = \langle V, E \rangle \), is there a path in \( V^* \) that starts and ends with the same vertex and that passes through every vertex exactly once?

\[
\begin{align*}
\text{Yes} & \quad \begin{array}{c}
\text{No}
\end{array}
\end{align*}
\]

- A brute-force solution is to generate and test every possible sequence of \(|V| + 1\) vertices to see if it is a Hamiltonian path
  - What is the complexity of the test operation?
  - What is the complexity of generate-and-test?
  - Can you think of a faster algorithm?

The traveling-salesperson problem

- For a weighted digraph \( G \), and an integer \( B \), is there a cycle that passes through all vertices and whose weight is not more than \( B \)?

\[
\begin{align*}
\text{All paths: } \{ (a,b,a) \} \\
\text{All paths: } \{ (a,b,c,a), (a,c,b,a) \}
\end{align*}
\]

- This problem is considered *intractable*, but a claimed solution can easily be checked
- At least as hard: What is the *least-cost* such path?
Set partition

- Given a finite set \( S \subseteq \mathbb{N} \), can \( S \) be partitioned into two sets, \( A \) and \( B \), s.t. \( \sum_{x \in A} x = \sum_{y \in B} y \)?
- Problem has brute-force \( \Theta(2^n) \) solution: sum every subset and its complement, see if any sums match.
- No faster solution is reported.
- Challenge: Reduce SAT to set-partition.

Independent set of vertices

- Independent set: a set of vertices in a graph, of which no two are adjacent.
- Problem: Does graph \( G \) contain an independent set of size at least \( k \)?
- Brute-force solution (\( \Theta(2^{\lvert V \rvert}) \)): test all subsets of \( V \).
- Challenge: Reduce SAT to independent-set.

[pic]
Matching problem

- **Matching** (in a graph): a subset of edges s.t. no two edges share a vertex
- **Maximum matching**: A matching with maximum number of edges
- **Example**: Where \( E \) represents common interests among people, find best pairing of a set of people so that the most pairs share a common interest. **Complexity**: \( O(2^{|E|}) \)
- \([pic]\)

Maximum matching

- **Matching** (in a graph): a subset of edges s.t. no two edges share a vertex
- **Maximum matching**: A matching with a maximum number of edges
- **Example**: Where \( E \) represents common interests among people, find best pairing of a set of people so that the most pairs share a common interest. **Complexity**: \( O(2^{|E|}) \)
- **Brute-force solution**: Generate all subsets \( S \) of \( E \); for each subset check all \( ((u, v), (w, x)) \in S \times S \) for a common vertex
### Channel assignment (telcom)

**Problem:**
- Given $b$ base stations, each operating $t$ cell phones, each of which has $f$ frequencies available and $f'$ frequencies in use ...
- Which frequency should be assigned to a given cell-phone call while avoiding interference among frequencies that are the same or close to each other?

### Public-key cryptography

- Used for transmission, e.g., of credit-card numbers
- Receiver sends sender a key, $k$, a product of primes
- Sender computes RSA function $y = f(k, x)$, where $x$ is message to send; sends $y$
- Only the receiver can extract $x$ from $y$ by computing $x = f^{-1}(k, y)$
- To extract $x$ from $y$, a snooper would have to factor the prime $k$ (an intractable problem)
- Sender can factor $k$ because it derived $k$ from primes in the first place
Observations about these problems

• Each seems to require generating and checking a very large set of candidate solutions
• Checking a given candidate solution is easy enough
• But the number of candidates seems to be $O(2^n)$
• For $n > 100$, the time required could be too much for practical purposes

2. Complexity classes of problems

• What are the running times of some search and sort algorithms?
• What are the fastest running times for the searching and sorting problems?
• How hard is the Towers of Hanoi problem?
Subtopic objectives

6.2a Describe complexity classes for tractable and intractable problems*

6.2b Use a reduction to explain why a problem is intractable

Classifying problems

- **Defn:** the set of problems/ languages with \(O(f(n))\) solutions (algorithms) for input of size \(n\)
- **Example:** sorting is in \(TIME(n \lg n)\) and in \(TIME(n^2)\)
- \(SPACE(f(n))\): the set of languages recognized by algorithms using space \(O(f(n))\)
- **Example:** Fibonacci is in \(SPACE(O(1))\) and \(TIME(n)\) (though no algorithm to compute it is both linear in time and constant in space)
 Polynomial-time complexity

- \( \mathcal{P} = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k) \)

- That is, \( \mathcal{P} \) is the set of problems decidable in \( O(n^k) \) time (polynomial time), where \( n \) is the size of the problem and \( k \) is a constant

- **Examples of problems in class \( P \)**
  - Searching a collection is in \( \text{DTIME}(n) \)
  - Sorting an array, \( \text{TIME}(n \lg n) \)
  - Searching a balanced BST, \( \text{TIME}(\lg n) \)
  - Generating a graph reachability matrix, \( \text{TIME}(n^3) \)

---

Does \( P \) capture tractability?

- **Cook-Karp Thesis**: The problem of deciding membership of strings in a language is tractable only if the language is polynomial-time decidable

- Counter argument: what if \( T(n) = O(n^{1000}) \)?

- But almost all algorithms in actual use have running times that are polynomials of degree 3 or less, with small constant terms or factors

- Hardware architectures are of equivalent speed, within a polynomial function
**P is closed under complement**

**Proof:**
- Suppose language $L$ is in $P$, $L(P) = L$ for program $M$; $M$ runs in polynomial time
- Modify $M$ to $M'$, which accepts the complement of $L$ by returning $true$ whenever $M$ is about to return $false$; returning $false$ whenever $M$ is about to return $true$
- $M'$, like $M$, runs in polynomial time
- **Example:** If SAT is in $P$, then so are TAUT, UNSAT

---

**Exponential time**

- $EXP = EXPTIME = TIME(2^n)$
- $SAT$ seems to be in $EXPTIME$ but not $P$, because it seems that to solve SAT, $\Theta(2^n)$ candidate solutions need to be generated and tested
- Problems that are in $EXP$ but not in $P$ are considered intractable
- We say that these problems are $\Omega(2^n)$
Deterministic time complexity

- Computers in general are deterministic, because their output at each step depends on input and variable values

- \( DTIME(T(n)) = \{ L \subseteq \Sigma^* \mid L \text{ is accepted by a deterministic algorithm in time } T(n) \} \)

- \( DTIME (T(n)) \) is a complexity class of problems for any function \( T(n) \)

Polynomial-time verifiability

- Some problems are such that a possible solution can be verified as correct or not in polynomial time

- Many hard problems with only EXP-time known solutions nevertheless have this property

- A verifier is an algorithm \( V \), for language \( A \), where \( A = \{ w \mid V \text{ accepts } (w, c) \text{ for some } c \} \)

- Here \( c \) is a certificate that verifies the solution
**Nondeterministic algorithms**

- … may generate values arbitrarily (by “guessing”) during execution
- … always return “yes” on some execution in instances of decision problems whose solutions are “yes”
- … never return “yes” when answer is “no”
- … may fail to say “no” when answer is “no”

**Nondeterministic time complexity**

- $NTIME(T(n)) = \{ L \subseteq \Sigma^* | L \text{ is accepted by a nondeterministic algorithm in time } T(n) \}$
- To solve SAT, a nondeterministic algorithm could “luckily” start by generating and testing a set of variable assignments that satisfy the formula and answer “yes” in $O(n)$ time
- SAT is in $NTIME(n)$ for propositional formulas of length $n$, because evaluation of formulas is linear-time
Intuitive notion of \textit{NTIME}

- \textit{NTIME} \((T(n))\) is the set of problems for which a correct solution can be verified in time \(T(n)\)
- The nondeterminism corresponds to the assumption that a correct solution is generated nondeterministically ("magically") and only needs to be verified

\textbf{The \textit{NP} complexity class}

- \(\textit{NP} = \bigcup_{k \in \mathbb{N}} \textit{NTIME}(n^k)\)
- That is, \(\textit{NP}\) is the set of problems decidable by nondeterministic algorithms in polynomial time
- \textit{Intuition}: \(\textit{NP}\) problems are those for which a particular candidate solution, once found, can be verified in polynomial time
- \textit{Question}: Is \(\mathcal{P} = \textit{NP}\)?
**NP** and efficient certifiers

- A **certifier** for problem $L$ (a set of strings) is an algorithm $C$ that takes evidence (certificate) $P$ and a string $x$, as inputs, and outputs $true$ iff $p$ is conclusive evidence that $x \in L$
- **Example**: for SAT, $p$ is a truth-value variable assignment that shows that formula $\phi$ is satisfiable, i.e., $\phi(p) = true$
- Efficient certifiers are algorithms with polynomial running times
- $NP = \{L \mid L$ has an efficient certifier$\}$

**Reducibility**

- Problem $A$ is **reducible** to problem $B$ if there is an algorithm that can transform any instance of $A$ to an instance of $B$
- If $A$ is reducible to $B$ ($A \leq B$), then we may say that $B$ is at least as hard as $A$
- **Example**: Finding the largest element of an array is reducible to sorting the array, so the maximum-value problem is reducible to the sorting problem.
Reductions

- We reason as follows: If problem $B$ is reducible to problem $A$, then $A$ is at least as hard as $B$ to solve.
- Example:
  - Making millions from pennies every day on the stock market is reducible to accurately forecasting future stock prices.
  - But hardly anyone makes millions from pennies on the stock market.
  - Hence accurately forecasting stock prices is a very hard problem.

Polynomial-time reducibility

- Problem $P$ is polynomial-time reducible to problem $Q$ iff some polynomial-time algorithm transforms “yes” instances of $P$ to “yes” instances of $Q$ and “no” instances of $P$ to “no” instances of $Q$.
- Example: A class of problems expressed in English can be translated into French in polynomial time.
- Example: 3SAT is reducible to independent-set in polynomial time.
Theorem: SAT ≤ 3SAT

Proof: Any propositional formula may be rewritten in PTIME as the conjunction of clauses of three literals, using extra variables.

Example: \( p \lor q \lor r \lor s = (p \lor q \lor x) \land (r \lor \neg x \lor y) \land (s \lor \neg y \lor \neg x) \)

3SAT ≤ Independent-set

• Given \( G = (V,E) \), \( S \subseteq V \) is independent iff \( \neg \exists u, v \in S \) \( ((u, v) \in E) \)

• Instance: Does \( G = (V,E) \), \( V = \{a, b, c\} \) contain a 2-vertex independent set?

• Conditions: For a 3-vertex graph:
  (1) \( S \) contains at least 2 vertices:
      \( (a \land b) \lor (a \land c) \lor (b \land c) \)
  (2) No pair \( \{u, v\} \in S \) is an edge in \( E \):
      \( \neg ((a \land b \land (a,b)) \lor (a \land c \land (a,c)) \lor (b \land c \land (b,c))) \)

• \( S \) is independent iff \((1) \land (2)\) is satisfiable
**NP completeness**

- For many problems in NP, e.g., SAT, no polynomial time solution is known
- Problems to which SAT or similar problems are reducible are called **NP-complete**
- *Example:* Traveling Salesperson
- If any of these has a polynomial-time solution, then all do, hence $P = NP$
- Most researchers think $P \neq NP$, i.e., NP-complete problems are believed to have no polynomial-time solutions

**SAT, NPC, and $P = \_? \_N P$**

- *Theorem* (Cook): *Every* problem in NP is reducible to SAT
- *Proof:* Construct a nondeterministic program that decides the problem, convert its computation steps to a CNF formula [Explain]
- *Theorem:* $P = NP$ iff SAT $\in P$
If *any* NPC problem is in \( P \), then so are *all* NPC problems

- **Theorem:**
  - Suppose problem \( L \in P \) and \( f : \Sigma^* \rightarrow \Sigma^* \in O(n^k) \) for some constant \( k \) (i.e., \( f \) is in \( P \)), and \( Q = \{ x \in \Sigma^* \mid f(x) \in L \} \)
  - Then \( Q \in P \)

- **Intuition:** If problem \( Q \) is polynomial-time reducible to \( L \), and \( L \) is polynomial time, then \( Q \) is polynomial time too.

- **Note:** Another theorem shows by construction that SAT is reducible to *any* NPC problem

---

**Proof that SAT is NPC**

A. \( (SAT \in NP) \): If a propositional-logic formula is satisfiable, then a nondeterministic program can guess and verify a model for it in \( O(n) \) time

B. (Any NPC problem \( L \) is reducible to SAT):
   1. Let \( P \) be a program that decides \( L \) given input \( w \)
   2. Convert \( P \)'s sequence of steps leading to an output of *true* into a (large) logic formula in which variables tell whether \( P \) has given states
   3. This formula is satisfiable if \( P \) accepts \( w \) in polynomially many steps
Intractability, NPC, and EXP

- Certain problems are thought or known to have only exponential-time, \( O(2^n) \), solutions
- \((EXPTIME - P)\) and NPC are the \textit{NP-hard} problems, considered \textit{intractable}
- Problems of planning, scheduling, routing, drawing inferences, understanding language, etc., are in general intractable
- \textit{What to do:} Replace intractable problem with a simpler one, e.g., one with a probabilistic or approximate solution

Some other NPC problems

- Hamiltonian Path; Traveling Salesperson, because HP \( \leq \) TSP
- Graph coloring
- Vertex cover, set cover, set packing
- \textit{Proof of NPC-ness:} SAT can be reduced to each
- \textit{Other classes of NPC problems:} partitioning, sequencing, numerical, channel assignment
Graph coloring is NPC

- Vertices correspond to regions, edges to contiguity of regions
- \( k \)-coloring: a function \( f: V \rightarrow \{1, 2, \ldots, k\} \) s.t. \( \forall (u, v) \in E \ f(u) \neq f(v) \)
- Coloring problem: Does \( G \) have a \( k \)-coloring?
- Applications: To determine conflicts in resource allocation situations, e.g., allocation of variables to registers in compiler design; wavelength assignment for wireless devices

Sequencing problems that are NPC

- Hamiltonian Path, TSP, Hamiltonian Cycle
- These work with permutations
- Proofs of intractability reduce SAT or 3SAT to the problem
- Technically, we represent the \( 2^n \) Hamiltonian-Cycle paths, for example, as possible variable assignments that correspond to direction traversing path
### PSPACE

- **Definition:** the set of problems solvable in storage space that is polynomial in size of input
- $P \subseteq PSPACE$, because no more space could be required than that for which time is available
- *Counting* to an exponential value is solvable in PSPACE by reusing space
- $NP \subseteq PSPACE$
- A problem is in $PSPACE$ iff its complement is too
- $PSPACE$-complete problems: planning with conditions, games, quantified SAT

### Co-NP and more open problems

- **Co-NP:** Problems whose complements are NP
- *Open* (like $P = NP$): $(NP = co-NP)$
- *Open problems* in mathematics are assertions neither proven nor refuted
- *Known:* If $NP \neq co-NP$ then $P \neq NP$
- $(NP \cap co-NP)$: well-characterized problems, because either a *true* or a *false* result has a certificate
- Also open: $P = (NP \cap co-NP)$?
P, NP, NPC, co-NP, co-NPC

- Suspected relationship, assuming $P \neq NP$

- *Theorem*: $NP = co-NP$ iff every complement of an NPC problem is NP

- Examples of problems in co-NP but not in NP:
  - USAT, unsatisfiable logic formulas
  - TAUT, tautologies in propositional logic

3. Approximation algorithms

- How do you solve the basketball-shooting decision problem?
- Do you usually seek *optimal* or *good enough* solutions to problems in your life?
6. Intractability and approximation

Subtopic objective

6.3 Explain an approximation algorithm

A strategy for designing polynomial-time algorithms

• Try to reduce the problem in PTIME to a PTIME problem; then solve the PTIME problem
• Theoretical basis: $L \in \text{PTIME}$ iff $(L \leq_p M) \land (M \in \text{PTIME})$
• Contrapositive of this is: $(L \leq_p M) \land (L \not\in \text{PTIME}) \Rightarrow (M \not\in \text{PTIME})$
Tractable versions of intractable problems

- **Vertex cover** seeking small-sized cover, because \( C(n, k) \) with constant \( k \) is a polynomial of \( n \)
- **Note:** \( k \) that is small but still too large can lead to impractical polynomial times, e.g., \( n^{100} \)
- **Independent set on trees:**
  Converting graphs \( G \) to their *tree decompositions* whose vertices are sets of vertices in \( G \)

Approximation algorithms

- Many use **heuristics**: rules of thumb based on human experience; e.g., packing largest items first
- **Accuracy ratio (AR)**: \( f(s_a) / f(s^*) \), where heuristic \( s_a \) minimizes \( f \), and \( s^* \) is an exact solution
- **Performance ratio**: lowest upper bound on AR
- **c-approximation algorithm**: one whose perf. ratio is at most \( c \), i.e., \( f(s_a) \leq f(s^*) \) for any instance
- **Continuous functions**: Newton’s method
  \[
  x_{n+1} = \frac{x_n + f(x_n)}{f'(x_n)}
  \]
  gives approximate sequence; e.g., for square root
Types and analysis of approximation algorithms

• *Types:* greedy, with approximate results; center selection; set cover
• *Priced* for constraint enforcements
• Linear programming and rounding
• Dynamic programming on rounded input
• *Analysis of performance:* In maximization problems, we seek to set a lower bound on ratio of approximate results to optimal ones

Minimum set cover problem

• *Given:* A set $U$ of $n$ items, with weights $w_{1..n}$, and a collection of $m$ sets $S_{1..m}$, whose union is $U$
• *Problem:* Find a minimum-weight collection of sets $S_i \subseteq S$ whose union is $U
Greedy approximate solution to minimum set cover problem

- Algorithm to generate minimum set cover for set $S \subseteq U$ with $w_i = \text{weight}(s_i)$:
  
  ```
  y \leftarrow \emptyset
  R \leftarrow U
  
  \text{while } R \neq \emptyset 
  
  \quad y \leftarrow y \cup s_i \text{ s.t. } s_i \text{ minimizes } (s_i \div |w_i \cap R|)
  
  \quad R \leftarrow R - \{s_i\}
  
  \text{return } y
  ```

- Where $d^*$ is optimal maximum set size, $y$’s weight is within $O(\lg d^*)$ of optimal weight

Approximate load balancing

- **Problem:** assign jobs to machines to balance machines’ loads; problem is NPC
- **Greedy strategy:** Assign jobs in sequence to machines with smallest current load
- In most cases, load is balanced reasonably well
- [pix: pp. 601; 604 (pathological)]
- **Enhancement:** sort jobs in descending order of processing time
Linear programming and Simplex

- **Linear-programming problem:** maximize or minimize \(c_1 x_1 + \ldots + c_n x_n\) s.t. for \(i = 1 \ldots m\),
  \[a_{i1} x_1 + \ldots + a_{in} x_n \leq b_i\) or \(\geq b_i\) or \(= b_i\)

- **Example:** Maximize \(3x + 5y\), given simultaneous equations \(x + 3y \leq 6, \ x + y \leq 4, \ x \geq 0, \ y \geq 0\)

- **Solution:** Feasible region (p. 338a), with maximal point (3, 1)

- **Simplex method:** Find one extreme point in feasible region; choose adjacent extreme points as long as they give improved value

Simplex method algorithm

- **Algorithm:**
  Initialize tableau
  While some entries in “objective row” are < 0
  Find entering variable (< 0) in obj. row
  Find departing var. in entry var.’s column
  Compute new tableau from old

- **Complexity:**
  – Non-terminating in rare cases
  – Worst case: \(O(n^2)\)
  – Expected in practical use: \(O(m^2n)\)
The maximum-flow problem

- Assume connected weighted digraph with \( n \) vertices, edge set \( E \)
- Digraph has exactly one sink, one source
- Weight of edge, \( u_{ij} \), is capacity to handle flow
- Maximum-flow problem is to maximize flow \( v = \sum x_{ij} \) for all \( j \) where \((1, j) \in E\), subject to constraint that inflow, outflow of each vertex are equal

Solving maximal-flow problem

- Repeatedly find flow-augmenting paths starting at zero flow
- Cut: set of edges with tail in one partition of vertices (containing source) and head in the other partition (containing sink)
- Edmunds-Karp: Shortest-augmenting path algorithm, using breadth-first search to find shortest augmenting paths
- Complexity: \( O(nm^2) \)
- Edmunds-Karp uses theorem that value of maximum flow equals capacity of minimum cut
4. State-space search

Q: Rationality in economic theory is
   a. The use of predicate logic
   b. The use of rational numbers
   c. Acting in one’s best interests using heuristics
   d. Acting in one’s best interests using perfect information
   e. Following instructions

Subtopic objective

6.4 Describe the notion of state-space search
6. Intractability and approximation

State spaces and pruning

- **Space**: set of possible arrangements of values, e.g.:
  - Board configurations in board games
  - Paths in a graph
  - Arrangements of items in a knapsack
  - Assignments of truth values in a formula

- **Transitions** from state to state may be defined, we can express them as a tree with start state at root

- **Goal**: Reduce size of state-space tree explored by algorithm, by pruning useless branches

- **Example**: n-queens chess problem

State transition systems

- Let $S$ be state space of solutions to a problem, then $S_1$ are the 1-component prefixes to solutions (1-edge paths, first moves in a game, etc.)

- Let $T = \langle S, \Rightarrow \rangle$ be a transition system where
  - $S$ is a set of states
  - $\Rightarrow$ is a transition relation in $(S \times S)$

- For $q \in S$, $\text{next}(q) = \{ \sigma \in S \mid q \Rightarrow \sigma \}$

- **Example**: Rules of chess, edges in graph

- A **solution path** is a sequence $X[1..n]$ of states that satisfies a goal constraint, and $(\forall i \leq n) \ X[i] \Rightarrow X[i + 1]$
Example: Finding a path

- **Problem:** In graph $G$, is there a path from vertex $a$ to vertex $b$?
- **Generate-and-test** solution consists of exploring the tree of paths from vertex $a$ to others
- One way to **generate** paths is **depth-first**: the next possible solution to generate is the previous one, extended by an edge
- To **test** a path, just see if it ends at $b$
- **Note:** This problem is easier than most state-space searches

Generate and test

A general-purpose algorithm for state-space search

To solve a given problem:

1. Design:
   a. a **test** of proposed solutions
   b. a **generator** of possible solutions

2. While ($\neg$problem-solved $\land$ $\neg$time-expired)
   - generate a possible solution
   - test it
Iterative improvement

- **Problem class:** Optimization under constraints
- **Strategy:** Find a *feasible* solution, improve it by successive steps
- **Obstacle:** *Global* maxima/minima (the objective) may differ from *local* ones
- **Cases:**
  - Simplex method for linear programming
  - Maximal flow
  - Maximal bipartite matching
  - Stable marriage
  - Hill climbing

Local search

- Explores space of possible solutions by moving from one candidate solution to a “nearby” one
- Often uses *heuristics* (“rules of thumb”)
- Intuitive model: energy minimization in physics (e.g., falling objects are losing potential energy)
- Algorithms are often without proven performance functions
- Design strategies define *neighbor relation* for states of solution space
Iterated local search (hill climbing)

**Search (S)**

```
repeat
  c ← random (S)
  repeat
    changed ← false
    for each t ∈ T
      c' ← t(c)
      if eval(c') > eval(c)
        c ← c'
        changed ← true
    until ¬changed
  until eval(c) is acceptable
return c
```

- Explores search space $S$ randomly using transformation set $T$ and fitness function $eval$
- Addresses $n$-queens, TSP, SAT
- A similar algorithm, simulated annealing, solves the independent-set problem

Vertex cover

- **Intuition:** A vertex cover is a subgraph that contains all the vertices of its parent graph
- **Given:** $G = (V, E), S ⊆ V$
- **Definition:** $S$ is a vertex cover of $G$ iff $(u, v) ∈ E$ $(u ∈ S ∨ v ∈ S)$
- Large vertex covers are easy to find; e.g., $V$ is a vertex cover of $G$
- **Problem:** Does $G$ have a vertex cover of size $≤ k$?
- **Note:** $S ⊆ V$ is an independent set iff $(V − S)$ is a vertex cover of $V$
Solution to vertex cover

• *Neighbor relation*: a possible solution with a vertex added or deleted

• *Greedy heuristic*:
  – choose neighboring solution with minimal cost (gradient descent)
  – find local minima
  – in a tree-shaped graph, only luckily deleting a leaf can lead to optimal solution

Approaches to some hard problems

• *Backtracking* and *branch-and-bound* solve some search problems in large state spaces

• It is very hard to predict whether these approaches will solve a particular problem in reasonable time

• *Backtrack example*: Finding an English word that satisfies a certain definition [why?]

• *Branch-and-bound example*: Finding best-score word in a turn at Scrabble
6. Intractability and approximation

**Backtrack** \((X [1..i])\)

If \(X [1..i]\) is a solution
    return \(X [1..i]\)
else
    for each state \(q\) in next\((X [1..i])\)
        \(X [i+1] \leftarrow q\)
        return **Backtrack** \((X [1..i+1])\)
Return *fail*

- How does this differ from depth-first search?
- Why does recursive call have *larger* parameter than original parameter?

**Branch-and-bound**

- In optimization problems, solutions may be
  - *Feasible* (satisfy constraints)
  - *Optimal* (yield best value of objective function, e.g., least-weight path)
- In branch-and-bound, search of a state-space tree stops when
  - Node represents no feasible solution, or one;
  - Value of node’s *bound* [define] is not better than best found so far
- *Example*: Assign \(n\) people to \(n\) jobs with best overall fit
**Knowledge, goals, and rationality**

- **Principle of rationality**: An agent will select an action if it has *knowledge* that the action will lead to a system *goal*.
- **Knowledge**: Whatever an agent has that enables it to *compute* its actions according to the principle of rationality.
- **Concept of rationality** derived from economics; later applied in game theory.

**Bounded rationality**

- Notion suggested by Herbert Simon, 1972, as alternative to classical rationality assumption of economic theory.
- **Argument**: Humans have limited knowledge and resources for decision making.
- Alternative goal to optimality: *satisficing* (good enough).
- **Rational agent**: one that chooses actions that yield maximum expected utility averaged over all outcomes.
Game theory

• Originated in economic theory
  – Mathematical methods for study of decisions
  – Includes notion of *rational* behavior (acting in own interests)
  – Includes notion of *utility*
  – Generalizes the notion of game strategy
• *Variants* (Von Neumann-Morgenstern, 1947):
  – zero-sum, non-zero-sum
  – 2-player, multi-player
  – perfect-information, imperfect-information

**Example: Playing a game**

• To choose a move by *Generate-and-test*, generate possible moves, test them with an evaluation (utility) function (e.g., allocating points for pieces won by the move)
• Finding good moves often entails *lookahead* and *backtrack* in the game tree
The minimax algorithm

- Used for problems with an adversary; e.g., two-player games
- Assuming an optimal opponent, let the best move be the one that would yield the best-valued situation if the opponent replies with his/her best move
- To determine that, apply the same algorithm from opponent’s viewpoints

Example: Tic-tac-toe

- State space: the set of all possible board positions
- Let a position’s value be 1.0 if we win, 0.0 if we lose
- Also, if we can force a win (as at right), value is 1.0

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>O</td>
</tr>
</tbody>
</table>
Game trees

- In a game tree, vertices are board positions
- Edges are possible moves (transitions between board positions)

5. Randomized and evolutionary algorithms

- Do you ever just “try something” to solve a problem?
- How is natural evolution said to work?
- Is programming like this in any way?
Subtopic objective

6.5 Explain a randomized approach to solving intractable problems

Decision by coin flips

- Use of probabilistic control usually leads, with sufficient time, to good approximate solutions
- Often used for optimization problems
- Examples: natural evolution; evolutionary computation, which generates a population of solutions stochastically, tests them, and uses test results to generate more
Randomized solutions to median and Quicksort

- Uses pivot (*splitter*) value to partition array as in Quicksort; recursing
- Use of random splitter enablers expected time of $O(n)$
- May be generalized to order-statistic problem
- Choosing random pivot location in Quicksort improves probability that expected running time $O(n \lg n)$ occurs rather than worst case $O(n^2)$

Randomized solution to SAT

- Flip coins repeatedly to set variables’ truth-values, seeking a set of values that satisfy formula
- *Advantage:* If many interpretations satisfy formula, this algorithm will find a solution quickly
- *Disadvantage:* Is likely to run long or give false negative result if a small number of interpretations satisfy formula
Max-SAT

- **Given:** A set of $k$ disjoint clauses in propositional logic $C_1, \ldots, C_k$
- **Problem:** What is the set of variable assignments that satisfies the largest number of clauses?
- **Randomized solution:** Expected number of disjunctive clauses with 3 variables satisfied, out of $k$ clauses: $7/8 \cdot k$
- Also, some variable assignment exists that satisfies $7/8$ of all clauses

Cache maintenance

- **Cache** contains $k$ locations for short-term high-speed data storage
- **Cache hit:** successful request for cache data
- **Cache miss:** unsuccessful request
- **Eviction:** replacement of an item in the cache
- **Goal:** to minimize the number of misses, using an efficient eviction policy
Cache-maintenance strategies

- Optimal greedy algorithm would evict item needed farthest in future; but this hard to predict
- *Heuristic* is often used: least-recently used (LRU); achieves performance of average one miss per access
- An efficient randomized algorithm is in a class of *marking* algorithms, where items are marked when requested and random unmarked ones evicted periodically; $1 / \lg k$ misses per access

Randomized closest-pair

- Consider points in random order, tracking closest pair so far
- Use the observation that if $(u, v)$ are closer than current minimum distance $m$, then $v$ is within a $5m \times 5m$ square that has $u$’s $m \times m$ square at center
- *Data structure*: dictionary to store squares
- *Expected running time*: $O(n) + O(n)$ dictionary operations; this resolves to $O(n)$ with hashing
Randomized vertex cover

- **Intuition:** A vertex cover is a subgraph that contains all the vertices of its parent graph
- **Given:** $G = (V, E), S \subseteq V$
- **Problem:** Does $G$ have a vertex cover of size $\leq k$?

\[
S \leftarrow \emptyset
\]
While $S$ is not a vertex cover of $G = (V, E)$
\[
e \leftarrow \text{Random}((u,v) \in E) \text{ s.t. } u, v \notin S
\]
\[
S \leftarrow S \cup \{e\}
\]

Universal classes of hash function

- Good hash functions don’t generate clusters and collisions
- A mathematical definition of a good set $H$ of hash functions (called “universal set”):
  - If $h$ is chosen at random from $H$ then probability that $h(a) = h(b)$ for any $a, b$, is not greater than $1/n$ for a hash table of size $n$
  - All $h \in H$ are compactly represented
Simulated annealing

- Strategy to avoid defect of local minima in gradient descent, by generating perturbations in state, semi-randomly updating state when perturbation yields a lesser-valued (better) state
- Perturbation algorithm may behave badly when close to local optima; “flinches” requiring greater probability of getting new state
- Solution to flinching: start system “hot” (volatile, favoring local moves less), allow it to “cool” when closer to global minima

Problems susceptible to randomized solutions

- \(RP\) (random polynomial) languages are those for which \(\text{PTIME}\) algorithms exist that accept with probability greater than 0
- \(ZPP\) (zero-error, probabilistic polynomial): problems for which solutions exist whose expected time is polynomial, but whose worst-case time is not
Testing primality

- Used extensively in cryptography
- Problem is both NP and co-NP (problems/languages that are complements of NP problems)
- It is also in RP, with easily implemented probabilistic algorithm

Monte Carlo algorithms

- Many RP problems are in P; those in (RP-P) are rare
- We may reduce problem of a false rejection by running algorithm multiple times, so that for \( n \) runs the probability of false negative is \( 1/2^n \)
Monte Carlo algorithm to find average

- Find average of $m$ values, chosen randomly
- Accuracy: $\frac{1}{\sqrt{m}}$
- Hence to achieve accuracy $\varepsilon$, $m = \varepsilon^{-2}$
- If $|A| < \varepsilon^{-2}$, then Monte Carlo algorithm is slower than adding $A$ and dividing

Monte Carlo method to approximate $\pi$

- $\pi = 4 \frac{\text{hits}}{\text{tries}}$ in throwing a dart at a circle whose radius is 1
- Iterate the following
  ```java
double rand = randGen.nextDouble();
double x = 2 * rand - 1,
y = 2 * rand - 1;
```
- Check for hit:
  ```java
  if (x * x + y * y <= 1)
      hits++;
  ```
Las Vegas random algorithms

• Always produce correct output
• Expected time is a polynomial of the input size; running time varies with random data values used
• Associated with problem class ZPP (zero-error, probabilistic, polynomial)
• Theorem: $ZPP = (RP \cap co-RP)$
  (Complements of randomized PTIME problems; i.e., Monte Carlo problems)

Random and deterministic algorithms

• Efficient deterministic algorithms with precise correct results are a special case of:
  – Efficient deterministic algorithms that are correct with high probability
  – Random algorithms that are always correct and are efficient with high probability
**Probably $P$-solvable problems**

- BPP: the class of decision problems that are solvable in polynomial time with a probability of over 50%
- Error probability can be reduced to $1/2^n$ by running the algorithm $n$ times
- *Example:* Does a given program, in a certain class of problems, halt in fewer than $k$ steps on a given input of size less than $m$?

**Evolutionary computation**

- A probabilistic, population-based way to develop approximate solutions to difficult problems using notions of utility and fitness
- Modeled on natural evolution of species and behaviors of some life forms
- A growing research area encompassing genetic algorithms, evolutionary programming, evolution strategies, ant computing, swarm computing, particle swarm optimization
6. Intractability and approximation

The function-optimization problem

• Let $f : \mathbb{N}^k \rightarrow \mathbb{R}$ for some $k$ (the arity of $f$)

• Problem: Find some $x \in \mathbb{N}^k$ s.t. $f(x)$ is maximal

• Example: Suppose $x$ is the set of proportions of ingredients in a fuel mixture, $f(x)$ is fuel cost-efficiency under this mixture

• Optimizing $f(x)$ means finding the most efficient mixture

• For an algorithm to optimize a function, we must have $f : X \rightarrow Y$ with $X, Y$ finite

Fitness and function optimization

• Example: Suppose $f$ is viewed as a fitness function and $x$ is the set of attributes of individuals of a population

• Then finding $x$ s.t. $f(x)$ is maximal is finding the fittest possible individual of the species, i.e., those with the best attributes to assure survival

• Evolution by natural selection tends to optimize fitness, over many generations
Evolutionary computation

- State space is explored using and comparing a population of states rather than one at a time
- Evolutionary algorithm repeatedly modifies and selects from a population of solutions
- Selection is driven by fitness of each member of the population
- Randomization is used to explore the state space

The evolutionary algorithm

\[
\begin{align*}
t &\leftarrow 0 \\
\text{Initialize} \ (P_0) \\
V &\leftarrow \text{Evaluate} \ (P_0) \\
\text{While not } \text{Terminate} \ (V, t) \text{ do} \\
\quad t &\leftarrow t + 1 \\
\quad P_t &\leftarrow \text{Select} \ (P_{t-1}, V) \\
\quad P_t &\leftarrow \text{Alter} \ (P_t) \\
\quad V &\leftarrow \text{Evaluate} \ (P_t) \\
\text{Return } P_t
\end{align*}
\]

- \text{Evaluate} applies fitness function to individuals
- \text{Select} chooses some members of \( P \) to survive
- \text{Alter} changes \( P \) randomly, e.g., by mutation or crossover
- \text{Terminate} ends algorithm after a given goal or a time deadline is reached
Genetic algorithms

• At each step, the population is selected and regenerated using genetic operators, e.g., mutation and crossover
• Parameters: % of population to retain at each generation, which mate and produce offspring, and which probability measures to use, if any

Example: Checkers (Samuel, 1950s)

• Let fitness function $f : \mathbb{N}^k \rightarrow \mathbb{R}$ be an evaluator of checkers board positions from a black or red angle
• Let $x \in \mathbb{N}^k$ be a $k$-tuple of weights for each of $k$ different criteria for evaluating a checkers position
• Example: let $x_1$ be relative importance of number of kings, $x_2$ be relative importance of number of opponent checkers threatened, etc.
• Then writing a good checkers-playing program reduces to finding a good set of relative weights $x_1, ..., x_k$ for these criteria; Samuels’ played well
Summary

• Some problems are intractable: so hard that the fastest algorithm that solves them is $O(2^n)$ or more
• We may add to a class of such problems, showing how to add to it by reductions
• Search and optimization tend to be intractable
• To solve these problems we design approximate solutions: iterative improvement, backtracking, branch-and-bound, evolutionary computation
• These solutions are often probabilistic

References (intractability)

J. Hopcroft, Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley.
### References (approximation)


