Practical Solutions for Hard Problems

Chapter 30
Approaches

- Compromise on generality
  - Practical instances of the TSP
  - OBDDs for large Boolean formulas

- Compromise on optimality

- Maximizing an approximate objective function
  \[
  \text{cost}(s) = 4 \cdot \text{dollar-cost}(s) - 2 \cdot \text{number-of-lives-saved-by}(s) - 1.5 \cdot \text{commuting-hours-saved-per-week}(s).
  \]

- Compromise on both

- Compromise on total automation
Approaches

• The space is structured randomly. Exploit that randomness.

• The space isn’t structured randomly and we have some knowledge about the structure that exists. Exploit that knowledge.
Quicksort

quicksort(list: a list of \( n \) elements) =

1. If \( n \) is 0 or 1, return list. Otherwise:
2. Choose an element from list. Call it the pivot.
3. Reorder the elements in list so that every element that is less than pivot occurs ahead of it and every element that is greater than pivot occurs after it. If there are equal elements, they may be left in any order.
4. Recursively call quicksort with the fragment of list that includes the elements up to, but not including pivot.
5. Recursively call quicksort with the fragment of list that includes all the elements after pivot.
Quicksort

• Worst case: $O(n^2)$ time.
• Best case: $O(n \log n)$ time.

Choose pivot randomly to avoid:
• Malicious users.
• The list is already sorted.

If the pivot is chosen randomly:
• Expected case: $O(n \log n)$ time.
Randomized Algorithms

Randomized algorithms are used when:

• The problem can usually be solved without exhaustively considering all paths to a solution.

• A systematic way of choosing paths would be vulnerable to common kinds of bad luck or to a malicious attacker that would explicitly construct worst-case instances if it knew how to do so.
Randomized Turing Machines

A *randomized Turing machine* is a nondeterministic Turing machine $M$ where:

- At every choice point, there are exactly two moves from which to choose.
- At every choice point, $M$ (figuratively) flips a fair coin and uses the result of the coin toss to decide which of its two branches to pursue.
The Probability of a Branch

- $b$ is a single path in a randomized Turing machine $M$,
- $k$ is the number of choice points along $b$.

The probability that $M$ will take $b$ is:

$$\Pr(b) = 2^{-k}.$$ 

The probability that $M$ accepts is:

$$\sum_b \Pr(b)$$

where $b$ ranges over all accepting paths.

The probability that $M$ rejects is: $1 - \Pr(M \text{ accepts})$. 
Error Probabilities

- $M$ accepts $L$ with a *false positive* probability, $\varepsilon_p$, iff:
  $$(w \notin L) \rightarrow (\Pr(M \text{ accepts } w) \leq \varepsilon_p).$$

- $M$ accepts $L$ with a *false negative* probability, $\varepsilon_N$, iff:
  $$(w \in L) \rightarrow (\Pr(M \text{ rejects } w) \leq \varepsilon_N).$$
Randomized Algorithms

- **Monte Carlo algorithms** always run efficiently but may (with small probability) deliver an incorrect answer.

- **Las Vegas algorithms** never return an incorrect answer but may (with small probability) be very expensive to run.
BPP

$L \in \text{BPP}$ iff there exists some probabilistic Turing machine $M$ that runs in polynomial time and that decides $L$ with:

- a false positive probability, $\varepsilon_P$, and
- a false negative probability, $\varepsilon_N$, both less than $\frac{1}{2}$.

The name BPP stands for Bounded-error, Probabilistic, Polynomial time.

Let $\varepsilon = \max(\varepsilon_P, \varepsilon_N)$. If $\varepsilon < \frac{1}{2}$ then $L(M)$ is in BPP.
Reducing the Error Rate

**Theorem:** Let $M$ be a randomized, polynomial-time Turing machine with error rate $\varepsilon$ that is a constant equal to $\max(\varepsilon_P, \varepsilon_N)$. If $0 < \varepsilon < \frac{1}{2}$ and $f(n)$ is any polynomial function, then there exists an equivalent randomized, polynomial-time Turing machine $M'$ with error rate $2^{-f(n)}$.

**Proof:** $M'$ will run $M$ some polynomial number of times and return the answer that appeared more often. If the runs are independent, then the probability of error decreases exponentially as the number of runs of $M$ increases.
One-Sided Errors

$L \in \mathbf{RP}$ iff there exists a randomized Turing machine $M$ that runs in polynomial time and decides $L$ and:

- If $w \in L$ then $M$ accepts $w$ with probability $1 - \varepsilon_N$, where $\varepsilon_N < \frac{1}{2}$, and

- If $w \notin L$ then $M$ rejects $w$ with probability 1 (i.e., with false positive probability $\varepsilon_P = 0$).

The name RP stands for Randomized, Polynomial time.
One-Sided Errors

$L \in \text{co-RP}$ iff there exists a randomized Turing machine $M$ that runs in polynomial time and decides $L$ and:

- If $w \in L$ then $M$ accepts $w$ with probability 1 (i.e., with false negative probability $\varepsilon_N = 0$), and

- If $w \notin L$ then $M$ rejects $w$ with probability $1 - \varepsilon_P$, where $\varepsilon_P < \frac{1}{2}$. 
Using Las Vegas Algorithms

$L \in ZPP$ iff there exists a randomized Turing machine $M$ such that:

- If $w \in L$ then $M$ accepts $w$ with probability 1,
- If $w \notin L$ then $M$ rejects $w$ with probability 1, and
- There exists a polynomial function $f(n)$ such that, for all inputs $w$ of length $n$, the expected running time of $M$ on $w$ is less than $f(n)$. It is nevertheless possible that $M$ may run longer than $f(n)$ for some sequences of random events.

The name ZPP stands for Zero-error, Probabilistic, Polynomial time.
Other Ways to Define ZPP

• ZPP is the class of languages that can be recognized by some randomized Turing machine $M$ that runs in polynomial time and that outputs one of three possible values: Accept, Reject, and Don’t Know. $M$ must never accept when it should reject nor reject when it should accept. Its probability of saying Don’t Know must be less than $\frac{1}{2}$.

• $\text{ZPP} = \text{RP} \cap \text{co-RP}$. 
Proving that $\text{ZPP} = \text{RP} \cap \text{co-RP}$

$(L \in \text{ZPP}) \rightarrow (L \in \text{RP} \cap \text{co-RP})$: If $L$ is in ZPP, then there is a Las Vegas-style TM $M$ that accepts it.

Construct Monte Carlo TMs $M_1$ and $M_2$: on input $w$, $M_1$ will run $M$ on $w$ for its expected running time or until it halts. If $M$ halts naturally in that time, $M_1$ will accept or reject as $M$ would have done. Otherwise, it will reject.

The probability that $M$ will have halted is at least $\frac{1}{2}$, so the probability that $M_1$ will falsely reject a string that is in $L$ is less than $\frac{1}{2}$.

Since $M_1$ runs in polynomial time, $L$ is in RP.

Similarly, construct $M_2$ that shows that $L$ is in co-RP except that, if the simulation of $M$ does not halt, $M_2$ will accept.
Proving that $\text{ZPP} = \text{RP} \cap \text{co-RP}$

$\left( L \in \text{RP} \cap \text{co-RP} \right) \rightarrow \left( L \in \text{ZPP} \right)$: If $L$ is in RP, then there is a Monte Carlo TM $M_1$ that decides it and that never accepts when it shouldn’t.

If $L$ is in co-RP, then there is another Monte Carlo TM $M_2$ that decides it and that never rejects when it shouldn’t.

From these two, construct a Las Vegas TM $M$ that shows that $L$ is in ZPP. On input $w$, $M$ will first run $M_1$ on $w$. If it accepts, $M$ will halt and accept. Otherwise $M$ will run $M_2$ on $w$. If it rejects, $M$ will halt and reject. If neither of these things happens, it will try again.
Relating Language Classes

- $P \subseteq BPP$.
- $P \subseteq ZPP \subseteq RP \subseteq NP$.
- $P \subseteq ZPP \subseteq co-RP \subseteq co-NP$.
- $RP \cup co-RP \subseteq BPP$.

Unknowns:

- Is $BPP$ or $NP$ a subset of the other?
- Is $P$ a proper subset of $BPP$? Widely conjectured that $P = BPP$. 
Primality Testing

PRIMES = 
\{w : w is the binary encoding of a prime number\}.

COMPOSITES = 
\{w : w is the binary encoding of a composite number\}.

PRIMES is in P. But before that result was known, primality testing was done efficiently using randomized algorithms that show that:

• PRIMES is in co-RP.
• COMPOSITES is in RP.
Fermat’s Little Theorem

If \( p \) is prime, for any positive integer \( a \), if \( \gcd(a, p) = 1 \):

\[ a^{p-1} \equiv_p 1. \]

We’ll say that \( p \) passes the Fermat test at \( a \) iff \( a^{p-1} \equiv_p 1 \).

Example 1: let \( p = 5 \) and \( a = 3 \). Then \( 3^{(5-1)} = 81 \equiv_5 1 \).

Example 2: let \( p = 8 \) and \( a = 3 \). Then \( 3^{(8-1)} = 2187 \equiv_8 3 \).
Liars and Witnesses

\( p \) passes the Fermat test at \( a \) iff \( a^{p-1} \equiv_p 1 \).

\( a \) is a \textit{Fermat witness} that \( p \) is composite iff \( p \) fails the Fermat test at \( a \).

If \( p \) is prime, then it must pass the Fermat test at every appropriately chosen value of \( a \). If \( p \) passes the Fermat test at some value \( a \), do we know that \( p \) is prime? No.

\( a \) is a \textit{Fermat liar} that \( p \) is prime iff \( p \) is composite and yet it passes the Fermat test at \( a \).
**SimpleFermat**

\[ \text{simpleFermat}(p: \text{integer}, k: \text{integer}) = \]
  
  Do \( k \) times:
  
  Randomly select a value \( a \) in the range \([2: p-1]\).
  
  If it is not true that \( a^{p-1} \equiv_p 1 \), then return \text{composite}.
  
  All tests have passed. Return \text{probably prime}.

\[ \text{simpleFermat} \text{ runs in polynomial time.} \]
Modular Exponentiation

Two important facts:

\[ n^{i+j} = n^i \cdot n^j. \]

\[ (n \cdot m) \pmod{k} = (n \pmod{k} \cdot m \pmod{k}) \pmod{k}. \]

Combining these, we have:

\[ n^{i+j} \pmod{k} = (n^i \pmod{k} \cdot n^j \pmod{k}) \pmod{k}. \]
Modular Exponentiation

Suppose that we want to compute $65^{49} \pmod{589}$. $49$ can be expressed in binary as $110001$. So $49 = 1 + 16 + 32$. Thus $65^{49} = 65^{1+16+32}$.

$65^1 \pmod{589} = 65$.
$65^2 \pmod{589} = 4225 \pmod{589} = 102$.
$65^4 \pmod{589} = 1022 \pmod{589} = 10404 \pmod{589} = 391$.
$65^8 \pmod{589} = 3912 \pmod{589} = 152881 \pmod{589} = 330$.
$65^{16} \pmod{589} = 3302 \pmod{589} = 108900 \pmod{589} = 524$.
$65^{32} \pmod{589} = 5242 \pmod{589} = 274576 \pmod{589} = 102$.

$65^{49} \pmod{589} = 65^{(1+16+32)} \pmod{589}$.
$= (65^1.65^{16}.65^{32}) \pmod{589}$.
$= ((65^1 \pmod{589}) \cdot (65^{16} \pmod{589}) \cdot (65^{32} \pmod{589}))$ \pmod{589}$.
$= (65 \cdot 524 \cdot 102) \pmod{589}$.
$= ((34060 \pmod{589}) \cdot 102) \pmod{589}$.
$= (487 \cdot 102) \pmod{589}$.
$= 49674 \pmod{589}$.
$= 198$. 
Error Rate of \textit{SimpleFermat}

A \textit{Carmichael number} is a composite number that passes the Fermat test at all values. Every value of $a$ is a Fermat liar for every Carmichael number.

If $n$ is not a Carmichael number, then the chance that an arbitrary $a$ is a Fermat liar for $n$ is less than $\frac{1}{2}$.

So, unless $n$ is a Carmichael number, the error rate of \textit{simpleFermat} is less than $\frac{1}{2^k}$. 
Working with Carmichael Numbers

If $p$ is prime, then 1 has exactly two square roots (mod $p$):

$$1 \text{ and } -1.$$  

If $p$ is composite, 1 may have $\geq 3$ square roots (mod $p$).

For example, let $p = 8$. Then we have:

$$1^2 = 1.$$  
$$3^2 = 9 \equiv_8 1.$$  
$$5^2 = 25 \equiv_8 1. \text{ Note that } 5 \equiv_8 -3.$$  
$$7^2 = 49 \equiv_8 1. \text{ Note that } 7 \equiv_8 -1.$$  

So we can write the four square roots of 1 (mod 8) as:

$$1, -1, 3, -3.$$
Working with Carmichael Numbers

Every Carmichael number has more than two square roots.

For example, the smallest Carmichael number is 561.

The square roots of 1 (mod 561) are:

• 1,
• -1 (560),
• 67,
• -67 (494),
• 188 (-373),
• 254, and
• -254 (-307).
Checking for Carmichael Numbers

Simple check: Choose random values and see whether they are square roots of 1 (mod \( p \)). If any is, then \( p \) isn’t prime.

More efficient check: Suppose that \( a \) has passed the \textit{simpleFermat} test. Then \( a^{p-1} \equiv_p 1 \).

So \( a^{(p-1)/2} \) is a square root of 1 (mod \( p \)).

If \( a^{(p-1)/2} \equiv_p 1 \), we haven’t learned anything. But then we can again take the square root of both sides and continue until:

- We get a root that is -1. We stop.
- We get a noninteger. We stop
- We get a root that is neither 1 nor -1. \( p \) is composite.
Miller-Rabin

\[\text{Miller-Rabin}(p: \text{integer}, k: \text{integer}) =\]

1. If \( p = 2 \), return \textit{prime}. Else, if \( p \) is even, return \textit{composite}.
2. Rewrite \( p-1 \) as \( d \cdot 2^s \), where \( d \) is odd.
3. Do \( k \) times:
   
   Randomly select a value \( a \) in the range \([2: p-1]\).
   
   Compute the following sequence (mod \( p \)):
   \[a^{d \cdot 2^0}, a^{d \cdot 2^1}, \ldots, a^{d \cdot 2^s}\]
   
   If the last element of the sequence is not 1, then \( a \) fails the simple Fermat test. Return \textit{composite}.
   
   For \( i = s-1 \) down to 0 do:
   
   If \( a^{d \cdot 2^i} = -1 \), exit loop. Else, if it is not 1, return \textit{composite}.
4. All tests have passed. Return \textit{probably prime}. 
Heuristic Search

Consider problems that can be described generically as:

• A space of states that correspond to configurations of the problem situation.

• A start state.

• One or more goal states. If there are multiple goal states, then the set of them must be efficiently decidable.

• A set of operators (with associated costs) that describe how it is possible to move from one state to another.
Examples

• 15-puzzle
Examples

• 15-puzzle

• Instant Insanity
Examples

• 15-puzzle

• Instant Insanity

• Airline scheduling
Given an explicit search graph $G$, \textsc{Shortest-Path} = \{<G, u, v, k>: there exists a path from $u$ to $v$ whose length is at most $k}\} is in P.

The extension to weighted graphs is also in P.

But if we have a concise description of a state space, then the problem of searching it may be NP-hard.
Practical Search Techniques

Simple problems like the $n$-puzzle and Instant Insanity, as well as real problems, like airline scheduling, are only solvable in practice when:

- There exists a succinct problem description, and

- There exists a search technique that can find acceptable solutions without expanding the entire implicitly defined space.
Heuristic Search

A heuristic is a rule of thumb. The word “heuristic” comes from the Greek word εὑρίσκ-ειν, meaning, “to find” or, “to discover”.

A heuristic search algorithm is a search algorithm that exploits knowledge of its problem space.

Knowledge may be encoded in:

• operators, or

• a heuristic function.
Heuristic Functions

The job of a heuristic function $f(n)$ is to evaluate a node $n$ in a search tree so that the “best” node can be selected for expansion.

Two approaches to defining $f$:

- $f$ measures the **value** of the state contained in the node.

- $f$ measures the **cost** of the state (and possibly the path to it).
Heuristic Functions

\( f^*(n) = \text{cost of getting from the start state to a goal state via a path that goes through } n. \)

\[ f^*(n) = g^*(n) + h^*(n), \text{ where:} \]

- \( g^*(n) \) is the cost of getting from the start state to \( n \), and
- \( h^*(n) \) is the cost of getting the rest of the way.
Heuristic Functions

We’ll denote an estimate of a function by omitting the * symbol. So:

\[ f(n) = g(n) + h(n). \]

\( f(n) \) will guide the search process.

\( h(n) \) will evaluate a state and return an estimate of the cost of getting from it to a goal.
Two Assumptions

• There is some positive number $c$ such that all operator costs are at least $c$.

• Every state has a finite number of successor states.
Graph Search vs. Tree Search

(a) Tree Search

(b) Tree Search

(c) Graph Search
Best-First Search

![Tree Diagram]

- Node A
  - Edge to B with weight 1
  - Edge to C with weight 3
- B
- C

Weights:
- B: 1+3
- C: 3+2
Best-First Search

(a) 

(b)

A

B

C

1

3

(1+3)

(3+2)

D

E

2

3

2

3

(3+3)

(4+2)

A

B

C

1

3

(1+3)

(3+2)
Best-First Search

(b)

(c)
When to Stop
The A*-tree Algorithm

A*-tree(\(P\): state space search problem) =

1. Start with \(OPEN\) containing the node corresponding to \(P\)'s start state. Set \(g\) to 0 and \(f\) to \(0 + h = h\).

2. Until an answer is found or there are no nodes left in \(OPEN\) do:
   1. If there are no nodes left in \(OPEN\), return Failure.
   2. Choose and remove from \(OPEN\) a node, \(BESTNODE\), with a lowest \(f\) value.
   3. If \(BESTNODE\) is a goal node, halt and return the path from the initial node to \(BESTNODE\).
   4. Generate the successors of \(BESTNODE\). For each of them do:
      Compute \(f\), \(g\), and \(h\), and add the node to \(OPEN\).
Properties of $A^*-tree$

- $A^*-tree$ is complete.
- $A^*-tree$ is admissible iff $h(n)$ is.
Admissibility of $h(n)$

$h(n)$ is **admissible** iff it never overestimates the true cost $h^*(n)$ of getting to a goal from $n$.

- If $h$ always returns 0, $A^*$-tree becomes breadth-first search.
- If $h(n)$ always returns $h^*(n)$, $A^*$-tree will walk directly down an optimal path and return it.
- If $h(n)$ overestimates $h^*(n)$, then it effectively “hides” a path that might turn out to be the cheapest.
Overestimating $h^*(n)$
A* - Searching a Graph
A*

- A* exploits two sets of nodes: OPEN, and CLOSED.

- A* must be able to trace backward from a goal. So it must explicitly record, at each node, the best way of getting to that node from the start node. Whenever a new path to node $n$ is found, its backward pointer may change.

- If, in A*, a new and cheaper path to node $n$ is found after node $n$ has been expanded, $n$'s $g$ value changes. But the cheaper path to $n$ may also mean a cheaper path to $n$'s successors. So it may be necessary to revisit them and update their backward pointers and their $g$ values.
A* (P: state space search problem) =

1. Start with OPEN containing the node corresponding to P's start state. Set CLOSED to ∅.
2. Until an answer is found or no nodes left in OPEN do:
   1. If there are no nodes left in OPEN, return Failure.
   2. Choose BESTNODE and place it in CLOSED.
   3. If BESTNODE is a goal node, halt and return the path from the initial node to BESTNODE.
   4. Generate the successors of BESTNODE. But do not add them to the search graph yet. For each SUCCESSOR do:
      1. Set SUCCESSOR to point back to BESTNODE.
      2. Compute g(SUCCESSOR).
      3. See if SUCCESSOR is in OPEN.
      4. See if SUCCESSOR is in CLOSED.
      5. If neither, compute f, and h, put it in OPEN, and add it to the list of BESTNODE's successors.
Properties of A*

• A* is complete.

• A* is admissible iff $h(n)$ is.

• A* is optimal iff $h(n)$ is monotonic.

$h(n)$ is monotonic iff, whenever $n_2$ is a direct successor of $n_1$,

$$h(n_1) \leq c(n_1, n_2) + h(n_2).$$
Is A* Good Enough?

Depending on the shape of the state space and the accuracy of $h$, it may still be necessary to examine a number of nodes that grows exponentially in the length of the cheapest path.

However, if the maximum error that $h$ may make is small, the number of nodes that must be examined grows only polynomially in the length of the cheapest path. More specifically, polynomial growth is assured if:

$$|h^*(n) - h(n)| \in \mathcal{O}(\log h(n)).$$
Other Heuristic Search Algorithms

The *minimax* algorithm searches game trees.