David M. Keil
Framingham State University

6. Intractability and approximation

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

Reading: Ch. 10-11

Objectives

6a. Relate the main complexity classes for tractable and intractable problems

6b. Explain how problems may be shown to be intractable using reductions

6c. Explain approximation or randomized approaches to solving intractable problems

Inquiry

- How hard is *optimization*?
- How helpful is it to mathematically characterize some problems as *hard* or *intractable*?
- Are approximate *satisficing* solutions a practical way to address hard computational problems?

Overview

- Some *problems* are so hard that the fastest algorithm that solves them is $O(2^n)$ or more
- As n approaches 100, running time is above astronomical; these problems are called *intractable*
- We define precisely a class of such problems and show how to add to it by *reduction*
- 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*

1. Undecidability and intractability

- Some functions are *uncomputable*
- Problems that are uncomputable *predicates* are called *undecidable*
- Decidable problems that take too much time to be worth solving are called *intractable*
- Recognizing undecidable and intractable problems can save unbounded effort

Notation

- ϕ_P is the function computed by program P
- $\#(P) \in \mathrm{N}$ corresponds to P's source code
- $f(x)\downarrow$ means that function f is defined for x
- $f(x)\uparrow$ means that f is undefined for x

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?
- *The characterization of these problems is surprising; they are all undecidable*

Undecidability of the HALT problem

- Let $HALT(P, x) = \begin{cases} \text{true} & \text{if program } P \text{ halts on input } x \ (\phi_P(x)\downarrow) \\ \text{false} & \text{otherwise} \end{cases}$
- If *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 input $\#(S)$
  - S hangs ($\phi_S(\#(S))\uparrow$) if S halts on input $\#(S)$
- Since S hangs iff S halts, S, hence H, cannot exist

Complexity and intractability

- Recall from Data Structures that *complexity of a problem* is the time function of the fastest algorithm that solves the problem
- *Example*: No search algorithm is better than O(n) for arbitrary collections, so the complexity of the search problem is $\Omega(n)$
- Some problems are *decidable* but take too long to solve *in practice*
- In this subtopic we characterize those *intractable* problems mathematically

Lower time bounds for problems

- For problems solved by repeated comparisons, algorithm may descend a *decision tree*
- *Example:* 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 $\Omega(n \lg n)$

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

Example intractable problems

- Satisfiability, 3SAT
- Hamiltonian cycle, TSP
- Set partition
- Independent sets
- Channel assignment

Satisfiability (SAT)

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

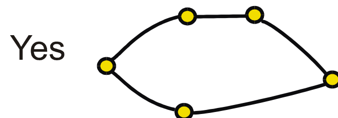
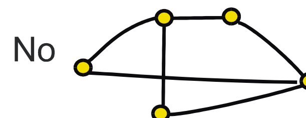
- Let $SAT = \{ \phi \mid (\exists\rho)\ \phi_\rho = \text{true} \}$ where ϕ is a formula in logic and ρ is a set of truth assignments
- *Example:* $\rho = \{ (p, \text{true}), (q, \text{true}), (r, \text{false}) \}$
- Then *SAT* is the set of formulas that are *satisfiable*
- Thus *SAT* is a *language*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

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:* $(p \vee q) \wedge (q \vee r \vee s)$ is in CNF
- 3-CNF: formulas in which every clause contains exactly three literals
- *3SAT*: Satisfiability problem for 3-CNF formulas

Hamiltonian-cycle problem

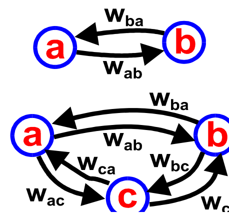
- *Problem:* Given a graph $G = \langle V, E \rangle$, is there a path in V^* that starts and ends with the same vertex and that passes through every vertex exactly once?

Yes  No 

- *One solution* is to generate and test every possible sequence of $|V| + 1$ vertices to see if it is a path
  - What is the complexity of the test operation?
  - What is the complexity of generate-and-test?
  - Can you think of a better 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?



All paths: $\{(a,b,a)\}$
All paths: $\{(a,b,c,a), (a,c,b,a)\}$

- This problem is considered *intractable,* but a claimed solution can easily be checked
- At least as hard: What is the *lowest-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*
- No one knows a faster solution

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* ($\Theta(2^{|V|})$): test all subsets of V

[pic]

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?

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

- *TIME*($f(n)$): the set of problems/ languages with solution (algorithms) of time $O(f(n))$ 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

- $P = \bigcup_{k \in N} TIME(n^k)$
- That is, 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 $DTIME(n)$
  - Sorting an array, $TIME(n \lg n)$
  - Searching a balanced BST, $TIME(\lg n)$
  - Generating a graph reachability matrix, $TIME(n^3)$
- The problems not in *P* are called *NP-Hard*

Exponential time

- $EXP = EXPTIME = TIME(2^n)$
- *SAT* seems to be in *EXPTIME* but not *P*, because it seems that to solve SAT, $O(2^n)$ candidate solutions need to be generated and tested
- Problems that are in *EXP* but not in *P* are considered intractable

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
- Almost all running times have small constant terms or factors
- Polynomials are closed under sum, composition
- Hardware architectures are of equivalent speed, within a polynomial function

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* is accepted by a *non*deterministic 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 *NTIME*

- *NTIME* (*T*(*n*)) is the set of problems for which a correct solution can be verified in time *T*(*n*)
- *Example: SAT* $\in$ *NTIME*($f(n) = n$) because time to check variable assignments that make ϕ true is linear in $|\phi|$
- The nondeterminism corresponds to the assumption that a correct solution is generated nondeterministically ("magically") and only needs to be verified

Deterministic time complexity

- $DTIME(T(n)) =$
  $\{ L \subseteq \Sigma^* / L$ is accepted by a *deterministic* algorithm in time $T(n)$ $\}$
- $DTIME\ (T(n))$ is a *complexity class* of problems for any function $T(n)$
- We think of all algorithmic problems as deterministic because we use deterministic algorithms and programs to solve real problems; see later for nondeterminism

The $\mathcal{NP}$ complexity class

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

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

Reducibility and NP-completeness

- 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.

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

Two example reductions

1. Theorem: SAT ≤ 3SAT

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

Example: $p \lor q \lor r \lor s =$

$(p \lor q \lor x) \land (r \lor \neg x \lor y) \land (s \lor \neg y \lor \neg x)$

2. Theorem: 3SAT ≤ Independent-set

For formula ϕ, of n clauses, create a graph G s.t. ϕ is satisfiable iff G has an independent set of size m

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 $P =_? NP$

- *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 $L \in \boldsymbol{P}$ 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 $Q \in \boldsymbol{P}$
- *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

Some NPC problems

- 3SAT, because SAT $\leq$ 3SAT and SAT is NPC
- Independent-set, because 3SAT $\leq$ IS and 3SAT is NPC
- Hamiltonian Path
- Traveling Salesperson, because HP $\leq$ TSP and HP is NPC
- Channel assignment
- Graph coloring

Intractability, NPC, EXPTIME

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

Challenges

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

```
Mystery(n)
while (n > 1)
  if n is even
      n ← n / 2
  else
      n ← 3n + 1
Return true
```

2. Is deciding whether to shoot, in a game of basketball, a tractable problem?

3. Approximation algorithms

Why compute an approximation rather than an exact result?

- Exact solution algorithm may take too much time (*Example*: shortest tour through *n* cities)
- Exact results may have infinitely large representation (*Example*: square root)
- Approximate solution may be used as part of a larger exact algorithm

Approximation algorithms

- Many use *heuristics*: rules of thumb based on human experience; e.g., putting largest item in box first
- *Accuracy ratio*: $f(s_a) / f(s^*)$, where heuristic s_a solves the problem of minimizing f and s^* is an exact solution
- *Performance ratio:* lowest upper bound on accuracy ratio
- *c-approximation algorithm:* one whose performance ratio is at most c, i.e., $f(s_a) \leq f(s^*)$ for any instance
- *Continuous functions:* Newton's method
  - $x_{n+1} = (x_n + f(x_n) \div f'(x_n)) \div 2$

  gives approximate sequence; e.g., for square root

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

Linear programming and Simplex

- Linear-programming problem: maximize or minimize $c_1x_1 + \ldots + c_nx_n$ s.t. for $i = 1 .. m$,
  $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ 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; cohose 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 $v = \Sigma\, x_{ij}$ 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

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

Maximum matching in bipartite graphs

- Simpler problem: $G = (V, E)$ where V is partitioned into disjoint sets V_1, V_2, s.t. $E \subseteq V_1 \times V_2$ (every element of V_1 is paired with one in V_2 and vice versa)
- Suppose M is a matching in G; then if every vertex in V_1 or every one in V_2 is matched, M cannot be improved
- But M can be improved if an *augmenting path* is found that yields a larger matching
- *Complexity:* $O(|V|^2 + |V||E|)$

Stable-marriage problem

- A *bipartite matching problem* (selecting pairs, one item from each of two sets)
- Each element of each partition has a totally ordered preference list
- *Stable pairing:* one in which no *unmatched* pair exists whose the members would prefer each other to their current matches
- *Algorithm:*
  While some element x of V_1 is unmatched
  x proposes to next-preferred $y \in V_2$
  y accepts if unmatched or if matched with a less-preferred partner
- *Complexity:* $O(n^2)$

4. State-space search

- *State space:* A 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* may be defined from state to state, which we can express as a *tree*
- *Goal:* Reduce size of state-space tree explored by algorithm, by pruning branches that cannot lead to a solution
- *Example:* n-queens chess problem

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 (¬*problem-solved* ∧ ¬*time-expired*)
   - generate a possible solution
   - test it

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

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 using randomization, transformation set T, and fitness function *eval*
- Addresses n-queens, TSP, SAT
- Similar algorithm, simulated annealing, solves independent-set problem

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

```
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 ] ← 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 only one; or
  - 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

- Levels of a computer system: device, circuit, symbol, … *knowledge* (Newell, 1982)
- *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

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

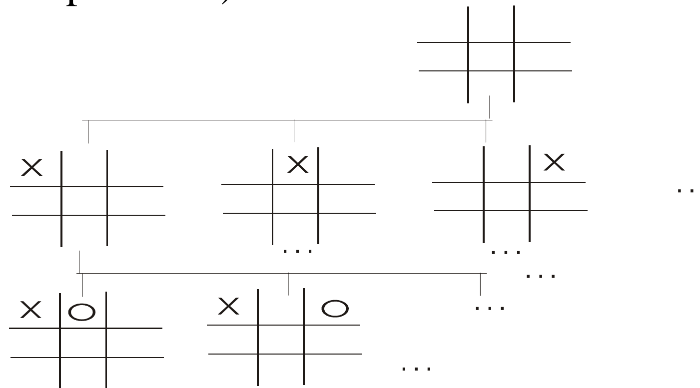
Example: Tic-tac-toe

- 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

X		X
	O	
X		O

Game trees

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



POMDPs

- *Partially observable Markov decision processes* are problems in inaccessible stochastic environments with the Markov property
- *Markov property:* Probability that system will be in a given state at next step depends only on current state, not on past history (L. Lipsky)

5. Randomized and evolutionary algorithms

- 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

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

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

The function-optimization problem

- Let $f : \mathbf{N}^k \rightarrow \mathbf{R}$ for some k (the *arity* of f)
- *Problem*: Find some $x \in \mathbf{N}^k$ s.t. $f(x)$ is maximal

graph

- *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

```
t ← 0
Initialize (P_0)
V ← Evaluate (P_0)
While not Terminate (V, t) do
    t ← t + 1
    P_t ← Select (P_{t−1}, V)
    P_t ← Alter (P_t)
    V ← Evaluate (P_t)
Return P_t
```


- *Evaluate* applies fitness function to individuals
- *Select* chooses some members of P to survive
- *Alter* changes P randomly, e.g., by mutation or crossover
- *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, which probability measures, if any, to use

Example: Checkers (Samuel, 1950s)

- Let fitness function $f : \mathbf{N}^k \rightarrow \mathbf{R}$ be an evaluator of checkers board positions from a black or red angle
- Let $x \in \mathbf{N}^k$ 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 $x_{1, \ldots,} x_k$ for these criteria
- *Result:* machine play at very high level

Complexity concepts

backtracking	intractable problem
branch-and-bound	nondeterministic algorithm
combinatorial problem	*NP*
complexity class of problems	*NP*-completeness
complexity of a problem	*NP*-hard problem
Cook-Karp Thesis	*NTIME* ($T(n)$)
decision tree	polynomial time (P)
DTIME	propositional logic formula
EXPTIME	reducibility
generate-and-test	satisfiability (SAT)
halting problem	set partition
Hamiltonian-cycle problem	*SPACE*($f(n)$)
independent set	*TIME*($f(n)$)
information-theoretic lower bound	undecidable problem

Approximation concepts

- approximation algorithm
- backtracking
- bounded rationality
- branch-and-bound
- evolutionary computation
- fitness
- function optimization
- game theory
- generate-and-test
- global maxima
- local maxima
- iterative improvement
- Markov property
- maximal flow
- maximal matching
- Minimax
- POMDP
- probabilistic algorithm
- rationality
- simplex method
- stable marriage
- state-space search

References (complexity)

M. Garey and D. Johnson. *Computers and Intractability.* H. Freeman, 1978.

H. Hamburger and D. Richards. *Logic and Language Models for Computer Science*. Prentice Hall, 2002.

J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

R. Johnsonbaugh and M. Schaefer. *Algorithms.* Prentice Hall, 2004.

A. Levitin. *The Design and Analysis of Algorithms*, 2nd ed. Addison Wesley, 2007, Chapters 11-12.

H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

M. Sipser. *Introduction to the Theory of Computation*. PWS, 1997.

References (approximation)

A. Levitin. *The Design and Analysis of Algorithms*. Addison Wesley, 2003, Chapter 11.

Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd Ed. Springer, 1996.

R. Johnsonbaugh and M. Schaefer. *Algorithms.* Pearson Prentice Hall, 2004.

A. Newell, H. Simon. Computer science as empirical enquiry: Symbols and search (1976). *In* George F. Luger, Ed., *Computation and Intelligence: Collected Readings*, 1995.

A. Newell. The knowledge level (1982). *In* Luger, 1995.

References (approximation)

Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality.* MIT Press, 1991.

S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Addison Wesley, 2003.

J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton, 1953.