

David M. Keil
Framingham State University

1. Classes of problems

1. Vector, matrix, and graph problems
2. Constraint and optimization problems
3. Logical specification of problems
4. Interactive problems

Reading: handouts

David Keil Analysis of Algorithms1. Problem classes1/121

Motivation

- Many problems in this set have different solutions using different design approaches
- In this course we will analyze the efficiencies of different algorithms to solve these problems
- Extremely difficult problems may require use of approximate or probabilistic solutions
- Some problems may be specified using the languages of predicate and temporal logic

David Keil Analysis of Algorithms1. Problem classes1/122

Classes of problems

- Basic functional problems
 - String, vector, and collection manipulation
 - Matrix multiplication
 - Geometric problems: convex hull, closest pair
 - Graph problems
 - State-space search and problem reducibility
- Constraint and optimization problems; e.g., satisfiability of logical formulas
- Interactive problems

Topic objectives

- 1a. Specify a computational problem using preconditions and postconditions
- 1b. Distinguish constraint from optimization problems
- 1c. Formally specify properties of an interactive system

Inquiry

How useful is the study of computational *problems*?

1. Vector, matrix, and graph problems

String manipulation

- *Count*: $n_0(x)$, $n_1(x)$ are the number of 0s and 1s, respectively, in string x
- *Match*: Determine whether two strings are identical
- *Search*: Find the first occurrence of string s_2 in string s_1
- *Pattern match*: May allow wild cards ('?', '*')
- Regular-expression match

Array manipulation

- *Search*: Find occurrence or lack of occurrence of value x in array A
- *Sort*: Rearrange array elements in ascending or descending order
- *Insertion*: Insert a new element into a sorted array such that the result is sorted
- *Order statistic*: find k^{th} largest element of array

Problems with bit vectors

For bit vector (Boolean array) A :

- Logical OR: $\bigvee_{k=1..|A|} A[k] = (\exists i \leq |A|) A[i] = 1$
- (Bitwise) OR(A, B) = $((A[1] \vee B[1]), (A[2] \vee B[2]), \dots, (A[|A|] \vee B[|A|]))$
(OR bits in same position to form a new bit vector C , composed of ORs of pairs in A, B)
- Logical AND and bitwise AND are similar
- Bitwise complement of A : sequence of bits that are negations of bits in A

Longest common subsequence

- *Vector, array, and sequence* may be considered to be equivalent
- Given sequences x_1, x_2 , what is the longest subsequence y s.t. y is a subsequence of both x_1 and x_2 ?
- This is a measure of similarity
- Elements of subsequences are not necessarily contiguous, e.g., “dab” is a subsequence of “database”

Partition problem

- Used in Quicksort
- Choose any element, x , of array
- All other elements less than x should go to its left
- All elements greater than x should go to its right
- *End result*: x ends up in what its position would be if the array were sorted; but *Partition* does not sort the array

Aligning DNA sequences

- *Definition*: “a pairwise match between the characters of each sequence”
- *Significance*: An alignment corresponds to a hypothesis about the evolutionary history connecting the sequences
- *Objective*: To find the best alignments between two sequences
- Techniques for alignment comparison of sequences are “a cornerstone of bioinformatics”

Priority queue problem

- A priority queue is a collection with restricted access
- Each item in priority queue has a key value denoting its relative *priority*
- Operations: *Insert*, *Extract*
- *Extract* retrieves and deletes the item of highest priority
- *Minimum queue* treats low-valued key as high-priority; maximum queue prioritizes high-valued key

Matrix multiplication

- Given two 2-dimensional matrices, their *product* is a matrix whose cells are each the sum of the products of several cells chosen from both matrices
- Given square matrices A and B , matrix C is the product, where
$$C[i, j] = A[i, 0] B[0, j] + \dots$$
$$A[i, k] B[k, j] + \dots$$
$$A[i, n-1] B[n-1, j] \text{ for } i, j \leq n - 1$$

Graph path search

- Finding a path in a graph from a given vertex to another vertex
- *Inputs*: $G = (V, E)$; origin vertex; destination vertex
- *Output*: a path (sequence of vertices) starting at origin, ending with destination; or a value denoting failure

Other graph problems

- *Traversal*, visiting all vertices
- *Cycles*: Is graph cyclic?
- *Connectivity*: Is graph connected?
- *Spanning tree*: a tree of edges that includes all vertices
- *Hamiltonian path*: In an unweighted graph, find a path that passes through all vertices
- Optimization versions (Minimal spanning tree; Traveling Salesperson; see subtopic 2)

Topological sorting of dag

- *dag*: directed acyclic graph, i.e., graph in which no cycles exist. Trees are dags but in a dag, contrary to a tree, two edges may have the same destination
- *Problem*: to arrange the vertices of a directed acyclic graph in a sequence such that for every pair (v_i, v_{i+1}) , there is no path in the dag from v_{i+1} to v_i
- *Example*: a listing of courses in a prerequisite relation so that no course is taken before its prerequisites

2. Constraint and optimization problems

- *Constraint problem*: To find some value that satisfies a set of constraints or conditions
- *Constraint problem examples*:
 - Search
 - Pattern matching
 - Sort of a set of records
- *Optimization problem*: to find maximum or minimum valued solution to a constraint problem, among all solutions

Constraint satisfaction problems

Problem:

- A set x of variables x_1, x_2, \dots, x_n
- A set C of constraints C_1, C_2, \dots, C_m , that each specifies acceptable combinations of values for a certain subset of x
- A variable assignment to x that does not violate any of C is *consistent* or legal
- A variable assignment to all of x is *complete*
- *Solution*: a complete consistent variable assignment

Problems in propositional logic

- *Evaluate*: given a particular set of variable assignments and a formula, evaluate formula
- *Satisfiability (SAT)*: Given a formula ϕ , does a set of variable assignments exist that satisfies ϕ (makes ϕ true)?
- *Validity*: does ϕ hold under *all* variable assignments?
- *Examples*: $(p \wedge \neg p)$ is not satisfiable;
 $(p \vee \neg p)$ is satisfiable and valid;
 $p \wedge q$ evaluates to *true* if p, q are *true*

General form of CSPs and solutions

- *Advantage* of formulating problems as CSPs: standard state representations enable generic transition functions, goal tests, heuristics
- *Form of state*: set of partial variable assignments
- *Initial state*: no assignments
- *Successor function*: assigns value to one variable while violating no constraint
- *Goal test*: variable assignments are complete
- Order-independence of variable assignments makes *backtracking* helpful

Optimization problems

Examples:

- *Closest pair*
Given a set of ordered pairs, points on a Cartesian plane, find two points that are closest together
- *Shortest path*: For vertices u , v , in *weighted* graph G , find *shortest* path from u to v
- *Bin packing*: Find a packing that minimizes the number of bins

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
- *Example*: Suppose x is the set of proportions of ingredients in a fuel mixture, $f(x)$ is fuel 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

Graph coloring

- An optimization problem
- What is the minimum number of colors needed to color all the vertices of a graph (equivalent to a map), such that no two adjacent vertices have same color?
- Country on a map is equivalent to vertex in graph; borders between countries are equivalent to edges in graph
- The compiler-design problem of *register allocation* is equivalent to the graph coloring problem

Paths through all vertices

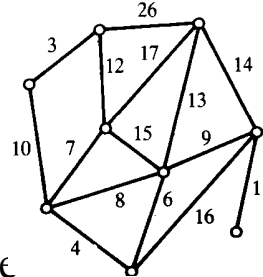
- *Hamiltonian path*: Find a path in a graph from a vertex to itself that passes through all vertices exactly once
- *Traveling Salesperson*: In a weighted graph, find minimum-cost path through all vertices
- TSP is the optimization version of HP

Convex hull

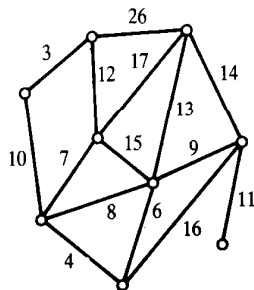
- The *Convex hull* of a set of points is a subset of ordered pairs (on a Cartesian plane) that define a polygon with no concave series of edges (“dents”)
- I.e., find smallest convex polygon that contains all of a set of points
- Equivalent problem: find the posts of the shortest fence that would enclose all the points
- [pic]

Minimum spanning tree

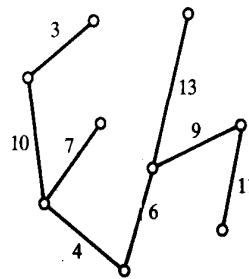
- Suppose you are designing a subway system
- You want to join n locations by rail at minimum expense
- The locations and the possible connections you will plan are a graph
- The optimal graph must be connected (Why?)
- The graph should be acyclic (Why?)



Example subway solution



City map



Optimal train network

Bin packing

- Given n items of sizes s_1, \dots, s_n , and k bins each of capacity < 1.0 , where $s_1, \dots, s_n < 1$, can all items be packed in the k bins?
- *Example:* If two bins are of capacity 0.5 and 0.3, can items of sizes 0.2, 0.2, 0.2, 0.2 fit?

Knapsack problem

- Find maximum-valued subset of weighted items totaling less than a given weight w
- Given n objects and a knapsack of capacity C , where object i has weight w_i and value v_i , find $x[1..i]$ to maximize sum of all values $v_i w_i$ where sum is less than C
- Two versions of problem:
 - Continuous, where $0 \leq x_i \leq 1$
 - 0/1 where $x_i \in \{0, 1\}$

The maximal-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

Maximum matching

- *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]

State-space search

- *State-space search*: problem involves search in an n -dimensional solution space for a structure that satisfies some specification
- *State* may be
 - game board configuration (*n-queens*)
 - way of partitioning a set (*set-partition*)
 - a set of variable assignments (*satisfiability*)
- Many optimization problems, are characterized by *combinatorial explosion*

n -queens and search trees

- *Problem*: place n queens on a chessboard s.t. no queen threatens another
- Many search problems can be expressed as searches of a *solution tree*; e.g., game playing, 8-queens.
- The tree represents possible transitions from one configuration of the game board to another
- Many games entail state-space search for “good” or winning positions

Set partition

- *Set partition:* Given a set $S \subseteq \mathbb{N}$, can S be partitioned into two subsets, A and B , s.t. the sums of the elements of A and B are the same?
- *Example:* If $S = \{1, 3, 4\}$ then *yes*;
if $S = \{1, 2, 4\}$, then *no*

Reducibility of problems

- If problem A is easily solved using a solution to problem B , then A is said to be *reducible* to B (intuition: A is as easy as B)
- *Example:* Solving a maze is reducible to the graph path-search (reachability) problem

3. Logical specification of algorithmic problems

- One part of specification of a problem is an *assertion* that holds as a *postcondition*
- Part of specification is a set of *preconditions* that must hold for any input
- *Examples*
 - Precondition for binary search, or postcondition for any sort: $(\forall i < |A|)(A[i] \leq A[i+1])$
 - Postcondition of any search:
(Return value = *true*) iff $(\exists i \leq |A|) A[i] = \textit{key}$

Predicate logic

- Predicate calculus (first-order logic, FOL) extends propositional calculus by enabling *quantifiers* and formulas in *predicate* (functional) form
- A *predicate* is a Boolean function, i.e., it returns a truth value; it is a *property*
- *Examples:*
 - $(\forall x) x < x + 1$ universal quantifier
 - prime*(5) predicate denoting a property
 - $(\exists x) x > 1$ existential quantifier
- *Arity* of a predicate is its number of parameters

Applications of predicate logic

- Theorems are expressed and proven in FOL
- Algorithm verification uses predicate logic; e.g., $Sorted(A) \Leftrightarrow (\forall i < |A|) A[i] \leq A[i + 1]$
- Theory of formal languages and automata uses FOL
- Artificial intelligence uses FOL to express knowledge and to implement automated reasoning

Variables in predicate logic

- In unquantified expressions $sum(x, 5)$, $odd(x)$, $prime(x)$, x is an *unbound placeholder* that stands for an unknown *constant*
- “ $sum(x, 5) = 8$ ”, “ $odd(x) = true$ ” are meaningless without having a value for x
- “ $x = 3 \rightarrow sum(x, 5) = 8$ ”, “ $x = 1 \rightarrow odd(x) = true$ ” are meaningful and true statements
- *Quantifiers* (\forall , \exists) bind variables: $(\exists x) P(x)$ means that every value x has property P
- *Boolean variables* are names for assertions

Quantifiers

- For sentence S ,
 - $(\exists x) S$ is true iff some assignment under I has an assignment to x s.t. S is true
 - $(\forall x) S$ is true iff S is true for all assignments of values to x under I
- *Correspondence between quantifiers:*
 - $(\exists x) P(x) \Leftrightarrow (\neg \forall x) \neg P(x)$
 - $(\forall x) P(x) \Leftrightarrow (\neg \exists x) \neg P(x)$

Notation

Logical operators (propositional logic)

- \neg *not* (negation)
- \wedge *and* (conjunction)
- \vee *or* (disjunction)
- \Rightarrow entails (implication)
- \Leftrightarrow iff (equivalence)

Quantifiers (predicate logic)

- \forall for all (universal quantifier)
- \exists there exists (existential quantifier)

Pre- and post- conditions

- **Precondition:** An assertion about inputs before an algorithm executes
- **Postcondition:** An assertion that is claimed to hold afterward if the precondition holds
- **Example:** Adding a series of numbers
 - Precondition: *total* is 0
 - Postcondition: *total* stores the sum of all input values

The terms of a contract

- A *precondition* of a method tells what must be true about parameters if the method is to work
- A *postcondition* asserts that the method has done its job correctly

- *Example:*

```
int sum_of_linear_series(int n)
// Precondition: n > 0
{
    int sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;
    // Postcondition: sum = 1 + 2 + ... + n
    return sum;
}
```

Hoare triples

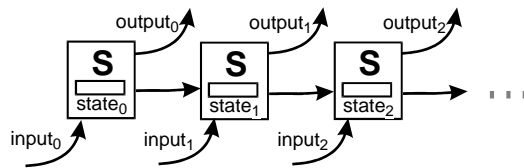
- A way to specify a problem as an (input, output) relation
- Let P be a program
- Let ϕ , ψ be assertions about the state of a program at a certain time, i.e., about the values of variables
- Specification $\langle \phi \rangle P \langle \psi \rangle$ asserts that if P runs starting in a state that satisfies ϕ , then the state after termination of P will satisfy ψ

4. Interactive problems

- A *service* (solving a sequential-interactive problem) is a series of responses to inputs
- A *protocol* defines rules for providing a service
- *Examples:*
 - Search engine
 - DBMS responses to database queries
 - Web server response to clicks
- *Note:* Response may reflect state of server, so that a service is not a series of algorithm executions generating output from input

Interactive computation

- Feature of computing today: Computation as an ongoing *service*, not assumed to terminate
- Must solve problems whose inputs cannot be completely specified a priori
- Dynamic input and output *during* computation
- *Persistence of state* between interaction steps
- *Environment* is an active partner in computation



Dynamic environments

- The function-optimization problem defined earlier is *static*, assumes f will be the same later as now
- Interaction is often in *dynamic* real-world environments, such as climate or ecosystem
- This increases the difficulty of verifying designs of reactive systems
- A separate, but related, category of environment is *stochastic* (imperfectly predictable)

Multi-stream-interactive problems

- A *mission* is a problem requiring services to multiple input streams, possibly under time constraints (quality of service)
- The sources of multiple input streams may interact among themselves
- *Streams* (connections) may be subject to creation and destruction
- Part of a mission may involve *scalability* in real time

Specification and verification of interactive systems

- *Certain properties* of reactive systems may be verified or invalidated by use of temporal logic and a technique called *model checking*
- Model checking uses *Kripke structures* and *computation trees*
- Predicate logic describes static situations, so must be augmented with temporal elements
- Specifications and proofs can be in Computation Tree Logic (CTL)

Formal specification of problems

- *Algorithmic problem (function)*: a set of (*input*, *output*) string pairs
- *Sequential-interaction problem (service)*: a set of dynamically generated *streams* of I/O pairs
- *Multi-stream interaction problem (mission)*: a set of possibly asynchronous I/O streams, possibly in real time and with dynamic creation/destruction of connections
- Part of spec for multi-stream interaction problem may include number of streams, constraints on computing power of agents, and time constraints

Temporal logic

- Temporal logic reasons about dynamic processes
- Used in verification of *reactive systems* in which input and output alternate
- In interactive systems, computation may be pictured as infinite
- *Model checking*, developed in U.S. and France in 1980s, uses notations Computation Tree Logic (CTL), CTL*, LTL, others
- Current primary use of model checking may be to verify hardware designs

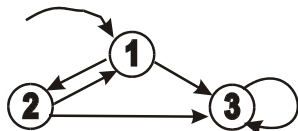
Kinds of properties verified

- *Safety* (undesirable states won't be visited)
- *Example*: Deadlock will never occur; microwave oven will not enter (power-on, door-open) state
- *Liveness* (desired states will obtain)
- *Example*: All resource requests will eventually be fulfilled
- Properties apply to *computational paths* through the tree of all possible paths

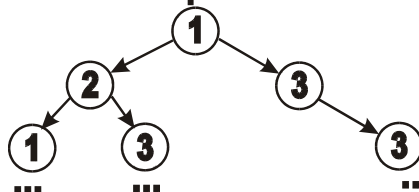
Computation Tree Logic (CTL)

- Supports assertions about Kripke structures (unlabeled transition systems)
- To a Kripke structure corresponds an infinite computation tree reflecting all possible paths through the system

Kripke structure

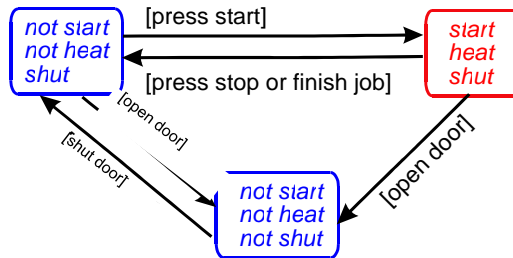


Its computation tree



CTL formulas

- *State* formulas express assertions about a state (see labels on states of statechart below)
- *Path* formulas express assertions about a set of paths in a computation tree



Key

start = start-button pressed *heat* = heating on *shut* = door shut

CTL quantifiers and operators

Where f and g are state formulas in predicate logic...

Quantifiers:

A on all paths

E there exists some path

Operators

Xf f is true in the next state

Ff f will eventually be true

Gf f is true globally (at each state on path)

$f U g$ f is true until g is true

CTL example formulas

$A F \phi$	on all paths ϕ will eventually hold
$E F \phi$	on some path ϕ will eventually hold
$A G \phi$	on all paths ϕ will always hold
$A X \phi$	on all paths ϕ will hold in the next state
$A \psi U \phi$	on all paths ψ will hold until ϕ does

Concepts: problems

array insertion	matrix product	satisfiability
array search	minimum spanning	scalability
bin packing	tree	sequential interaction
closest pair	mission	service
convex hull	multi-stream	set partition
dynamic	interaction	shortest path
environment	n -queens	sort
function	optimization	static environment
optimization	order statistic	string match
graph coloring	principle of	string search
Hamiltonian cycle	optimality	topological sort
independent set	quality of service	traveling salesperson
knapsack problem	reachability	traversal
matrix	reducibility	

Concepts: logic and formal specification

assertion	partial and total correctness
convergence	partition
correctness	postcondition
Hoare triple	precondition
insertion sort	predicate logic
liveness	quantifier
loop invariant	safety
model checking	temporal logic

References

- Clarke, Grumberg, & Peled. *Model Checking*. MIT Press, 1999.
- M. Huth & M. Ryan. *Logic in Computer Science: Modeling and Reasoning About Systems*. Cambridge Univ., 2000.
- R. Johnsonbaugh and M. Schaefer. *Algorithms*. Pearson Prentice Hall, 2004.
- A. Levitin. *The Design and Analysis of Algorithms*. Addison Wesley, 2007.
- Z. Manna & A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.
- Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Addison Wesley, 2003.
- P. Wegner. Why interaction is more powerful than algorithms. *Comm. ACM*, 5/97.