5. Greedy and other efficient optimization algorithms

David M. Keil
Framingham State University

5. Greedy and other fast optimization algorithms

1. Optimal-substructure property
2. Greedy graph algorithms
3. Compression and packing
4. Space/time tradeoffs and dynamic programming
5. Transform and conquer

Reading: Ch. 7-8

Constraint and optimization problems

- *Constraint problem:* To find some value that satisfies a set of constraints or conditions
- *Optimization problem:* to find maximum or minimum valued solution to a constraint problem, among all solutions
- Minimizing cost function *f*:
  $\min(f) = m$ s.t. $f(m) = \min_{x \in N}\{y \mid y = f(x)\}$
- Greedy algorithms iteratively build a minimal-cost structure by choosing a step that minimizes cost function

Coin changing

- For *some* sets of *n* denominations, this algorithm makes change with a minimum number of coins

Make-change (*amt*, *Denom*)

i ← |*Denom*|
while *i* > 0 and *amt* > 0
  write ⌊*amt* ÷ *Denom*[*i*] ⌋
  amt ← *amt* mod *Denom*[*i*]
  i ← *i* – 1

Topic objectives

5a. Explain the greedy approach to algorithm design

5b. Explain the optimal-substructure property

5c. Describe a greedy algorithm for graphs

5d. Explain an instance of the dynamic-programming approach

5e. Explain an instance of the transform-and-conquer approach

Inquiry

- Is *time scalability* a good way to measure the performance of computing systems?
- How much does studying *design approaches* help us create efficient solutions?
- How hard is *optimization*?

Greedy algorithms

- Solve optimization problems by seeking *local optima* as the next step of a state-space search
- Greedy algorithms expand a partial solution until problem is solved
- Choice at each step is feasible, irrevocable
- Not all greedy algorithms produce optimal results, or even results meeting minimal constraints
- *Cases:* Making change; two algorithms to find minimal spanning tree; Dijkstra's for shortest path; Huffman coding

1. Optimal-substructure property

- A problem π has the optimal-substructure property iff any structure *S,* that is an optimal solution to π, is composed of optimal solutions to subproblems of π
- This property holds for *some* problems, e.g., change making with U.S. denominations, graph coloring, shortest path
- It is the property required for greedy algorithms

Local optima in search

- *State-space search:* Finds *global optimum* in a very large set of possible solutions (*states*)
- Search goes from state to state
- *Example:* Finding highest location in a county
  - Greedy algorithm follows uphill path until at the top of a hill
  - Low hills are local optima; highest hill is global optimum
  - Greedy algorithm alone does not find global optimum

Example of optimal-substructure property

- Imagine a hill that has no dips anywhere from the base to the summit
- Thus, to get to the summit from any point on the side of the hill, the right way to go is always to choose an upward direction
- Thus any subpart of a path to a higher elevation is also a path to a relatively higher elevation
- The problem of climbing that hill has the optimal-substructure property

Generalized greedy algorithm

```
y ← null
Repeat
   augment y by appending "best"
        available value from some set
until y satisfies the problem's constraints
return y
```

y is a structure, e.g., a path in a graph

Limits of the greedy approach

- Some problems lack optimal-substructure property
- *Examples:* In baseball, chess, or checkers, a short-term sacrifice (e.g., bunt) may bring long-term net benefit
- Generalized state-space search is of an *n*-dimensional landscape full of local optima that are not global optima
- *Example:* Climbing to a higher ground or walking toward destination do not always optimally help us reach objective

2. Greedy graph algorithms

- *Graph coloring:* color the vertices so that no adjacent vertices have the same color
- *Minimal spanning tree:* Smallest-weight tree containing all vertices in a weighted graph
- *Single-source shortest path:* Dijkstra's algorithm builds a tree of shortest paths, starting from the source vertex. The problem of finding the shortest path to one destination is reducible to the problem of finding shortest paths to all

Graph coloring problem

- 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

Greedy-coloring (*G*)

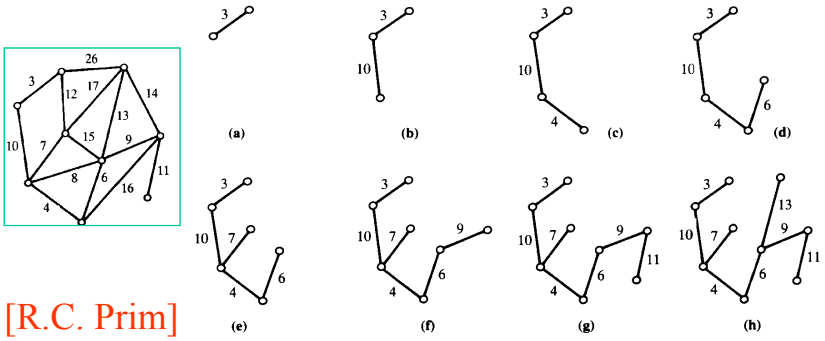
- Assumes an ordered sequence of colors
- Gives (possibly *non-optimal*) solution to graph-coloring problem
- Q: What is the fastest algorithm that solves the coloring problem *optimally*?

```
for i ← 1 to |G.V|
    v [ i ].color ←
        min { c | c is not the color of a
                  vertex in G adjacent to v [ i ] }
```

Minimal spanning tree

- MST is a subgraph of a graph
- *Example:* road or rail system connecting *n* locations with lowest construction cost
- Not *every path* in MST is minimal; only the *whole tree* is minimal
- [pic]

Prim's algorithm for minimal spanning tree



[R.C. Prim]

Strategy: repeatedly add to the growing tree the lowest-weight available edge adjacent to a vertex already in the tree w/o forming cycle

Prim's algorithm

Prim **(*G*) *G* = (*V*, *E*)**
$V_T \leftarrow \{ v_0 \}$ // any vertex
$E_T \leftarrow \varnothing$
while $|V_T| < |V|$ do
    $e \leftarrow$ min-wt-edge (E) s.t.
        $e = (v, u) \wedge (v \in V_T) \wedge (u \in V - V_T)$
    $V_T \leftarrow V_T \cup \{ v \}$
    $E_T \leftarrow E_T \cup \{ e \}$
Return (V_T, E_T)

Kruskal's algorithm

- Builds minimal spanning tree, bottom up, as an expanding sequence of subgraphs
- These are not necessarily connected until the algorithm terminates
- Greed applies to a weight-sorted list of all edges
- Each step connects two trees (possibly trivial) by the next edge in weight order
- Makes use of optimal-substructure property

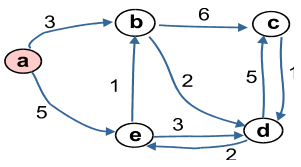
Kruskal (G) $G = (V, E)$

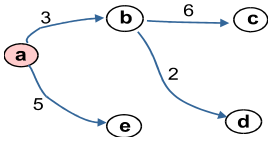
$E_T \leftarrow \varnothing$
Sort edges E into array e in weight order
$k \leftarrow 1$, $count \leftarrow 0$
while $count < \min\{|E|, |V| - 1\}$ do
    If $(V_T, E_T \cup \{ e_k \})$ is acyclic
        $E_T \leftarrow E_T \cup \{ e_k \}$
        $k \leftarrow k + 1$
    $count \leftarrow count + 1$
Return (V, E_T)

- Sorting time dominates run time, hence run time is $\Theta(|E| \lg |E|)$
- Alternative code: Johnsonbaugh, pp. 277, 280

Single-source shortest path

- In any weighted graph, from any source vertex, a tree exists composed of the set of shortest paths from the source to each other vertex
- The problem of a single shortest path reduces to building this tree (which is *not* the MST)
- *Dijkstra's algorithm* builds this tree of shortest paths, starting from the source vertex





The Dijkstra algorithm

- Uses breadth-first search, beginning by examining the edge from the source to each adjacent vertex, then from each of these to its neighbors
- Builds solution tree along path of least immediate cost (greedy)
- If current path from source to a vertex *v* is found to be costlier than path to *u* plus path from *u* to *v*, adds *v* to a minimum-path-so-far array, relaxing the estimate on path length

A shortest path's subpaths are minimal

- Optimal-substructure property applies to shortest-path problem as follows
- If path (*u, ..., u′, ..., v′,..., v*) is minimal, then (*u′,v′*) is minimal too (why?)
- So to minimize path *u..v*, find shorter segments *u′..v′*

- *Note:* more than one shortest path may connect two given vertices

5. Greedy and other efficient optimization algorithms

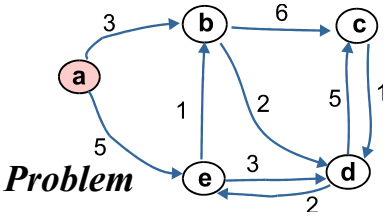
Dijkstra (graph, weights, source)

```
For i ← 1 to |V | do
    estimate[ i ] ← ∞
PQ-insert(Q, i )   // weight attribute is PQ key
estimate [source] ← 0
While ¬empty(Q)
    u ← Extract-min (Q )  //Greedy choice
    For each vertex v adjacent to u
        If estimate[v] > estimate[u] + weight (u,v)
            estimate[v] ← estimate[u] + weight (u,v)
```

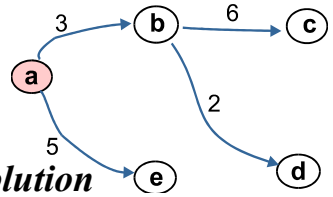
- Vector *estimate*..|*V* |] stores set of shortest known path lengths from *source* to each vertex
- Key to priority queue *Q* is *estimate*[*u*]

Application of Dijkstra algorithm

Step	*Q*	*u*	*v*	Distance from *a* *a*	*b*	*c*	*d*	*e*
1.	*abcde*	*a*	*b*	0	3	∞	∞	∞
			e	0	3	∞	∞	5
2.	*bcde*	*b*	*c*	0	3	9	∞	5
			d	0	3	9	5	5
3.	*cde*	*c*	*d*	0	3	9	5	5
4.	*de*	*d*	*e*	0	3	9	5	5
			c	0	3	9	5	5
5.	*e*	*e*	*d*	0	3	9	5	5



Problem



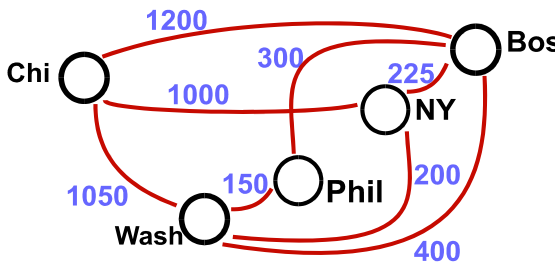
Solution

Dijkstra details

- Output is *estimate* array
- How are paths extracted from this array?

Find shortest paths

	Wash	Bos	NY	Phila	Chi
Wash		400	200	150	1050
Bos			225	300	1200
NY					1000
Phila					
Chi					



Correctness and analysis of Dijkstra

- *Theorem:* The Dijkstra algorithm finds a shortest path from *source* to u for all $u \in V$
- How would this be proved?
- Output is *estimate* array
- How are paths extracted from this array?
- *Running time, using heap, worst case:*
  $O((n+m) \lg n)$
  $O(n^2 \lg n)$ if $m = n^2$
  where n is # vertices, m is # edges

3. Compression and packing

- *Compression problem:* Map alphabet to a set of variable-length bit encodings s.t. most frequently used characters are represented by shortest encodings
- *Packing problem* (knapsack): Find a maximum-valued set of items, each with weights and values, that fit into a container with a given maximum weight

Prefix-free variable-length codes

- *Problem:* Find a space-optimal variable-length bit encoding for an alphabet, given a distribution of occurrences of characters in strings to be encoded
- *Example:* Morse Code solves a simpler problem in which *character delimiters* are allowed
- *Constraint:* Codes must be *prefix-free*, no codeword is a prefix of another codeword
- Binary tree can represent the code, where left branches are labeled 0 and right ones labeled 1
- A *Huffman code* is a prefix-free code represented by a binary tree

Huffman's compression algorithm

Create one-node trees, one for each character
Label nodes by character and its probability
Repeat until all trees are one:
  Find two trees with minimum weight so far
  Pair them as subtrees of a new node
  Weight (parent) ← sum of weights of subtrees

- This greedy algorithm achieves an encoding with optimum compression ratio by generating an optimal binary tree

[Pic Levitin, p. 326]

Greedy solution to continuous-knapsack problem

- *Problem:* Find maximum-valued subset of a set of weighted items, totaling less than weight *w*
- *Parameters: A*[1..*n*], where each element of *A* has value and weight attributes; *max-wt* ∈ **R**
- *Solution:*
  1. Sort *A* by ratio of value to weight
  2. While total weight so far is less than *max-wt*
     select next element of sorted *A*
     increment total weight so far
  3. Select some part of next element of *A*

4. Space/time tradeoffs and dynamic programming

- Time efficiency can sometimes be gained by making use of storage space
- *Tables* or *larger tree nodes* may be used to obtain improved running times
- *Cases:*
  - Sorting by counting
  - String matching
  - Hashing
  - B trees

Sorting by counting

- Suppose problem is to sort an array composed only of values in 1..*m*
- Then a solution is to count the occurrences of each value in 1..*m* and store in a table *T*
- Then write to the array *T*[1] 1's, *T*[2] 2's, etc.
- Running time O(*n*) is better than any comparison-based sort, provided that $m \leq O(n)$
- 2 5 1 2 8 7 5 1 5 $\Rightarrow$ 1 1 2 2 5 5 7 8

String matching

- *Problem:* Find first occurrence of string of length *m* in string of longer length
- *Brute-force solution*: Perform $(n - m + 1)$ string comparisons, each of length *m*
- Faster *Boyer-Moore algorithm* (simplified):
  - Construct a 26-element shift table for the search key, saying how far from the *right* of the key each letter is
  - Do string comparison from the right
  - Use the shift table to skip most string comparisons
- *Average case:* $\Theta(n)$ but "obviously faster"

Hashing

- Dictionary is array in which index is computed from key value
- Desirable attributes of hash function: speed, even distribution of keys
- *Two implementations:* Open addressing with linear probe; array of buckets (linked lists)
- *Load factor:* ratio of number of entries to table size
- *Time/space tradeoff:* High load factor costs time, low load factor wastes space

B trees

- Each node has m children
- All data is stored in leaves
- All leaves are at same tree level
- Used to store very large indexes for databases stored on disk
- *Advantage:* extremely short paths to leaves ($\lg_m n$)
- *Disadvantage:* Wasted space

Dynamic programming

- Some problems (e.g., Fibonacci) have overlapping subproblems
- *Dynamic programming* suggests solving each subproblem only once and storing solution in a table for later reference
- *Cases*:
  - Fibonacci
  - Binomial coefficient
  - Warshall's and Floyd's algorithms (graphs)
  - Optimal BSTs
  - Knapsack problem

Fibonacci

- Recall $Fib(x) =$

$$\begin{cases} 1 & \text{if } x \leq 1 \\ Fib(x-1) + Fib(x-2) & \text{otherwise} \end{cases}$$

- Running time is $\Theta(2^x)$
- Dynamic-programming algorithm is $\Theta(x)$:

```
DP-Fib(x)
F [0] ← 1, F [1] ← 1
For i ← 2 to x do
        F [ i ] ← F [ i – 1] + F [ i – 2]
Return F [ x ]
```

Longest common subsequence

- 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?
- Elements of subsequences are not necessarily contiguous, e.g., "dab" is a subsequence of "database"
- Dynamic programming solution: see Goodrich-Tamassia, pp. 568-572

Binomial coefficient

- $C(n, k)$ is the number of combinations (subsets) of k elements chosen from a set of n elements
- $C(n, k) =$

$$\begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ C(n-1, k-1) + & \\ \quad C(n-1, k) & \text{otherwise} \end{cases}$$

Binomial (*n*, *k*)

```
for i ← 0 to n do
  for   j ← 0 to min{i, k} do
    if j = 0 or j = k
       C [ i, j ] ← 1
    else
       C [ i, j ] ← C [ i – 1, j – 1] +C [ i – 1, j ]
Return C [ n, k ]
```

Time complexity: _____

Space complexity: _____

Warshall's algorithm

- Computes transitive closure (reachability matrix) of a digraph from its adjacency matrix
- Faster alternative to DFS or BFS for each pair
- *Principle*: If vertex *j* is reachable from *i,* and *k* is reachable from *j*, then *k* is reachable from *i*

Warshall (*M* [*n*, *n*])

```
for i ← 1 to n do
   for j ← 1 to n do
      for k ← 1 to n do
         if M [i, j ] ∧ M [ j, k ]
            M [i, k ] ← true;
Return M
```

Source vertex

Intermediate vertex

Destination vertex

Running time: Θ(___)

Floyd's algorithm

- Finds shortest paths between any pair of vertices in a weighted graph
- Computes a distance, cost, or weight matrix
- Principle: reduce cost estimate d_{ik} if shorter path found (greedy)

```
Floyd (G [n, n])
D ← G.W     // weights matrix
for i ← 1 to n do
   for j ← 1 to n do
      for k ← 1 to n do
         D [i, k] ← min {D [ i, k ], D [ i, j ] + D [ j, k ] }
Return D
```


Optimal BSTs

- *Problem:* Given probabilities that certain values will be search keys, find BST with minimum average search time
- *Solution:* Construct optimal subtree as one node with optimal left and right subtrees
- Dynamic-programming approach uses a table of average number of comparisons for a range of nodes
- Space complexity: $\Theta(n^2)$
- Time complexity: $\Theta(n^3)$

Knapsack with table

- *Problem:* Given a set of n items with weights w_1 .. w_n and values v_1 .. v_n, find greatest-valued set of items that fit in knapsack of capacity W
- *Solution:* Let V_{ij} be the optimal value of the first i items in a knapsack of capacity j
- $V[i, j]$ =

$$\begin{cases} \max \{ V[i-1, j], \\ \qquad v_i + V[i-1, j-w_i] \} & \text{if } j > w_i \\ V[i-1, j] & \text{otherwise} \end{cases}$$

- Time and space complexity: $\Theta(nW)$

Sequence matching

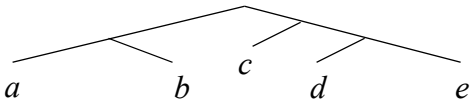
- A bioinformatics problem, in which phylogenetic (family) relationships among protein sequences in DNA are found by comparing
- It is a more sophisticated type of string comparison

DNA and computation

- Atoms and molecules have discrete forms
- *Example*: DNA strands are built from only four different molecules; alphabet is {C, A, G, P}
- In replicating, dividing, and recombining, DNA can be said to *compute* on discrete symbolic values as a digital computer computes, or as a mind manipulates symbols logically

Phylogenetic trees

- *Definition*: "typically a graphical representation of the evolutionary relationship among three or more genes or organisms" (p. 80)
- Terminal nodes are from empirical data, internal nodes are inferred common ancestors
- Newick format: ((*a, b*), (*c*, (*d, e*))) =



- May reflect substitutions in sequences: ABCD (ZBCD (ABYD, ZBCQ), ABXD)

Alignment between 2 sequences

- *Definition:* "a pairwise match between the characters of each sequence" (Krane and Raymer, p. 35)
- *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"

Alignment techniques

- Want to align a given two elements of language: Σ^* where Σ = {C, G, A, P}
- *Objective*: To insert gaps in either of two DNA sequences to maximize pairwise matches
- *Example*: align **AATCTATA** with **AAGATA**
- Possible solution: **AATCTATA** / **AA--GATA**
- A scoring method accounts for matches, mismatches, and gaps

Needleman-Wunsch algorithm

- *Overview*: Break down alignment problem into smaller problems by finding best alignment of subsequences; storing them in a table rather than computing repeatedly
- *Example:* Align CACGA, CGA (p. 42)
- There are 3 ways to start, beginning at the left:
  (1) C (…A…C…G…A)
  C (…G…A)
  (2) − (…C…A…C…G…A)
  C (…G…A)
  (2) C (…A…C…G…A)
  − (…C…G…A)

Needleman-Wunsch

- Global sequence alignment algorithm
- Assume match score is 1, mismatch is 0, gap is (−1)
- To evaluate alignments above,
  score(1) is +1 (C matches C) plus alignment score of ACGA and CGA
  score(2) = –1 + *score*(CACGA, CGA)
  score(3) = –1 + *score*(ACGA, CCGA)
- Fill out table: […]

Needleman-Wunsch table

Match bonus: 1 Gap penalty: -1

		g	t	c	a	t	a	g	a	c	g
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
t	-1	0	0	-1	-2	-3	-4	-5	-6	-7	-8
c	-2	-1	0	1	0	-1	-2	-3	-4	-5	-6
a	-3	-2	-1	0	2	1	0	-1	-2	-3	-4
t	-4	-3	-1	-1	1	3	2	1	0	-1	-2
a	-5	-4	-2	-1	0	2	4	3	2	1	0

5. Transform and conquer

Transformations:

- Instance simplification
- Representation change
- Problem reduction

Principle: Performance advantages can be gained by changing the form of the input

Problems: Uniqueness, mode, matrix inverses, determinants, BST balancing, polynomial evaluation, least common multiple

Reductions of problems

- Transform-and-conquer approach uses reducibility of some problems to others
- *Example: Least common multiple* problem is reducible to *greatest-common-divisor*:
  lcm(m, n) = mn / gcd(m, n)
- Finding extrema of *some* functions is reducible to finding derivative
- Problems like wolf-goat-cabbage (Levitin, p. 17, Problem 1) are reducible to state-space (graph) problems

Algorithms using presorted arrays

- *Uniqueness verification* is linear-time after array is transformed by presorting
- Compare brute-force O(n^2) algorithm with algorithm using sorted array:

```
Uniqueness ( A [0 .. n – 1] )
Sort (A)
For i ← 0 to n – 2 do
   if A[ i ] = A[ i + 1 ]
        return false
Return true
```

Finding mode

- *Mode:* most common element in array
- *Worst-case*: no duplications – brute force makes $\Theta(n^2)$ comparisons to compile list of frequencies of elements [*explain*]
- Better algorithm using sorted array: Find longest run of equal values – $\Theta(n)$
- Complexity: $\Theta(n \lg n)$ including sort

Mode (A [0 .. n – 1])

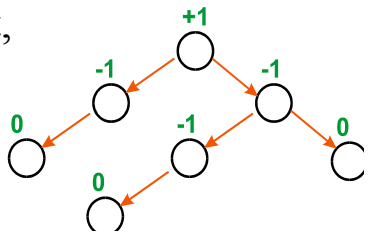
```
Sort (A)
i ← 0, mode_frequency ← 0
while i ≤ n – 1 do
   run_length ← 1
   run_value ← A[ i ]
   while i + run_length ≤ n – 1 and
                A[run] = run_value do
      run_length ← run_length + 1
      if run_length > mode_frequency
            mode_frequency ← run_length
            mode_value ← run_value
   i ← i + run_length
Return mode_value
```

Gaussian elimination

- Can find inverses and determinants of matrices by GE
- Assume linear equations
  $a_{11}x + a_{12}y = b_1$
  $a_{21}x + a_{22}y = b_2$
- Can solve by transforming equations into a system with an upper-triangular matrix with zeroes below the diagonal, solvable by backward substitution

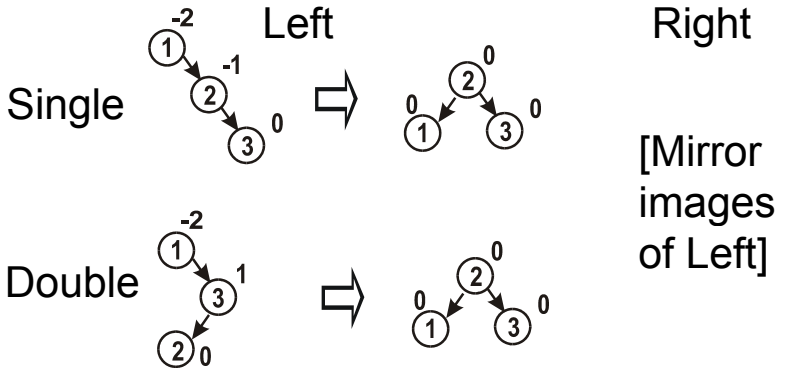
BST balancing

- A case of instance simplification
- *Note:* Transformation from a set to a BST is itself a case of representation change
- *Problem:* preserve O(lg *n*) properties of a balanced BST as it is built and updated
- *AVL tree:* BST with left, right subtrees differing in height by not more than 1



AVL trees

- Unbalanced BST subtree is transformed by *rotation* around root
- 4 kinds of rotation:



Horner's Rule

- Algorithm to evaluate a polynomial:

  ***Horner* (*P* [0.. *n*], *x*)**
  > *P*[0..*n*] are coefficients of degree-*n* polynomial
  $p \leftarrow P[n]$
  for $i \leftarrow n - 1$ downto 0 do
  $p \leftarrow xp + P[i]$
  return *p*
- Complexity: $\Theta(n)$
- Complexity of brute-force version: $\Theta(n^2)$
- H's Rule can be used to do binary exponentiation in $\Theta(\lg n)$ time

5. Greedy and other efficient optimization algorithms

Concepts (greedy)

acyclic graph
Dijkstra's algorithm
global optimum
greedy algorithm
Huffman coding
Kruskal's algorithm
local optima
minimal spanning tree
optimization problem
optimal-substructure property
prefix-free encoding
Prim's algorithm
single-source shortest path
state-space search

Concepts (dynamic, transform)

adjacency matrix
AVL tree
binomial coefficient
Boyer-Moore algorithm
BST balancing
B-trees
Build-Heap
dynamic programming
dynamic-programming Knapsack algorithm
Extract-min
Fibonacci
Floyd's algorithm
Gaussian elimination
hash function
hashing
Heapify
Heap-Sort
Horner's Rule
least common multiple
linear probe
load factor
minimum heap
mode
open addressing
optimal BST
reachability matrix
sequence matching
time/space tradeoff
transform and conquer
uniqueness verification
Warshall's algorithm

5. Greedy and other efficient optimization algorithms

References

T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

A. Levitin. *The Design and Analysis of Algorithms,* 2nd ed. Addison Wesley, 2007. Chapters 7-8, 10.

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