Dynamic Programming

Philosophy and how to use it

Solving subproblems like DnC but use previous answers so we are not solving the same subproblems multiple times.

Longest increasing subsequence - O(x^2) time O(x) space

For j = 1,2..n:

L(j) = 1 + max(L(i) : (i,j) in E)

Return maxjL(j)

\*Turn into DAG of subsequences, find the longest path along the DAG.

\*\*Array full of ones to start, start i at value checking, j at first element. If value at j < value at i, increment by 1.

Binomial coefficient – O(nk) time/space

1 – if k = 0 or k = n

C(n – 1, k – 1) + C (n -1, k) – if 0<k<n

0 -everything else.

Edit distance – O(mn) time/space

Gene sequencing

E[I,j] = min(diff(I,j) + E[i-1,j-1], 1 + E[I,j-1], 1 + E[I-1,j])

\*diff = 0 if letters match, 1 if they are different.

Knapsack w/Repetition – O(nW) time, O(W) space

(weights, values, W) (W == max weight limit)

K[0] = 0

For w = 1…. W do:

K[w] = max(for i:weights(i)<=w) (K[w-weights(i)] + values(i))

Return K[W]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| K0 | K1 | K2 | K3 | K4 | K5 | K6 | K7 | K8 | K9 | K10 |
| 0 | 0 | 9 | 14 | 18 | 23 | 30 | 32 | 39 | 44 | 48 |

Item 1 weight = 6 Value = 30

Item 2 weight = 3 Value = 14

Item 3 weight = 4 Value = 16

Item 4 weight = 2 Value = 9

0/1 Knapsack – O(nW) time/space

K[0,j] = 0

K[w,0] = 0

For j = 1,2…n do

For w = 1,2…W do

If (weights(j) > w then

K[w,j] = K[w,j-1]

Else

K[w,j] = max(K[w,j-1], K[w-weights(j), j-1]+values(j))

Return K[W,n] (bottom right most square of graph.

Recursion and Memoization – Memoization is using a hash table to store values that can be checked for in a recursive program so skip duplicate solving.

Linear Programming

Setting up an LP from an objective and constraints

Basic idea of Simplex algorithm (you should be able to solve a simple LP)

Graph constraints, check each vertex in counterclockwise order until next one decreases/increases, return. – O(m^n)

Primal and dual representations

Ex)

max x + y -> 2x+y <= 3, x+ 3y <= 5, x,y >= 0

\*Flip coefficients of max equation with opposite sides of inequalities.

max( 1, 1)

(2 1) (x) <= (3)

(1 3) (y) (5)

Min(z1,z2) (3,5) -> min(3z1 + 5z2)

(z1,z2) (2 1) >= (1) -> 2z1+z2 >= 1

(1 3) >= (1) -> z1+3z2 >= 1, z1,z2 >= 0

2 variables -> solve with substitution, answers should match.

Zero-sum game

Row player tries to maximize values, column tries to minimalize penalty.

Matching pennies

h t

h (1 -1)

t (-1 1)

rowplayer: max(min(x1-x2, -x1+x2))

z = min (x1-x2, -x1+x2)

max z Simplify:

x1-x2 >= z -2x1 + 1 + z <= 0

-x1+x2 >= z 2x1-1+z <= 0

x1+x2 = 1 x1 >=0 --> graph and solve

x1,x2 >=0

Intelligent Search

Backtracking – Same idea as BnB but no pruning

S = {P0}

While S is not empty do

P = S.eject

T = expand(P)

For each Pi in T do

If (test(Pi)) then

Return yes

If !test(Pi) then

T.remove(Pi)

S = S union T

Return No

Branch and Bound – You know this.

Beam search

Semi-greedy state space search keeps a limited number of states (beam width) in memory. Worst cost/heuristic values are dropped.

Bounding functions – should be optimistic. Quality bounding increases performance.

TSP with B&B – Project 5

Reduced cost matrix

Reduce Rows, reduce columns (reduce = 0 is each row/column).

Add min values to get cost bound.

State-space search with partial path: - Calculating lowerbound = O(n^2) Other work = O(b^n) – overall O(n^2 b^n) b = branching factor. N, number of nodes. Without pruning – O(n^2 n!)

Start with initial parent matrix reduced. Expand children states. Reduce child matrix. Cost = parent state cost + cost of new edge + cost to reduce. Set row/column of selected edge to inf, set edge and inverse to inf. Reduce. Put all states in queue and pop lowest cost state. Once BSSF is updated, prune.

Complexity Classes

P – Polynomial time, set of problems for which we can find and verify a solution in deterministic polynomial time.

NP – Non-deterministic polynomial time. We can verify a solution in deterministic polynomial time.

NP-complete – NP and other search problems reduce into it.

Bounded approximation algorithms

Approximation factor

Ex. Set cover -> greedy algorithm finds set cover of cadinality at most OPT(I)log(n) n = size of set.

Approximation ratio = maxi A(I)/OPT(I)

TSP – no efficient approximating unless P = NP

Local Search

Basic algorithm and properties

S = random solution

While s’ in neighbors and cost(s’) < cost(s) do

S = s’

Return s

What is a solution? Who is my neighbor? How to measure cost?

Local minima – Changing one factor of the algorithm results in higher cost (local min). Escape using randomness, try different starting states.

Advanced Algorithms

Randomized algorithms – Randomness included to speed up complicated tasks with no deterministic algorithm, multiple coupled variables, uncertainty in inputs.

Monte Carlo vs. Las Vegas – MC (Fermat, Local Min) = Consistent time complexity (set run times) chances of a bad answer, increase time = better answer , LV (Selection, Quicksort) = returns a correct answer, probability that it takes a relatively long time to find or finds no answer.

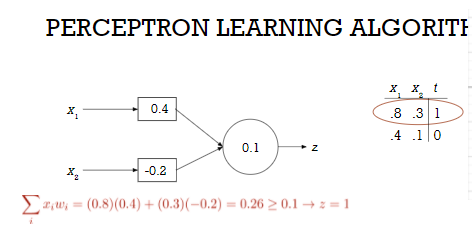
MC – randomly draw k samples, compute k, aggregate results.

Amplification of stochastic advantage – ex. Using k tests in fermat to make a probabilistic statement about likelihood of primality. (50% vs ½^k)

Quantum computing – Qubits represent multiple states as waves, run with gates, very fast non-deterministic computing.

Genetic algorithms – Algorithms that model nature. “Breed solutions” as binary strings/input and run for set number of generations to create more optimal solutions. Population size? Parent selection? Cross over mechanics? Mutation mechanics? Survival/fitness determining?

Perceptron learning algorithm – deltaw = learning rate( (error) t – z)xi <- input strength. Neural networks.



Analysis

Empirical analysis – Running tests, checking times etc.

Average case analysis (high level)

Overall summary

Divide and Conquer

Problem has natural hierarchy with independent branches

Speed-up happens when we can find short-cuts during partition/merge that can be taken because of the divide and conquer paradigm

Graph Algorithms

When finding reachability, paths, and properties of tasks represented as graphs, often fall under other approaches

Greedy

Common simple and fast approximation approach, occasionally optimal

Dynamic Programming

Overlapping subproblems (given by a recursive definition) that are only slightly (constant factor) smaller than the original problem, solved with the proper ordering

Linear Programming

Any optimization with linear objective and constraints

Intelligent Search

Effective when we have some heuristic knowledge of the search space to allow pruning

Local Search

Simple optimization technique for many complex search spaces – local optima issues

Stochastic Algorithms

Sampling problems, amplification of stochastic advantage, takes advantage of fast computers, etc.

**Midterm Review**

Primality testing

Fermats – if p is prime then for every 1<=a<p; a^(p-1) **≡** 1 mod p

Fermats(N,k) N = number to test, k number of tests = O(n^3)

Select k random numbers (x) between 0 and N – 1

If modexp(x, N-1, N) = 1 Mod N for all x in i=0..k

Return yes

Else: return no

\*\*Fermats is not iff, just if. It says only what happens when N is prime, not when it is composite. Error analysis says prob false positive <= 1/(2^k).

RSA = O(n^4**)**

1.Pick two primes p,q, let N=pq = O(n^4)

2. Choose e relatively prime to (p-1)(q-1) = O(n^3) Euclids

3. Find d such that de **≡** 1 mod (p-1)(q-1) = O(n^3) extended

4. Encrypt message x as x^e mod N **=** y = O(n^3) Modexp (5 also)

5. Decrypt code y as y^d mod N = (x^e)^d mod N = x^de Mod N = x O(n^3)

\*O(n^4) because choosing primes is slow

While True do

Choose random n bit number p O(n)

If PrimeTest2(p) = yes then return p O(n^3)

\*Private key encryption = shared secret key encrypts/decrypts data

\*Public key = public & private keys are different but mathematically linked, RSA=public

BFS(G, s)

For all u in V do

Dist(u) = infinity

Dist(start) = 0

Q.inject(S)

While Q is not empty do

U = Q.eject

For all edges(u,v) in E do

If dist(v) = infinity then

Q.inject(v)

Dist(v) = dist(u)+1

Return dist

BFS vs. DFS

BFS: backtracks immediately, guaranteed to find shortest path, deep goal may be slow, memory grows with width, no problems with deep path

DFS: goes deep before backtracking, typically not shortest path, can find deep goal quickly, only need memory = depth, infinite deep paths are bad.

Dijkstra’s Algorithm: O(|V|(cost insert+costdeletemin)+|E|(cost-decreasekey))

Dijkstra(G,l,s) – Working with edges with weight NO NEGATIVE

For all u in V do

Dist(u) = infinity

Prev(u) = NULL

Dist(start) = 0

H.makequeue(V) {Distance = key}

While H is not empty do

U = H.deletemin()

For all edges(u,v) in E do

If dist(v) > dist(u) + l(u,v) then

Dist(v) = dist(u) + l(u,v)

Prev(v) = u

H.decreasekey(v)

Return dist, prev

Dijkstra by data structure

Implementation | deletemin | insert/decreasekey | |V|x deletemin+|V|+|E|x insert

Array O(|V|) O(1) O(|V^2|)

Binary Heap O(log|V|) O(log(|V|) O((|V|+|E|)log|V|)

d-ary heap O(dlog|V|/logd) O(log|V|/logd) O((|V|d + |E|log|V|/logd)

Fibonacci heap O(log|V|) O(1) amortized O(|V| log |V| + |E|)

Bellman-Ford: O(|V||E|)

BellmanFord(G,l,s): Graphs with negative weights, just iterate over everything!

For all u in V do

For all u in V do

Dist(u) = infinity

Prev(u) = NULL

Dist(start) = 0

For I = 1 to |V|-1 do

For all edges(u,v) in E do

If dist(v) > dist(u) + l(u,v) then

Dist(v) = dist(u) + l(u,v)

Prev(v) = u

Return dist, prev

Master Theorem

If T(n) = aT([n/b]) + O(n^d) for a > 0, b > 1, d >=0, then

O(n^d) if 1 > a/(b^d) == d > logba

O(n^d log n) if 1 = a/(b^d) == d = logba

O(n^log­b­a) if 1 < a/(b^d) == d < logba

A = branching factor, b = number of sub problems, d = pre/post work

Text, letter

Description automatically generated

F(n) 756n^2 + 27n + 10003 g(n) = n^2

Limn->inf of f(n)/g(n) = limn->inf 756 + limn->inf 27/n + limn->inf 10003/n^2

= 756 + 0 + 0 🡪 756 is real number -> f(n) in Big theta (g(n))

**Greedy algorithms**

Philosophy: Do the best option available at the time!

MST = minimum spanning tree, find three that minimized total edge weights

Prims(G,l)

For all u in V do

Cost(u) = infinity

Prev(u) = NULL

Chose starting node (random/assigned)

Cost (start) = 0

H.makequeue(V) {cost as keys}

While H is not empty do

V = H.deletemin()

For each (v,z) in E where !visited(z) do

If cost(z) > l(v,z) then

Cost(z) = l(v,z)

Prev(z) = v

H.decreasekey(z)

Return cost,prev

Kruskals(G,l)

For all u in V do

Makeset(u)

X = []

Sort edges E by weight

For all edges (u,v) in E in increasing order do

If find(u) != find(v) then

X = X union (u,v)

Union(u,v)

Return X

Modexp(x,y,N) = x^y mod N = O(n^3)

If y = 0: return 1

z = modexp(x, [y/2], N)

if y is even: return z^2 mod N

else: return x \* z^2 mod N

Table

Description automatically generated

Euclid(a, b) = a >= b >= 0, gcd of a and b =

If b = 0: return a

Return Eculid (b, a mod b)

Extended-Euclid(a,b) a>=b>=0, int x,y,d that d = gcd(a,b) and ax+by=d

If b=0: return (1,0,a)

(x’,y’,d) = extended-Euclid(b, a mod b)

Return (y’,x’-[a/b]y’, d)