→ Backtracking [Reduce and Conquer]

* Can be applied to search and optimization problems.
* Gives more efficient runtime over exhaustive search or brute force but may not result in polynomial time. It is usually IMPROVED EXPONENTIAL time, and can solve NP complete problems.
* Uses the constraint to eliminate impossible solutions early on in their development.
* For recursive calls, input size is reduced by CONSTANT DIFFERENCE as opposed to CONSTANT FACTOR

Maximal Independent Set: For an undirected graph find the largest set of vertices such that there is no edge between any 2 of them

Greedy → Select person with fewest enemies, remove enemies, and repeat

$|GS| \le \frac{|OPT|}{D+1}$ D: maximum degree for a vertex ⟶ bad approximation ratio

$T(n) = 2T(n-1) + O(n) \longrightarrow O(2^n)$
$T(n) = T(n-1) + T(n-2) + O(n) \longrightarrow O(1.62^n)$
$T(n) = T(n-1) + T(n-3) + O(n) \longrightarrow O(1.46)^n$

IMPROVE ↓

MIS3(G, undirected graph)
  if $|V| = 0$:
    return ∅
  pick a vertex v.
  In = MIS3(G − {v and all of v's neighbors}) ∪ {v}   ← $T(n-1-\deg(v))$
  if deg(v) = 0 or deg(v) = 1:
    return In   ← new graph $O(n)$
  Out = MIS2(G − {v})   ← $T(n-1)$
  If lInl>lOutl:
    return In
  else:
    return Out

Best known algorithm is $O(1.2)^n$

Weighted Event Scheduling (no known greedy algorithm)

Worst case: $T(n) = 2T(n-1) + O(n) = O(2^n)$

BTWES($I_1, \ldots, I_n$) (sorted by end times.)
  if n = 0: return 0
  if n = 1: return $value(I_1)$
  OUT = BTWES($I_1, \ldots, I_{n-1}$)
  Let I_k be the last event to end before I_n starts.
  IN = BTWES($I_1, \ldots, I_k$) + $value(I_n)$
  return max(OUT,IN)

However, at most (n+1) recursive calls
$I_1 \ldots I_k$

DPWES($I_1, \ldots, I_n$; $v(I_1), \ldots, v(I_n)$) ordered by end times.
  $A[0] = 0$
  for $k = 1 \ldots n$:
    $j = 1$ ← find last event that ends before I_k starts
    while End(I_j) ≤ Start(I_k):
      $j = j + 1$
    $A[k] = \max(A[k-1], v(I_k) + A[j-1])$
  return $A[n]$
(out / in)

→ Dynamic Programming

(S1) The definition of subproblems is often just a restament of the original problem.

(S3) Defining the recurrence relation often involves case analyis: in/out, 3 paths, x books to place on last shelf

★ Iterating over all previously computed values

String Reconstruction

StringReconstruction(x[1…n])
Initialize all S(.) to be false and all prev(.) to be nil
S(0):=true
for k from 1 to n
- j:=k-1
- while (not S(k)) and j>0
  - if S(j) is true and x[j+1,…k] is a valid word, then
    - S(k):=true
    - prev(k):=j
  - else
    - j:=j-1

If S(n) then
- p:=n
- while p>0
  - print(p)
  - p:=prev(p)

Can a string of length n be broken up into words.

Key Idea → S[k] is true if there exists any $j \in [1,k]$ such that S[j] == True and x[j+1…k] is a word

Knapsack Problem

How to maximise value with a fixed capacity C.

BTKS($w[1 \ldots n], v[1 \ldots n], C$)
  IF $n = 0$:
    return 0
  IF $C = 0$:
    return 0
  IF $w[n] > C$:
    return BTKS($w[1 \ldots n-1], v[1 \ldots n-1], C$)
  IN = BTKS($w[1 \ldots n], v[1 \ldots n], C - w[n]$) + $v[n]$
  OUT = BTKS($w[1 \ldots n-1], v[1 \ldots n-1], C$)
  return max(IN,OUT)

Solving iteratively: KS(j, b) is the maximum value that can be fit in a b capacity knapsack using items 1…j

Knapsack($w[1 \ldots n], v[1 \ldots n], C$)
$KS[j, 0] = 0$ for all j
$KS[0, b] = 0$ for all b
for j from 1 to n:
  for b from 1 to C:
    if $w[j] > b$:
      $KS[j, b] = KS[j-1, b]$
    else:
      IN = $v[j] + KS[j, b - w[j]]$
      OUT = $KS[j-1, b]$
      $KS[j, b] = \max(IN, OUT)$
return $K[n, C]$

Runtime: $O(nC)$

Coin Denomination

dp[i, j] = ways to form amount j with denominations 1…i

if $(j - c_i \ge 0)$? IN = dp[i, j − c_i] : IN = 0
OUT = dp[i, j−1]
dp[i, j] = IN + OUT

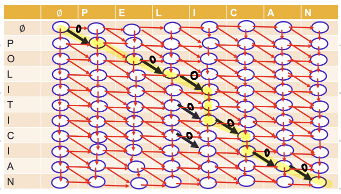
Edit Distance

Transform word x into word y using insertion, deletion, and substitution.

Let E[i, j] be the minimum cost to transform x[1…i] into y[1…j]

EditDist(x[1…n],y[1…m])
Initialize for i from 1 to n, E(i,0)=i and for j from 1 to m, E(0,j)=j
for i from 1 to n
  for j from 1 to m
    IF $x[i] == y[i]$:   deletion   insertion   substitution
      E(i,j)=min(1 + E(i-1,j), 1 + E(i,j-1), 0 + E(i-1,j-1))
    IF $x[i] \ne y[i]$:
      E(i,j)=min(1 + E(i-1,j), 1 + E(i,j-1), 1 + E(i-1,j-1))

Return E(n,m)



$|V| = O(nm)$
$|E| = O(3nm)$
Dijkstra's
$O(nm \log(nm))$
⇓
can be improved since this is a DAG

Shortest path in DAG

When we follow topological ordering, we can be certain that once we compute the dist for each vertex, we are done with that vertex.

DAGDP(G,s):
  $dist(s) = 0$
  Let $v_1, \ldots, v_n$ be a list of all vertices after s in topological ordering.
  for $i = 1 \ldots n$:
    $dist(v_i) = \min_{(v, v_i) \in E} \{dist(v) + \ell(v, v_i)\}$

Runtime = O(|V| + |E|) so even using this method edit distance can be solved in O(nm) time.

★ works for both ⊕ and ⊖ weights since being a DAG prevents the problem of negative cycles.

Longest Increasing Subsequence

Return the shortest path in DAG:
Vertices: Positions in array
Edges: (i, j) where i < j and a[i] < a[j] each with weight −1

→ Negative Cycles

A cycle in a graph such that the sum of edges is a negative number.

Dijkstra's efficiency relies on the fact that all edge weights are non-negative. With negative weights, runtime can be exponential

(assumption) BUT NO −ve CYCLES

→ Shortest Path (with −ve Weights)

We turn a graph with cycles into a DAG so we can use DP with a budget T.

B[i, t] = shortest path from v_0 to v_i with at most t edges

$$B(i,t) = \min \begin{cases} B(0,t-1) + w(v_0, v_i) \\ B(1,t-1) + w(v_1, v_i) \\ B(2,t-1) + w(v_2, v_i) \\ B(3,t-1) + w(v_3, v_i) \\ \ldots \\ B(n-1,t-1) + w(v_{n-1}, v_i) \end{cases} = \min_{(v_j, v_i) \in E} [B(j, t-1) + w(v_j, v_i)]$$

(when no edge exists $w(u,v) = \infty$)

★ Subproblems have to be ordered by column

Assuming there are no negative cycles, the shortest path is a simple path. So, T = n − 1 (budget)

In such a situation, distance never improves beyond a budget of n−1
ie. B[i, n−1] = B[i, n] = B[i, n+1] = …

→ Bellman Ford

detect −ve cycles, if none return shortest dist array

BFDP(G,v_0) (graph with edge weights)
$B[0,0] = 0$
$B[i,0] = \infty$ for all $i \ne 0$
for $t = 1, \ldots, n$: ← max edges
  for $i = 0, \ldots, n-1$: ← for each vertex
    $B(i,t) = \min_{(v_j, v_i) \in E} [B(j, t-1) + w(v_j, v_i)]$ O(indegree(v_i))
for $i = 0, \ldots, n-1$:
  if $B(i, n-1) \ne B(i, n)$:
    return "negative cycle!!!!"
return $[B(0,n), B(1,n), B(2,n), \ldots, B(n-1,n)]$

Runtime: $O(n)$ $n \sum_i \text{indegree}(v_i) = n(|V| + |E|) = n(n+m)$

Runtime = $O(|V||E|) = O(n^3)$

→ Shortest Path Between all Pairs Of Vertices — Floyd Warshall

Given a directed graph with n vertices and m edges we can determine the shortest path between any 2 vertices. (−ve edges are allowed in G but not cycles)

Bellman Ford n times $= O(n \cdot nm) = O(n^4)$
Floyd Warshall $= O(n^3)$

Let FW(i, j, t) be the length of the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as possible intermediate vertices.

The question is whether going through v_t improves the shortest distance.

$$FW(i,j,t) = \min[FW(i,j,t-1), FW(i,t,t-1) + FW(t,j,t-1)]$$

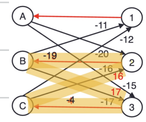
Algorithm:

Floyd-Warshall(G):
$FW[i, j, 0] = w(i, j)$ if $(v_i, v_j) \in E$
$FW[i, j, 0] = \infty$ if $(v_i, v_j) \notin E$
for $t = 1 \ldots n$:
  for $i = 1 \ldots n$:
    for $j = 1 \ldots n$:
      $FW[i,j,t] = \min(FW[i,j,t-1], FW[i,t,t-1] + FW[t,j,t-1])$
return $FW[i, j, n]$ for all i, j.

→ Maximum Bipartite Matching

Then create a graph with each red circle corresponding to a left positive edge and each other entry corresponding to a right negative edge. Look for negative cycles and if you find one, then flip around all edges in the cycle.

	1	2	3
A	14	20	15
B	11	19	17
C	12	16	4



Keep doing this until there are no more negative cycles left!!

To detect negative cycles, use Bellman-Ford which runs in OlVIlEl = O(n^3) time Each negative cycle will increase the total cookie amount by at least 1, so in the worst case, you will have to increase C times where C is the total weights of all cookies (can you think of a slightly tighter upper bound?).

So the total runtime of the algorithm is $O(Cn^3)$.

→ Hill Climbing

Start with any solution and make local changes to improve while Obeying constraints.
May give a LOCAL OPTIMUM, not optimal solⁿ

→ Linear Programming

↳ optimization problems

- It works in the case when the constraints and objective function are all linear equations or inequalities.
- Essentially, the constraints limit the solution space to a polygon or multi-dimensional polyhedron.
- Then since the objective function is linear as well, there are no local optimums so the global optimum occurs at a corner of the polyhedron.
- Linear programming is the process of traveling from one vertex to another, always improving until you cannot improve any more.

Linear programming reduces to solving the " simplex algorithm"

NETWORK FLOW

- Instance: Directed graph with non-negative edge weights, called the capacity of edges. Two vertices, s: source, t: sink
- Solution format: An assignment of non-negative values to each edge.
- Constraints: At any vertex except s, t total flowing in = total flowing out. Flow along edge cannot exceed capacity of edge.
- Objective: maximize total flow out of s = total flow into t

Residual flow graph

- Ford-Fulkerson insight: We can represent the problem of improving a flow as another flow problem, for the residual graph
- If $f(e)$ is flow on edge e and $c(e)$ is capacity of edge e then change the capacity to $c(e) - f(e)$ and add $f(e)$ to the capacity of the reverse edge.

We Stop when there is no path left from s to t.

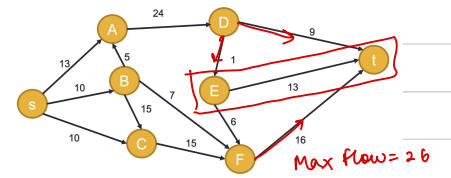
Let T =V-S, the unreachable vertices. Then all edges e=(u,v) with u in S and v in T are not in the residual graph. Therefore, they are all being used to full capacity.
Let Cut(S,T)= the total capacity of all such edges. Then Flow(f)=Cut(S,T).

Runtime and Algorithm

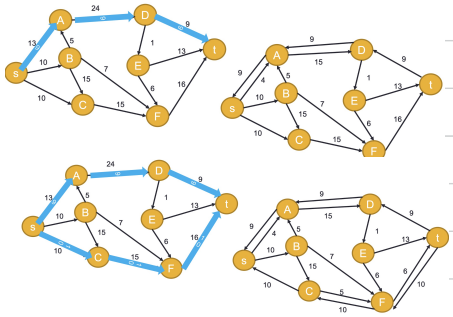
* May not terminate with irrational weights

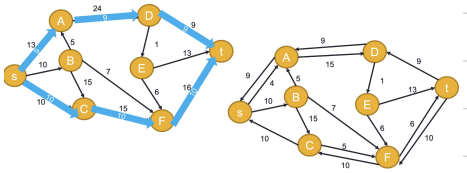
FF(G, c, s, t)
  Repeat until there is no path in the residual graph:
    Find a path in the residual graph (e.g., use DFS) from s to t
    Augment the flow along every edge in this path.
    Create new residual graph

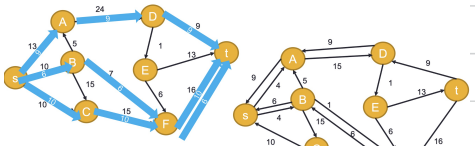
At most W|V| iterations of outside loop, each iteration takes O(|E|) time, so O(W|V||E|).

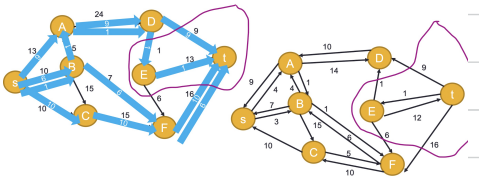


Max Flow = 26









→ P vs. NP

NP: the set of all decision problems that are verifiable in polynomial time

P: The set of all decision problems that can be solved in polynomial time.

* Any decision problem that can be solved in polynomial time can also be verified in polynomial time → P ⊆ NP

Alternately,

NP: The class of all *decision* problems that can be *decided* using a *non-deterministic Turing machine* in polynomial time.

P: The class of all *decision* problems that can be *decided* using a *deterministic Turing machine* in polynomial time.

Imp:
The decision version of TSP is in NP.

→ Pseudo Polynomial Time

- An algorithm is said to run in polynomial time if the runtime of the algorithm is polynomial with respect to the *size* of the input (in other words, the number of bits required to store the input)
- An algorithm is said to run in pseudo-polynomial time if its input is a value and the algorithm is polynomial with respect to the *value* of the input.

If an algorithm runs in O(C) it runs in $O(2^{\log_2(C)})$

Eg: Knapsack DP, Determining if a num is is prime by dividing by each of 2,3 ... $\sqrt{n}$

→ Reductions

* $A \leq_r B$: A reduces to B
B is "relatively harder" than A

- If $A \leq_r B$ and $B \in P$ then $A \in P$
- Proof: Let Alg_B be a polynomial time algorithm that solves B.
- Define $Alg_A(x) := Alg_B(F(x))$ using the reduction F.
- Since Alg_B and F are both polynomial time, then Alg_A is also polynomial time.

- If $A \leq_r B$ and $A \notin P$ then $B \notin P$
- Proof: By contradiction let Alg_B be a polynomial time algorithm that solves B.
- Define $Alg_A(x) := Alg_B(F(x))$ using the reduction F.
- Then we have built a polynomial time algorithm for A which was assumed to not exist.

NP Complete : A is NP complete if all problems in NP reduce to A

Cook Levin Thm : SAT is NP complete

→ SAT (The problem of Satisfiability)

- Instance: Boolean formula (in CNF)
- Solution format: Boolean assignment of the variables
- Constraint: The assignments make the formula True.
- Decision: If a solution exists return True, otherwise return False.

2 SAT : each clause has 2 literals
3 SAT : each clause has 3 literals

* SAT is NP complete

→ 2SAT

2 SAT ∈ P since SCC ∈ P and 2SAT $\leq_r$ SCC

Note
- Transitivity of implication: a→b and b→c then a→c
- Contradiction: $x \leftrightarrow \bar{x}$
- $x \vee y$ is the same as: $\neg x \rightarrow y$, $\neg y \rightarrow x$

- Create a graph G
- For each literal x create 2 vertices: one for x and one for $\bar{x}$.
- For each clause $(x \vee y)$ create two edges for $(\bar{x} \rightarrow y)$ and $(\bar{y} \rightarrow x)$
- There is no chance of assignment that makes the formula true if there is a contradiction.
- There is a contradiction if $x \rightarrow \bar{x}$ and $\bar{x} \rightarrow x$
- What does this mean in the graph we built?
- Run SCC algorithm on this graph. For each variable x, if x and $\bar{x}$ are in the same SCC then return False. Otherwise return True.

→ 3SAT

if you can solve 3SAT, you can also solve SAT

(3SAT) $\leq_r$ (SAT) trivially

(SAT) $\leq_r$ (3SAT) interesting

- Consider a clause in the form:
$$(a_1 \vee a_2 \vee \cdots \vee a_n)$$
- This clause has an assignment that makes it true iff the following clause has an assignment to make it true:
$$(a_1 \vee a_2 \vee y_1) \wedge (\bar{y_1} \vee a_3 \vee y_2) \wedge (\bar{y_2} \vee a_4 \vee y_3) \ldots (\bar{y_{n-2}} \vee a_{n-1} \vee a_n)$$

Eg $a_1 \vee a_2 \vee a_3 \vee a_4 \equiv (a_1 \vee a_2 \vee y_1) \wedge (\bar{y_1} \vee a_3 \vee a_4)$

→ NP Complete Problems

SAT, 3SAT, Hamiltonian Path, TSP (decision), Independent Set, Knapsack, 3 Coloring

Optimization → BUDGET → decision

→ Vertex Cover

Find smallest subset of vertices that cover all edges, → Optimization
Decision → can we cover all edges using B vertices

Given an instance I of 3SAT, create G:
- 1. Create an edge: $\{x, \bar{x}\}$ for each variable.
- 2. Create a triangle: $\{a,b\},\{b,c\},\{c,a\}$ for each clause $(a \vee b \vee c)$
- Then for each literal x in step 1, create an edge to each x in step 2.

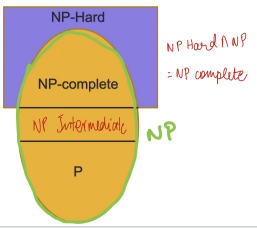
Then ask if G has a VC with B or fewer vertices where $B = \ell + 2m$ where ℓ is the number of literals and m is the number of clauses

- Each Δ has exactly 2 vertices in cover (2m)
- Each vertex in backbone has exactly 1 vertex in cover (ℓ)

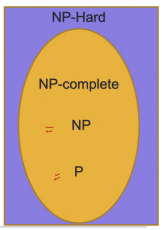
*A problem H is called NP Hard if all all problems in NP can be reduced to H

* NP Hard vs NP complete
NP Complete ⊆ NP NP Hard ⊄ NP

P ≠ NP?



P = NP?



→ Some T/F → all false

If a problem is in NP then there is no known polynomial time algorithm to solve it.

If you perform explore on any DAG starting at a source vertex, then all vertices in the DAG will be marked as visited. → what if there are 2 sources

If an undirected connected graph has a cycle, and e is the unique lightest edge of that cycle, then e must be part of all MSTs. → what if there are 2 cycles

For counter examples ⇒ focus on wording, consider tries / 2 options/etc

→ Try to keep exchange argument simple. Only add / remove / shift 1 element if that works

CONCRETE DP EXAMPLES

→ For dp, at times the final result may be the max / min of entire dp array
Eg: when dp[i,j] is max money earned ending at cell (i,j) of grid

→ For dp think about "states"
Eg1 Longest common subsequence :
LCS in prefix 1...i and 1...j

$$dp[i,j] = \begin{cases} 1 + dp[i-1, j-1], & a[i] == b[j] \\ \max(dp[i-1,j], dp[i,j-1]), & a[i] != b[j] \end{cases}$$

Ans: dp[n][m]

Eg2 Longest common substring
Longest common substring ending at (i,j)

$$dp[i,j] = \begin{cases} 1 + dp[i-1, j-1], & a[i] = b[j] \\ 0, & a[i] != b[i] \end{cases}$$

Ans: max over all values in dp

Eg3 Longest Palindromic Substring
dp[i,j] = T/F if there is a palindrome starting at i and ending at j

dp[i,j] = T if (i == j)
dp[i,j] = T if (s[i] == s[j] and dp[i+1][j-1] is true)

→ Greedy Proof with Cases

Case 3: c_1 is paired with c_j and c_2 is paired with c_i. Then create OS' from OS by removing the pairs $(c_1, c_j), (c_2, c_i)$ and replacing them with $(c_1, c_2), (c_i, c_j)$
OS' is valid: By assumption $c_2 - c_1 \leq T$ so (c_1, c_2) is a valid pair.
Subcase 1: $c_i \leq c_j$: Then $c_j - c_i \leq c_j - c_1 \leq T$.
Subcase 2: $c_i > c_j$: Then $c_i - c_j \leq c_i - c_2 \leq T$.
So by these two cases, (c_i, c_j) make a valid pair, and with the validity of OS, OS' is valid.
OS' is just as good as OS: there the same number of pairs in OS' than OS so $|OS'| = |OS|$.

Case Analysis *

→ Max Product in Partition

- **Recursion and Case analysis:** To compute $P[i]$, we can do a case analysis based on what is the biggest part in the product.
Case 1: one of the parts is 1. Then $P[i] = 1 * P[i-1]$.
Case 2: one of the parts is 2. Then $P[i] = 2 * P[i-2]$.
...
Case $i-1$**:** one of the parts is $i-1$. then $P[i] = (i-1) * P[1]$.
Case i**:** the only part is i in which case $P[i] = i$.
Since we do not know in advance which case results in the biggest product, we take the maximum over all cases:

→ Maximize items purchased with budget

Define Potatoes$P_1[0,...,B], P_2[0,...,B], ...P_n[0,...,B]$
(1) **for** $i = 0, \ldots, n$:
(2) $M[i,0] = 0$
(3) **for** $b = 0, \ldots, B$:
(4) $M[0,b] = 0$
(5) **for** $i = 1, \ldots, n$:
(6) **for** $b = 1, \ldots, B$:
(7) $tempmax = 0$.
(8) $prev(i,j) = \emptyset$
(9) **for** $k = 0, \ldots, b$:
(10) **if** $P_i(k) + M[i-1, b-k] > tempmax$:
(11) $tempmax = P_i(k) + M[i-1, b-k]$
(12) $prev(i,b) = (i-1, b-k)$
(13) $M[i,b] = tempmax$
(14) **print** $M[n,B]$

How to reconstruct answer using prev pointer