CSE 417 Final Prep

zjmcnulty

February 2019

Contents

1 Misc Algorithms

1.1 Stable Matching Algorithm

Definition

Given two sets X and Y of equal size $|X| = |Y| = n$, where each element $x \in X$ has a list of preferences for each element $y \in Y - \{y_1, y_2, ..., y_n\}$ and vice versa, find a perfect matching (i.e. pair each x with one of the y uniquely) $\{(x_i, y_j) : i, j \in [n]\}$ that is *stable*. A perfect matching is stable if:

- There exists no set of pairs $(x, y), (x', y')$ such that x prefers y' to y and y' prefers x to x'
- i.e. given the choice, no two people would want to leave their current pairing to pair with each other instead.

Implementation

1. Choose an $x \in X$ that is not currently paired with a $y \in Y$.
2. Have x propose to join with their first preference y_i.
   (a) If y_i is unpaired, tentatively pair them with x
   (b) If y_i is paired to x_j but prefers x to their current pairing, tentatively pair y_i to x and mark x_j as unpaired
   (c) If y_i does not prefer x to their current partner, have x propose to join their next highest preference.
3. Repeat steps 1 and 2 until no such x is unpaired.

Properties

- Guaranteed to exist
- the proposing party, i.e. X, gets the highest possible preferences in a stable matching. The other party, Y, gets their lowest possible preferences in a stable matching.
- Fairly easy to modify to consider other possible pairings involving some kind of preference list.

2 Graph Algorithms

2.1 Breadth First Search - Connected Components

Definition
Graph traversal algorithm that explores nearby nodes in the tree before ones farther away.

Implementation

1. Initialization
   - toVisit = $Queue(startNode)$
   - mark $startNode$ as discovered and all other nodes as not discovered.
2. While toVisit is not empty, choose the first element v in the Queue.
   - For each edge (v, u) if u is not discovered, mark it as discovered and add it to the end of the queue toVisit. Here you could also keep track of predecessors and other information depending on what you are using BFS for.

Properties

- $O(|V| + |E|)$ runtime ($O(|V|)$ for sparse graphs and $O(|V|^2$ for dense graphs).
- The traversal path of BFS forms a tree called the BFS tree. All edges of the graph not in the tree must be between the same or adjacent levels (depths) in the tree.
- When a node is first discovered, the path taken from the start to reach that node is (one of) the shortest possible path.
- level i in the BFS tree is exactly the set of vertices that are i edges away from the start vertex.

BFS(s) Implementation

Global initialization: mark all vertices "undiscovered"

```
BFS(s)
    mark  s "discovered"
    queue = { s }
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u fully explored
```

Exercise: modify code to number vertices & compute level numbers

27

2.2 Depth First Search - Connected Components

Definition

Graph traversal algorithm that explores nodes farther from starting point first before nearby nodes. Traversals all nodes in a given connected component.

Implementation

1. Initialization
   - toVisit = $Stack(startNode)$
   - mark $startNode$ as discovered and all other nodes as not discovered.
2. While toVisit is not empty, choose the top element v in the Stack.
   - For each edge (v, u) if u is not discovered, mark it as discovered and add it to the top of the stack toVisit. Here you could also keep track of predecessors and other information depending on what you are using DFS for.

Properties

- $O(|V| + |E|)$ runtime ($O(|V|)$ for sparse graphs and $O(|V|^2$ for dense graphs).
- When traversing a graph with DFS, the edges you choose form a tree called the DFS tree. Any edge in the graph G that is not part of this tree must go between a vertex and its ancestor in the DFS tree.
- When a node is first discovered, we are NOT gauranteed that it was discovered along the shortest possible path

DFS(v) – Recursive version

```
Global Initialization:
    for all nodes v, v.dfs# = -1   // mark v "undiscovered"
    dfscounter = 0   this is for numbering the vertices in the order we visit
                     them
DFS(v):
    v.dfs# = dfscounter++          // v "discovered", number it
    for each edge (v,x)
        if (x.dfs# = -1)           // tree edge (x previously  undiscovered)
            DFS(x)
        else …                     // code for back-, fwd-, parent-
                                   // edges, if needed; mark v
                                   // "completed," if needed
```

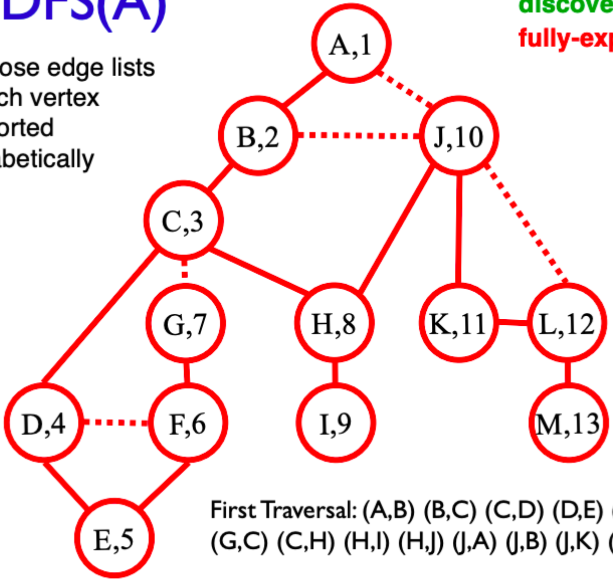
do whatever computations you need to do for whatever application you are using DFS for.

72

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

Color code:
undiscovered
discovered
fully-explored



Call Stack:
(Edge list)

TA-DA!!

First Traversal: (A,B) (B,C) (C,D) (D,E) (E,F) (F,D) (F,G) (G,C) (C,H) (H,I) (H,J) (J,A) (J,B) (J,K) (K,L) (L,J) (L,M)

105

2.3 Articulation Points

Definition

Given a graph G, an **articulation point** is any vertex where removing it (and all its adjacent edges) disconnects the graph (or increases the number of connected components of G is already disconnected).

Implementation

1. run DFS while keeping track of a few values for each vertex.
    - dfs# = the order the given vertex was visited in; i.e. if dfs# = 5, then the given vertex was the 5th vertex to be visited and expanded in DFS (NOT the order it was discovered in)
    - LOW(v) = the lowest vertex (in terms of dfs#) that v is adjacent to (besides its parent in the DFS tree) or that is adjacent to a vertex in the subtree rooted at v.

    $$LOW(v) = min(dfs\#(v), LOW(u)) \quad \forall (u, v) \in E \quad : u \neq parent(v)$$

2. When expanding a vertex v and exploring an edge (v, u)
    - If u is undiscovered, expand it. Set $LOW(V) = min(LOW(V), LOW(u))$. If $LOW(u) \geq dfs\#(v)$ then v is an articulation point
    - Else, u is a back-edge: if u is not $v's$ parent in the DFS tree, set $LOW(v) = min(LOW(v), dfs\#(u))$

Properties

- $O(|E| + |V|)$ i.e. speed of DFS.

DFS To Find Articulation Points

```
Global initialization: dfscounter = 0; v.dfs# = -1 for all v.
DFS(v):
    v.dfs# = dfscounter++
    v.low = v.dfs#                // initialization
    for each edge {v,x}
        if (x.dfs# == -1)         // x is undiscovered
            DFS(x)
            v.low = min(v.low, x.low)
            if (x.low >= v.dfs#)
                print "v is art. pt., separating x"
        else if (x is not v's parent)
            v.low = min(v.low, x.dfs#)
```

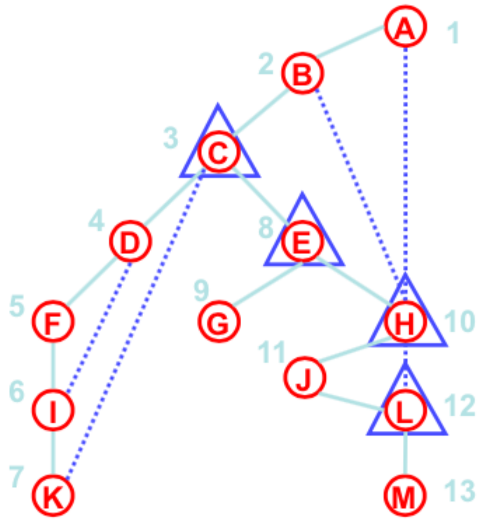
set v.low to the min of its childs LOW values

Except for root. Why?

What if G is not connected?

Equiv: "if({v,x} is a back edge)" Why?

Articulation Points



Vertex	DFS #	Low
A	1	1
B	2	1
C	3	1
D	4	3
E	8	1
F	5	3
G	9	9
H	10	1
I	6	3
J	11	10
K	7	3
L	12	10
M	13	13

2.4 Biconnected Components

Definition
A **biconnected component** is a **maximal induced subgraph** such that removing any of the vertices does NOT disconnect the subgraph. i.e. it is a set of vertices and all their corresponding edges such that removing any of the vertices does NOT disconnect the graph, but if we were to add any other vertices from the graph G to this set that would no longer be true.
Implementation

1.

Properties

•

2.5 Testing for Bipartiteness

Definition
A **bipartite graph** is a graph that is 2-colorable, i.e. a graph that can be separated into two parts X, Y such that all edges go between a vertex in X and a vertex in Y. Note, a graph is bipartite IFF it contains no odd-length cycles.

Implementation

1. Label the start vertex X.
2. run BFS
3. If a vertex is labeled X, label all its undiscovered children Y and vice versa.
4. If there is ever a point where a vertex has a (discovered) child whose label is the same as its own, then there is an odd-length cycle and the graph is not bipartite. Else, the labeling we have generated creates the two parts, X and Y.

Properties

- Bipartite graphs can be separated into two parts.

2.6 Topological Sort - Precedence Graphs

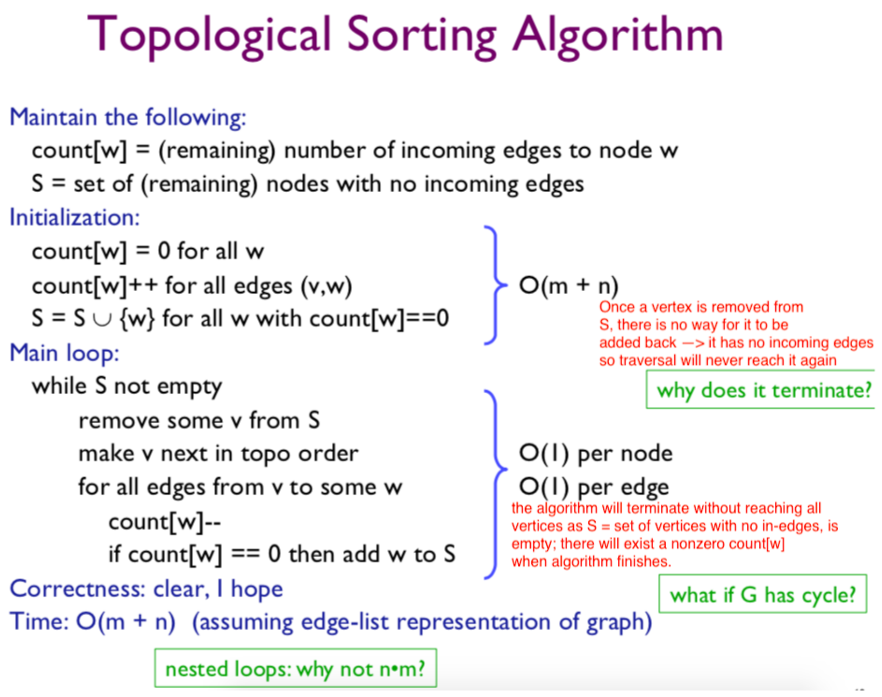
Definition
Given a directed graph G where the edges represent precedence (i.e. if there is an edge (u, v) this implies u must come before v in some ordering), a topological sorting is an order of the vertices that respects all these precedences (i.e. if the ordering is $(v_1, v_2, ..., v_n)$ then v_i comes before v_j implies that there is no precedence relation/edge in the graph (v_j, v_i)). It is a way of ordering the vertices that respects these precedence requirements.

Implementation

1. Count the number of incoming edges for each vertex
2. If one exists, remove one of the vertices that has no incoming edges and delete all edges adjacent to it. Add this vertex to the end of our topological sort. If one does not exist, and vertices remain, a topological sort is not possible
3. continue until all vertices are gone (success) or there are no vertices with zero incoming edges remaining (failure).

Properties

- Any directed graph can be topologically sorted iff it is a directed acyclic graph.



Topological Sorting Algorithm

```
Maintain the following:
  count[w] = (remaining) number of incoming edges to node w
  S = set of (remaining) nodes with no incoming edges
Initialization:
  count[w] = 0 for all w
  count[w]++ for all edges (v,w)                    } O(m + n)
  S = S ∪ {w} for all w with count[w]==0
Main loop:
  while S not empty
      remove some v from S
      make v next in topo order                     O(1) per node
      for all edges from v to some w                O(1) per edge
          count[w]--
          if count[w] == 0 then add w to S
Correctness: clear, I hope
Time: O(m + n)  (assuming edge-list representation of graph)
```

Once a vertex is removed from S, there is no way for it to be added back —> it has no incoming edges so traversal will never reach it again

why does it terminate?

the algorithm will terminate without reaching all vertices as S = set of vertices with no in-edges, is empty; there will exist a nonzero count[w] when algorithm finishes.

what if G has cycle?

nested loops: why not n•m?

2.7 Minimum Spanning Tree - Kruskals

Definition

A minimum spanning tree of a graph G with weighted edges (edge costs) is an acyclic connected subgraph of G (i.e. a tree that contains all the vertices of G) whose edges have the least possible total weight.

Implementation

1. Sort the edges by their cost in ascending order.
2. Take the lowest cost edge and if it does not create a cycle, add it to the tree.
3. Continue until $|V| - 1$ edges have been chosen, i.e. the number of edges in a tree.

Properties

- $O(Elog(E))$ runtime; sorting the edges is the slowest step
- Interestingly, the minimum spanning tree places some bounds on solutions to the Traveling Salesman Problem. Given a graph G it holds that:

$$MST(G) \leq TSP(G) \leq 2 * MST(G)$$

3 Greedy Algorithms

3.1 Minimum Spanning Tree - Kruskals

see section in Graph Algorithms.

3.2 Interval Scheduling

Definition
Given a set of intervals (that are allowed to overlap), find the largest possible subset that has no conflicting intervals.
Implementation

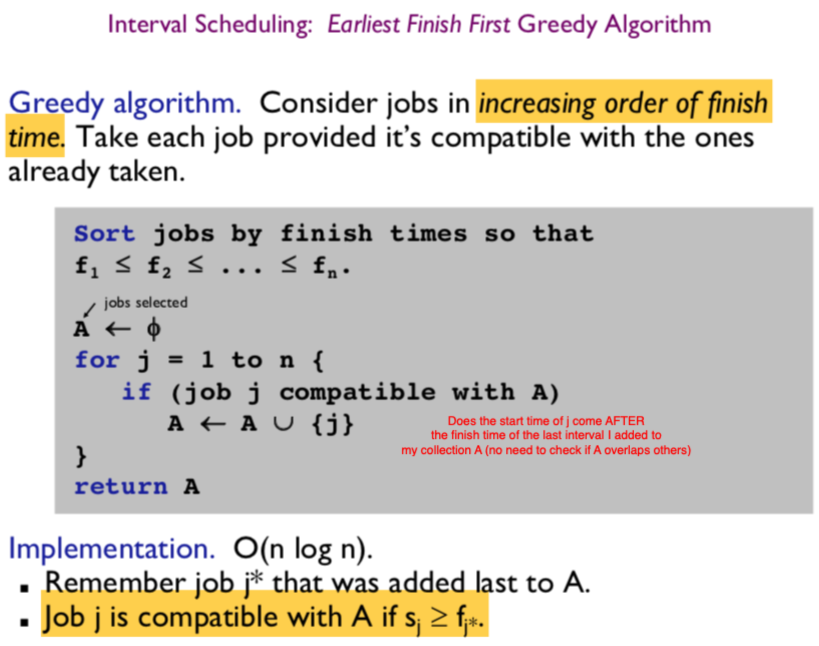
1. Order the intervals by finish time ascending $f_1 \leq f_2 \leq f_3 \leq ... \leq f_n$
2. Keep track of the previous finish time of the set of selected jobs.
3. If the start time of the next interval is after the previous finish time in the set, add the next interval to the set (and update previous finish time)
4. Repeat for all intervals

Properties

- $O(nlog(n))$ sorting the intervals is the slowest part.
- We can actually encode this problem into a graph. Create a vertex v_i for each interval i and connect v_i by an edge to v_j iff intervals i and j conflict. Then, finding the optimal interval schedule is the same as finding the largest possible independent set in this graph.

Proof of Correctness
Let $g_1, g_2, ..., g_n$ be the greedy choice of intervals and $j_1, ..., j_k$ be some other choice of (compatible) intervals. Let $f(x), s(x)$ denote the finish/start time of interval x respectively.

- $f(g_1) \leq f(j_1)$ as our greedy alg chose the min finish time.
- Induction: Assume $f(g_i) \leq f(j_i)$
- Thus, $s(g_{i+1}) \leq s(j_{i+1})$ which implies interval j_{i+1} would have been compatible with $g_1, g_2, .., g_i$. Thus, $f(g_{i+1}) \leq f(j_{i+1})$ as the algorithm chooses the compatible interval with the minimum finish time.
- As $f(g_n) \leq f(j_n)$, $k \leq n$ as otherwise job j_{n+1} would have been compatible with jobs $g_1, ..., g_n$ and I would have chosen to add it to my interval set.



Interval Scheduling: *Earliest Finish First* Greedy Algorithm

Greedy algorithm. Consider jobs in *increasing order of finish time.* Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that
$f_1 \le f_2 \le ... \le f_n$.

  jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Does the start time of j come AFTER
the finish time of the last interval I added to
my collection A (no need to check if A overlaps others)

Implementation. O(n log n).

- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \ge f_{j*}$.

3.3 Interval Partitioning

Definition

Given a set of intervals (that can overlap) find a way to partition the intervals into the fewest number of compatible groups. i.e. groups where no intervals overlap with each other. Define the **depth** of a set of intervals to be the maximum number of intervals that all overlap at a given time.

Implementation

1. Sort the intervals by start time
2. Generate a priority queue of all the current parts, sorted by the current finish time of the intervals in the part (i.e. the finish time of the last interval added to the part)
3. For each interval, check if it is compatible with the first part in the priority queue (i.e. if the intervals start time is greater than the finish time of that part)
    - If it is, add it to the part and resort the priority queue (i.e. deleteMin, add interval to part, add part back to priority queue)
    - If it is not, make a new part with just that interval in it and add it to the priority queue.

Properties

- $O(nlog(n))$; sorting + n deleteMin/add from priority queue at $O(logn)$.
- Greedy algorithm generates a partition with $d = depth$ parts, the minimum theoretically possible.

Greedy algorithm. Consider lectures *in increasing order of start time*: assign lecture to any compatible classroom.

```
Sort intervals by start time so $s_1 \le s_2 \le ... \le s_n$.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lect j is compatible with some room k, 1≤k≤d)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

For each part, keep track of the most recent finish time. Order the parts in a priority queue sorted by this finish time so that we only have to check if the given interval is compatible with the first part (by checking that the parts finish time <= the next intervals start time). If it is not compatible with the first part which has the earliest finish time, it certainly will not be compatible with any of the other parts, so we have to make a new part to put the interval into.

3.4 Lateness Minimization

Definition
Given a set of tasks where each task has a length/duration and a due date, what order should the tasks be completed in such that the maximum lateness is minimized (NOT total lateness; minimize the largest lateness of a single task)

Implementation

1. Sort the tasks by due date ascending.
2. Do the jobs in that order.

Properties

- $O(nlogn)$

Proof of Correctness

Consider the order of tasks chosen by our algorithm $g_1, g_2, ..., g_n$ and any other order $j_1, ..., j_n$. We can then convert the other order to our algorithm's order without making it any worse, thus showing the greedy order is optimal. First, note that as long as $j_1, ..., j_n$ is not nondecreasing in due date, there is always a pair j_k and j_m with $m > k$ such that j_m is due before j_k. This implies there exists a pair j_p and j_{p+1} such that j_{p+1} is due before j_p. Swapping the two can only reduce (or not change) the max lateness. Repeat until in order.

3.5 Huffman Codes/Compression

Definition
The Huffman code is a form of lossless compression. Given a file, find an optimal lossless

Implementation

1.

Properties

•

4 Divide and Conquer Algorithms

4.1 Merge Sort

Definition
Sorting a list of elements fast!
Implementation

1. Split the list in two and recursively sort the two halves
2. With the two sorted halves, keep track of an index i_1 and i_2 with where you are in the two lists.
3. if $L_1[i_1] \leq L_2[i_2]$ take the next element from L_1 and increment i_1. Else, take the next element from L_2 and increment i_2.
4. When you reach the end of one list, append the remaining elements of the other list to the end of our sorted list. Return.

Properties

- NOT inplace
- $O(nlogn)$

4.2 Closest Pair of Points

Definition

Given a set of n points (x_i, y_i) find the pair of points that have the minimum euclidean distance.

Implementation

1. Find a vertical line L such that half the points are $\leq L$ and half the points are $> L$. i.e. sort the points by x coordinate and separate into two, choosing L to be $mean(points[n//2], points[n//2+1])$
2. recursively find minimum distance between two points in left half and two points in right half. Call this distance δ
3. Find all points $p_1, ..., p_m$ who's x coordinate is less than δ from L. i.e. all (x_i, y_i) such that $|L - x_i| < \delta$. Sort these points by y coordinate.
4. For each p_i found in last step, while $p_{i+k}.y - p_i.y < \delta$, check if p_{i+k} and p_i are a distance less than δ apart. Update δ if necessary.

Properties

- $O(nlogn)$ if presort the points by x and recursively merge lists to sort by y.

4.3 Integer Multiplication (Karatsuba)

Definition

Implementation

1.

Properties

-

5 Dynamic Programming Algorithms

The idea of dynamic programming is that many algorithms that lend themselves naturally to recursion often involve a lot of redundant work. Within the recursive tree there are many nodes which are calculating the same value. For example, if we want to calculate the *nth* Fibonacci Number, we could use the simple recurrence:

$$f_n = f_{n-1} + f_{n-2}$$

With this, we could just recurse down to the base case of $f_1 = f_2 = 1$ and get our solution. However, this recursive tree is pretty large as we reduce the size of our problem by at most 2 each time $f_n \to f_{n-2}$. Fortunately, you will notice this tree includes many nodes that calculate the value f_c for some $2 \leq c \leq n$. Why not just compute it once? Each of these subproblems, i.e. calculate f_c, could be calculated in an optimal way to reduce redundant calculations an eliminate recursion. Its easy to see if we start calculating $f_3, f_4, ...$ we always have the subproblems we need to calculate the next term in the sequence and need no recursion. This only takes $O(N)$ time then! This is the heart of dynamic programming.

5.1 Change Making

Problem Definition

Given an integer s and a set of coin denominations $\{w_m\}$, what is the minimum number of coins required to make change for s? More formally, the problem is find a set of non-negative integer counts c_i :

$$min\left(\sum_{i}^{n} c_i\right) : \sum_{i}^{n} c_i * w_i = s$$

Algorithm

Use dynamic programming and define a subproblem $OPT[i]$ to be the optimal collection of coins (minimum total number) that sum to i. Then, there are several cases.

1. $i == 0$ and no coins are required.
2. There is at least one coin of denomination w_1
3. There is at least one coin of denomination w_2
4. ...
5. There is at least one coin of denomination w_m

Thus:

$$OPT[i] = min\begin{cases} 0 & i == 0 \\ 1 + OPT[i - w_1] & i \geq w_1 \\ \;... & \\ 1 + OPT[i - w_m] & i \geq w_m \\ & \end{cases}$$

Properties

- $O(nm)$ runtime which is $O(n)$ if $m << n$

5.2 Weighted Interval Scheduling

Problem Definition

Given a set of intervals I where each interval $i \in I$ has an associated weight w_i and takes up a specific portion of time ($start = s_i, finish = f_i$) find a non-conflicting (don't overlap in time) subset of I that has the largest total weight.

Algorithm

First, order the jobs by finish time. Use dynamic programming and define $OPT[k]$ to be the maximum possible weight obtained from a non-conflicting subset of I using only the first k intervals. Furthermore, for each interval $i \in I$ define $p(i)$ to be the highest indexed interval $j < i$ such that j does not conflict with i ($f_j \leq s_i$). Then to calculate $OPT[k]$ note that there are two cases:

1. interval k is part of the optimal subset using only the first k intervals. Thus $OPT[k] = w_k + OPT[p(k)]$ as by the definition of $p(k)$ only the intervals less than or equal to $p(k)$ are non-overlapping with interval k.
2. Otherwise, k is not part of the optimal subset and thus $OPT[k] = OPT[k-1]$

Properties

1. Runtime is $O(N)$ for dynamic programming but sorting takes $O(NlogN)$

5.3 Knapsack Problem

Problem Definition

Given a set of weights $W = \{w_1, ..., w_n\}$ and a capacity C find a subset of the weights that maximizes the sum without exceeding capacity.

Algorithm

Define a dynamic programming problem in the following way: Let $OPT[i, w]$ be the optimal subset of only the first i weights that sum to at most w. Then note to find $OPT[i, w]$ there are two cases

1. Weight w_i is in the optimal subset of the first i weights that sum to at most w. In this case then, $OPT[i, w] = w_i + OPT[i-1, w-w_i]$
2. Weight w_i is not in the optimal subset. Thus $OPT[i, w] = OPT[i-1, w]$

If $w_i > w$ then of course the second one of these possibilities must be true and if $i = 0$ or $w = 0$ then clearly no weights are included in the optimal subset. Thus :

$$OPT[i, w] = \begin{cases} 0 & i == 0 \cup w == 0 \\ OPT[i-1, w] & w_i > w \\ max(OPT[i-1, w], w_i + OPT[i-1, w-w_i]) & otherwise \end{cases}$$

Properties

- Running time is pseudopolynomial at $O(nW)$

5.4 Nussinov's Algorithm - RNA seconday Structure

Problem Definition

Given an RNA primary structure (i.e. a series of bases $\{A, U, G, C\}$) find an "optimal" RNA secondary structure (one that allows the most possible base pairings) satisfying the following conditions

- **NO tight turns**: any pairs must be between bases with at least 4 bases between them.
- **Complementary Pairing**: Only A can pair with U and vice versa; only G can pair with C and vice versa.
- **No crossing**: If (b_i, b_j) and (b_k, b_ℓ) are two pairs then we cannot have $i < k < j < \ell$ (a pseudoknot).

Algorithm

Define $OPT[i, j]$ to be the optimal (maximum) number of bases pairs possible between bases $\{b_i, ..., b_j\}$. Then there are a few cases:

1. $j - i \leq 4$ in which case the two are too close for any base pairs to form and thus $OPT[i, j] = 0$
2. No base pair occurs at base j: in which case $OPT[i, j] = OPT[i, j-1]$
3. A base pair occurs at j with base b_t for some $i \leq t < j - 4$. Thus, $OPT[i, j] = max(\{OPT[i, t-1] + 1 + OPT[t+1, j-1] : i \leq t < j-4\})$. NOTE you also have to check that b_j and b_t are base complements (A and U or G and C).

Properties

- Runtime $= O(N^3)$

5.5 String Alignment

Problem Definition

Given two strings s_1 and s_2, and a scoring function $\sigma(s_{1i}, s_{2i})$ what is the optimal way to align the two such that the total score is maximized. An **alignment** of two strings S, T is a pair of strings S', T' that include dashes such that $|S'| = |T'|$ and removing the dashes from S', T' leaves S, T respectively. NOTE: BOTH strings may have dashes in them

Algorithm

Define $OPT[i, j]$ be the score for the optimal alignment of the first i letters of s_1 and the first j letters of s_2. Then there are three possible cases:

1. $s_1[i]$ and $s_2[j]$ are paired in the optimal arrangement. Thus,
$$OPT[i, j] = \sigma(s_1[i], s_2[j]) + OPT[i-1, j-1]$$
2. $s_1[i]$ is paired with a dash. In which case:
$$OPT[i, j] = OPT[i-1, j] + \sigma(s_1[i], -)$$
3. $s_2[j]$ is paired with a dash. In which case:
$$OPT[i, j] = OPT[i, j-1] + \sigma(-, s_2[j])$$
4. **BASE CASE:**
In the case $i == 0$, we are out of letters from the first string so all the remaining letters from the second string must pair with a dash. Thus:
$$OPT[0, j] = \sum_{k=1}^{j} \sigma(-, s_2[k])$$
Similarly, for $j == 0$ we get:
$$OPT[i, 0] = \sum_{k=1}^{i} \sigma(s_1[k], -)$$

Properties

- Runtime : $O(|s_1||s_2|)$

6 P vs NP

Polynomial problems (P) are the subset of all problems that can be solved in polynomial time (i.e. the big-O runtime is $O(n^k)$ for some fixed k) while the set NP is a collection of problems that can be verified in polynomial time (given a possible solution to the problem, it can tell if it is a solution or not). Note that NP problems do are not required to run slowly. All polynomial time algorithms run in NP, but there are also problems in NP whose best known solution runs in exponential time.

6.1 Proving Something is in NP

Given some hint that is polynomial in size (if it was not then how could the verifier run polynomially?) there should exist a function (the verifier) that can determine if the hint represents a valid solution to the specific problem in polynomial time. Moreover, there are no hints representing a NO instance of the decision problem that can get the verifier to response YES this is a solution (i.e. the verifier works...)

1. Check that the input/problem instance is a well-formed representation of the problem to be solved
2. check that the hint h is a well-formed representation of the hint; i.e. it is in the format we are expecting
3. From the hint, determine if the given solution specified by h is valid or not

Below is an example for the KNAP problems proof of being in NP.

Proof: If the KNAP problem is defined as the set of weights $\{w_1, w_2, ..., w_n\}$ with a capacity C. Then let the hint h be a binary string where digit i is one if the proposed solution includes weight w_i and zero otherwise. Then the verifier simply has to check if :

$$\sum_{i=1}^{n} h[i]w_i = C$$

6.2 NP Problems

6.2.1 Boolean Satisfiability (SAT)

This problem deals with trying to find a satisfying assignment (or determining one does not exist) to a given boolean equation in **Conjunctive Normal Form (CNF)**. This form is:

$$clause_1 \wedge clause_2 \wedge .... \wedge clause_n$$

$$clause = literal_1 \lor literal_2 \lor ... \lor literal_m$$

Where each literal can be x_i or $\neg x_i$. Thus, a satisfying assignment (one that evaluates as true) has at least one literal in each clause be true and no contradicting literals (i.e. x_1 and $\neg x_1$ cannot both be true). Sometimes, there are limitations placed on the number of literals per clause. For example, a 3SAT problem is one in which each clause has AT MOST three literals.

6.2.2 Independent Set

Given a graph $G = (V, E)$ an independent set is a subset of the vertices $S \subseteq V$ such that no two vertices in S are adjacent (i.e. $v_1, v_2 \in S \leftrightarrow (v_1, v_2) \notin E$). The goal of the independent set problem is to find the independent set of the maximum size within the graph G.

6.2.3 Knapsack

Given a set of weights w_n and a capacity C, the goal of the knapsack problem is to find a subset of the weights S that has the maximum possible weight without going over the capacity.

6.3 Reductions and Decision Problems

The idea of reductions is that I can convert a problem I do not know how to solve into one that I do know how to solve. If I can do this conversion relatively quickly and the algorithm for the latter problem is fast, now I have a fast way to solve the former problem. Essentially, these reductions show that these problems are only superficially different. In the case of two NP problems, a polynomial runtime is what we consider fast. In this way, if I can find a proper polynomial time reduction and can solve an NP problem in polynomial time, I can solve any of the problems that reduce to it in polynomial time as well! If I can convert problem A to problem B in polynomial time, we say that:

$$A \leq_p B$$

Not only does this tell us that the running time of A is upperbounded by B (so any fast/polynomial algorithm for B gives a fast/polynomial algorithm for A) but it also shows us that the running time of A provides a lowerbound for the running time of B (If A has no fast/polynomial algorithms, neither does B). Rather than work with general problems, many reductions work with **decision problems**. These are problems whose solutions are simple YES or NO answers. Here is how we would interpret the common NP problems as decision problems

IndSet: what is the largest independent set? $\rightarrow$ is there an independent set of size k?

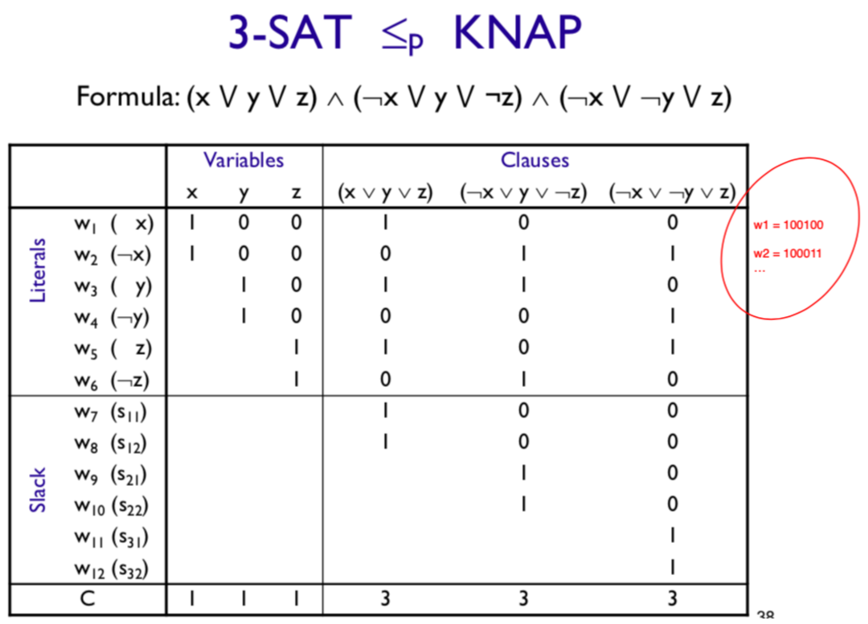
SAT: what is a satisfying assignment to this SAT $\rightarrow$ is there a satisfying assignment?

KNAP: what set of weights has the maximum sum below capacity? $\rightarrow$ is there a subset of weights that sum to exactly C?

To show a reduction from one decision problem to another is valid we must due several things:

1. The runtime of the reduction process is polynomial; it shows how to convert a given instance of problem A to an instance of problem B
2. If I have a valid solution to problem A, then the reduction specifies a valid solution to problem B
3. If I have a valid solution to B, then the reduction specifies a valid solution to problem A

NOTE: A solution to either problem is NOT required to run the reduction. We can convert problem A to problem B without knowing solutions to either of them.



3-SAT $\leq_p$ KNAP

Formula: $(x \lor y \lor z) \land (\neg x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z)$

		Variables			Clauses		
		x	y	z	$(x \lor y \lor z)$	$(\neg x \lor y \lor \neg z)$	$(\neg x \lor \neg y \lor z)$
Literals	w_1 (x)	1	0	0	1	0	0
	w_2 ($\neg$x)	1	0	0	0	1	1
	w_3 (y)		1	0	1	1	0
	w_4 ($\neg$y)		1	0	0	0	1
	w_5 (z)			1	1	0	1
	w_6 ($\neg$z)			1	0	1	0
Slack	w_7 (s_{11})				1	0	0
	w_8 (s_{12})				1	0	0
	w_9 (s_{21})					1	0
	w_{10} (s_{22})					1	0
	w_{11} (s_{31})						1
	w_{12} (s_{32})						1
	C	1	1	1	3	3	3

w1 = 100100

w2 = 100011

...

Figure 1: KNAP reduction of SAT problem

6.3.1 3SAT $\leq_p$ k-IndSet

Given a 3SAT problem with k clauses generate a graph G in the following way. For each literal (even duplicates) in every clause, make a vertex corresponding to that literal. Connect two vertices if their corresponding literals are in the same clause OR are negations of each other (i.e. x_1 and $\neg x_1$). The associated IndSet problem will be: does there exist an independent set of size k?

6.3.2 3SAT $\leq_p$ KNAP

Given a 3SAT problem with k clauses and variables $\{x_1, x_2, ..., x_p\}$ that make up the literals of those clauses (i..e x_i or $\neg x_i$ are the literals in the clauses for some $1 \leq i \leq p$) then we can define a KNAP problem in the following way. Define a table as above. Each clause gets two slack weights with a one in the clauses column and zeros elsewhere. For each varible x_i, place a one under each clause column if that literal is in that clause and a zero otherwise. Set the capacity C to be one in the variable columns (this guarantees that exactly one of x_i and $\neg x_i$ is chosen) and a three in the clause columns (since there are only two slacks, this guarantees at least one literal is true in each clause).

6.4 NP Completeness

A problem is NP complete if it is in NP and every problem in NP is reducible to it. Thus, finding a fast algorithm to an NP complete problem finds a fast solution to all NP problems. NOTE: once we prove one problem is NP-complete, any problem that we can reduce that NP complete problem to must also be NP complete.

7 Other Details

7.1 Important Data Structures

7.1.1 Graph Adjacency Matrix

Store a matrix where of size $|V| \times |V|$ where entry (i, j) is the number of edges going from v_i to v_j (or can represent the weight of the edge going between the two). Note, for undirected graphs this will naturally be a symmetric matrix.

Pros

Cons

7.1.2 Graph Adjacency List

7.1.3 Priority Queues

7.2 Where Greed Fails

Coin-changing example, weighted interval scheduling,

7.3 Proof Techniques

7.3.1 Greedy Stays Ahead

7.4 Exchange Arguments

inversions and whatnot

7.5 Complexity Analysis

Def: Big-O runtime

Master Theorem

7.6 Random Stuff

$$\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}$$

divide and conquer – master recurrence

$T(n) = aT(n/b)+cn^k$ for $n > b$ then

$a > b^k \Rightarrow T(n) = \Theta(n^{\log_b a})$	[many subprobs → leaves dominate]
$a < b^k \Rightarrow T(n) = \Theta(n^k)$	[few subprobs → top level dominates]
$a = b^k \Rightarrow T(n) = \Theta(n^k \log n)$	[balanced → all $\log n$ levels contribute]

Fine print:
$T(1) = d$; $a \geq 1$; $b > 1$; $c, d, k \geq 0$; $n = b^t$ for some $t > 0$;
a, b, k, t integers. True even if it is $\lceil n/b \rceil$ instead of n/b.