


Info

14 October 2020 10:14

Algorithm Design by J Kleinberg and E Tardos

Algorithms by Dasgupta

Assessment

- Coursework 1 (5%): due in week 5, peer assessed
 - Coursework 2 (5%): due in week 7
 - **Class test (20%): 1h in week 9**
 - Exam (70%): 2h, early in term 3
- 

[Module page](#)

[Lectures](#)

[Moodle](#)

[Book](#)

Greedy Algorithms

21 October 2020 14:15

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Interval Partitioning Problem

21 October 2020 14:04

Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

When jobs overlap, you can say that they are incompatible.

The inputs are the same as to the scheduling problem – sequences of pairs of (start time, finish time)

The desired outputs: we want to allocate lectures to classrooms such that they don't overlap, but now we want to reduce the number of classrooms.

This is subject to the constraint that the lectures allocated to the classroom do not overlap.

A greedy algorithm to solve the interval partitioning problem

Earliest-start-time-first algorithm

- Loop over the lectures, ordered by start time
- If a classroom is available, i.e. there is no incompatibility issue, then allocate that lecture to that classroom.
- If all classrooms are incompatible, then just add a new classroom and increment the classroom counter.

We can always produce an allocation of lectures to classrooms that uses the minimum number of classrooms possible.

We must first study the depth of a set of open intervals – the maximum number of intervals that contain any given point.

So at any point in time, some number between 0 and the max number of intervals, will contain that point in time.

This is the depth of the set.

- The depth is the lower bound on the number of classrooms needed.
- The depth is also the upper bound on the number of classrooms needed.

Thus, if we can prove that the earliest-start-time-first algorithm always produces an answer that equals the depth of the set, then we know it always produces an optimal solution.

Proof that the algorithm produces an optimal solution

This is a structural proof – we use information about the structure of the problem to show that it produces an optimal solution

Theorem. Earliest-start-time-first algorithm is optimal.

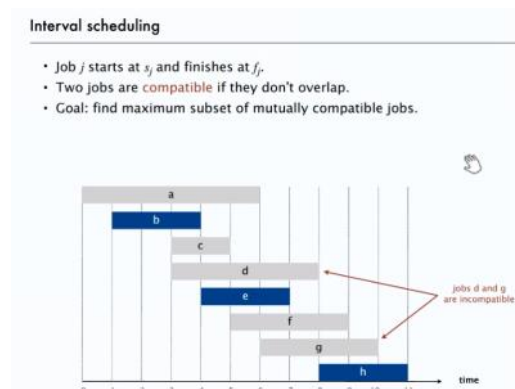
Pf.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j , that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these d lectures each end after s_j .
- Since we sorted by start time, each of these incompatible lectures start no later than s_j .
- Thus, we have d lectures overlapping at time $s_j + \epsilon$.
- Key observation \Rightarrow all schedules use $\geq d$ classrooms. \blacksquare

Interval Scheduling Problem

14 October 2020 10:33

Interval scheduling



We have a set of requests corresponding to a start time and a finish time. A subset of requests is compatible if there is no overlap in time. Our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called optimal.

Designing a Greedy Algorithm

The idea is to use a simple rule to accept a first request, and then reject all subsequent requests that are not compatible with this first request. We then select the first compatible request and reject everything after that, that is not compatible, and so on. We continue until we run out of requests.

The challenge in creating a good greedy algorithm is finding a good simple rule to use for the initial selection

Some rules we could use:

1. Select the available request that starts earliest – the one with the minimal start time. This does not yield an optimal solution. If the earliest request l is for a very long interval, the by accepting request l we may have to reject a lot of requests for shorter time intervals. Since our goal is to satisfy the maximum number of requests possible, we will end up with a suboptimal solution.
2. We could therefore start out by accepting the request that requires the least amount of time first. This is better than the previous rule, but it can still produce suboptimal schedules e.g. if a short request blocks two longer request that could run together
3. Alternatively, we could count the number of requests that are not compatible with each request – e.g. count the conflicts. This is a better solution than the previous two, but is still not optimal.

Earliest Finish Time First

A greedy rule that does lead to the optimal solution is based on a fourth idea – we should accept the first request that finish first – that is, the request i for which $f(i)$ is as small as possible.

This reflects a natural idea – we ensure that our resource becomes free as soon as possible while still satisfying one request.

Interval scheduling: earliest-finish-time-first algorithm

```
EARLIEST-FINISH-TIME-FIRST ( $n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$ )
  SORT jobs by finish times and renumber so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
   $S \leftarrow \emptyset$ .  $\leftarrow$  set of jobs selected
  FOR  $j = 1$  TO  $n$ 
    IF (job  $j$  is compatible with  $S$ )
       $S \leftarrow S \cup \{j\}$ .
  RETURN  $S$ .
```

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

- Keep track of job j^* that was added last to S .
- Job j is compatible with S iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.

Why Is EFTF a Good Solution?

You can prove by contradiction by assuming greedy is not optimal

When you reach a contradiction, you show that the earliest finish time first algorithm is indeed optimal

If the algorithm is not optimal then an optimal set must have more requests

Proof. We will prove the statement by contradiction. If A is not optimal, then an optimal set \mathcal{O} must have more requests, that is, we must have $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request j_{k+1} in \mathcal{O} . This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests i_1, \dots, i_k , the set of possible requests R still contains j_{k+1} . But the greedy algorithm stops with request i_k , and it is only supposed to stop when R is empty—a contradiction. ■

Assume there is a set with a larger size – an optimal solution

Align the jobs with each other

We say that the size of the new optimal solution is larger than our solution, since it is optimal

Then we have to prove there is a job in the new solution set that is not compatible

You can then prove that there is a job that is not in the greedy algorithm solution but which is compatible with the greedy algorithm

The greedy algorithm contains

Scheduling To Minimize Lateness

21 October 2020 14:46

<https://echo360.org.uk/lesson/29b902d7-bdf3-4a6a-8237-41f9ea4481a7/classroom>

The Problem

A scheduling problem. We have a single resource and a set of n requests to use the resource for an interval of time.

Assume the resource is available at starting time s .

Each request is now more flexible than previously. Instead of a start time and a finish time, the request just has a deadline, and it requires a contiguous time interval of length t , but it is willing to be scheduled at any time before the deadline.

Each accepted request must be assigned an interval of time of length t , and the different requests must now be assigned nonoverlapping intervals.

Goal: Schedule all jobs to minimize maximum lateness

- Inputs: sequence of pairs of jobs
- Output sequence of numbers
- We want to provide start times such that no jobs overlap
- We want to minimise maximum lateness L

Designing the Algorithm

What would a greedy algorithm for this problem look like?

- Schedule in order of increasing length t , so as to get the short jobs out of the way quickly – this ignores deadlines
- We could schedule in order of increasing slack time $d - t$. However, this fails too which you can see if you imagine a job with $t = 10$ and $d = 10$. Thus slack is 0 so it would go first regardless of a job $t = 1$, $d = 2$ that would result in a lateness of 9.

The most efficient way of doing this is with an earliest-deadline-first greedy algorithm

Proof: <https://echo360.org.uk/lesson/b581324b-6cb9-49c5-a7a4-d21720ba25a9/classroom>

Theorem. The earliest-deadline-first schedule S is optimal.

Pf. [by contradiction]

optimal schedule can
have inversions

Define S^* to be an optimal schedule with the fewest inversions.

- Can assume S^* has no idle time. ← Observation 1
- Case 1. [S^* has no inversions] Then $S = S^*$. ← Observation 3
- Case 2. [S^* has an inversion]
 - let $i-j$ be an adjacent inversion ← Observation 4
 - exchanging jobs i and j decreases the number of inversions by 1 without increasing the max lateness ← key claim
 - contradicts "fewest inversions" part of the definition of S^* ✱

Proving Greedy Algorithms

21 October 2020 22:37

Proof Techniques

Greedy algorithm stays ahead

Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm

Structural

Discord a simple structural bound asserting that every possible solution must have a certain value. Then show that your algorithm achieves this bound

Exchange argument

Gradually transform any solution to the one found by the greedy algorithm without hurting its quality

Shortest Path Problems

04 November 2020 14:23

Algorithms for shortest paths work on weighted paths

Graphs with a set of vertices and edge weights.

There are a variety of applications for shortest path algorithms

- Map routing
- Robot navigation
- Texture mapping
- Typesetting in latex
- Urban traffic planning
- Seam carving

Single-Pair Shortest Path

The shortest path from a start node to a goal node

Single-Source Shortest Path Problems

The shortest directed path from s to every node

Dijkstra's Algorithm

10 November 2020 14:35

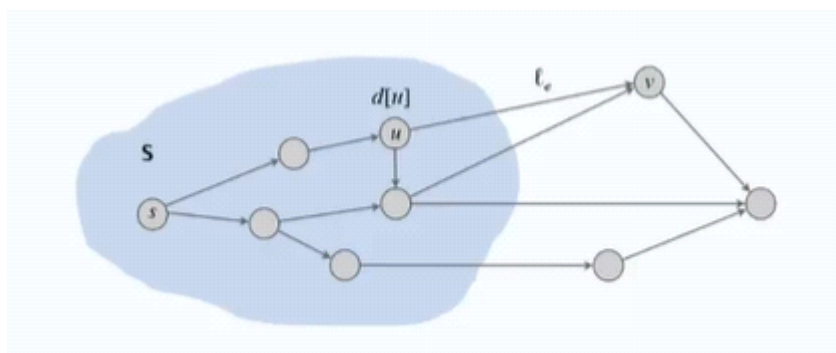
For **single-source** shortest paths problem

- Mark all nodes as unvisited
- Set the distance to s to 0, and ∞ to all other nodes. Set s as current node
- For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the new distance to the current assigned value and assign the smaller one
- When we are done considering all the unvisited neighbours of the current node, mark the current node as visited. A visited node will never be visited again
- If the destination node has been marked visited or if the smallest tentative distance among the nodes in the unvisited set is infinity,

Greedy Approach

- maintain a **set of explored nodes S** for which algorithm has determined $d[u]$ = length of a shortest path
- Initialise the set of explored nodes as s , with $d[s] = 0$
- Repeatedly **choose unexplored node v that are not in S , which minimizes the cost.**

The length of a shortest path from s to some node u in explored part S , followed by a single edge $e = (u, v)$



At each iteration of the algorithm we consider all vertices v which are not in S . We want to pick the v that minimises some parameter.

The param is defined to be the minimum over all edges e , that enter vertex v (note we talk about directed graph), but we only consider edges incoming to v , whose source is in S .

So we consider all edges that come from the cloud S , into the vertex V . We consider the smallest value for the expression $d[u] + \text{length}(e)$ (length of edge from u to v)

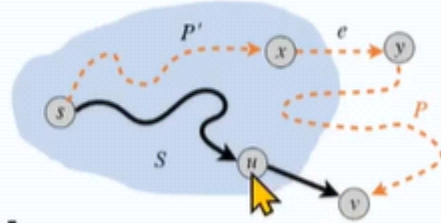
Note that $d[u]$ is simple the shortest path of the path from s to u .

Proof of Correctness

- **Invariant:** For each node u in S , $d[u]$ = length of a shortest path from s to u
- Proved by induction on size of S
- **Base case** $|S| = 1$ is easy since $S = \{s\}$ and $d[s] = 0$

Inductive hypothesis: Assume true for $|S| \geq 1$.

- Let v be next node added to S , and let (u, v) be the final edge.
- A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider **any** other $s \rightsquigarrow v$ path P . We show that it is no shorter than $\pi(v)$.
- Let $e = (x, y)$ be the first edge in P that leaves S , and let P' be the subpath from s to x .
- The length of P is already $\geq \pi(v)$ as soon as it reaches y :



$$\ell(P) \geq \ell(P') + \ell_e \geq d[x] + \ell_e \geq \pi(y) \geq \pi(v) \quad \blacksquare$$

\uparrow non-negative lengths \uparrow inductive hypothesis \uparrow definition of $\pi(y)$ \uparrow Dijkstra chose v instead of y

Efficient Implementation

For each unexplored node v not in S , explicitly maintain $\pi[v]$ instead of computing directly from definition

We maintain this $\pi[v]$ as $d[u] + \text{Length}(e)$ where e is minimum of all edges entering v from u where u is in S

Specifically, suppose u is added to S , and there is an edge $e=(u,v)$ leaving u . Then, it suffices to update $\pi(v) \leftarrow \min\{\pi(v), \pi(u) + \ell_e\}$

Remember that for all u in S , $\pi(u) = d(u) = \text{length of shortest path from } s \text{ to } u$

Optimisation: use a min-oriented priority queue to choose an unexplored node that minimises $\pi(v)$

- Algorithm maintains $\pi(v)$ for each node v
- Priority queue stores unexplored nodes using π as priorities
- Once u is deleted from PQ, $\pi(u) = \text{length of shortest path from } s \text{ to } u$

DIJKSTRA (V, E, ℓ, s)

FOREACH $v \neq s$: $\pi[v] \leftarrow \infty, \text{pred}[v] \leftarrow \text{null}; \pi[s] \leftarrow 0$.

Create an empty priority queue pq .

FOREACH $v \in V$: **INSERT**($pq, v, \pi[v]$).

WHILE (**IS-NOT-EMPTY**(pq))

$u \leftarrow \text{DEL-MIN}(pq)$.

 FOREACH edge $e = (u, v) \in E$ leaving u :

IF ($\pi[v] > \pi[u] + \ell_e$)

DECREASE-KEY($pq, v, \pi[u] + \ell_e$).

$\pi[v] \leftarrow \pi[u] + \ell_e; \text{pred}[v] \leftarrow e$.



Dijkstra's algorithm: which priority queue?

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- Array implementation optimal for dense graphs. $\leftarrow \Theta(n^2)$ edges
- Binary heap much faster for sparse graphs. $\leftarrow \Theta(n)$ edges
- 4-way heap worth the trouble in performance-critical situations.

priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
node-indexed array ($A[i]$ = priority of i)	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman-Tarjan 1984)	$O(1)$	$O(\log n)^\dagger$	$O(1)^\dagger$	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

assumes $m \geq n$ † amortized

13

Minimum Spanning Trees

10 November 2020 15:06

Cycles

- A path is a sequence of edges which connects a sequence of nodes
- A cycle is a path with no repeated nodes or edges other than the starting and ending nodes

Cuts & Cutsets

- A cut is an arbitrary partition of the nodes into two nonempty subsets S and $V - S$
- A cutset of a cut S is the set of edges with exactly one endpoint in S – so the edges have one end on either side of the cut.

Cut-Cycle Intersection

Proposition – a cycle and a cutset interact in an even number of edges

Consider a cycle c . If c is outside or entirely inside s , then the cutset is empty

If there is some edge in the cycle, that goes from the cut to outside the cut, then there must be an edge that comes back into the cut.

Thus these edges come in pairs

Spanning Tree Definition

A spanning tree of a graph is a subgraph that is acyclic and connected.

1. H is a spanning tree of G
2. H is acyclic and connected
3. H is connected and has $V - 1$ edges
4. H is acyclic and has $V - 1$ edges
5. H is minimally connected: removal of any edge disconnects it
6. H is maximally acyclic: addition of any edges creates a cycle

Minimum Spanning Tree

Given a connected undirected graph with edge costs, an MST is a spanning tree of G such that the sum of the edge costs is minimized

Cayley's Theorem – the complete graph on n nodes has n^{n-2} – thus we can't solve by brute force

Applications

- Network design – communication, computer, road etc
- Dithering
- Cluster analysis
- Max bottleneck paths
- Real-time face verification
- Finding road networks in satellite imagery
- Approximation algorithms for NP-hard problems

A Generic Greedy Algorithm for the MST Problem

Fundamental cycle property

- Results from observing that a ST is maximally acyclic subgraph of a spanning graph
- If we add an edge to a spanning tree, we create a unique cycle. If we then remove another edge, we create a new spanning tree

If the cost of the edge we removed is less than the new edge added, the original tree was not an MST

Fundamental Cutset

- For any spanning tree, if we remove an edge we get two connected components
- Let D denote the corresponding cutset created
- A spanning tree is a minimally

Red Rule

- Let C be a cycle with no red edges
- Select an uncoloured edge of C of max cost and color it red

Blue Rule

- Let D be a cutset with no blue edges
- Select an uncoloured edge in D of min cost and colour it blue

Greedy algorithm

- Apply the red and blue rules non-deterministically until all edges are coloured. The blue edges form an MST
- Note: we can stop once $n-1$ edges are coloured blue

Correctness of the Algorithm

Colour invariant: there exists an MST containing every blue edge and no red edge

Proof by induction on number of iterations

Base case – no edges coloured \Rightarrow every MST satisfies invariant

Induction step – blue rule

- Suppose colour invariant true before blue rule
- Let D be chosen cutset, and let f be edge coloured blue
- If f is in the MST T^* , then T^* still satisfies invariant
- Otherwise, consider fundamental cycle C by adding f to T^*
- Let e in C , be another edge in D
- e is uncoloured and the cost is $\geq f$, since e is in T^* $\Rightarrow e$ is not red
- Blue rule $\Rightarrow e$ not blue and cost $e \geq f$
- Thus, T^* and f minus e satisfies invariant

Now we will argue that there is always an edge we can colour red or blue

The greedy algorithm terminates – blue edges form an MST

We need to show that either the red or blue rule applies

Suppose edge e is left uncoloured

Blue edges form a forest

Case 1 – both endpoints of e are in the same blue tree

- apply red rule to cycle formed by adding e to blue forest

Case 2 – both endpoints of e are in different blue trees

- Apply blue rule to cutset induced by either of two blue trees

Prims Algorithm for the MST Problem

Prims is similar to Dijkstra's

Initialise $S = \{ s \}$ for any node s , $T =$ empty set

Repeat $n - 1$ times

- Add to T a min-cost edge with exactly one endpoint in S
- Add the other endpoint to S

Theorem – prim's algorithm computes an MST

Special case of greedy algorithm – blue rule repeatedly applied to S

We can implement Prim's in a similar manner to Dijkstra's.

We hold a set S of nodes and say every node is infinitely far from the start node s

We have a priority queue

We iterate over the vertices connected to s and add the cheapest known edge to the cloud

We then iterate over the priority queue

Kruskal's Algorithm

Consider edges in ascending order of cost

- Add to tree unless it would create a cycle

Kruskal's algorithm computes an MST

Case 1 – both endpoints of e are in the same blue tree

- Colour e red by applying red rule to unique cycle

Case 2 – both endpoints of e are in different blue trees

- Colour e blue by applying blue rule to cutset defined by either tree

Implementing Kruskal's

Can be implemented in $O(m \log n)$ time

- Sort edges by cost
- Use union-find data structure to dynamically maintain connected components

KRUSKAL (V, E, c)

Sort m edges by cost and renumber so that $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$.

$T \leftarrow \emptyset$.

FOREACH $v \in V$: **MAKE-SET**(v).

FOR $i = 1$ **TO** m

$(u, v) \leftarrow e_i$.

IF (**FIND-SET**(u) \neq **FIND-SET**(v)) \leftarrow are u and v in same component?

$T \leftarrow T \cup \{e_i\}$.

UNION(u, v). \leftarrow make u and v in same component

RETURN T .

Reverse-Delete Algorithm

Start with all edges in T and consider them in descending order of cost

Delete edge from T unless it would disconnect T

The reverse-delete algorithm computes an MST

Divide & Conquer

05 January 2021 17:20

Refers to a class of algorithmic techniques in which one breaks the input down into several parts and solves each part of the problem recursively

When we combine these solutions to the subproblems in each part, we have an overall solution.

When we use divide and conquer, generally we're trying to move from polynomial time to a lower polynomial, rather than exponential time to any kind of polynomial.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.

- Divide problem of size n into **two** subproblems of size $n/2$. $\leftarrow O(n)$ time
- Solve (conquer) two subproblems recursively.
- Combine two solutions into overall solution. $\leftarrow O(n)$ time

Consequence.

- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $O(n \log n)$.



attributed to Julius Caesar

2

Mergesort

20 November 2020 09:59

"Divide the input into two pieces of equal size; solve the two subproblems on these pieces separately by recursion and then combine the two results into an overall solution, spending only linear time for the initial division and final recombining. "

We also need a base case for the recursion. In the merge sort case, we can assume that once the input has been reduced to size 2, we can stop the recursion and sort the two elements by simply comparing them.

- Let us say $T(n)$ is the worst case running time for the algorithm.
- Supposing the input n is even, the algorithm spends $O(n)$ time to divide the input into two pieces of size $n/2$
- Then, the algorithm spends time $T(n/2)$ to solve each one since.
- Finally, it spends $O(n)$ combining the solutions from the two recursive calls.
- Thus the running time $T(n)$ satisfies the following recurrence relation:

(5.1) For some constant c ,

$$T(n) \leq 2T(n/2) + cn$$

when $n > 2$, and

$$T(2) \leq c.$$

To get an asymptotic running time we need to solve the relation so that T only appears on the left side.

Solving this recurrence is a standard task, but it is very useful to understand the process of solving recurrences and to recognize which recurrences lead to good running time, since the design of an efficient divide and conquer algorithm is heavily intertwined with an understanding of how a recurrence relation determines a running time.

Approaches to Solving Recurrences

1. The most natural way to solve a recurrence is to unroll the recursion. We account for the running time at a few levels, find a common pattern and then deduce the running time by finding out where the recurrence bottoms out.
2. A second way is to start with a guess for a solution, substitute it into the recurrence relation, and check it works. Formally, this is induction on n . This is useful when we don't have exact values for all the parameters.

Unrolling

At the first level, we have a single problem of size n , taking cn plus time in subsequent levels
At the next level, we have two problems each of size $n/2$, taking $2c/n$ taking cn time to complete, again plus time at subsequent levels

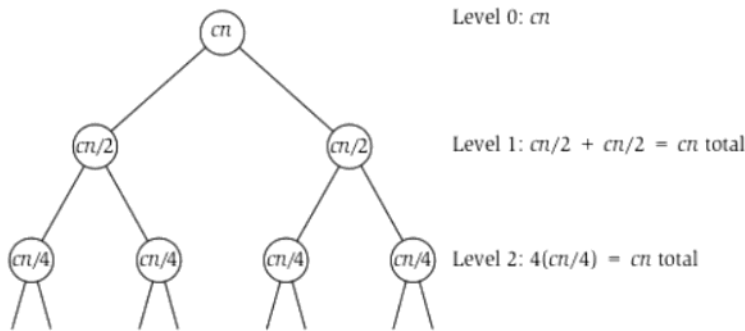


Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

We have 2^j nodes at level j of the recursion. The size of each node has shrunk by a factor of 2^j times and so each has a size of $n/2^j$, and hence each takes time at most $cn/2^j$. Thus level j contributes a total of at most $2^j(cn/2^j) = cn$ to the total running time. When we sum all the levels, we see there are $\log_2(n)$. Thus, the running time is $O(n \log n)$.

Proving $T(n) = n \log n$

Proof by induction

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

← assuming n is a power of 2

Pf. [by induction on n]

- Base case: when $n = 1$, $T(1) = 0 = n \log_2 n$.
- Inductive hypothesis: assume $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned}
 T(2n) & \stackrel{\text{recurrence}}{=} 2T(n) + 2n \\
 & \stackrel{\text{inductive hypothesis}}{=} 2n \log_2 n + 2n \\
 & = 2n (\log_2 (2n) - 1) + 2n \\
 & = 2n \log_2 (2n). \quad \blacksquare
 \end{aligned}$$

11

Substituting a Solution

1. We have a guess for the running time we want to verify – plug it into the recurrences
2. Plug in a number like 2, check it works
3. Suppose by induction that it holds, and we want to establish it for $T(n)$
4. We do this by writing the recurrence for $T(n)$ and plugging in the inequality
5. Simplify the resulting expression
- 6.

$$\begin{aligned}
 T(n) & \leq 2T(n/2) + cn \\
 & \leq 2c(n/2) \log_2(n/2) + cn \\
 & = cn[(\log_2 n) - 1] + cn \\
 & = (cn \log_2 n) - cn + cn \\
 & = cn \log_2 n.
 \end{aligned}$$

Master Theorem

20 November 2020 10:57

We need a recipe for solving common divide-and-conquer recurrences

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

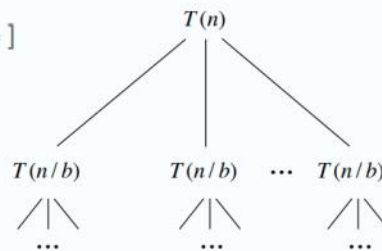
$a \geq 1$ is the number of subproblems

$B \geq 2$ is the factor by which the subproblem size decreases

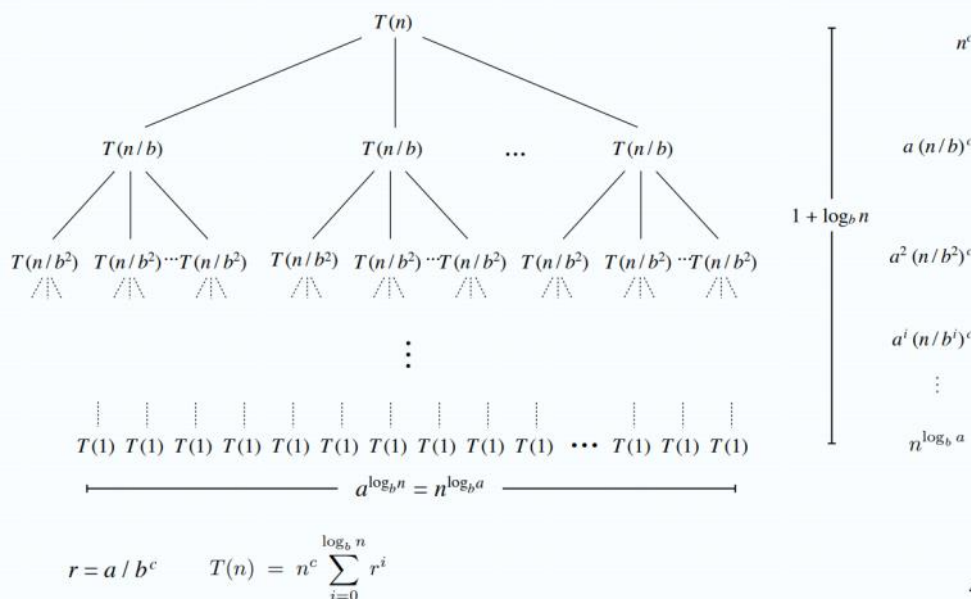
$F(n) \geq 0$ is the work to divide and combine subproblems

Recursion tree. [assuming n is a power of b]

- a = branching factor.
- a^i = number of subproblems at level i .
- $1 + \log_b n$ levels.
- n / b^i = size of subproblem at level i .



Suppose $T(n)$ satisfies $T(n) = a T(n/b) + n^c$ with $T(1) = 1$, for n a power of b .



All the work done by each level is designated by the value on the right

At the bottom, you see the constant r .

The value of $T(n)$ is the sum of $n^c \cdot r^i$ for all the levels (0 to $\log_b(n)$)

Let $r = a / b^c$. Note that $r < 1$ iff $c > \log_b a$.

$$T(n) = n^c \sum_{i=0}^{\log_b n} r^i = \begin{cases} \Theta(n^c) & \text{if } r < 1 & c > \log_b a & \leftarrow \text{cost dominated by cost of root} \\ \Theta(n^c \log n) & \text{if } r = 1 & c = \log_b a & \leftarrow \text{cost evenly distributed in tree} \\ \Theta(n^{\log_b a}) & \text{if } r > 1 & c < \log_b a & \leftarrow \text{cost dominated by cost of leaves} \end{cases}$$

Proving the value of $T(n)$ when $r > 1$ – note r is a/b^c (

Geometric series.

- If $0 < r < 1$, then $1 + r + r^2 + r^3 + \dots + r^k \leq 1 / (1 - r)$.
- If $r = 1$, then $1 + r + r^2 + r^3 + \dots + r^k = k + 1$.
- If $r > 1$, then $1 + r + r^2 + r^3 + \dots + r^k = (r^{k+1} - 1) / (r - 1)$.

Gaps in master theorem.

- Number of subproblems is not a constant.

$$T(n) = \textcircled{n} T(n/2) + n^2$$

- Number of subproblems is less than 1.

$$T(n) = \left(\frac{1}{2}\right) T(n/2) + n^2$$

- Work to divide and combine subproblems is not $\Theta(n^c)$.

$$T(n) = 2T(n/2) + \textcircled{n \log n}$$

When the number of subproblems is not a constant, e.g. is some function of a variable n

Ex 3. $T(n) = 48 T(\lfloor n / 4 \rfloor) + n^3$.

- $a = 48, b = 4, c = 3, \log_b a > 2.79$.
- $T(n) = \Theta(n^3)$.

Integer Multiplication Problem

04 January 2021 10:35

Addition – given two n -bit integers a and b , compute $a + b$

Subtraction – given two n -bit integers a and b , compute $a - b$

Multiplication – given two n -bit integers, a and b , compute $a \times b$ - $O(n^2)$ operations

Conjecture – Grade-school algorithm is optimal

Theorem [Karatsuba 1960] – conjecture is false

This is the basic premise for the algorithm:

To multiply two n -bit integers x and y :

- Divide x and y into low- and high-order bits.
- Multiply **four** $\frac{1}{2}n$ -bit integers, recursively.
- Add and shift to obtain result.

$$m = \lceil n / 2 \rceil$$

$$a = \lfloor x / 2^m \rfloor \quad b = x \bmod 2^m$$

$$c = \lfloor y / 2^m \rfloor \quad d = y \bmod 2^m$$

← use bit-shifting
to compute 4 terms

$$xy = (2^m a + b)(2^m c + d) = \underbrace{2^{2m} ac}_{\textcircled{1}} + \underbrace{2^m (bc + ad)}_{\textcircled{2} \textcircled{3}} + \underbrace{bd}_{\textcircled{4}}$$

$$2^m \cdot a = \begin{array}{r} 10000000 \\ \underline{1101} \\ a \end{array}$$

$$x = 2^m \cdot a + b \quad y = 2^m \cdot c + d$$

Ex. $x = \underbrace{1000}_{a} \underbrace{1101}_{b} \quad y = \underbrace{1110}_{c} \underbrace{0001}_{d}$

I.e, we take two numbers and divide them into equal sections of m bits.
Then, the product of xy :

$$Xy = (2^m a + b)(2^m c + d)$$

We can use bit shifting to compute the 4 terms which in binary is very cheap. So we can multiply/divide by 2 in constant time





This forms the basis:

MULTIPLY(x, y, n)

IF ($n = 1$)

RETURN $x \times y$.

ELSE

$m \leftarrow \lceil n / 2 \rceil$ 
 $a \leftarrow \lfloor x / 2^m \rfloor; b \leftarrow x \bmod 2^m$.  $\Theta(n)$
 $c \leftarrow \lfloor y / 2^m \rfloor; d \leftarrow y \bmod 2^m$.
 $e \leftarrow \text{MULTIPLY}(a, c, m)$.
 $f \leftarrow \text{MULTIPLY}(b, d, m)$.  $4 T(\lceil n / 2 \rceil)$
 $g \leftarrow \text{MULTIPLY}(b, c, m)$.
 $h \leftarrow \text{MULTIPLY}(a, d, m)$.
 RETURN $2^{2m} e + 2^m (g + h) + f$.  $\Theta(n)$

A = 4

B = 2

C = 1

We compared $c = 1$ to $\log_b(a) = 2$

So $\log_b(a) > c$ so $t(n) = O(n^2)$

So the above algorithm doesn't give us any performance increase

Karatsuba

But, with one minor alteration:

Karatsuba trick

To multiply two n -bit integers x and y :


- Divide x and y into low- and high-order bits.
- To compute middle term $bc + ad$, use identity:

$$bc + ad = ac + bd - (a - b)(c - d)$$


- Multiply only **three** $\frac{1}{2}n$ -bit integers, recursively.

$$\begin{aligned}
 m &= \lceil n / 2 \rceil \\
 a &= \lfloor x / 2^m \rfloor \quad b = x \bmod 2^m \\
 c &= \lfloor y / 2^m \rfloor \quad d = y \bmod 2^m
 \end{aligned}$$

$x = \underbrace{1000}_a \underbrace{1101}_b$
 $y = \underbrace{1110}_c \underbrace{0001}_d$

 middle term

$$\begin{aligned}
 \underline{xy} &= (2^m a + b)(2^m c + d) = \underbrace{2^{2m}(ac)}_{\text{1}} + \underbrace{2^m(bc + ad)}_{\text{1 3}} + \underbrace{bd}_{\text{2}} \\
 &= 2^{2m} ac + 2^m (ac + bd - (a - b)(c - d)) + bd
 \end{aligned}$$



1
1
3
2
3

Most of the computation goes into the circled multiplications – this is where we recurse.

If we look at $bc + ad$, we see that we can construct it with the existing values of ac and bd that we've done already.

$$bc + ad = ac + bd - (a - b)(c - d)$$

Thus we only have 3 multiplications of $\frac{1}{2}$ n -bit integers recursively, rather than 4.

Karatsuba analysis

Proposition. Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two n -bit integers.

Pf. Apply Case 1 of the master theorem to the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} a &= 3 \\ b &= 2 \\ c &= 1 \end{aligned}$$

$$\Rightarrow T(n) = \Theta(n^{\log_2 3}) = O(n^{1.585})$$

$$\log_b a$$

Practice.

- Use base 32 or 64 (instead of base 2).
- Faster than grade-school algorithm for about 320–640 bits.



Dynamic Programming

19 November 2020 09:38

Greedy algorithms are a natural approach but often times there will be no natural greedy algorithm that works for a problem

Another approach is divide and conquer, but these tend to not be strong enough to reduce exponential brute-force search down to polynomial time. Instead, these algorithms are good at reducing polynomial time algorithms down to a faster running time.

6.1 Dynamic Programming

Implicitly explore the space of all possible solutions by decomposing the problem into a series of subproblems and then building up correct solutions to larger and larger subproblems.

Weighted Interval Scheduling Problem

Similar to the interval scheduling problem except we now have weighted intervals and we want to pick a non-overlapping set of intervals with the maximum weight

We need to sort our intervals into an order of non-decreasing finish time

We have interval I that comes before a request if $I < j$. This will be the natural left-to-right order in which we'll consider intervals

We define $p(j)$ for an interval to be the largest index $I < j$ such that intervals I and j are disjoint. In other words $p(j)$ indicates the largest compatible interval with j .

We can already make the claim that in a solution to this problem, either the last interval will be in the solution, or it will not be.

Case 1: The last interval is in the solution.

In this case we can be sure that no interval after $p(n)$ will be in the solution.

Further, if n is in the solution, then the solution will contain an optimal solution to the problem containing intervals $\{1 \dots p(n)\}$

$$\text{OPT}(n) = v_n + \text{OPT}(p(j))$$

Case 2: The last interval is not in the solution

In this case, the optimal solution is simply the same as the one with requests $\{1 \dots n-1\}$

This is by similar reasoning – we're assuming that the solution does not include request n so if it does not choose the optimal set of requests from $\{1 \dots n-1\}$ we could replace it with a better one

$$\text{OPT}(n) = \text{OPT}(n-1)$$

We can call this optimal solution $\text{OPT}(j)$ - we define $\text{OPT}(0)$ as 0.

Therefore, the optimum solution for n intervals is

$$\text{OPT}(n) = \max \{ v_n + \text{OPT}(p(n)), \text{OPT}(n-1) \}$$

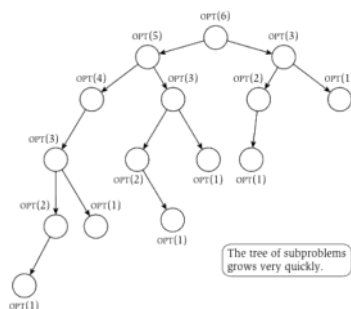
```

Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max( $v_j + \text{Compute-Opt}(p(j))$ ,  $\text{Compute-Opt}(j-1)$ )
  Endif

```

We can work out the correctness of the algorithm by induction on j

1. $\text{OPT}(0) = 0$
2. Take some $j > 0$
3. Suppose by way of induction that $\text{Compute-Opt}(j)$ correctly computes $\text{OPT}(j)$ for all $i < j$
4. By the induction hypothesis, we know that:
 - a. $\text{Compute-Opt}(p(j)) = \text{OPT}(p(j))$ and
 - b. $\text{Compute-Opt}(j-1) = \text{OPT}(j-1)$
 - c. Hence we know that $\text{OPT}(j) = \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) = \text{Compute-Opt}(j)$



The recursive tree grows very quickly. We can use memoization to fix this.

We can store the value of compute-opt for each value we compute and check if it exists before we re-calculate it.

This gives us a linear running time of $O(n)$

6.2 Principles of Dynamic Programming

Memorization or Iteration over Subproblems

We now use the algorithm for the WISP developed in the previous section to summarise the basic principles of dynamic programming

We will also discuss a fundamental concept – iterating over subproblems rather than computing solutions recursively.

In the previous algorithm we developed a recursive algorithm that ran in polynomial time and then optimised it with memorization to create a linear time algorithm

Instead of doing this, we can instead start from the beginning with $M[0]$ and build up. We can clearly see that this runs in linear time as it contains n steps of constant time.

```
Iterative-Compute-Opt
M[0] = 0
For  $j = 1, 2, \dots, n$ 
     $M[j] = \max(v_j + M[p(j)], M[j - 1])$ 
Endfor
```

FIND-SOLUTION(j)

IF ($j = 0$)

RETURN \emptyset .

ELSE IF ($w_j + M[p[j]] > M[j - 1]$)

RETURN $\{j\} \cup \text{FIND-SOLUTION}(p[j])$.

ELSE

RETURN FIND-SOLUTION($j - 1$).

$M[j] = \max \{ M[j - 1], w_j + M[p[j]] \}.$

This has a linear time complexity since we just make 1 recursive call per function call

An Outline of Dynamic Programming

The previous two approaches are conceptually very similar – they both stem from the insight gained by the recursive, brute-force solution.

The second solution uses an iterative building-up of subproblems which is easier to express. But for every problem that we solve using this iterative building-up procedure, we can do something analogous with a memorized recursion.

To create an algorithm based on dynamic programming, we need a collection of subproblems derived from the original problem, that satisfy some properties.

1. There must be only a polynomial number of subproblems
2. The solution to the original problem can be easily computed from the solutions to the subproblems
3. There is a natural ordering on subproblems from smallest to largest, and an easy to compute recurrence that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems

Subset Sums & Knapsacks

29 November 2020 13:38

6.4 Subset Sums and Knapsacks

We will consider a version of the problem where we have a request for a time interval on a resource that have a deadline and a duration, but do not mandate a particular interval in which they need to be done.

We will use dynamic programming, but we will see that the obvious set of subproblems will turn out to be insufficient, so we will need to create a better set of subproblems. This will be done by adding a new variable to the recurrence underlying dynamic programming

Problem

- We have a scheduling problem with requests and a single machine that can process them.
- We are only able to use resources between time 0 and W
- Each request is a job that requires w_i time to process
- If our goal is to process jobs so as to keep the machine as busy as possible up to the cut-off W , which jobs should we choose?

This is a natural special case of the Knapsack Problem, in which we have requests with a value and a weight. We want to maximise the value, while keeping the weight below some threshold W .

Subset Sums show up as subproblems to more complex problems.



Designing the Algorithm

One strategy which worked for us in the WISP was to consider subproblems involving only the first i requests.

However, to find out the $OPT(n)$ value for the Knapsack Problem we not only need the value of $OPT(n-1)$ but we also need to know the best solution we can get using a subset of the first $n-1$ items and total allowed weight $W - w_n$. We are therefore going to use many more subproblems: one for each initial set $\{1, \dots, i\}$ of the items and each possible value for the remaining available weight w

- If $n \notin \mathcal{O}$, then $OPT(n, W) = OPT(n-1, W)$, since we can simply ignore item n .
- If $n \in \mathcal{O}$, then $OPT(n, W) = w_n + OPT(n-1, W - w_n)$, since we now seek to use the remaining capacity of $W - w_n$ in an optimal way across items $1, 2, \dots, n-1$.

When the n th item is too big, i.e. $W < w_n$, we must have $OPT(n, W) = OPT(n-1, W)$. That is, the weight is larger than the weight available to us, so the optimal solution can't contain that request.

With these rules we can derive a recurrence which we can use in an algorithm:

If $w < w_i$ then $\text{OPT}(i, w) = \text{OPT}(i - 1, w)$. Otherwise

$$\text{OPT}(i, w) = \max(\text{OPT}(i - 1, w), w_i + \text{OPT}(i - 1, w - w_i)).$$

Subset-Sum(n, W)

Array $M[0 \dots n, 0 \dots W]$

Initialize $M[0, w] = 0$ for each $w = 0, 1, \dots, W$

For $i = 1, 2, \dots, n$

For $w = 0, \dots, W$

Use the recurrence (6.8) to compute $M[i, w]$

Endfor

Endfor

Return $M[n, W]$

Analysing the Algorithm

We can see using this array the values that are computed by the algorithm for each weight and item:

Knapsack size $W = 6$, items $w_1 = 2, w_2 = 2, w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

As before with the WISP, we build up a table M of solutions and we compute each of the values $M[i, w]$ in $O(1)$ time using the previous values. Thus, the running time is proportional to the number of entries in the table.

The Subset-Sum(n, W) algorithm correctly computes the optimal value of the problem and runs in $O(nW)$ time.

Note that this is not as efficient as the dynamic program for the WISP. It is a polynomial function of n and W . We call such algorithms pseudo-polynomial, and they can be reasonably efficient when the numbers w involved in the input are reasonably small. However, they become less practical when w grows large.

To recover the optimal set of items we can trace back through M as before in the WISP.

The Knapsack Problem

The Knapsack Problem is a bit more complex because we have a value as well as a weight. The goal is to find a maximum subset S of max value subject to the restriction that the total weight of the set should not exceed the maximum weight W .

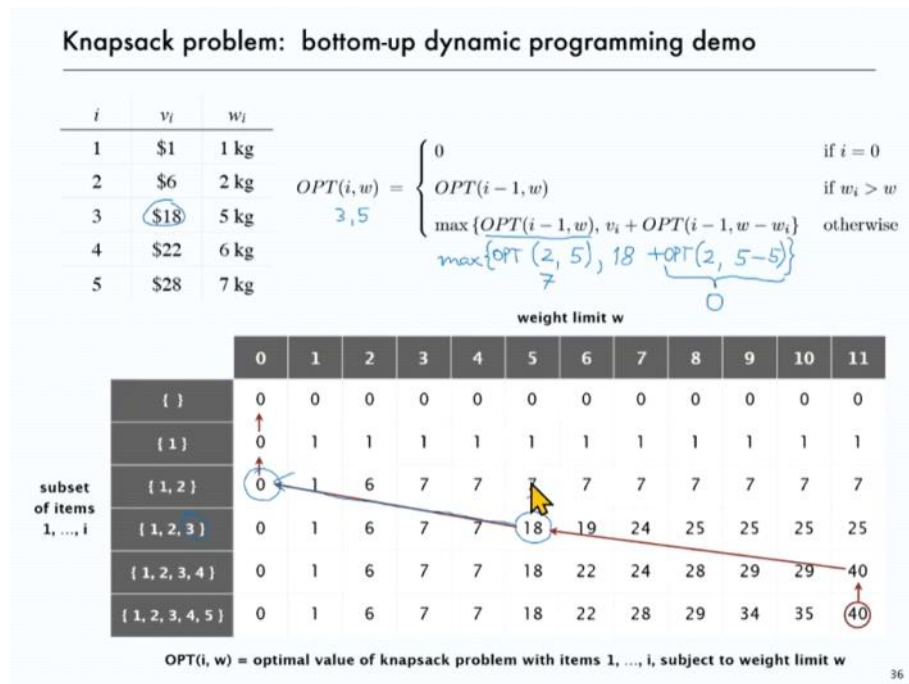
We can extend our dynamic programming algorithm to this more general problem.

If $w < w_i$ then $OPT(i, w) = OPT(i - 1, w)$. Otherwise

$$OPT(i, w) = \max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)).$$

Note the only difference between this recurrence and the one in the previous subset sum problem is the v_i that is added in the calculation of the $OPT(i, w)$ rather than w_i

1. So we have two cases here – either $opt(i, w)$ does not select i (maybe because w of i is too big)
2. Or, $opt(i, w)$ does select i – we collect the value, get a new weight limit by doing $W - w_i$, then select the best of $\{1, 2 \dots i - 1\}$ subject to the new weight limit



Running time

The DP algorithm solves the knapsack problem with n items and a maximum weight w in $O(nW)$ time and $O(nW)$ space

- Takes $O(1)$ time per table entry
- There are $O(nW)$ table entries
- After computing optimal values can trace back to find solution

Sequence Alignment

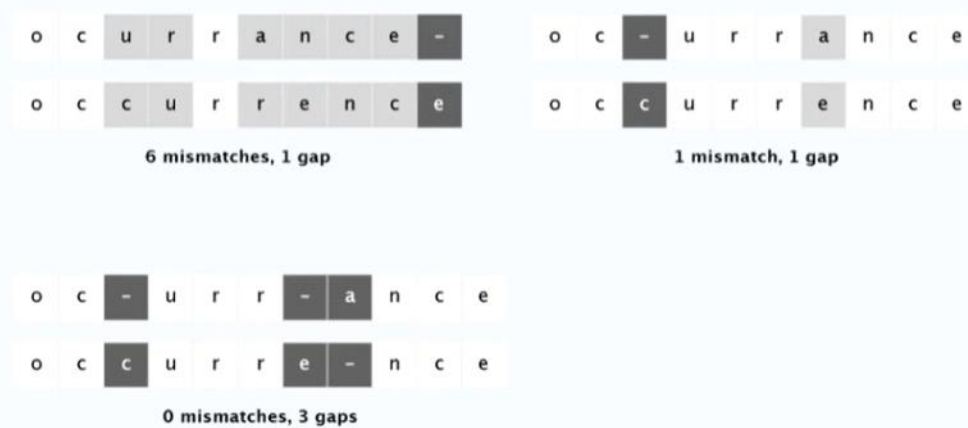
29 November 2020 13:06

An alignment M is a set of ordered pairs such that each character appears in at most one pair and there are no crossings

String similarity

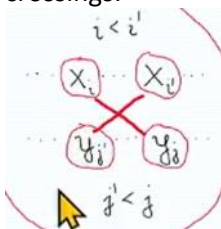
Q. How similar are two strings?

Ex. occurrence and occurrence.



Goal: given two strings, find the minimum cost alignment

An alignment M is a set of ordered pairs such that each character appears in at most one pair and no crossings:

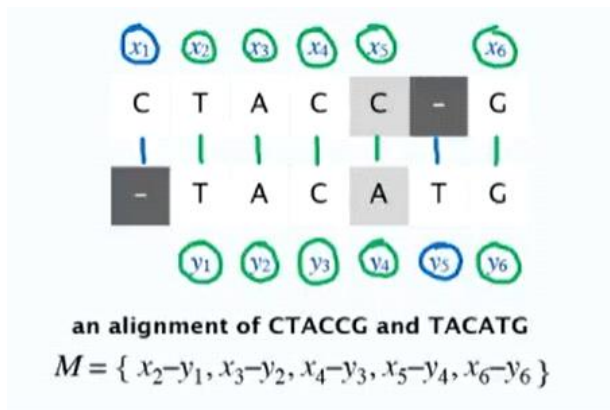


The cost of an alignment M is

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Alpha is the cost of a mismatch between x and y , and delta is the cost of an unmatched character

So the cost is the sum of the mismatch costs, plus the unmatched (gapped) letters from x , plus the unmatched letters from y .



x1 and y5 are matched to gaps.

Problem Structure

We need a small set of subproblems to solve. There are 3, which are as follows:

$OPT(i, j)$

Sequence alignment: problem structure

$x_1 x_2 \dots x_i$
 $y_1 y_2 \dots y_j$

Def. $OPT(i, j)$ = min cost of aligning prefix strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

Goal. $OPT(m, n)$.

Case 1. $OPT(i, j)$ matches $x_i - y_j$.
 Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$.

Case 2a. $OPT(i, j)$ leaves x_i unmatched.
 Pay gap for x_i + min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$.

Case 2b. $OPT(i, j)$ leaves y_j unmatched.
 Pay gap for y_j + min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$.

Bellman equation.

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$

$OPT(i-1, j-1)$ $\alpha_{x_i y_j}$
 $x_1 x_2 \dots x_{i-1}$ x_i
 $y_1 y_2 \dots y_{j-1}$ y_j
 $OPT(i-1, j)$ δ
 $x_1 x_2 \dots x_{i-1}$ x_i
 $y_1 y_2 \dots y_j$ y_j
 $OPT(i, j-1)$ δ
 $x_1 x_2 \dots x_i$ x_i
 $y_1 y_2 \dots y_{j-1}$ y_{j-1}
 $OPT(i, j-1)$ δ

optimal substructure property
(proof via exchange argument)

8

Sequence alignment: bottom-up algorithm

SEQUENCE-ALIGNMENT($m, n, x_1, \dots, x_m, y_1, \dots, y_n, \delta, \alpha$)

FOR $i = 0$ TO m

$M[i, 0] \leftarrow i\delta.$

FOR $j = 0$ TO n

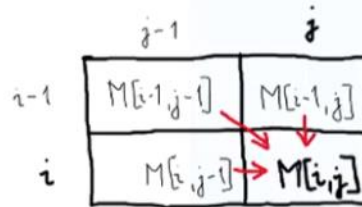
$M[0, j] \leftarrow j\delta.$

FOR $i = 1$ TO m

FOR $j = 1$ TO n

$M[i, j] \leftarrow \min \{ \alpha_{x_i y_j} + M[i-1, j-1],$
 $\delta + M[i-1, j],$
 $\delta + M[i, j-1] \}.$

RETURN $M[m, n].$



9

Sequence alignment: analysis

Theorem. The DP algorithm computes the edit distance (and an optimal alignment) of two strings of lengths m and n in $\Theta(mn)$ time and space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ■

Theorem. [Backurs-Indyk 2015] If can compute edit distance of two strings of length n in $O(n^{2-\epsilon})$ time for some constant $\epsilon > 0$, then can solve SAT with n variables and m clauses in $\text{poly}(m) 2^{(1-\delta)n}$ time for some constant $\delta > 0$.

Edit Distance Cannot Be Computed
in Strongly Subquadratic Time
(unless SETH is false)*

Arturs Backurs[†]
MIT

Piotr Indyk[‡]
MIT

which would disprove SETH
(strong exponential time hypothesis)

11

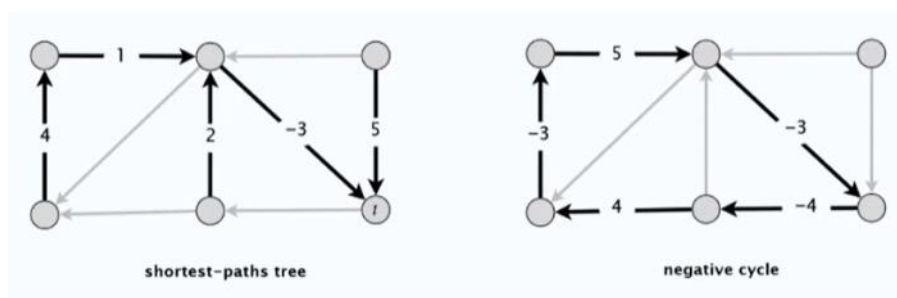
Shortest Paths with Negative Weights

29 November 2020 13:40

- Dijkstra's will work for positive edge weights.
- It won't work for negative edge weights for numerous reasons.
- There are problems in negative graphs such as negative cycles, that the algorithm can get stuck in.

Single-destination shortest-path problem – given a digraph with edge lengths but no negative cycles, and a distinguished node t , find a shortest path for every node v

Negative cycle problem – given a digraph with edge lengths, find a negative cycle if one exists



If we make the assumption the graph contains no negative cycles, then we are guaranteed that there exists a shortest path between any two vertices, that contain at most l edges.

Thus there is an upper bound on the number of edges we need to consider
Goal is $OPT(n-1, v)$ for each v , where $n-1$ is the number of nodes - 1

Shortest paths with negative weights: dynamic programming

Def. $OPT(i, v)$ = length of shortest $v \rightarrow t$ path that uses $\leq i$ edges.

Goal. $OPT(n-1, v)$ for each v .

by Lemma 2, if no negative cycles, there exists a shortest $v \rightarrow t$ path that is simple

Case 1. Shortest $v \rightarrow t$ path uses $\leq i-1$ edges.

- $OPT(i, v) = OPT(i-1, v)$.

optimal substructure property (proof via exchange argument)

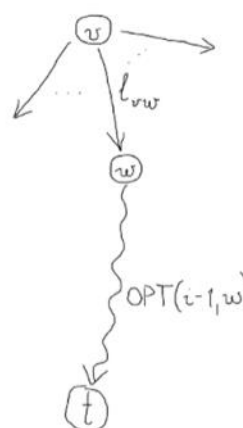
Case 2. Shortest $v \rightarrow t$ path uses exactly i edges.

- if (v, w) is first edge in shortest such $v \rightarrow t$ path, incur a cost of ℓ_{vw} .
- Then, select best $w \rightarrow t$ path using $\leq i-1$ edges.

Bellman equation.

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min \left\{ OPT(i-1, v), \min_{(v,w) \in E} \{ OPT(i-1, w) + \ell_{vw} \} \right\} & \text{if } i > 0 \end{cases}$$

39



Time complexity of the DP approach for computing shortest paths with negative weights, for every node v is $O(n^2)$, and the space complexity is $O(n^2)$

Shortest paths with negative weights: implementation

Theorem 1. Given a digraph $G = (V, E)$ with no negative cycles, the DP algorithm computes the length of a shortest $v \rightarrow t$ path for every node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Pf.

- Table requires $\Theta(n^2)$ space.
- Each iteration i takes $\Theta(m)$ time since we examine each edge once. ▀

Finding the shortest paths.

- Approach 1: Maintain $successor[i, v]$ that points to next node on a shortest $v \rightarrow t$ path using $\leq i$ edges.
- Approach 2: Compute optimal lengths $M[i, v]$ and consider only edges with $M[i, v] = M[i - 1, w] + \ell_{vw}$. Any directed path in this subgraph is a shortest path.

Intractability

05 January 2021 13:52

Basic reduction strategies.

- Simple equivalence: $\text{INDEPENDENT-SET} \equiv_p \text{VERTEX-COVER}$.
- Special case to general case: $\text{VERTEX-COVER} \leq_p \text{SET-COVER}$.
- Encoding with gadgets: $3\text{-SAT} \leq_p \text{INDEPENDENT-SET}$.

Transitivity. If $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$.

Pf idea. Compose the two algorithms.

Ex. $3\text{-SAT} \leq_p \text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER} \leq_p \text{SET-COVER}$.

Independent Set

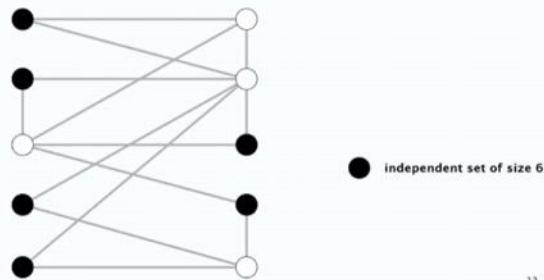
05 January 2021 12:52

Independent Sets

Given a graph $G = (V, E)$ and an integer k , is there a subset of k or more vertices such that no two are adjacent?

Ex. Is there an independent set of size ≥ 6 ?

Ex. Is there an independent set of size ≥ 7 ?



The black vertices are an independent set, because when we take any pair of black nodes they are never connected to each other.

Vertex Cover

Given a graph $G = (V, E)$ and an integer k , is there a subset of k or fewer vertices such that each edge is incident to at least one vertex in the subset?

In other words, is there a set of k nodes that connect to all edges?

Vertex Cover and Independent Set Reduce to One Another

We show **S is an independent set of size k iff $V - S$ is a vertex cover of size $n - k$**

A polynomial-time reduction proves that the first problem is no more difficult than the second one, because whenever an efficient algorithm exists for the second problem, one exists for the first problem as well.

Because this is an iff relationship, there is an implication and a reverse implication to prove. To prove the first implication:

- Let S be any independent set of size k .
- $V - S$ is of size $n - k$.
- Consider an arbitrary edge $(u, v) \in E$.
- S independent \Rightarrow either $u \notin S$, or $v \notin S$, or both.
 \Rightarrow either $u \in V - S$, or $v \in V - S$, or both.
- Thus, $V - S$ covers (u, v) . ■

We can then argue the other way:

- Let $V - S$ be any vertex cover of size $n - k$.
- S is of size k .
- Consider an arbitrary edge $(u, v) \in E$.
- $V - S$ is a vertex cover \Rightarrow either $u \in V - S$, or $v \in V - S$, or both.
 \Rightarrow either $u \notin S$, or $v \notin S$, or both.
- Thus, S is an independent set. ■

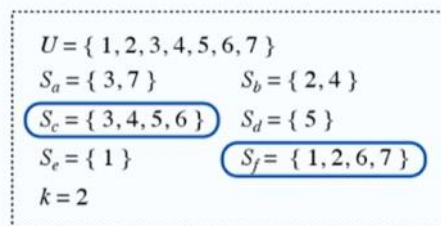
Set Cover

05 January 2021 13:29

Given a set U of elements, a collection S of subsets of U , and an integer k , are there $\leq k$ of these subsets whose union is equal to U ?

Sample application.

- m available pieces of software.
- Set U of n capabilities that we would like our system to have.
- The i^{th} piece of software provides the set $S_i \subseteq U$ of capabilities.
- Goal: achieve all n capabilities using fewest pieces of software.



a set cover instance

Vertex Cover Reduces to Set Cover

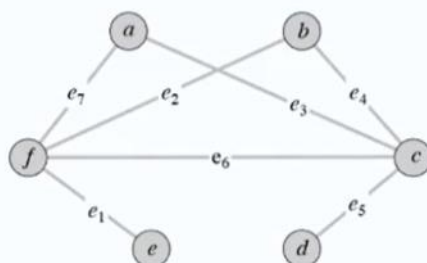
You have to do some pre-processing to get the vertex cover problem into a format acceptable for the set cover problem – i.e. let E from G be the U (niverse of edges) and let V from G be S (ets of edges, i.e. the edges connecting to the node).

Theorem. $\text{VERTEX-COVER} \leq_p \text{SET-COVER}$.

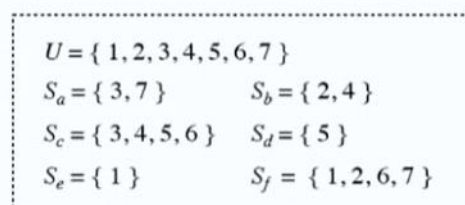
Pf. Given a VERTEX-COVER instance $G = (V, E)$ and k , we construct a SET-COVER instance (U, S, k) that has a set cover of size k iff G has a vertex cover of size k .

Construction.

- Universe $U = E$.
- Include one subset for each node $v \in V$: $S_v = \{e \in E : e \text{ incident to } v\}$.



vertex cover instance
($k = 2$)



set cover instance
($k = 2$)

Proof Vertex Cover Reduces to Set Cover

Lemma. $G = (V, E)$ contains a vertex cover of size k iff (U, S, k) contains a set cover of size k .

Pf. \Rightarrow Let $X \subseteq V$ be a vertex cover of size k in G .

- Then $Y = \{S_v : v \in X\}$ is a set cover of size k . ▀

"yes" instances of VERTEX-COVER
are solved correctly

Pf. \Leftarrow Let $Y \subseteq S$ be a set cover of size k in (U, S, k) .

- Then $X = \{v : S_v \in Y\}$ is a vertex cover of size k in G . ▀

"no" instances of VERTEX-COVER
are solved correctly

Satisfiability

05 January 2021 13:51

What's Satisfiability?

- **Literal** – a Boolean variable or its negation
- **Clause** – a disjunction of literals
- **Conjunctive Normal Form** – a propositional formula that is a conjunction of clauses

$$x_i \text{ OR } \overline{x_i}$$

$$C_j = x_1 \vee \overline{x_2} \vee x_3$$

$$\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$$

- **SAT** – given a CNF formula, does it have a satisfying truth assignment?
- **3SAT** – SAT where each clause contains exactly 3 literals and each literal corresponds to a different variable

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

yes instance: $x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{false}, x_4 = \text{false}$

Key application – electronic design automation (EDA)

Satisfiability is hard

It is conjectured that there is no poly-time algorithm for 3SAT
P vs. NP – this hypothesis is equivalent to $P \neq NP$ conjecture

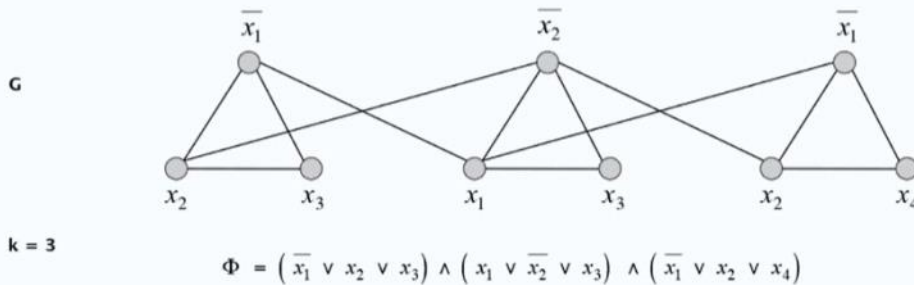
3SAT Reduces to Independent Set

Theorem. $3\text{-SAT} \leq_P \text{INDEPENDENT-SET}$.

Pf. Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size $k = |\Phi|$ iff Φ is satisfiable.

Construction.

- G contains 3 nodes for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.



Proof

Lemma. Φ is satisfiable iff G contains an independent set of size $k = |\Phi|$.

Pf. \Rightarrow Consider any satisfying assignment for Φ .

- Select one true literal from each clause/triangle.
- This is an independent set of size $k = |\Phi|$. ■

"yes" instances of 3-SAT
are solved correctly

Logically, this holds because when you select a literal from each clause you assert that it's true. The only edges that connect literals from one clause to another are those that travel between clauses and their inverses. Since we're picking true literals, we're never going to pick a literal and its inverse.

Thus, there can never be a connection between any of the true literals that we chose.

Pf. \Leftarrow Let S be independent set of size k .

- S must contain exactly one node in each triangle.
- Set these literals to *true* (and remaining literals consistently).
- All clauses in Φ are satisfied. ■

"no" instances of 3-SAT
are solved correctly