Contents

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 2: Solving Recurrence

We start with the merge sort example, and then we study some simple techniques to solve recurrence formulas that come from analysis of time complexity of algorithms.

Merge Sort

Sorting is a fundamental algorithmic task, and merge sort is a classical algorithm using the idea of divide and conquer.

This divide and conquer approach works if there is a nice way to reduce an instance of the problem to smaller instances of the same problem.

For sorting, suppose the first $\frac{n}{2}$ numbers and the last $\frac{n}{2}$ numbers are already sorted, the observation is that we can then merge these two halves easily in $O(n)$ iterations.

But how do we assume that the two halves are already sorted? The idea is to apply the same procedure (break into two halves, sort each, then merge) recursively.

So, the merge sort algorithm can be summarized as follows.

Merge sort

```
sort ( A[1,n] )
    if n=1, return.
    sort ( A[1, ⌈n/2⌉] )
    sort ( A[⌈n/2⌉+1, n] )
    merge ( A[1, ⌈n/2⌉], A[⌈n/2⌉+1, n] )
```

The correctness of the algorithm can be proved formally by a standard induction.

We focus on analyzing the running time.

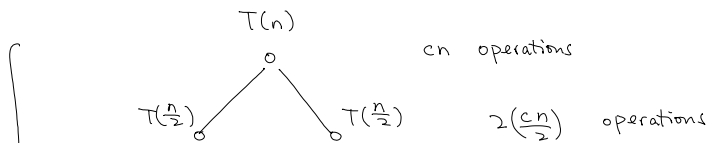
Let $T(n)$ be the time required to sort n numbers, with $T(1) = 1$.

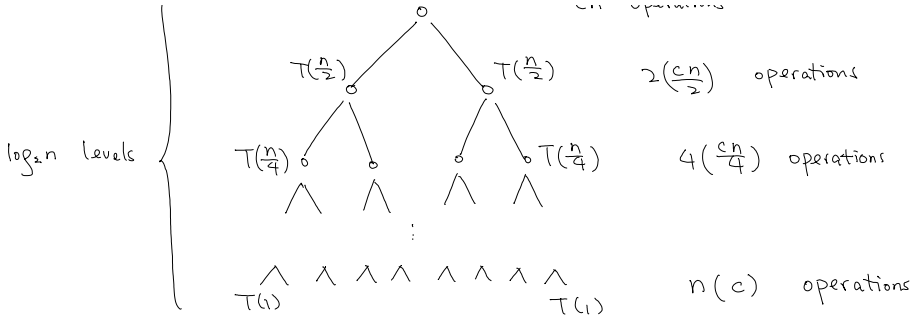
Then $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n)$.

For simplicity, we assume n is a power of two and so the relation becomes

$$T(n) = 2T(\tfrac{n}{2}) + O(n).$$

One way to solve this recurrence is to draw the recursion tree.





Each level requires cn operations, and there are $\log_2 n$ levels, and so the total time complexity is $cn\log_2 n$.

We can assume n is a power of two by at most doubling the input size, and this shows that the asymptotic complexity of merge-sort is $O(n\log n)$.

One can also prove it by induction, by using the "guess and check" method.

Induction hypothesis: $T(n) = cn\log_2 n$, where c is the constant in $O(n)$.

Induction step: $T(m) = 2T(\frac{m}{2}) + cm = 2c(\frac{m}{2})\log_2(\frac{m}{2}) + cm = cm(\log_2 m - 1) + cm = cm\log_2 m$.

Question: What is wrong with the following proof?

Induction hypothesis: $T(n) = O(n)$.

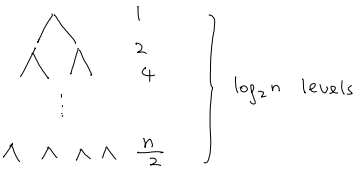
Induction step: $T(m) = 2T(\frac{m}{2}) + O(m) = 2O(\frac{m}{2}) + O(m) = O(m)$.

Exercises: Solve $T(n) = 4T(\frac{n}{2}) + n$, $T(n) = 3T(\frac{n}{2}) + n$, $T(n) = 2T(\frac{n}{2}) + n^2$, assuming n is a power of 2 and $T(1) = 1$.

Solving Recurrence [DPV 2.2]

In the following, we assume that say $T(i) \le C_2$ for $i \le C_1$ where $C_1 \ge 10$ and C_2 are absolute constants, i.e. constant size problems can be solved in constant time.

Let's try to solve $T(n) = 2T(\frac{n}{2}) + 1$, assuming $n = 2^k$ for some k.



Then we see from the picture that $T(n) \le n - 1$.

We may use the guess and check method and try $T(n) \le cn$.

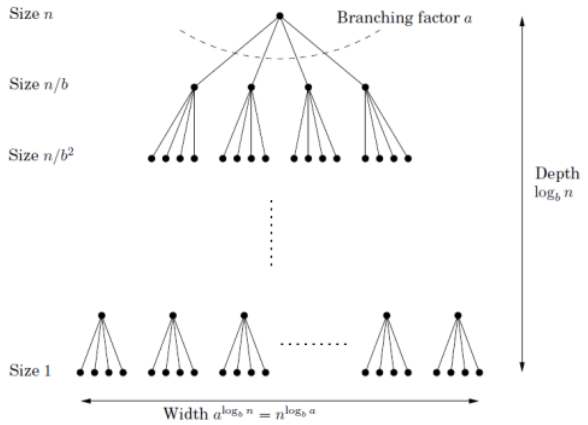
But then the induction step $T(m) = 2T(\frac{m}{2}) + 1 = 2c(\frac{m}{2}) + 1 = cm + 1$, not working.

Instead, we need to use the correct hypothesis $T(n) \le n - 1$ to make the induction works.

Let's consider a more general setting.

Consider the recurrence relation $T(n) = aT(\frac{n}{b}) + n^c$ for some constants $a > 0$, $b > 1$, $c \ge 0$.

picture from [CDPU22]



n^c operations

$a\left(\frac{n}{b}\right)^c$ operations

$a^2\left(\frac{n}{b^2}\right)^c$ operations

$a^{\log_b n}$ operations $= n^{\log_b a}$

If we sum the number of operations, we see that it is a geometric sequence, with the ratio $\frac{a}{b^c}$.

Now we analyze the sum based on whether the ratio is greater than 1, smaller than 1, or equal to 1.

- If $\frac{a}{b^c} = 1$, then every term is the same, and we have the sum is $n^c \cdot \log_b n$. This is what we have seen in merge sort.
- If $\frac{a}{b^c} < 1$, then it is a decreasing geometric sequence. and it is dominated by the first term, and we have the sum is $O(n^c)$, with the hidden constant depending on a, b, c (this is where we need to assume they are constant).
- If $\frac{a}{b^c} > 1$, then it is an increasing geometric sequence, and it is dominated by the last term, and we have the sum is $O(a^{\log_b n}) = O(n^{\log_b a})$.

To summarize, we have proved the result known as the master theorem.

Master Theorem If $T(n) = a\,T(\frac{n}{b}) + n^c$ for constants $a > 0$, $b > 1$, $c \geq 0$, then

$$T(n) = \begin{cases} O(n^c) & \text{if } c > \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^{\log_b a}) & \text{if } c < \log_b a . \end{cases}$$

Remark: It is more important to remember the method than the result. It is easy to derive the result back.

Also, there are scenarios that the theorem does not apply but the method still works as we will see.

More Recurrences

Single subproblem: This is common in algorithm analysis.

Examples: $T(n) = T(\frac{n}{2}) + 1$, we have $T(n) = O(\log n)$, binary search.

$T(n) = T(\frac{n}{2}) + n$, we have $T(n) = O(n)$, geometric sequence.

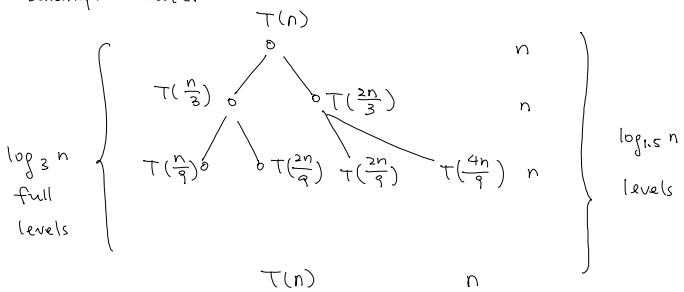
$T(n) = T(\sqrt{n}) + 1$, we have $T(n) = O(\log\log n)$, counting levels.

(In level i, the subproblem is of size $n^{2^{-i}}$. When $i = \log\log n$, it becomes $n^{\frac{1}{\log n}} = O(1)$.)

Non-even subproblems: We will see one interesting example later.

$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + n$

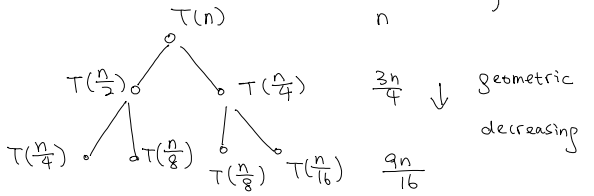
So, $T(n) = O(n \log n)$.



$T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n$

So, $T(n) = O(n)$.

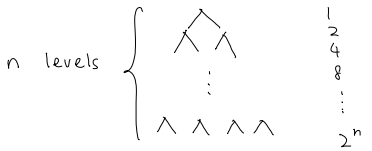
Could check by induction that $T(n) \leq 4n$.



Exponential time

$T(n) = 2T(n-1) + 1$

So, $T(n) = O(2^n)$.



Optional

Can we improve the runtime if we have $T(n) = T(n-1) + T(n-2) + 1$?

This is the same recurrence as the Fibonacci sequence.

Using the techniques that you have learnt in MATH 239 (computing roots of polynomials), it can be shown that $T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = O(1.618^n)$, faster exponential time.

This kind of recurrence shows up often in analyzing faster exponential time algorithms.

Consider the maximum independent set problem, where we are given a graph $G = (V, E)$, and our task is to find a maximum subset of vertices $S \subseteq V$ such that there are no edges between every pair of vertices $u, v \in S$.

A naive algorithm is to enumerate all subsets, and this takes $\Omega(2^n)$ time.

Now consider a simple variant.

Pick a vertex v with maximum degree.

There are two possibilities: either $v \in S$ or $v \notin S$.

In the latter case, we delete v and reduce the graph size by one.

In the former case, we choose v, and then we know that all neighbors of v cannot be chosen, and so we can delete v and all its neighbors, so that the graph size is reduced by at least two.

So, $T(n) \leq T(n-1) + T(n-2) + O(n)$, and it is strictly smaller than $O(2^n \cdot n)$.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 3: Divide and Conquer

We will study some divide and conquer algorithms, including the interesting median of median algorithm.

Counting Inversions [KT 5.3]

Input: n distinct numbers $a_1, a_2, \ldots a_n$.

Output: number of pairs with $i<j$ but $a_i > a_j$.

For example, given $(3, 7, 2, 5, 4)$, there are five pairs of inversion $(3,2), (7,2), (7,5), (7,4), (5,4)$.

You can think of this problem as computing the "unsortedness" of a sequence.

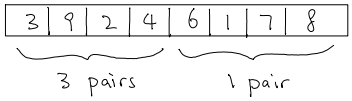
You may also imagine that this is measuring how different are two rankings; see [KT 5.3] for an elaboration of this connection.

For simplicity, we again assume that n is a power of two.

Using the idea of divide and conquer, we try to break the problem into two halves.

Suppose we could count the number of inversions in the first half, as well as in the second half.

Would it then be easier to solve the remaining problem?



It remains to count the number of inversions with one number in the first half and the other number in the second half.

These "cross" inversion pairs are easier to count, because we know their relative positions.

In particular, to facilitate the counting, we could sort the first half and the second half, without worrying losing information as we already counted the inversion pairs within each half

Now, for each number c in the second half, the number of "cross" inversion pairs involving c is precisely the number of numbers in the first half that is larger than c.

In this example, 1 is involved in 4 cross pairs. 6, 7, 8 are all involved in 1 cross pair (with 9), and so the number of "cross" inversion pairs is 7.

How to count the number of cross inversion pairs involving a number a_j in the second half efficiently?

Idea 1: As the first half is sorted, we can use binary search to determine how many numbers in the first half are greater than a_j.

This takes $O(\log n)$ time for one a_j, and totally $O(n \log n)$ time for all numbers in the second half.

This is not too slow, but we can do better.

Idea 2: Observe that this information can be determined when we merge the two sorted list in merge sort.

When we insert a number in the second half to the merged list, we know how many numbers in the first half that are greater than it.

For example, 

, we know that there is only one number in the first half that is greater than 6.

As in merge sort, this can be done in $O(n)$ time.

As in merge sort, this can be done in $O(n)$ time.

Algorithm: Now we are ready to describe the algorithm we have so far.

Count ($A[1,n]$) $\leftarrow T(n)$

if $n=1$, return 0. // base case

count($A[1, \frac{n}{2}]$) $\leftarrow T(\frac{n}{2})$

count ($A[\frac{n}{2}+1, n]$) $\leftarrow T(\frac{n}{2})$

Sort ($A[1, \frac{n}{2}]$) $\leftarrow O(n \log n)$

Sort ($A[\frac{n}{2}+1, n]$) $\leftarrow O(n \log n)$

merge-and-count-cross-inversions ($A[1, \frac{n}{2}]$, $A[\frac{n}{2}+1, n]$) $\leftarrow O(n)$

Total time complexity is $T(n) = 2T(\frac{n}{2}) + O(n \log n)$.

Solving this will give $T(n) = \Theta(n \log^2 n)$. Try this using the recursion tree method.

Perhaps you have already noticed that the sorting step is the bottleneck and is unnecessary, as we have already sorted them in the merge step.

As it turns out, we can just modify the merge sort algorithm to count the number of inversion pairs.

Final algorithm

Count-and-sort ($A[1,n]$) $\leftarrow T(n)$

if $n=1$, return 0.

S_1 = Count-and-sort($A[1, \frac{n}{2}]$) $\leftarrow T(\frac{n}{2})$

S_2 = Count-and-sort($A[\frac{n}{2}+1, n]$) $\leftarrow T(\frac{n}{2})$

S_3 = merge-and-count-cross-inversions($A[1, \frac{n}{2}]$, $A[\frac{n}{2}+1, n]$) ← $O(n)$

return $S_1 + S_2 + S_3$

Total time complexity $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$ as in merge sort.

After all - it is just a simple modification of the merge sort algorithm. but I hope the (long) thought process clarifies how we approach the problem and come up with the algorithm.

Exercise : Write pseudocode for the merge-and-count-cross-inversions subroutine.

Maximum Subarray [CLRS 4.1]

Input : n numbers $a_1, \ldots, a_n$

Output : i, j that maximizes $\sum_{k=i}^{j} a_k$.

For example, 1, -3, 4, 5, 6, -20, 5, 7, 7, -3, 9, 10, 11, -10, 3, 3
(5, 7, 7, -3, 9, 10, 11 : this is the optimal solution)

The problem is trivial if all numbers are positive.

A naive algorithm is to try all i, j and compute the sum, and this takes $O(n^3)$ time - too slow.

We can speed it up by realizing that for one i, we can compute the sum from i to j for all j in $O(n)$ time, and so the total time is $O(n^2)$.

We now apply the divide and conquer approach to this problem.

We again assume that n is a power of two.

Suppose we break the array into two halves, and find the maximum subarray within each half.

Would it be easier to solve the remaining problem?

1, -3, 4, 5, 6, -20, 5, 7 | 7, -3, 9, 10, 11, -10, 3, 3

An optimal solution is either contained in the first half, contained in the second half, or crossed the mid-point.

So, after solving the two subproblems, we just need to find the maximum subarray that crosses the mid-point - i.e. $i \le \frac{n}{2} < j$.

Each such crossing subarray can be broken into two pieces - $[i, \frac{n}{2}]$ and $[\frac{n}{2}+1, j]$

Assume that $[i, j]$ is an optimal solution with $i \le \frac{n}{2} < j$.

Observe that $[i, \frac{n}{2}]$ must be a maximum subarray ending at $\frac{n}{2}$.

Suppose not. If the sum from $[i', \frac{n}{2}]$ is even bigger, then the sum from $[i', j]$ would be bigger than that of $[i, j]$, contradicting the optimality of the sum from $[i, j]$.

This observation leads to the following claim which leads to a simple algorithm.

Claim For $i \le \frac{n}{2} < j$, $[i, j]$ is a maximum subarray crossing the mid-point if and only if $[i, \frac{n}{2}]$ is a maximum subarray ending at $\frac{n}{2}$ and $[\frac{n}{2}+1, j]$ is a maximum subarray starting at $\frac{n}{2}+1$.

Finding a maximum subarray with a fixed starting / ending point can be done in $O(n)$ time, as we discussed in the $O(n^2)$ algorithm above by computing the partial sums.

The correctness of the algorithm is based on the claim above.

The total time complexity is $T(n) = 2T(\frac{n}{2}) + O(n)$, where $O(n)$ is for maximum "crossing" subarray, and it follows that $T(n) = O(n \log n)$.

Challenge: Provide an $O(n)$ time algorithm for the maximum subarray problem.

Question: Can you extend the algorithm to work in the circular setting, where the array wraps around? E.g. 6 1 2 7 -3 -10 8 4

Finding Median [CLRS 9.3]

Input: n distinct numbers $a_1, a_2, \ldots, a_n$.

Output: the median of these numbers.

It is clear that the problem can be solved in $O(n \log n)$ time, by first sorting the numbers.

But it turns out that there is an interesting $O(n)$-time algorithm.

To solve the median problem, it is more convenient to consider a slightly more general problem.

Input: n distinct numbers $a_1, \ldots, a_n$ and an integer $k \ge 1$.

Output: the k-th smallest number in $a_1, \ldots, a_n$.

The reason is that the median problem doesn't reduce to itself (and so we can't recurse), while the k-th smallest number lends itself to reduction as we will see.

The idea is similar to that in quicksort (which is a divide and conquer algorithm).

We choose a number a_i. Split the n numbers into two groups, one group with numbers smaller than a_i, called it S_1. and the other group with numbers greater than a_i, called it S_2.

Let r be the rank of a_i, i.e. a_i is the r-th smallest number in $a_1, \ldots, a$

If $r=k$, then we are done.

If $r>k$, then find the k-th smallest number in S_1.

If $r<k$, then find the $(k-r)$-th smallest number in S_2.

Observe that when $r>k$, the problem size is reduced to $r-1$ as $|S_1| = r-1$,

and when $r<k$, the problem size is reduced to $n-r$ as $|S_2| = n-r$.

So, if somehow we could choose a number "in the middle" as a pivot as in quicksort, then we can reduce the problem size quickly and making good progress, but finding a number in the middle is the very question that we want to solve.

But observe that we don't need the pivot to be exactly in the middle, just that it is not too close to the boundary.

Suppose we can choose a_i such that its rank satisfies say $\frac{n}{10} \le r \le \frac{9n}{10}$, then we know that the problem size would have reduced by at least $\frac{n}{10}$, as $|S_1| = r-1 \le \frac{9n}{10}$ and $|S_2| = n-r \le \frac{9n}{10}$.

So, the recurrence relation for the time complexity is $T(n) \le T\left(\frac{9n}{10}\right) + P(n) + cn$, where $P(n)$ denotes the time to find a good pivot and cn is the number of operations for splitting.

If we manage to find a good pivot in $O(n)$ time, i.e. $P(n) = O(n)$, then it implies that $T(n) = O(n)$.

We have made some progress to the median problem, by reducing the problem of finding the number exactly in the middle to the easier problems of finding a number not too far from the middle.

It remains to figure out a linear time algorithm to find a good pivot.

Randomized solution: If you have seen randomized quicksort before, then you would have guessed that keep choosing a random pivot would work. This is indeed the case and you can take a look at [DPV 2.4] for a proof. We won't discuss randomized algorithms in this course.

Deterministic solution: There is an interesting deterministic algorithm that would always return a number a_i with rank $\frac{3n}{10} \le r \le \frac{7n}{10}$ in $O(n)$ time.

As seen above, this means $P(n) = O(n)$ and it follows that $T(n) = O(n)$.

Finding a good pivot: The idea of the algorithm is to find the median of the medians.

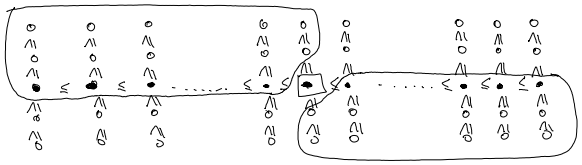
① Divide the n numbers into $\frac{n}{5}$ groups, each of 5 numbers. Time: $O(n)$.

② Find the median in each group. Call them $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $O(n)$.

③ Find the median of these $\frac{n}{5}$ medians $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $T(\frac{n}{5})$.

Lemma Let r be the rank of the median of medians. Then $\frac{3n}{10} \le r \le \frac{7n}{10}$.

Proof



The square is the median of the medians.

In the picture, we sort each group, and then order the groups by an increasing order of the medians.

We emphasize that this is just for the analysis, and we don't need to do sorting in the algorithm.

It should be clear that the square is greater than the numbers in the top-left corner, and is smaller than the numbers in the bottom-right corner.

There are about $3\cdot(\frac{n}{10}) = \frac{3n}{10}$ numbers in the top-left and the bottom-right corners.

This implies that $\frac{3n}{10} \le r \le \frac{7n}{10}$. □

This lemma proves the correctness of the pivoting algorithm.

Time Complexity

We have $P(n) = T(\frac{n}{5}) + c_1 n$, where c_1 is a constant.

By the reduction above, $T(n) \le T(\frac{7n}{10}) + P(n) + c_2 n = T(\frac{7n}{10}) + T(\frac{n}{5}) + (c_1+c_2)n$.

Using the recursion tree method in L02, we see that $T(n) = O(n)$. We could check it by induction.

This completes the analysis of the median of medians algorithm.

Exercise: What happens if we divide into groups of 3 elements, 7 elements, or $\sqrt{n}$ elements?

Exercise: It would be good to trace out the recursion to understand the algorithm more deeply.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 4: Divide and Conquer II

We will see more examples of divide and conquer algorithms: one in computational geometry and others in computer algebra, for which divide and conquer gives the fastest known algorithms.

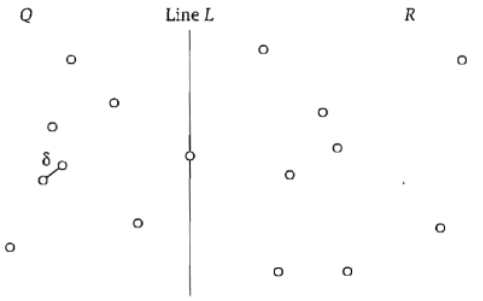
Closest Pair [KT 5.4]

Input: n points $(x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)$ on the 2D-plane.

Output: $1 \le i < j \le n$ that minimizes the Euclidean distance $\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$

It is clear that this problem can be solved in $O(n^2)$ time by trying all pairs.

We use the divide and conquer approach to give an improved algorithm.



picture from [KT 5.4]

We find a vertical line L to separate the point set into two halves: call the set of points on the left of the line Q, and the set of points on the right of the line R.

For simplicity, we assume that every point has a distinct x-value. We leave it as an exercise to see where this assumption is used and also how to remove it.

The vertical line can be found by computing the median based on the x-value, and put the first $\lceil \frac{n}{2} \rceil$ points in Q, and the last $\lfloor \frac{n}{2} \rfloor$ points in R.

Now, we recursively find the closest pair within Q, and the closest pair within R.

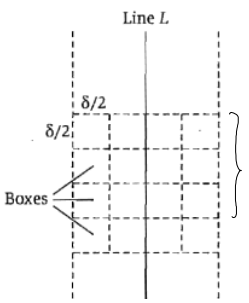
Suppose the closest pair in Q is of distance δ, and it is smaller than that in R.

To solve the closest pair problem in all n points, it remains to find the closest "crossing" pair with one point in Q and the other point in R.

It doesn't seem that the closest crossing pair problem is easier to solve.

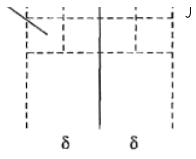
The idea is that we only need to determine whether there is a crossing pair with distance $< \delta$.

This allows us to restrict attention to the points with x-value within δ to the line L, but still all the points can be here.



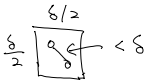
This allows us to restrict attention to the points with x-value within δ to the line L, but still all the points can be here.

We divide the narrow region into square boxes of side length $\frac{\delta}{2}$ as shown in the picture. Here comes the important observations.



(picture from [KT])

Observation 1 Each square box has at most one point.

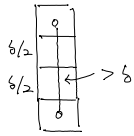


Proof If two points are in the same box, their distance is at most $\sqrt{\left(\frac{\delta}{2}\right)^2+\left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$.

This would contradict that the closest pairs within Q and within R have distance $\geq \delta$. □

Observation 2 Each point needs only to compute distances with points within two horizontal layers.

Proof For two points which are separated by at least two horizontal layers (see picture), then their distance would be more than δ and would not be closest. □



With observation 2, every point in a square box needs only to check with points in at most eleven other boxes (boxes in the same layer and the next two layers).

Observation 1 says that there is at most one point in each square.

Combining these, each point only needs to compute distances with at most eleven other points, in order to search for the closest pairs (i.e. pairs with distance $\leq \delta$).

This cuts down the search space from $\Omega(n^2)$ pairs to $O(n)$ pairs.

Algorithm

1. Find the dividing line L by computing the median using the x-value. Time: $O(n)$.
2. Recursively solve the closeset pair problem in Q and in R. Get δ. Time: $T(\frac{n}{2})$.
3. Using a linear scan, remove all the points not within the narrow region defined by δ. Time: $O(n)$.
4. Sort the points in non-decreasing order by their y-value. Time: $O(n \log n)$.
5. For each point, we compute its distance to the next eleven points in this y-ordering. Time: $O(n)$.
   (Note that two points within two layers must be within 11 points in the y-order, as ≤ 10 boxes in between.)
6. Return the minimum distance found.

The correctness of the algorithm is established by the two observations, justifying that it suffices for each point to compute distance to $O(1)$ other points as described in step 5.

Time complexity: $T(n) = 2T(\frac{n}{2}) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$.

Note that the bottleneck is in the sorting step and it is not necessary to do sorting within recursion.

We can sort the points by y-value once in the beginning and use this ordering throughout the algorithm.

This reduces the time complexity to $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$.

Similarly, we don't need to compute the medians within the recursion.

We can sort the points by x-value once in the beginning and use it for the dividing step.

This would not improve the worst case time complexity but would improve its practical performance.

Questions: 1. Where did we use the assumption that the x-values are distinct?

2. What do we need to change so that the algorithm would work without this assumption?

Remark: There is a randomized algorithm to find a closest pair in expected $O(n)$ time. See [KT 13.7].

Arithmetic Problems

Arithmetic problems are where the divide and conquer approach is most powerful.

Many fastest algorithms for basic arithmetic problems are based on divide and conquer.

Today we will see some basic ideas how this approach works. But unfortunately we will not see the fastest algorithms as they require some background in algebra (although not too much).

Integer Multiplication [KT 5.6]

This is a really fundamental problem that our computers solve everyday.

Given two n-bit numbers $a = a_1 a_2 \cdots a_n$ and $b = b_1 b_2 \cdots b_n$, we would like to compute ab efficiently.

The multiplication algorithm that we learnt in elementary school takes $\Theta(n^2)$ bit operations.

```
   1100
 x 1101
------
   1100
  0000
 1100
1100
--------
10011100
```

} n n-bit numbers. adding one by one, each addition requires $\Theta(n)$ bit operations.

Let's apply the divide and conquer approach to integer multiplication.

Suppose we know how to multiply n-bit numbers efficiently.

We would like to use it to multiply 2n-bit numbers quickly.

Given two 2n-bit numbers x and y, we write $x = x_1 x_2$ and $y = y_1 y_2$, where x_1, y_1 are the higher-order n-bits and x_2, y_2 are the lower-order n-bits.

Written mathematically, $x = x_1 \cdot 2^n + x_2$ and $y = y_1 \cdot 2^n + y_2$.

Then, $xy = (x_1 \cdot 2^n + x_2)(y_1 \cdot 2^n + y_2) = x_1 y_1 \cdot 2^{2n} + (x_1 y_2 + x_2 y_1) \cdot 2^n + x_2 y_2$.

Since x_1, x_2, y_1, y_2 are n-bit numbers, the products x_1y_1, x_1y_2, x_2y_1, x_2y_2 can be computed recursively.

Therefore, $T(n) = 4T(\frac{n}{2}) + O(n)$, where the additional $O(n)$ bit operations are used to add the numbers.

(Note that $x_1y_1 \cdot 2^{2n}$ is simply shifting x_1y_1 to the left by $2n$ bits; we don't need a multiplication operation.)

Solving the recurrence will give $T(n) = O(n^2)$, not improving the elementary school algorithm.

This should not be surprising, since we haven't done anything clever to combine the subproblems, and we should not expect that just by doing divide and conquer some speedup would come automatically.

Kuratsuba's algorithm: A clever way to combine the subproblems.

Instead of computing x_1y_1, x_1y_2, x_2y_1, x_2y_2 using four subproblems, Kuratsuba's idea is to use three subproblems to compute x_1y_1, x_2y_2 and $(x_1+x_2)(y_1+y_2)$.

After that, we can compute the middle term $x_1y_2 + x_2y_1$ by noticing that

$$(x_1 + x_2) \cdot (y_1 + y_2) - x_1y_1 - x_2y_2 = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2 - x_1y_1 - x_2y_2 = x_1y_2 + x_2y_1.$$

That is, the middle term can be computed in $O(n)$ bit operations after solving the three subproblems.

Therefore, the total complexity is $T(n) = 3T(\frac{n}{2}) + O(n)$, and it follows from master theorem that $T(n) = O(n^{(\log_2 3)}) = O(n^{1.59})$.

This is the first and significant improvement over the elementary school algorithm.

Polynomial Multiplication

Given two degree n polynomials, $A(x) = a_n x^n + a_{n-1}x^{n-1} + \dots + a_1 x + a_0$ and $B(x) = \sum_{i=0}^{n} b_i x^i$.

we can use the same idea to compute $A \cdot B(x)$ in $O(n^{1.59})$ word operations,

where we assume that $a_i b_j$ can be computed in $O(1)$ word operations.

You will need to work out the details in the programming problem.

Matrix multiplication [DPV 2.5]

We all know the $O(n^3)$ word operations algorithm to multiply two nxn matrices.

The divide and conquer approach can also be successfully applied to matrix multiplication.

Given two 2nx2n matrices A and B, we can think of each matrix as consisting of four nxn blocks such that $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$.

Then, $$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

There are eight subproblems to solve: $A_{11}B_{11}$, $A_{11}B_{21}$, $A_{12}B_{21}$, $A_{12}B_{22}$, $A_{21}B_{11}$, $A_{21}B_{12}$, $A_{22}B_{21}$, $A_{22}B_{22}$.

After that, we just need to do $O(n^2)$ word operations to obtain AB by adding the subproblems.

The time complexity is $T(n) = 8T(\frac{n}{2}) + O(n^2)$, which implies that $O(n^3)$.

Again, this should not be surprising, as we are just doing the standard algorithm in block form.

We should not expect to get an improvement without doing anything clever.

Strassen's Algorithm

For some time, the $O(n^3)$ algorithm was thought to be optimal, but Strassen surprised the world by his magic formula: $AB = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$,

where $P_1 = A_{11}(B_{12} - B_{22})$, $P_2 = (A_{11} + A_{12})B_{22}$, $P_3 = (A_{21} + A_{22})B_{11}$, $P_4 = A_{22}(B_{21} - B_{11})$,

$P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$, $P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$, $P_7 = (A_{11} - A_{21})(B_{11} + B_{12})$.

The point is that each subproblem can be computed using one multiplication and $O(n^2)$ additional operations.

The time complexity is $T(n) = 7T(\frac{n}{2}) + O(n^2)$.

It follows from the master theorem that $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$!

After Strassen's algorithm, there is a long line of research (with some recent developments) pushing the time complexity of matrix multiplication to $O(n^{2.37})$.

Some researchers believe that matrix multiplication can be done in $O(n^2)$ word operations.

This is currently of theoretical interest only, as the algorithms are too complicated to be implemented.

Strassen's algorithm can be implemented and it will be faster than the standard algorithm when $n \gtrsim 5000$.

Applications

There are many combinatorial problems that can be reduced to matrix multiplication, and Strassen's result implies that they can be solved faster than $O(n^3)$ time.

As an example, the problem of determining whether a graph has a triangle can be reduced to matrix multiplication, and we leave it as a puzzle to you to figure out how.

There are many combinatorial problems in the literature where the fastest known algorithm is by matrix multiplication.

Fast Fourier Transform [DPV 2.6] [KT 5.7]

There is a very nice algorithm to solve integer multiplication and polynomial multiplication in $O(n \log n)$ time.

The presentation in [DPV 2.6] is highly recommended for those who are interested in learning it.

Waterloo has a strong symbolic computation group and you can learn it in CS 487.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 5 : Breadth First Search

We study simple graph algorithms based on graph searching.

There are two most common search methods : breadth first search (BFS) and depth first search (DFS).

Today we study BFS and see some applications. Next time we will study DFS.

Graphs

Many problems in computer science can be modeled as graph problems. See [KT 3.1] for discussions.

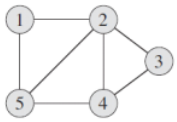
Graph Representations

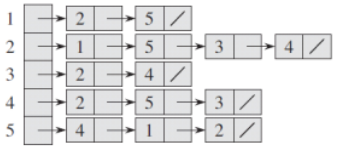
Let $G=(V,E)$ be an undirected graph. We use throughout that $n=|V|$ and $m=|E|$.

There are two standard representations of a graph.

One is the adjacency matrix : It is an $n \times n$ matrix A with $A[i,j] = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{if } ij \notin E \end{cases}$.

Another is the adjacency list : Each vertex maintains a linked list of its neighbors.





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

(picture from [CLRS])

We will mostly use the adjacency list representation, as its space usage depends on the number of edges, while we need to use $\Theta(n^2)$ space to store an adjacency matrix.

Only the adjacency list representation allows us to design algorithms with $O(m+n)$ word operations.

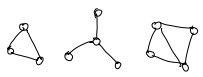
Graph Connectivity

Given a graph, we say two vertices are connected if there is a path from u to v.

A subset of vertices $S \subseteq V$ is connected if $u,v \in S$ are connected for all $u,v \in S$.

A graph is connected if $s,t \in V$ are connected for all $s,t \in V$.

A connected component is a maximally connected subset of vertices.



3 Components

Some of the most basic questions about a graph are :

1. to determine whether it is connected.
2. to find all the connected components.
3. to determine whether u,v are connected for given $u,v \in V$.

④ to output a shortest path between u and v for given $u, v \in V$.

Breadth first search (BFS) can be used to answer all these questions in $O(n+m)$ time.

Breadth First Search

To motivate breadth first search, imagine we are searching for a person in a social network. A natural strategy is to ask our friends, and then ask our friends to ask their friends, and so on. A basic version of BFS is described as follows.

Input: $G = (V, E)$, $s \in V$.

Output: all vertices reachable from s.

Initialization: visited[v] = false for all $v \in V$.

queue Q is empty. enqueue(Q, s). visited[s] = true.

```
While Q ≠ empty do
    u = dequeue(Q)
    for each neighbor v of u
        if visited[v] = false
            enqueue(Q, v).   visited[v] = true.
```

Time Complexity

Each vertex is enqueued at most once (when visited[v] = false).

When a vertex is dequeued, the for loop is executed for deg(v) iterations.

So, the total time complexity is $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$.

Lemma There is a path from s to v if and only if visited[v] = true at the end.

Proof

$\Leftarrow$) We prove by induction on the number of steps of the algorithm that if visited[v] = true, then there is a path from s to v.

The base case is at the beginning when only visited[s] = true.

Now, supposed visited[v] is set to be true in the current step, inside the for loop of u.

Then, visited[u] = true at that time, because u was put in the queue.

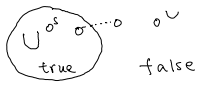
By the induction hypothesis, there is a path from s to u.

Extending this path with the edge uv, we have found a path from s to v.

$\Rightarrow$) Let U be the set of vertices such that visited[v] is set to be true.

We would like to show that there is no path from s to any vertex in $V \setminus U$.

Note that there are no edges with one endpoint in U and another endpoint in $V \setminus U$, as otherwise we would have enqueued the other endpoint and set it to true.



Suppose for contradiction that there is a path from s to $v \in V \setminus U$.

Then there must exist an edge in the path that "crosses" the set U, a contradiction. □

The correctness of BFS is supported by the lemma.

With this claim, we see that this basic version of BFS can already be used to answer:

- whether the graph is connected or not, by checking whether visited[v] = true for all $v \in V$;
- the connected component containing s, by returning all the vertices with visited[v] = true;
- whether there is a path from s to v, by checking whether visited[v] = true.

Exercise: Find all connected components of the graph in $O(n+m)$ time.

BFS Tree

How to trace back a path from s to v (if such a path exists)?

This follows from the proof of the lemma above.

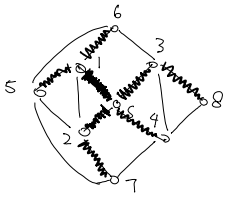
We can add an array parent[v].

When a vertex v is first visited, within the for loop of vertex u, then we set parent[v] = u.

Now, to trace out a path from v to s, we just need to write a for loop that starts from v, and keep going to its parent until we reach vertex s.

For all vertices v reachable from s, the edges $(v, \text{parent}[v])$ form a tree, called the BFS tree.

Why is it a tree in the connected component containing s?



Say the connected component has n vertices.

Every vertex, except s, has one edge to its parent.

Theses edges can't form a cycle because the parent of a vertex is visited earlier.

So, these edges form an acyclic subgraph and there are $n-1$ edges (as s has no parent).

Therefore, the edges $(v, \text{parent}[v])$ must form a tree in the component containing s.

Shortest Paths

Not only can we trace back a path from v to s using a BFS tree, this path is indeed a shortest path from s to v!

To see this, let's think about how a BFS tree was created.

These edges record the first edges to visit a vertex.

Initially, s is the only vertex in the queue, and then every neighbor of s is visited with s being their parent, and these edges are put in the BFS tree.

At this time, all vertices with distance one from s are visited and are put in the queue, before all other vertices with distance at least two from s are put in the queue.

A vertex v is said to have distance k from s if the shortest path length from s to v is k.

Then, all vertices with distance one will be dequeued, and then all vertices with distance two will be enqueued before all other vertices with distance at least three.

Repeating this argument inductively will show that all the shortest path distances from s are computed correctly, and a shortest path can be traced back from the BFS tree.

This is also very intuitive (friends before friends of friends before friends of friends of friends etc).

Being able to compute the shortest paths from s is the main feature of BFS.

We summarize below the BFS algorithm with shortest path distances included.

Input: $G = (V, E)$, $s \in V$.

Output: all vertices reachable from s and their shortest path distance from s.

```
Initialization:   visited[v] = false for all v ∈ V.
                  queue Q is empty.  enqueue(Q,s).  visited[s] = true.  distance[s] = 0.
While Q ≠ empty do
    u = dequeue(Q)
    for each neighbor v of u
        if visited[v] = false
            enqueue(Q,v).  visited[v] = true.  parent[v] = u.  distance[v] = distance[u] + 1.
```

Exercise: Write the code for printing a shortest path from v to s.

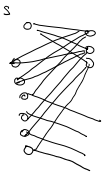
Bipartite Graphs

One application of BFS is to check whether a graph is bipartite or not.

There is not much freedom allowed to design an algorithm for checking bipartiteness.

Given a vertex s, all its neighbors must be on the other side, and then neighbors of neighbors must be on the same side as s, and so on.

With this observation, we can run the BFS algorithm above and put all vertices with even distance from s on the same side as s and all other vertices on the other side.



Algorithm

- Let $L = \{ v \in V \mid \text{dist}(s,v) \text{ even} \}$ and $R = \{ v \in V \mid \text{dist}(s,v) \text{ odd} \}$.
- If there is an edge with both endpoints in L or both endpoints in R, then return "non-bipartite".
- Otherwise, return "bipartite" and (L,R) as the bipartition.

We assume the graph is connected, as otherwise we can solve the problem in each component.

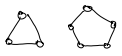
The time complexity is $O(m+n)$ as we just do a BFS and then check every edge once.

Correctness

It is clear that when the algorithm says "bipartite" it is correct, as (L,R) is indeed a bipartition.

The more interesting part is to show that when the algorithm say "non-bipartite", it is also correct.

When can we say for sure that a graph is non-bipartite?

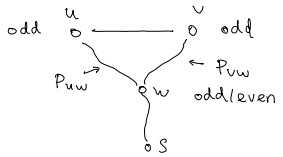
A necessary and sufficient condition is when the graph has an odd cycle (MATH 239) 

Thus we would like to show that when the algorithm says "non-bipartite", the graph has an odd cycle.

Suppose WLOG that there is an edge uv between two vertices u,v in L.

We look at the BFS tree T.

Let w be the lowest common ancestor of u,v in T.



Since $\text{dist}(s,u)$ and $\text{dist}(s,v)$ are both odd (i.e. having the same parity), regardless of whether $\text{dist}(s,w)$ is even or odd, the sum of the path lengths of uw and vw on T is a even number.

This implies that $P_{vw} \cup P_{uw} \cup \{uv\}$ is an odd cycle.

So, when the algorithm says "non-bipartite", it is correct as the graph has an odd cycle. □

Some Remarks

① This provides an algorithmic proof that a graph is bipartite iff it has no odd cycles.

② This also provides a linear time algorithm to find an odd cycle of an undirected graph.

③ Having an odd cycle is a "short proof" of non-bipartiteness.

It is much better than saying "we tried all bipartitions but all failed".

References: [KT 3.1, 3.2], [DPV 4.1, 4.2], [CLRS 22.1, 22.2].

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 6: Depth First Search

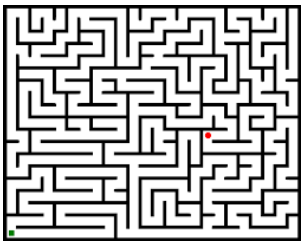
Depth first search is another basic search method in graphs, and this will be useful in identifying more refined connectivity structures as we will see today and next time.

Motivating Example

Last time we imagined that we would like to search for a person in a social network, and BFS is a very natural strategy (asking friends, then friends of friends, and so on).

There are other situations that using depth first search is more natural.

Imagine that we are in a maze searching for the exit.



We could model this problem as a s-t connectivity problem in graphs.

Each square of the maze is a vertex, and two vertices have an edge if and only if the two squares are reachable in one step.

Then, finding a path from our current position to the exit is equivalent to finding a path between two specified vertices in a graph (or determine that none exists).

How would you search for a path in the maze?

There are no friends to ask, and it doesn't look efficient anymore to explore all vertices with distance one, then distance two and so on (as we have to move back and forth).

Assuming we have a chalk and can make marks on the ground. Then it is more natural to keep going on one path bravely until we hit a dead end, and make some marks on the way and also on the way back so that we won't come back to this dead end again, and only explore yet unexplored places.

This is essentially depth first search (DFS).

Depth First Search

As for BFS, we define DFS by an algorithm. DFS is most naturally defined as a recursive algorithm.

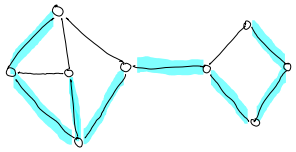
DFS Algorithm

Input: an undirected graph $G=(V,E)$, a vertex $s \in V$.

Output: all vertices reachable from s.

[Main program] visited[v] = false $\forall v \in V$. visited[s] = true. explore(s).

```
explore(u)        // recursive function explore.
    for each neighbor v of u
        if visited[v] = false
            visited[v] = true .   explore(v).
```



Time Complexity

The analysis of the time complexity is similar to that in BFS.

For each vertex u, the recursive function explore(u) is called at most once.

When explore(u) is called, the for loop is executed at most $deg(u)$ times.

Thus the total time complexity is $O(n + \sum_{v \in V} deg(v)) = O(n+m)$ word operations.

Stack and Queue

There is a way to write DFS non-recursively.

The idea, not surprisingly, is to use a stack, as recursive programs are implemented using stacks in our computers.

The resulting program using stack is syntatically very similar to that of BFS.

So, one can think about the two fundamental search methods correspond to two fundamental data structures.

Try to write it out or see [kT] for the solution.

BFS and DFS

The basic lemma about graph connectivity still holds for DFS.

Lemma There is a path from s to t if and only if visited[t] = true at the end.

The proof is the same as in BFS and is left as an exercise.

The lemma shows that DFS can also be used to check s-t connectivity, to find the connected component containing s, and to check graph connectivity, all in $O(m+n)$ time.

And we can also find all connected components in $O(m+n)$ time using DFS (exercise).

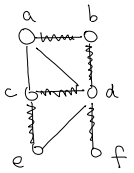
The main difference from BFS is that DFS cannot be used to compute the shortest path distances, and this is the main feature of BFS.

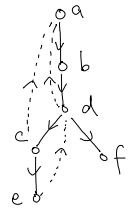
But as we shall see, DFS can be used to solve some interesting problems that BFS cannot do.

DFS Tree

As for BFS, we can construct a DFS tree to trace out the path from s.

Again, when a vertex v is first visited when we explore vertex u,
we say vertex u is the parent of vertex v.





By the same argument as in BFS, these edges (v, parent[v])
form a tree, and we can use them to find a path to s.
We call this a DFS tree of the graph.

Note that a graph could have many different DFS trees depending on the order of exploring the neighbors of vertices. The same can be said for BFS trees.

Definitions / Terminology for DFS trees

- The starting vertex s is regarded as the root of the DFS tree.
- A vertex u is called the parent of a vertex v if the edge uv is in the DFS tree, and u is closer to the root than v is to the root.
- A vertex u is called an ancestor of a vertex v if u is closer to the root than v, and u is on the path from v to the root.

  In this situation, we also say v is a descendant of vertex u.
- A non-tree edge uv is called a **back edge** if either u is an ancestor or descendant of v.

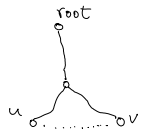
  It is called a back edge because this edge from the descendant to the ancestor.

In the above example, b is an ancestor of e and f, but c is neither an ancestor nor descendant of f.

The following is a simple but important property that we will use.

Property (back edges) In an undirected graph, all non-tree edges are back edges.

Proof Suppose by contradiction that there is an edge between u and v but
u and v are not an ancestor-descendant pair.



WLOG assume that u is visited before v.

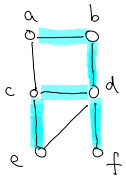
Then, since $uv \in E$, v will be explored before u is finished,
and thus u will be an ancestor of v, a contradiction. □

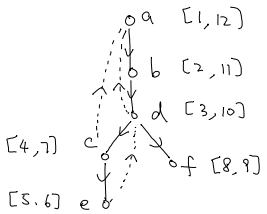
Starting Time and Finishing Time

We record the time when a vertex is first visited and the time when its exploring is finished.
These information will be very useful in design and analysis of algorithms.
To be precise, we include the pseudocode in the following.

```
[ Main program ]   visited[v] = false ∀ v∈V .   time = 1 .   visited[s] = true.   explore(s).

explore(u)        // recursive function explore.
    start[u] = time.     time ← time + 1.
    for each neighbor v of u
        if visited[v] = false
            visited[v] = true .    explore(v).
    finish[u] = time.    time ← time + 1.
```





Property (parenthesis) The intervals [start(u), finish(u)] and [start(v), finish(v)] for two vertices u and v are either disjoint or one is contained in another.

The latter case happens precisely when u, v are an ancestor-descendant pair.

Cut Vertices and Cut Edges

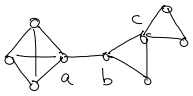
Suppose an undirected graph is connected.

We would like to identify vertices and edges that are critical in the graph connectedness.

A vertex v is a cut vertex (aka an articulation point, or a separating vertex) if G-v is not connected, i.e. removal of v and its incident edges disconnects the graph.

An edge e is a cut edge (aka a bridge) if G-e is not connected.

In the example, vertices a, b, e are cut vertices and edge ab is a cut edge.

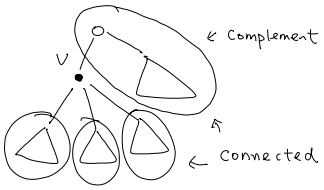


Observations and Ideas

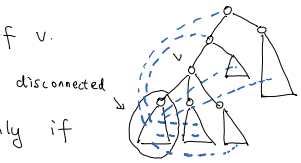
The idea is to use a DFS tree to identify all cut vertices and cut edges.

Consider a vertex v which is not the root. We would like to determine whether v is a cut vertex.

When we look at the DFS tree, all the subtrees below v are connected, as well as the complement of the subtree at v (see the picture).



The main observation is the property that all the non-tree edges are back edges (proved above), and so the only way for a subtree below v to be connected outside is to have edges going to an ancestor of v.



Claim A subtree T_i below v is a connected component in G-v if and only if there are no edges with one endpoint in T_i and another endpoint in a (strict) ancestor of v.

Proof ⇐) By the back edge property, all the non-tree edges are back edges, so there are no edges going to another subtree below v nor edges going to another subtree of the root.

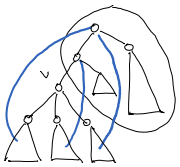
So, if there are no such edges going to a (strict) ancestor of v, then T_i must be a component in $G-v$.

⇒) On the other hand, if such edges exist, then T_i is connected to the complement even after v is removed, and so T_i won't be a connected component in $G-v$. □

The same argument applies to each subtree below v gives the following characterization of a cut vertex.

Lemma For a non-root vertex v in a DFS tree, v is a cut vertex if and only if there is a subtree below v with no edges going to a (strict) ancestor of v.

Proof ⇒) If every subtree below v has some edges going to an ancestor of v, then every subtree is connected to the complement. See picture.



So, $G-v$ is connected and thus v is not a cut vertex.

⇐) If some subtree T_i below v has no edges going to an ancestor of v, then T_i will be a connected component in $G-v$ by the previous claim. and thus v is a cut vertex. □

It remains to consider the root vertex of the DFS tree. The proof is left as an exercise.

Lemma For the root vertex v of a DFS tree, v is a cut vertex if and only if v has at least two children.

With these lemmas, we know how to determine if a vertex is a cut vertex by looking at a DFS tree

---

Algorithm

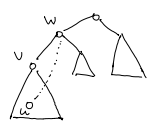
We are ready to use the above lemmas to design a $O(n+m)$ time algorithm to report all cut vertices. To have an effieient implementation, the idea is to process the vertices of a DFS following a bottom up ordering. and keep track of how "far up" the back edges of a subtree can go (i.e. how close to the root).

By the lemma, for a non-root vertex v, v is not a cut vertex if and only if all subtrees below v have an edge that goes above v.

What would be a good parameter to keep track of how far up we can go?

The starting time would be a good measure, because an ancestor always has an earlier / smaller starting time than its descendants.

(We could also do it in other ways, e.g. by recording the distance to the root instead.)



Let us define a value early[v] for each vertex on the DFS tree.

$$early[v] := \min \left\{ start[v],\ \min \{ start[w] \mid uw \text{ is a back edge with } u \text{ being a descendant of } v \text{ or } u = v \} \right\}$$

Informally, early[v] records how far up we can go from the subtree rooted at v.

We will be done if we can prove the following two things:

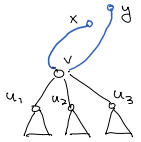
① We can compute early[v] for all $v \in V$ in $O(n+m)$ time.

② We can identify all cut vertices in $O(n+m)$ time using the early array.

For ①, we compute the early values from the leaves of the DFS tree to the root of the DFS tree.

The base case is when v is a left. Then we can compute early[v] by considering all the edges incident on v and taking the minimum of the starting time of the other endpoint. This takes $O(\deg(v))$ time.

By induction, suppose the early values of all children of v are computed.



Then, to compute early[v], we just need to take the minimum of the early value of its children, as well as the start time for all back edges involving v. This takes $O(\deg(v))$ time.

In the example given in the picture, $early[v] = \min \{ early[u_1], early[u_2], early[u_3], start[x], start[y] \}$.

It should be clear that early[v] is computed correctly, assuming the early values of all its children are correct, and so the correctness can be established by induction.

By this bottom-up ordering, every vertex on the tree is only processed once, and thus the total time complexity is $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$.

For ②, to check whether a non-root vertex v is a cut vertex, we just need to check whether $early[u_i] < start[v]$ for all children u_i of v.

If so, then v is not a cut vertex, as all subtrees below v have a back edge going above v.

Otherwise, if $early[u_i] \geq start[v]$, then the subtree rooted at u_i will be a connected component in G-v, and thus v is a cut vertex. These arguments are all covered in the first lemma.

The root vertex is handled using the other lemma.

This completes the description of a linear time algorithm to identify all cut vertices given the low array.

Exercise: Extend the algorithm to identify all the cut edges.

References: [DPV 3.2]. Cut vertices and cut edges are from the exercises of [DPV].

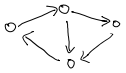
CS 341 - Algorithms, Winter 2023. University of Waterloo

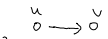
Lecture 7: Directed Graphs

We study directed graphs and what BFS/DFS can do in directed graphs.

A highlight is a very clever algorithm to identify all strongly connected components in linear time.

Directed Graphs

In a directed graph, each edge has a direction. 

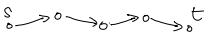
When we say uv is a directed edge, we mean the edge is pointing from u to v, 

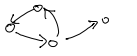
and u is called the <u>tail</u> and v is called the <u>head</u> of the edge.

Given a vertex v, $\text{indeg}(v)$ denotes the number of directed edges with v as the head and we call them the incoming edges to v. Similarly, $\text{outdeg}(v)$ denotes the number of directed edges with v as the tail and we call them the outgoing edges of v.

Directed graphs are useful in modeling asymmetric relations (e.g. web page links, one-way streets, etc).

We are interested in studying the connectivity properties of a directed graph.

We say t is <u>reachable</u> from s if there is a directed path from s to t. 

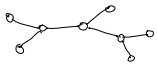
A directed graph is called <u>strongly connected</u> if for every pair of vertices $u, v \in V$, u is reachable from v and v is reachable from u.  strongly connected  not strongly connected

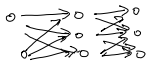
A subset $S \subseteq V$ is called strongly connected if for every pair of vertices $u, v \in S$, u is reachable from v and v is reachable from u.

A subset $S \subseteq V$ is called a <u>strongly connected component</u> if S is a maximally strongly connected subset, i.e. S is strongly connected but $S+v$ is not strongly connected for any $v \notin S$.  2 strongly connected components

A directed graph is a <u>directed acyclic graph</u> (DAG) if there are no directed cycles in it.

Note that a directed acyclic graph, unlike its undirected counterpart, could have many edges.

 undirected acyclic graph

 directed acyclic graph

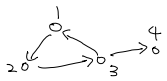
We are interested in designing algorithms to answer the following basic questions:

1. Is a given graph strongly connected?
2. Is a given graph directed acyclic?
3. Find all strongly connected components of a given directed graph.

As in undirected graphs, it will turn out that there are $O(n+m)$-time algorithms to solve

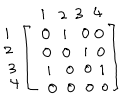
these problems, but they are not as easy as the algorithms for undirected graphs.

Graph Representations



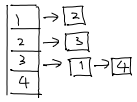
Both adjacency matrix and adjacency list can be defined for directed graphs.

In the adjacency matrix A, if ij is a directed edge, then $A_{ij} = 1$; otherwise $A_{ij} = 0$.



In the adjacency list, if ij is an edge, then j is on i's linked list.

As in undirected graphs, we will only use adjacency list in this part of the course, as only this allows us to design $O(n+m)$-time algorithms.



Reachability

Before studying the above questions, we first study a simpler question of checking reachability.

Given a directed graph and a vertex s, both DFS and BFS can be used to find all vertices reachable from s in $O(n+m)$ time.

Both BFS and DFS are defined as in for undirected graphs, except that we only explore out-neighbors.

DFS Algorithm

Input: A directed graph $G = (V, E)$ and a vertex s.

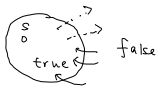
Output: All vertices reachable from s.

```
[ Main program ]   visited[v] = false  ∀ v∈V.   time = 1.   visited[s] = true.   explore(s).

explore(u)        // recursive function explore.
    start[u] = time.    time ← time + 1.
    for each out-neighbor v of u
        if visited[v] = false
            visited[v] = true.   explore(v).
    finish[u] = time.    time ← time + 1.
```

The time complexity is $O(n+m)$, and a vertex t is reachable from s if and only if visited[t] = true.

When we look at all vertices reachable from s, the subset form a "directed cut" with no outgoing edges (but could have incoming edges into the subset).



We leave as exercises in checking these claims. The proofs are the same as in undirected graphs.

We can define BFS for directed graphs analogously, by only exploring out-neighbors.

An important property of BFS is that it computes the shortest path distances from s to all other vertices. We leave it as an important exercise to check this claim.

BFS / DFS Trees

As in for undirected graphs, when a vertex v is first visited, we remember its parent as the vertex u when v is first visited from.

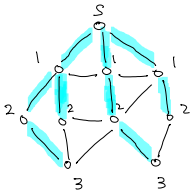
The edges (v, parent[v]) form a tree, and both BFS trees and DFS trees are defined in this way.

BFS trees

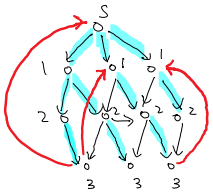
By setting dist[v] = dist[parent[v]] + 1, we compute all shortest path distances from s.

In undirected graphs, for all non-tree edges uv, $dist[v] - 1 \le dist[u] \le dist[v] + 1$.

In directed graphs, there could be non-tree edges uv with large difference between dist[u] and dist[v], but in this case it must be dist[u] > dist[v] as they must be "backward edges".



undirected BFS tree

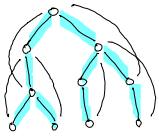


directed BFS tree

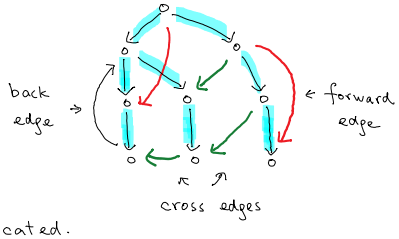
DFS trees

In undirected graphs, all non-tree edges are back edges (see L06).

In directed graphs, some non-tree edges are "cross edges" and "forward edges".



undirected DFS tree



directed DFS tree

Structured in directed graphs are more complicated.

Strongly Connected Graphs

We are ready to consider the first problem of checking whether a directed graph is strongly connected.

From the definition, we need to check $\Omega(n^2)$ pairs and see if there is a directed path between them.

In undirected graphs, it is enough to pick an arbitrary vertex s, and check whether all vertices are reachable from s. So we just check reachability for O(n) pairs.

What would be a corresponding "succinct" condition to check in directed graphs?

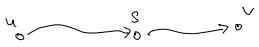
It is easy to find examples for which just checking reachability from s is not enough.

Checking reachability from every vertex would work, but it would take $\Omega(n(n+m))$ time, too slow.

The following observation allows us to reduce the number of pairs to check to $O(n)$.

Observation G is strongly connected if and only if every vertex v is reachable from s and s is reachable from every vertex v, where s is an arbitrary vertex.

Proof $\Rightarrow$) is trivial by the definition of a strongly connected graph.

$\Leftarrow$) For any u, v, by combining a path from u to s and a path from s to v, we obtain a path from u to v, so G is strongly connected. 

□

We know how to check whether all vertices are reachable from s in $O(n+m)$ time by BFS or DFS.

How do we check whether s is reachable from all vertices efficiently?

There is a simple trick to do it - by reversing the direction of the edges.

Claim Given G, we reverse the direction of all the edges to obtain G^R.

There is a directed path from v to s in G iff there is a directed path from s to v in G^R.

So, s is reachable from all vertices in G iff every vertex is reachable from s in G^R.

With this claim, we can check whether s is reachable from every vertex in G by doing one BFS / DFS in G^R from s.

To summarize, we have the following algorithm.

Algorithm (strong connectivity)

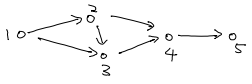
1. Check whether all vertices in G are reachable from s by one BFS/DFS.
2. Reverse the direction of all the edges in G to obtain G^R.
3. Check whether all vertices in G^R are reachable from s by one BFS/DFS.
4. If both yes, return "strongly connected"; otherwise return "not strongly connected".

The correctness of the algorithm follows from the observation and the claim above.

The time complexity is $O(n+m)$ time.

We leave it as a simple exercise to construct G^R in linear time.

Directed Acyclic Graphs

Directed acyclic graphs are directed graphs without directed cycles. 

They are useful in modeling dependency relations (e.g. course prerequisities. software installation).

In such situations, it would be useful to find an ordering of the vertices so that all the edges go forward.

This is called a <u>topological ordering</u> of the vertices (e.g. an ordering to take the courses).

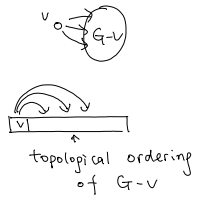
<u>Proposition</u> A directed graph is acyclic if and only if there is a topological ordering of the vertices.

<u>Proof</u> $\Leftarrow$) Since all the directed edges go forward, there are no directed cycles.

$\Rightarrow$) We will prove that any directed acyclic graph has a vertex v of indegree zero.

Then, we can put v as the first vertex in the ordering, and we consider $G-v$.

Since $G-v$ is also acyclic, there is a topological ordering of $G-v$ by induction on the number of vertices, and we are done.



It remains to argue that every directed acyclic graph has a vertex of zero indegree.

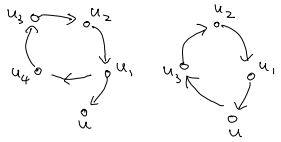
Suppose by contradiction that every vertex has indegree at least one.

Then, we start from an arbitrary vertex u, and go to an in-neighbor u_1 of u, and then go to an in-neighbor u_2 of u_1, and so on.

It is always possible since every vertex has in-degree at least one.

If some in-neighbor repeats, then we find a directed cycle, a contradiction.

But it must repeat at some point, since the graph is finite. □



There are at least two good approaches to find a topological ordering of a directed acyclic graph efficiently.

<u>Approach 1</u> (Sketch) Just follow the procedure in the above proof.

That is, keep finding a vertex of indegree zero in the remaining graph and put it in the beginning of the ordering.

We leave it as a problem for you to implement this algorithm in $O(n+m)$ time.

<u>Approach 2</u> This is perhaps less intuitive, but the ideas will be useful in the next section as well.

The idea is to do a DFS on the whole graph (i.e. start a DFS on an arbitrary vertex, but if not all vertices are visited, start a DFS on an unvisited vertex, and so on, until all vertices are visited, just like what we would do in finding all connected components of an undirected graph).

Note that this DFS can be done in any ordering of vertices. In particular, we don't need to start at a vertex of indegree zero nor do we need any information about a topological ordering.

In the following proof, we use that the parenthesis property of starting and finishing time holds

for directed graphs as well. Please check.

Lemma If G is directed acyclic, then for any directed edge uv, finish[v] < finish[u] for any DFS.

Proof We consider two cases.

Case 1: start[v] < start[u].

Since the graph is acyclic, u is not reachable from v.

So, u cannot be a descendant of v.

By the parenthesis property, the intervals [start[v], finish[v]] and [start[u], finish[u]] must be disjoint.

The only possibility left is start[v] < finish[v] < start[u] < finish[u], proving the lemma in this case.

Case 2: start[u] < start[v].

Then, since v is unvisited when is started and uv is an edge, v will be a descendant of u in the DFS tree. This is the same as the argument used in the back edge property in L06.

By the parenthesis property, we have start[u] < start[v] < finish[v] < finish[u]. □

Using the lemma, we have the following simple algorithm for computing a topological ordering.

Algorithm (Topological Ordering / Directed Acyclic Graphs)

1. Run DFS on the whole graph.
2. Output the ordering with decreasing finishing time.
3. Check if it is a topological ordering. If not, return "not acyclic".

Correctness: The lemma proves that if the graph is acyclic, then all edges go forward in this ordering.

On the other hand, if the graph is not acyclic, then there is no topological ordering by the proposition.

Time complexity: The algorithm can be implemented in O(n+m) time.

Note that we don't need to do sorting for the second step.

Just put a vertex in a queue when it is finished, by adding one line in the code.

Strongly Connected Components (SCC)

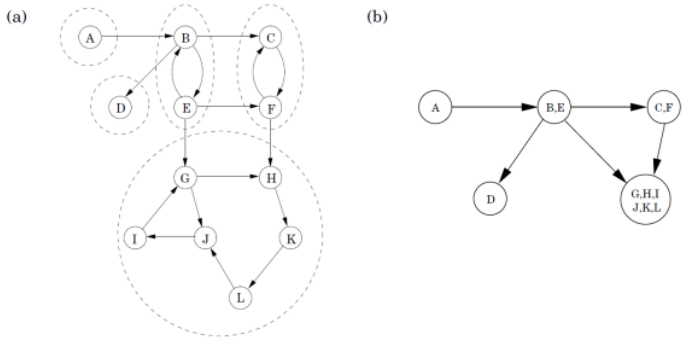
Finally, we consider the more difficult problem of finding all strongly connected components.

We will combine and extend the previous ideas to obtain an O(n+m) time algorithm.

First, let's get a good idea about how a general directed graph looks like.

Observation: Two strongly connected components are vertex disjoint. If two strongly connected components

Observation: Two strongly connected components are vertex disjoint. If two strongly connected components C_1 and C_2 share a vertex, then $C_1 \cup C_2$ is also strongly connected, contradicting maximality of C_1, C_2.



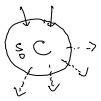
picture from [DPV 3.4]

In the picture, when every strongly connected component is "contracted" into a single vertex, then the resulting directed graph is acyclic.

This is true in general. We leave the proof as an exercise.

So, a general directed graph is a directed acyclic graph on its strongly connected components.

Idea 1: Suppose we start a DFS/BFS in a "sink component" C (a component with no outgoing edges), then we can identify the strongly connected component C.



This is because every vertex in C is reachable from the starting vertex, but no vertices outside. So, just read off vertices with visited[v] = true will identify C.

This suggests the following strategy.

1. Find a vertex v in a sink component C.
2. Do a DFS/BFS to identify C.
3. Remove C from the graph and repeat.

So, now, the question is how to find a vertex in a sink component efficiently?

It doesn't look easy. Can you think of how?

Idea 2: Do "topological sort".

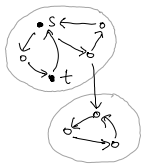
As discussed above, if each strong component is "contracted" to a single vertex, the resulting graph is acyclic. From the previous section about directed acyclic graphs, we know that if we do a DFS on the whole graph, the node with the earliest finishing time is a sink.

This suggests the following strategy.

1. Run DFS on the whole graph and obtain an ordering in increasing finishing time.

2. Use this ordering in the previous strategy in idea 1 to take out one sink component at a time.

This is a very nice strategy, but unfortunately it doesn't work.

For a counterexample, consider the graph 

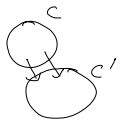
, if we start the DFS at s, then node t has the earliest finishing time, but t is not in a sink component.

Idea 3: The natural strategy doesn't work, but a modification of it may still work.

The observation is that the DFS ordering still gives us useful information about a topological ordering of components.

In particular, although we couldn't say that a vertex with smallest finishing time is in a sink component, we can say that a vertex with the largest finishing time is in a source component.

The proof of the following lemma is similar to the proof of the lemma in topological ordering.

Lemma If C and C' are strong components and there are edges from C to C', then the largest finishing time in C is bigger than the largest finishing time in C'. 

Proof Again, we consider two cases.

Case 1: The first vertex v visited in $C \cup C'$ is in C'.

Note that vertices in C are not reachable from v but all vertices in C' are reachable from v.

By the time when v is finished, all vertices in C' are finished, while all vertices in C haven't started.

Case 2: The first vertex v visited in $C \cup C'$ is in C.

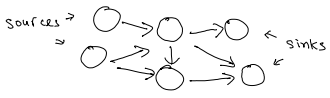
Since vertices in $C \cup C'$ are reachable from v, all vertices in $C \cup C'$ will be finished before v is finished, and so $v \in C$ will have the largest finishing time in $C \cup C'$. □

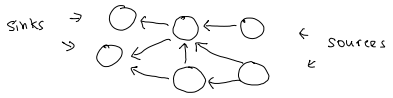
With this lemma, we know that if we first do a DFS and order the vertices in decreasing order finishing time, and then do a DFS again using this ordering, then we will visit "ancestor components" before we visit "descendant components".

But this is not what we want, as we want to start in a sink component and cut it out first.

Idea 4: Reverse the graph so that sources become sinks!

First, observe that the strong components in G are the same as the strong components in G^R.

G 

$\Rightarrow$ G^R 

Very important for us, source components in G become sink components in G^R and vice versa.

Therefore, the ordering we have in G following a topological ordering of the components from

sources to sinks becomes an ordering in G^R following a topological ordering of the components from sinks to sources.

Now, we can just follow this ordering to do the DFS in G^R to cut out sink components one at a time, as we wished in idea 1 and idea 2.

Finally, we can summarize the algorithm.

Algorithm (Strong Components)

1. Run DFS on the whole graph G using an arbitrary ordering of vertices.
2. Order the vertices in decreasing order of finishing times obtained in step 1.
3. Reverse the graph G to obtain the graph G^R
4. Follow the ordering in step 2 to explore the graph G^R to cut out the components one at a time.

To be more precise, we expand step 4 in more details.

(i) Let i be the vertex of i-th largest finishing time in step 2.

(ii) Let $c = 1$. // It is a variable counting the number of strong connected components.

(iii) For $1 \leq i \leq n$ do

```
If visited[i] = false
    DFS(G^R, i)
    Mark all the vertices reachable from i in G^R in this iteration to be in component c.
    c ← c+1
```

The proof of correctness follows from our long discussion.

We leave it to the reader that all the steps can be implemented in $O(n+m)$ time.

This algorithm is very clever and it may take more time to fully understand it.

Reference : [DPV 3.3-3.4]

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 8: Greedy Algorithms

We start our study of greedy algorithm using scheduling problems as examples.

Interval Scheduling [KT 4.1]

Input: n intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$

Output: a maximum set of disjoint intervals.

For example, in  , the highlighted intervals form a maximum set of disjoint intervals. There are multiple optimal solutions in this example.

For this problem, we can imagine that we have a room, and there are people who like to book our room and they tell us the time interval that they need the room, and our objective is to choose a maximum subset of activities with no time conflicts.

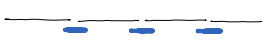
Generally speaking, greedy algorithms work by using simple and/or local rules to make decisions and commit on them. An analogy is to make maximum profit in short term.

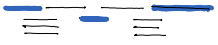
There are multiple natural greedy strategies for this problem including

- earliest starting time (choose the interval with $\min_i s_i$)
- earliest finishing time (choose the interval with $\min_i f_i$)
- shortest interval (choose the interval with $\min_i f_i - s_i$)
- minimum conflicts (choose the interval that overlaps with the minimum number of other intervals)

It turns that only one of these strategies would work.

Earliest starting time is the easiest to find counterexamples, because the interval with earliest starting time could be very long, e.g.  .

It is not difficult to find counterexamples for shortest intervals, e.g. 

It is a bit harder to find counterexamples for minimum conflicts, e.g. 

There are no counterexamples for earliest finishing time.

The intuition is that we leave maximum space for future intervals.

But how could we argue that this will always give an optimal solution (that there are no

solutions doing better)?

Algorithm

- Sort the intervals so that $f_1 \le f_2 \le \dots \le f_n$. Current solution $S = \phi$.
- For $1 \le i \le n$ do

  if interval i $[s_i, f_i]$ has no conflicts with other intervals in S, add i to S.
- Return S.

Correctness

The idea is to argue that any (optimal) solution would do no worse by using the interval with earliest finishing time.

Let $[s_{i_1}, f_{i_1}] < [s_{i_2}, f_{i_2}] \dots < [s_{i_k}, f_{i_k}]$ be the solution returned by the greedy algorithm.

Let $[s_{j_1}, f_{j_1}] < [s_{j_2}, f_{j_2}] \dots < [s_{j_\ell}, f_{j_\ell}]$ be an optimal solution with $\ell \ge k$.

Since $f_{i_1} \le f_{j_1} < s_{j_2}$ (the first inequality is because $i_1 = 1$ and is the interval with earliest finishing time, and the second inequality is because $[s_{j_1}, f_{j_1}]$ and $[s_{j_2}, f_{j_2}]$ are disjoint), so $[s_{i_1}, f_{i_1}] < [s_{j_2}, f_{j_2}] < \dots < [s_{j_\ell}, f_{j_\ell}]$ is still an optimal solution.

Thus we have the following claim, showing that it is no worse by choosing $[s_1, f_1]$.

Claim There exists an optimal solution with $[s_1, f_1]$ chosen.

We will use this argument inductively to prove the following lemma.

Lemma $[s_{i_1}, f_{i_1}], [s_{i_2}, f_{i_2}], \dots [s_{i_k}, f_{i_k}] [s_{j_{k+1}}, f_{j_{k+1}}] \dots [s_{j_\ell}, f_{j_\ell}]$ is an optimal solution with $f_{i_k} \le f_{j_k}$

Informally, the lemma says that the greedy solution always "stays ahead".

Before we prove the lemma, let's see how it implies that the greedy solution is optimal.

Suppose, by contradiction, that the greedy solution is not optimal, i.e. $\ell > k$.

Since $f_{i_k} \le f_{j_k} < s_{j_{k+1}}$, we see that the interval $[s_{j_{k+1}}, f_{j_{k+1}}]$ has no overlapping with the greedy solution, and it should have been added to the greedy solution by the greedy algorithm, a contradiction that the greedy algorithms only finds k disjoint intervals.

Proof of lemma The base case holds because of the previous claim.

Assume the claim is true for $c \ge 1$, and we are to prove the inductive step.

Since $[s_{i_1}, f_{i_1}] \dots [s_{i_c}, f_{i_c}] [s_{j_{c+1}}, f_{j_{c+1}}) \dots [s_{j_\ell}, f_{j_\ell}]$ is an optimal solution by the induction hypothesis with $f_{i_c} \le f_{j_c}$, we have $f_{i_{c+1}} \le f_{j_{c+1}}$ as the interval

$[s_{j_{c+1}}, f_{j_{c+1}}]$ has no overlapping with $[s_1, f_1], \ldots, [s_{i_c}, f_{i_c}]$ and the greedy algorithm chooses the next interval with earliest finishing time.

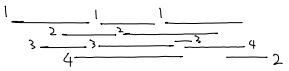
As $f_{i_{c+1}} \le f_{j_{c+1}}$, by replacing $[s_{j_{c+1}}, f_{j_{c+1}}]$ with $[s_{i_{c+1}}, f_{i_{c+1}}]$, the resulting solution $[s_{i_1}, f_{i_1}] \ldots [s_{i_{c+1}}, f_{i_{c+1}}] [s_{j_{c+2}}, f_{j_{c+2}}] \ldots [s_{j_k}, f_{j_k}]$ is feasible and optimal. □

Time complexity: It is easy to check that the greedy algorithm can be implemented in $O(n \log n)$ time.

Interval Coloring

Input: n intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$

Output: use the minimum number of colors to color the intervals, so that each interval gets one color and two overlapping intervals get two different colors.



(the numbers are the colors)

We can imagine this is the problem of using the minimum number of rooms (colors) to schedule all the activities (intervals).

One natural greedy algorithm is to use the previous algorithm to choose a maximum subset of disjoint intervals and use only one color for them, and repeat.

Find a counterexample for this algorithm!

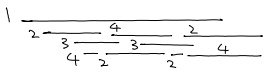
There is another greedy algorithm that will work.

Algorithm (interval coloring)

- Sort the intervals by starting time so that $s_1 < s_2 < \ldots < s_n$.
- for $1 \le i \le n$ do

  use the minimum available color c_i to color the interval i.

  (i.e. use the minimum number to color the interval i so that it doesn't conflict with the colors of the intervals that are already colored.)



Suppose the algorithm uses k colors.

To prove the correctness of the algorithm, we must prove that there are no other ways to color the intervals using at most $k-1$ colors.

How do we argue that ?

A nice and simple way is to show that when the algorithm uses k colors, there exists a time t such that it is contained in k intervals.

Since these k intervals are pairwise overlapping, we need at least k colors just to color them, and therefore a (k-1)-coloring doesn't exist.

Proof of correctness: Suppose the algorithm uses k colors.

Let interval ℓ be the first interval to use color k.

This implies that interval ℓ overlaps with intervals with colors $1, \ldots, k-1$.

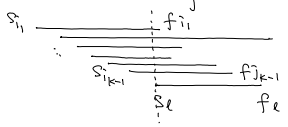
Call them $[s_{i_1}, f_{i_1}], \ldots, [s_{i_{k-1}}, f_{i_{k-1}}]$.

As we sort the intervals with increasing order of starting time, we have $s_{i_j} \le s_\ell$ for all $1 \le j \le k-1$.

Since all these intervals overlap with $[s_\ell, f_\ell]$, we also have $s_\ell \le f_{i_j} \quad \forall \; 1 \le j \le k-1$.

Therefore, s_ℓ is a time contained in k intervals.

This implies that there is no k-1 coloring. □



Find a counterexample showing that using an arbitrary ordering of the intervals would not work.

Minimizing Total Completion Time

Input: n jobs, each requiring processing time p_i

Output: An ordering of the jobs to finish so as to minimize the total completion time.

(The completion time of a job is defined as the time when it is finished.)

For example, given four jobs with processing times 3, 4, 6, 7, and if we process the jobs in this order, then the completion times are 3, 7, 13, 20, and the total completion time is 43.

It is very intuitive that we should process the jobs in increasing order of processing time.

(Imagine we are in a supermarket with four customers with 3, 4, 6, 7 items.)

But how do we argue that there are no better solutions?

Here we use an "exchange argument" to show that the greedy solution is optimal.

(Again, we imagine that we have 1 item but the customer in front of us has 10 items.)

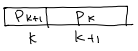
Proof of correctness

Consider a solution which is not sorted by non-decreasing processing time.

This implies that there is an "inversion pair": the i-th job and the j-th job with $i<j$ but $p_i > p_j$.

This implies that there exists $i \le k < j$ with $p_k > p_{k+1}$.

By swapping the jobs k and k+1, we will prove that the total completion time is smaller.

Note that all the completion times, except for k and k+1, are unchanged. (swap) ···· 

Let C be the completion time of job k-1.

Before swapping, the completion times of job k and k+1 are $C+p_k$ and $C+p_k+p_{k+1}$ respectively. and the total of these two jobs are $2C + 2p_k + p_{k+1}$.

After swapping, the total of these two jobs are $2C + 2p_{k+1} + p_k$.

Since $p_{k+1} < p_k$, it is clear that the solution after swapping is better.

So, any ordering which is not sorted by non-decreasing processing times is not optimal. □

The exchange argument is quite nice and useful.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 9: Single Source Shortest Paths

We study Dijkstra's algorithm that computes shortest paths from a single vertex to all vertices, on graphs with non-negative edge lengths.

Shortest Paths

Input: A directed graph $G=(V,E)$, with a non-negative length l_e for each edge $e \in E$, and two vertices $s,t \in V$.

Output: A shortest path from s to t, where the length of a path is equal to the sum of edge lengths.

We can think of the graph as a road network, and the edge length represents the time needed to drive pass the road (may depend on traffic).

Then the problem is to find a fastest way to drive from point s to point t, which is the type of queries that Google Maps answers every day.

It will be more convenient to solve a more general problem.

Input: A directed graph $G=(V,E)$, with a non-negative length l_e for each edge $e \in E$, and a vertex $s \in V$.

Output: A shortest path from s to v, for every vertex $v \in V$.

This problem seems to be harder, because each path could have $\Omega(n)$ edges, and so the output size could already be $\Omega(n^2)$.

But it will turn out that there is a succinct representation of these paths and this single source shortest path problem can be solved in the same time complexity as the shortest s-t path problem.

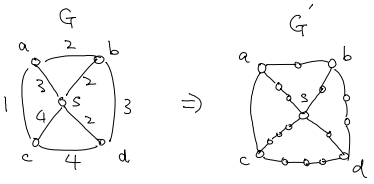
Breadth First Search and Dijkstra's Algorithm

We have seen in LOS that BFS can be used to solve the single-source shortest paths problem, when every edge has the same length (so we count the number of edges on the paths).

It is not difficult to reduce the non-negative edge length problem to this same-length special case.

Let's say all the edge lengths are positive integers.

We can reduce our problem to the special case by replacing each edge of length l_e by a path of l_e edges.



Then there is an s-t path of length k in G iff there is an s-t path of length k in G'.

And then we can solve the problem in G' by simply doing BFS starting from s.

The only problem of this reduction is that it may not be efficient as G' may have many more vertices. The number of vertices in G' is $n + \sum_{e \in E} (l_e - 1)$, so when l_e is large then G' has many more vertices. And then a linear time algorithm for G' does not correspond to a linear time algorithm for G.

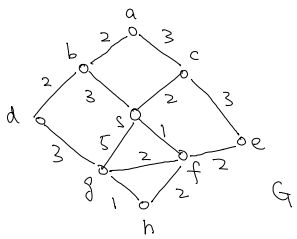
Physical Process

To design an efficient algorithm through this reduction, instead of constructing G' explicitly and run BFS on it, we just keep G' in mind and try to simulate BFS on G' efficiently.

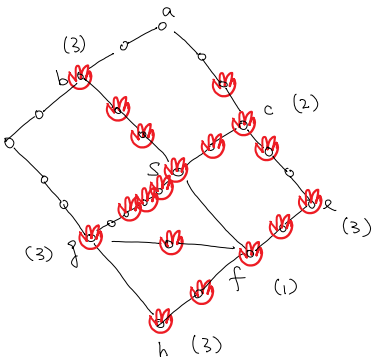
We can think of the process of doing BFS on G' as follows.

- We start a fire at vertex s at time 0. The fire will spread out.
- It takes one unit of time to burn an edge in G'.

Then the shortest path distance from s to t is just the first time when vertex t is burned.



G

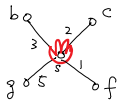


G' at time $t=3$

To simulate this process efficiently, the idea is that we just need to be able to keep finding out what is the next vertex to be burned and when (rather than simulating it faithfully on the paths).

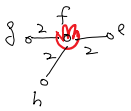
Initially, it is the source vertex s.

Then, knowing that s is burned at time 0, in the example above, we know that b, c, g, f will be burned in <u>at most</u> time $3, 2, 5, 1$ respectively.



Then, the next vertex to be burned is vertex f.

Knowing that f is burned at time 1, we know that g, h, e will all be burned in <u>at most</u> 2 time units (by time step 3) through the edges fg, fh, fe.



In the above example, we update the <u>upper bound</u> on g to be burned to be 3 (replacing the initial value of 5 which was set when s was burned), as vertex g will be burned earlier through the fire from g than the fire from s.

Then, with this updated information after g is burned, we find out the next vertex to be burned (in this example c) and then update the upper bound on the neighbors of c as the fire can now go from c to its neighbors.

Repeating this (i.e. find out next vertex, update upper bound on neighbors) gives us an efficient simulation of the process of G'. and this is exactly Dijkstra's algorithm.

Dijkstra's Algorithm (or BFS Simulation)

```
dist(v) = ∞ for every v ∈ V      // the initial upper bound on the time v is burned
dist(s) = 0                      // start a fire at vertex s at time 0
Q = make-priority-queue(V)       // all vertices are put in the priority queue,
                                 //   using dist(v) as the key value (priority value) of v
While Q is not empty do
    u = delete-min(Q)            // dequeue the vertex with minimum dist() value
                                 //   i.e. find out the next vertex to be burned
    for each (out-)neighbor v of u   // it works for directed graphs as well
        if dist(u) + l_uv < dist(v)  // find a shorter way to get to v through u
            dist(v) = dist(u) + l_uv
            decrease-key(Q, v)
            parent(v) = u        // note that this may be updated multiple times
```

Dijkstra's algorithm is very similar to that of the BFS algorithm, except that we use a priority queue (which can be implemented by a min-heap) to replace a queue.

Correctness

I hope it is clear that the algorithm is an efficient simulation of BFS in G'.

Also we argued before that shortest paths from s in G' are shortest paths from s in G.

So, the correctness of this algorithm in G just follows from the correctness of using BFS to solve shortest paths in G' (where every edge is of the same length), which we proved in L05.

Anyway, we will do a more traditional and formal proof as well.

Time Complexity

Each vertex is enqueued once (in the beginning) and dequeued once (when it is burned).

When a vertex is dequeued, we check every edge uv once and may use the value $dist(u) + l_{uv}$ to update the value of $dist(v)$ using a decrease-key operation.

All the enqueue, dequeue, decrease-key operations can be implemented in $O(\log n)$ time by a min-heap, and thus the total time complexity is $O\left(\left(n + \sum_v outdeg(v)\right)\log n\right) = O((n+m)\log n)$ time.

So, the cost of simulating BFS in the big graph is just an extra factor of $\log n$.

Take a look at [DPV 4.5] or [CLRS] on background of priority queues.

With more advanced data structures (namely Fibonacci heaps), the runtime could be improved to $O(n \log n + m)$. See [CLRS] for Fibonacci heaps.

Analysis

Here we present a more traditional approach to prove the correctness of Dijkstra's algorithm.

Besides being more formal, the proof technique is also useful in analyzing other problems, e.g. MST.

The algorithm will turn out to be the same, but the way of thinking about it is slightly different.

This is also the way we think of Dijkstra's algorithm as a greedy algorithm.

The idea is to grow a subset $R \subseteq V$ so that $dist(u)$ for $u \in R$ are computed correctly.

Initially, $R = \{s\}$, and then at each iteration we add one more vertex to R.

Which vertex to be added in an iteration?

We will add the vertex which is closest to R, so in this sense we are growing R greedily.

Algorithm

$dist(v) = \infty \quad \forall v \in V. \quad dist(s) = 0.$

$R = \phi.$

While $R \neq V$ do

pick the vertex $u \notin R$ with smallest $dist(u)$. $R \leftarrow R \cup \{u\}$.

for each edge $uv \in E$

if $dist(u) + \ell_{uv} < dist(v)$

$dist(v) = dist(u) + \ell_{uv}$.

Equivalently, $u \notin R$ is the vertex with $dist(u) = \min_{v \notin R, w \in R} \{ dist(w) + \ell_{wv} \}$.

We leave it as a simple exercise that the two versions of Dijkstra's algorithm are equivalent.

Induction

We prove the correctness by induction, maintaining the following invariant.

Invariant: For any $v \in R$, $dist(v)$ is the shortest path distance from s to v.

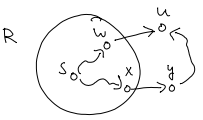
Base case: It is true when $R = \{s\}$.

Induction step: Assume that the invariant holds for the current R.

We would like to prove that the invariant remains true after a new vertex u is added to R.

We need to argue that $dist(u) = \min_{v \in R, w \in R} \{ dist(w) + l_{wv} \}$ is the shortest path distance from s to u.

Let $w \in R$ be the vertex in a minimizer, i.e.



$dist(u) = dist(w) + l_{wu} \leq dist(a) + l_{ab} \quad \forall a \in R$ and $b \notin R$.

Since $dist(w)$ is the shortest path distance from s to w, the shortest path distance from s to u is at most $dist(w) + l_{wu}$ (i.e. using the path from s to w and edge wu).

So, it remains to prove that the shortest path distance from s to u is at least $dist(w) + l_{wu}$.

Consider any path P from s to u.

Since $s \in R$ and $u \notin R$, there must be an edge xy in P such that $x \in R$ and $y \notin R$.

(Note that x could be w and/or y could be u.)

The length of path P is at least $dist(x) + l_{xy}$, as $dist(x)$ is computed correctly by the invariant.

* Here, observe that we crucially use the fact that the length of each edge is non-negative.

But we know that $dist(w) + l_{wu} \leq dist(x) + l_{xy}$ as (w, u) is a minimizer.

Therefore, we conclude that $length(P) \geq dist(x) + l_{xy} \geq dist(w) + l_{wu} = dist(u)$.

This inequality is true for any path from s to u, and thus the shortest path distance from s to u is at least $dist(u)$. □

Note that the proof applies to directed graphs as is.

Shortest Path Tree

Now, we think about how to store the shortest paths from s for all $v \in V$.

From the proof, when a vertex v is added to R, any vertex u that minimizes $dist(u) + l_{uv}$ is the parent on a shortest path from s to v.

We keep track of this parent information throughout Dijkstra's algorithm, by always keeping a vertex that attains the minimum of $dist(u) + l_{uv}$ for $u \in R$ as the current parent, and will update whenew vertex is put in R and makes this value smaller.

After the algorithm finished (i.e. $R = V$), by tracing the parents until we reach the source s, we can find a shortest path from s to v.

Every vertex can do the same to find a shortest path from s.

As in BFS tree, the edges $(v, parent[v])$ form a tree. The proof is left as an exercise.

So, we have a succinct way to store all the shortest paths from s, using only $n-1$ edges

to store n paths.

This shortest path tree is very useful.

Question: Can you see clearly how the algorithm will fail if there are some negative edge lengths?

We will come back to this more difficult setting when we study dynamic programming algorithms.

References: [KT 44], [DPV 44]

CS 341 - Algorithms, Winter 2023. University of Waterloo

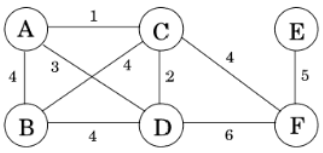
Lecture 10: Minimum Spanning Trees

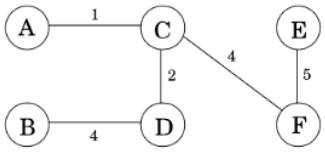
We design and analyze greedy algorithms for the classical problem of finding a minimum spanning tree. This is a good example where we first observe some mathematical property of the problem, then use it to come up with (different) algorithms. then implement them efficiently using data structures.

Minimum Spanning Trees

Input: An undirected graph $G=(V,E)$ with a cost $c_e>0$ on each edge $e \in E$.

Output: A minimum-cost spanning tree T, where the cost of T is defined as $\sum_{e \in T} c_e$.





[DPV, chapter 5]

This is a basic network optimization problem where the goal is to find a cheapest way to build a subgraph that connects all the vertices of a graph.

When the cost on the edges are positive, any optimal solution must be a spanning tree because of the following simple property.

Property Removing an edge of a cycle in a connected graph cannot disconnect the graph.

Cheapest Edge

The most tempting greedy strategy is to choose an edge of minimum cost.

Could it go wrong? The following claim shows that it can't go wrong.

The proof is by a simple exchange argument.

Claim 1 There is a minimum spanning tree that contains a cheapest edge.

Proof Let $e=uv$ be a cheapest edge, and T be a minimum spanning tree. If $e \in T$, then we are done. So suppose $e \notin T$. Consider $T+e$.

Since there is a (unique) path connecting u and v in T, there is a (unique) cycle containing e in $T+e$. Let $f \neq e$ be an edge in this cycle, then $T+e-f$ is connected (by the property) and a spanning tree. As e is a cheapest edge, $\text{cost}(T+e-f) \leq \text{cost}(T)$ and so $T+e-f$ is also a minimum spanning tree, and $T+e-f$ contains e, thus proving the claim. □

Actually, this simple claim alone already leads to a polynomial time algorithm for finding a MST. The idea is to find a cheapest edge $e=uv$, then "contract" its two endpoints to obtain a graph with one fewer vertices, and recursively find a MST T' using the same procedure (i.e. repeatedly find a cheapest edge and contract it), and return $T'+e$ as a MST in the original graph.

We leave it to the reader to prove the correctness of this algorithm formally.

The reason that we don't do this is that directly implementing this idea doesn't give as fast an algorithm. We will see that the Kruskal's algorithm that we will present later is essentially the same algorithm.

Cheapest Edge on a Vertex

The above claim is not as easy to use because it only concerns the cheapest edges of the whole graph. Let's generalize the idea so that it can be applied more broadly to other edges.

Claim 2 For each vertex v, there is a minimum spanning tree containing a cheapest edge incident on v.

We won't prove this claim as we will prove a more general claim and also the proof is similar to Claim 1.

This claim leads to a fast MST algorithm, called the Borůvka's algorithm, the first MST algorithm according to wikipedia.

Assume the edge costs are distinct. The idea is to add the cheapest edge incident to each vertex to the (partial) solution.

This will form a forest F where each vertex is of degree at least one, and hence at least $n/2$ edges.

Then, we can "contract" each component in this forest F into a single vertex, recursively applying the same procedure to find a MST T' in the contracted graph, and return $T' \cup F$ as a MST.

This algorithm can be implemented in $O(m \log n)$ time, as each iteration can be implemented in $O(m+n)$ time to find and contract the forest, and there are at most $O(\log n)$ iterations as each iteration we decrease the graph size by at least half.

We leave the correctness proof and the time complexity analysis to the reader, as the correctness proof also follows from a more general claim that we will prove later.

Cheapest Edge of a Cut

We prove a generalization of Claim 2 that is even more flexible to use.

A subset of vertices $\phi \neq S \subset V$ defines a bipartition of the vertex set $(S, V-S)$.

We say the set of edges with one endpoint in S and one endpoint in $V-S$ the cut of S.

denoted by $\delta(S) := \{ uv \in E \mid u \in S, v \notin S \}$.

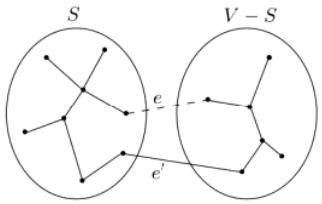
Claim 3 For every subset $\phi \neq S \subset V$, there is a minimum spanning tree containing a cheapest edge in $\delta(S)$.

Note that Claim 2 is a special case where $S = \{v\}$ is a singleton.

We also won't prove Claim 3, but a slightly more general version that is more useful in analyzing algorithms.

The Cut Property Suppose edges in $F \subseteq E$ are part of a MST of $G = (V, E)$. For any $\phi \neq S \subset V$ with $\delta(S) \cap F = \phi$ (i.e. no edges in F are in the cut of S), let e be a cheapest edge in $\delta(S)$, $F + e$ is part of some MST.

Proof The proof is by a simple exchange argument.



Let T be a MST that contains the edges in F.

If $e \in T$, then we are done. So assume $e \notin T$. Consider $T + e$.

There is a unique path P connecting the two endpoints of e in T, one endpoint in S and the other endpoint in $V - S$.

Therefore, there must exist an edge f in P with one endpoint in S and one endpoint in $V - S$.

That is, $f \in \delta(S)$. Since e is a cheapest edge in $\delta(S)$, it holds that $\text{cost}(e) \leq \text{cost}(f)$.

This implies that $T' = T + e - f$ is also a MST in G.

Finally, note that $F + e \subseteq T'$ as $F \subseteq T$ and $f \notin F$ because $f \in \delta(S)$ but $F \cap \delta(S) = \phi$. □

Algorithms

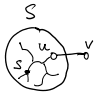
With the cut property, we can prove that several algorithms work to find a minimum spanning tree.

Prim's algorithm

The idea of Prim's algorithm is to start from an arbitrary vertex s and to grow the component containing s one vertex at a time.

$F := \phi$, $S = \{s\}$.

while $S \neq V$ do



let $e = uv$ be a cheapest edge in $\delta(S)$ with $u \in S$ and $v \notin S$.

$F \leftarrow F + \{e\}$, $S \leftarrow S + \{v\}$.

return F

The correctness of Prim's algorithm follows from repeatedly applying the cut property.

We will discuss how to implement Prim's algorithm efficiently in the next section.

Kruskal's algorithm

The idea of Kruskal's algorithm is to consider the edges from cheapest to most expensive, and add an edge to the solution as long as it doesn't create a cycle.

$F := \emptyset$.

Sort the edges in non-decreasing cost so that $cost(e_1) \leq cost(e_2) \leq \cdots \leq cost(e_m)$.

for $1 \leq i \leq m$ do

    if $F + e_i$ does not have a cycle, then $F \leftarrow F + e_i$.

return F

Note that the correctness of Kruskal's algorithm also follows from repeatedly applying the cut property, because if $F + e_i$ does not create a cycle, then e_i is a cheapest edge leaving a component of F, and so by the cut property $F + e_i$ is also contained in some MST.

We will discuss how to implement Kruskal's algorithm efficiently in the next section.

Remark: The correctness of Kruskal's algorithm also follows from the "cycle property", which says that a most expensive edge in a cycle can be removed from the graph and the remaining graph still has a MST. See (4.20) in [KT] or wikipedia for a proof.

Borůvka's algorithm

We can also prove the correctness of Borůvka's algorithm by repeatedly applying the cut property. We leave it as an optional exercise.

Reverse-delete algorithm

The idea is to keep removing a heaviest edge as long as the remaining graph is still connected. The correctness of this algorithm can be established by repeatedly applying the cycle property. We also leave it as an optional exercise.

Implementations

Finally, we show how to implement the algorithms efficiently using appropriate data structures.

Prim's algorithm

Note that Prim's algorithm has exactly the same structure as Dijkstra's algorithm, as both involve

adding a vertex $v \notin S$ "closest" to S and growing $S \leftarrow S + \{v\}$.

In Dijkstra's algorithm, for each $v \notin S$, we defined $dist[v] = \min_{w \in S} \{ dist[w] + l_{wv} \}$ and add the vertex $v \notin S$ with smallest $dist[v]$ into S.

In Prim's algorithm, for each $v \notin S$, we can define $connect[v] := \min_{w \in S} \{ c_{wv} \}$ and add the vertex $v \notin S$ with smallest $connect[v]$ into S.

```
connect(v) = ∞ for every v ∈ V , connect(s) = 0 .
F := φ , S := {s} , parent[s] = s
Q = make-priority-queue (V)          // with key value of v being connect[v]

While Q is not empty do
    u = delete-min (Q)
    for each neighbor v of u
        if cost(uv) < connect(v)
            connect(v) = cost(uv)
            decrease-key (Q, v)
            parent(v) = u
    S ← S + {u} , F ← F + (u, parent[u]).
```

Time complexity: This is the same as Dijkstra's algorithm, with running time $O((m+n)\log n)$ as each each priority queue operation can be done in $O(\log n)$ time.

kruskal's algorithm

For kruskal's algorithm, the main step that we need to speed up is to check whether the two endpoints of an edge uv belong to the same component in the forest F or not.

There is an interesting data structure called "disjoint sets" that supports the following operations:

- makeset (x) : create a singleton set containing just x.
- find(x) : return which set that x belongs to.
- union (x,y) : merge the sets containing x and y.

All of these operations can be done in $O(\log n)$ time when there are at most n elements.

With this disjoint-sets data structures, we can implement kruskal's algorithm as follows.

```
F := φ
```

makeset(v) for each $v \in V$.

Sort the edges by cost so that $cost(e_1) \leq cost(e_2) \leq \dots \leq cost(e_m)$

for $1 \leq i \leq m$ do

let $e_i = uv$

if find(u) $\neq$ find(v),

then $F \leftarrow F + \{uv\}$, union(u,v)

return F

Time complexity: As each operation of the data structure can be done in $O(\log n)$ time, it is clear that the total time complexity is $O((m+n)\log n)$.

The details of the disjoint-set data structure can be found in [DPV 5.1] and [KT 4.4].

Borůvka's algorithm

This can be implemented in $O((m+n)\log n)$ time without any non-trivial data structures.

We leave it as an optional exercise. (I wonder why this algorithm is not discussed more in books...)

References: [DPV 5.1], [KT 4.4].

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 11: Dynamic Programming

We introduce the technique of dynamic programming through some well-known examples.

Introduction

On a high level, we can solve a problem by dynamic programming if there is a recurrence relation with only a small number of subproblems (i.e. polynomially many).

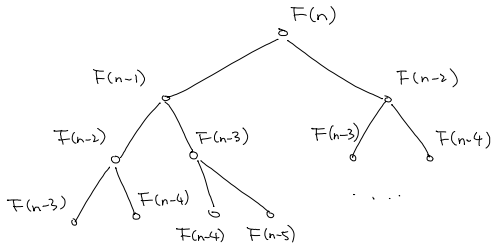
This is a general and powerful technique, and also simple to use once we learnt it well.

To illustrate the idea using a toy example, consider the problem of computing the Fibonacci sequence

$$F(n) = F(n-1) + F(n-2); \qquad F(1) = F(2) = 1.$$

The function is defined recursively with the base cases given.

It is then natural to compute it using recursion, but if we trace the recursion tree, we find out that it is huge.



If we solve the recurrence relation (MATH 239), then we find out the runtime of this algorithm is $\Theta(1.618^n)$.

Observe that the recursion tree is highly redundant, many subproblems are computed over and over again.

There are only n subproblems. Why do we waste so much time?

There are two approaches to solve the problem efficiently.

Top-Down Memorization

As in BFS / DFS, we use an array `visited[i]` to ensure that we only compute each subproblem at most once, and we use an array `answer[i]` to store the value $F(i)$ for future lookup.

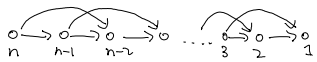
```
visited[i] = false   for 1 ≤ i ≤ n
F(n).
```
} main program

```
F(i)    // recursive function
    if visited[i] = true, return answer[i].
    if i = 1 or i = 2, return 1.
    answer[i] = F(i-1) + F(i-2).
    visited[i] = true.
    return answer[i].
```

Time Complexity Each subproblem is only computed once, when visited[i] = false.

When it is computed, it looks up two values. So, the total time complexity is O(n).

Alternatively, we can think of it as doing a graph search on a directed acyclic graph, with only n vertices and 2(n-1) edges.



Bottom-Up Computation

For Fibonacci sequence, there is a straightforward algorithm to solve the problem in O(n) time.

F(1) = F(2) = 1

for $3 \le i \le n$ do

F(i) = F(i-1) + F(i-2).

It is clear that this solves the problem in O(n) additions.

(Note that the values grow exponentially in n, so we cannot assume that each addition can be done in O(1) time.)

Dynamic Programming

So, basically, this is the framework of dynamic programming, to store the intermediate values so that we don't need to compute it again.

From the top-down approach, it should be clear that if we write a recursion with only polynomially many subproblems with additional polynomial time processing, then the problem can be solved in polynomial time.

In this viewpoint, designing an efficient dynamic programming algorithm amounts to coming up with a nice recursion. This is similar to what we have done in designing divide and conquer algorithms, coming up with a right recursion.

We will see that many interesting and seemingly difficult problems have a nice recurrence relation.

Of course, it requires some skills and practices to write a nice recursion to solve the problems, and this is what we will focus on in the many examples to follow.

In practice, the bottom-up implementation is preferred as it is non-recursive and usually more efficient.

In some cases, it will be easy to translate a top-down solution into a bottom-up solution.

In some cases, however, it requires clear thinking to find a correct ordering to compute the subproblems, especially when the recurrence relation is complicated, although in principle we just need a topological ordering.

Our main focus will be to come up with the right recurrence, as that would already imply an efficient algorithm.

We will also mention the bottom-up implementations as much as possible.

Weighted Interval Scheduling [KT 6.1]

Input: n intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$ with weights $w_1, w_2, \ldots, w_n \in \mathbb{R}$.

Output: a subset S of disjoint intervals that maximizes $\sum_{i \in S} w_i$.

This is a generalization of the interval scheduling problem in L08, when $w_i = 1 \ \forall i$.

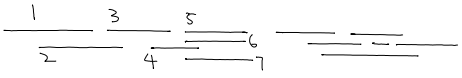
We can think of the intervals are requests to book our room from time s_i to f_i, and the i-th request is willing to pay w_i dollars if the request is accepted.

Then, our objective is to maximize our income, while ensuring that there are no conflicts for the accepted requests.

Unlike the special case when $w_i = 1 \ \forall i$, there are no known greedy algorithms for this problem.

Exhaustive Search

To come up with a good recursion for this problem, we start by running an exhaustive search algorithm and see why it is wasteful and how to improve it.

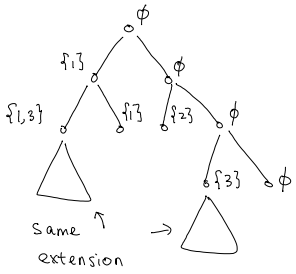


First, we make a decision on interval 1, either choose it or not.

If we choose interval 1, then we cannot choose interval 2, and we need to make a decision on interval 3.

If we do not choose interval 1, then we need to make a decision on interval 2.

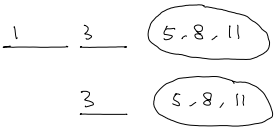
Doing this recursively, we have a recursion tree as shown.



This is an exponential time algorithm, but observe a lot of redundancy.

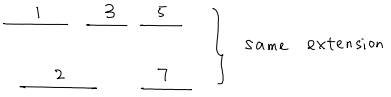
For example, in the branches of $\{1,3\}$ and $\{3\}$, the subtrees extending these partial solutions are exactly the same.

This is because to determine how to extend these partial solutions, what really matter is the last interval of the current partial solution, but not anything on the left, since they won't interact with anything on the right.



So, we just need to keep track of the "boundary" of the solution.

In this problem, the boundary is simply the last interval.



This suggests that there should be a recursion with only 1 parameter!

Better Recurrence

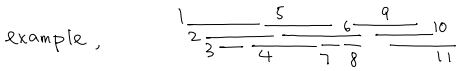
To facilitate the algorithm description, we use a good ordering of the intervals and pre-compute useful information.

We sort the intervals by starting time so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

For each interval i, we use next[i] to denote the smallest j such that $j > i$ and $f_i < s_j$.

i.e., the first interval on the right of interval i that is not overlapping with interval i.

If no such intervals exist, next[i] is defined as n+1, representing the end.

In this example, 

, next[1] = 5, next[2] = 6, next[3] = 4, next[9] = 12, etc.

Since we sort the intervals by non-decreasing starting time, for an interval i, next[i] has the property that intervals $\{i, \ldots, next[i]-1\}$ overlap with interval i while the intervals $\{next[i], \ldots, n\}$ are disjoint from interval i.

Now, we are ready to write a recursion with only one parameter, resulting in only n subproblems!

Let opt(i) := maximum income that we can earn using the intervals in $\{i, i+1, \ldots, n\}$ only.

Then opt(1) is the optimal value that we would like to compute.

To compute opt(1), there are only two options for the solutions:

① The solutions that choose interval 1.

For these solutions, we earn w_1 dollars by choosing interval 1.

But then we cannot choose the intervals in $\{2, \ldots, next[1]-1\}$ since these intervals overlap with interval 1.

So, to find the optimal value of choosing interval 1, we need to find an optimal way to choose from $\{next[1], \ldots, n\}$.

Therefore, the optimal value for solutions choosing interval 1 is $w_1 + opt(next[1])$.

② The solutions that don't choose interval 1.

Then, by definition of opt(2), the optimal value for solutions not choosing interval 1 is opt(2).

Combining the two cases, we get that $opt(1) = \max\{w_1 + opt(next[1]),\ opt(2)\}$.

This recurrence relation is true for every i, and so we have the following recursive algorithm to solve the problem.

Algorithm (top-down weighted interval scheduling)

1. Sort the intervals by non-decreasing starting time so that $s_1 \le s_2 \le \ldots \le s_n$.
2. Compute next[i] for $1 \le i \le n$. Set visited[i] = false for $1 \le i \le n$.
3. Return opt(1)

```
opt(i)    // recursive function
    if i = n+1, return 0.          // base case
    if visited[i] = true, return answer[i].
    answer[i] = max{ w_i + opt(next[i]), opt(i+1) }.
    visited[i] = true.
    return answer[i].
```

Correctness The correctness of the algorithm follows from the explanation of the recurrence relation above.

Time complexity Sorting and the next[] array can be computed in $O(n \log n)$ time. (why?)

After that, the top-down memorization implements the recursion in $O(n)$ time, since there are only n subproblems and each subproblem only needs to look up two values.

Bottom-up implementation It is simple in this problem.

$opt(n+1) = 0$

for i from n downto 1 do

$opt(i) = \max \{ w_i + opt(next[i]),\ opt(i+1) \}$.

Quite surprisingly, this has the same time complexity as the greedy algorithm!

That is, somehow we can implement an exhaustive search algorithm as efficient as the greedy algorithm!

We can see why dynamic programming is general and powerful, because it is very systematic (so that we may not need problem specific insight) and yet we get very competitive algorithms.

Exercise: Write a program to print out an optimal solution (i.e. which intervals to choose).

Subset-Sum and Knapsack Problem [KT 6.4]

We consider two related and useful problems.

Subset-Sum

Input: n positive integers $a_1, a_2, \ldots, a_n$, and an integer K.

Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$, or report that no such subset exists.

For example, given 1, 3, 10, 12, 14, is there a subset with sum 27? Yes, $\{3, 10, 14\}$. Sum 29? No.

This problem can be modified to ask for a subset S with $\sum_{i \in S} a_i \le K$ but maximizes $\sum_{i \in S} a_i$.

This is the version in [KT] and is slightly more general, but once we solve our equality version it should be clear how to solve the inequality version as well.

Knapsack

Input: n items, each of weight w_i and value v_i, and a positive integer W.

Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \le W$ that maximizes $\sum_{i \in S} v_i$.


(from cliparthut)

We can think of W as the weight that the Knapsack can hold.

Then, the problem asks us to find a maximum value subset that we can fit in the knapsack.

Alternatively, we can think of W as the total time that we have, and our objective is to choose a subset of jobs that can be finished on time while maximizing our income.

Knapsack is more general than subset-sum as there are two parameters to consider, but again it will not be difficult once we solved subset-sum. So, let's start with subset-sum.

Recurrence

To come up with the recurrence, we start with the exhaustive search algorithm for the subset-sum problem. We will consider all possibilities.

Start with the first number a_1. Then, either we choose it or not.

- If we choose a_1, then we need to choose a subset from $\{2,\dots,n\}$ so that the sum is $k-a_1$.
- Otherwise, if we don't choose a_1, then we need to choose a subset from $\{2,\dots,n\}$ so that the sum is k.

A naive implementation will consider all subsets, and this gives an exponential time algorithm.

The observation is, we don't really need to keep track of which subset we have chosen so far, as long as they have the same sum.

This suggests that we can just keep track of the (partial) sum in the recursion, which allows us to reduce the search space significantly.

The subproblems that we will consider are subsum[i, L] for $1 \le i \le n$ and $L \le K$, where subsum[i,L] returns true if and only if there is a subset in $\{i,\dots,n\}$ with sum L.

Then the original problem that we would like to solve is subsum[1, K].

The recurrence relation is subsum[i,L] = (subsum[i+1, L-a_i] OR subsum[i+1, L]).

The former case corresponds to choosing a_i, while the latter case corresponds to not choosing a_i.

If either case returns true, then return true. Otherwise, when both return false, return false.

Algorithm (top-down subset-sum)

From the recurrence relation to a correct algorithm, we just need to be careful about the base cases.

```
subsum(i, L)        // recursive function
    if (L=0)    return true.
    if (i>n or L<0)  return false      // running out of numbers, or partial sum too large.
    return (subsum(i+1, L-a_i) OR subsum(i+1, L)).
```

Correctness follows from the recurrence relation explained above.

Time complexity: There are totally nk subproblems, n choices for i and k choices for L.

Each subproblem can be solved by looking up two values.

Using top-down memorization, the time complexity is $O(nk)$.

Pseudo-polynomial time: Note that the time complexity is $O(nk)$, which depends on K.

When k is small, this is fast.

But k could be exponential in n (as n-bit number can be as big as 2^n), in which case this is even slower than the naive exhaustive search (and also uses much more space).

We call this type of time complexity pseudo-polynomial.

This is probably unavoidable, as we will see that the subset-Sum problem is NP-complete.

Implementations

Bottom-up computation

We use a 2D-array subsum[n][K] to store the values of all subproblems.

We can compute these values in reverse order from n to 1.

```
subsum[i][L] = false  for all 1≤i≤n and 0≤L≤K      // initialization
subsum[n][a_n] = subsum[n][0] = true      // the YES case for the size 1 problem where we only have a_n.
subsum[i][0] = true  for all 1≤i≤n      // the YES case when the target is zero.
for i from n downto 1 do
    for L from 1 to K do
        if subsum[i+1][L] = true, then subsum[i][L] = true.
        if (L-a_i ≥ 0 and subsum[i+1][L-a_i] = true), then subsum[i][L] = true.
```

Space-Efficient Implementation

With this bottom-up implementation, we see that we don't need to use a $n \times k$ array.

When we are computing subsum[i][*] in the outer for-loop, we just use the values of subsum[i+1][*].

So, we can throw away the values subsum[≥i+2][*] and only use a $2 \times k$ array, a significant saving.

Top-Down vs Bottom-Up

The bottom-up implementations don't need recursions, and also leads to a space-efficient algorithm.

But the run-time is always exactly nk, to compute all the subproblems.

When there is a solution, the top-down approach may run faster, as it may be able to find

a solution by solving only a few subproblems.

From this perspective, it may make a difference by solving subsum(i+1, L-a_i) before subsum(i+1, L). (Why?)

Tracing a Solution

By following a path of "true" from subsum(1, K), we can find a subset of sum K.

- If subsum(2, K-a_1) = true, then put a_1 in our solution and recurse.
- Otherwise, don't put a_1 in the solution, and follow a "true" path from subsum(2, K).

Dynamic Programming and Graph Search

Dynamic programming reduces the search space to polynomial size. We can draw a graph.

Each vertex is a subproblem, and the edges are added according to the recurrence relation.

That is, there is a direct edge from (i, L) to (i+1, L) and from (i, L) to (i+1, L-a_i) if L-$a_i \geq 0$.

Then the problem is equivalent to determining whether there is a directed path from the starting state (1, K) to a "true" "base state" (e.g. (i, 0) for some i).

The way the top-down memorization algorithm works is basically doing a DFS on this "subproblem graph", by recursing and marking subproblems visited.

Knapsack (Sketch)

We outline how to solve the knapsack problem, which is quite similar to solving Subset-Sum.

From now on, we won't start from the exhaustive search again.

First Approach

We use a similar recurrence as in Subset-Sum.

The subproblems are knapsack(i, W, V), which is true if and only if there is a subset in $\{i, i+1, \ldots, n\}$ with total weight W and total value V.

With this 3D-table, you should be able to write a recurrence as in subset-sum to solve the problem.

Better recurrence

It is possible to just use 2 parameters as in subset-sum.

Note that some subproblems dominate other subproblems, e.g. if knapsack(i, W, V+1) = true, then we can ignore the subproblem knapsack(i, W, V).

So, the idea is to only have subproblems knapsack(i, W) and keep track of the max value achievable.

Define knapsack(i, W) as the maximum value that we can earn using items in $\{i, \ldots, n\}$ with total weight $\leq W$.

More precisely, let knapsack(i, W) = $\max_{S \subseteq \{i, \ldots, n\}} \left\{ \sum_{j \in S} v_j \mid \sum_{j \in S} w_j \leq W \right\}$.

Define knapsack(i,w) as the maximum value that we can carry using items in {i,...,n} with weight ≤ w.

More precisely, let $\text{knapsack}(i,w) = \max_{S \subseteq \{i,\dots,n\}} \left\{ \sum_{j \in S} v_j \mid \sum_{j \in S} w_j \le w \right\}$.

Then, the recurrence relation is $\text{knapsack}(i,w) = \max\{ v_i + \text{knapsack}(i+1, w-w_i),\ \text{knapsack}(i+1,w) \}$.

The first case corresponds to choosing item i, thus earning v_i and the maximum value of using items from i+1 to n when the capacity left in the knapsack is $w-w_i$.

The second case corresponds to not choosing item i.

With this recurrence relation, it is not difficult to complete the algorithm and the analysis as in Subset-Sum.

Just need to be careful in the base cases: when $w<0$, return $-\infty$; when $i>n$, return 0.

We leave the details to the reader. Go through the subset-sum section again if necessary.

The time complexity is $O(nW)$.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 12: Dynamic Programming II

We will use dynamic programming to design efficient algorithms for basic sequence and string problems.

Longest Increasing Subsequence [DPV 6.2]

Given n numbers $a_1, \dots, a_n$, a subsequence is a subset of these numbers taken in order of the form $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ where $1 \le i_1 < i_2 < \dots < i_k \le n$, and a subsequence is increasing if $a_{i_1} < a_{i_2} < \dots < a_{i_k}$.

Input: n numbers $a_1, a_2, \dots, a_n$

Output: an increasing subsequence of maximum length.

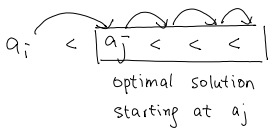
For example, given 5, 1, 9, 8, 8, 8, 4, 5, 6, 7 (recognize this?), the longest increasing subsequence is 1, 4, 5, 6, 7.

By now, it should be relatively straightforward to solve this problem using dynamic programming.

Subproblems: Let $L(i)$ be the length of a longest increasing subsequence starting at a_i and only using the numbers in $a_i, \dots, a_n$. So, there are only n subproblems.

Final answer: After we compute $L(1), L(2), \dots, L(n)$, the final answer is $\max_{1 \le i \le n} \{L(i)\}$.

Recurrence relation: Given we start at a_i, we try all possible numbers a_j with $j > i$ and $a_j > a_i$, and form an increasing subsequence starting from a_i by concatenating with a longest increasing subsequence starting at a_j.



More precisely, $L(i) = 1 + \max_{i+1 \le j \le n} \{ L(j) \mid a_j > a_i \}$.

Note that the subsequences formed must be increasing.

The correctness can be proved by induction, i.e. if $L(i+1), \dots, L(n)$ are correct, then $L(i)$ is also correct.

Bottom-up implementation

```
L(i) = 1   ∀ 1 ≤ i ≤ n    // initialization
for i from n downto 1 do
    for j from i+1 to n do
        if a_j > a_i and L(j) + 1 > L(i)
            then update L(i) ← L(j) + 1.
```

For example, given 3, 8, 7, 2, 6, 4, 12, 14, 9.

the L-values are 4, 3, 3, 4, 3, 3, 2, 1, 1

Time complexity: It should be clear that it is bounded by $O(n^2)$.

Printing a longest increasing subsequence We leave it as an exercise to write out the details.

One way to do it is to keep track of the next number (e.g. parent[i] = j) when we update $L(i) \leftarrow L(j)+1$.

We can also directly trace back a longest increasing subsequence using the L-values in $O(n)$ time without using extra storage.

Longest path in DAG: An alternative way to think about this problem is to find a longest path in a DAG.

Given n numbers $a_1, \ldots, a_n$, we create a graph of n vertices, each corresponding to a number.

There is a directed edge from i to j if $j > i$ and $a_j > a_i$.

Then, an increasing subsequence corresponds to a directed path in this directed acyclic graph, and vice versa.

So, a longest path in the graph gives us a longest increasing subsequence.

For example, given 3, 8, 7, 2, 6, 4, 12, 14, 9, the graph is

In general, the longest path problem in DAG can be solved by dynamic programming efficiently, and it is a useful exercise to work out the details.

A Faster Algorithm for Longest Increasing Subsequence (Optional)

There is a clever algorithm to solve the problem in $O(n \log n)$ time.

The observation is that we don't need to store all the subproblems, as some subproblems are "dominated" by other subproblems.

For each length k, we will only store the "best" position to start an increasing subsequence of length k.

Then, it will turn out that these best positions satisfy a monotone property, and this allows us to use binary search to update these values in $O(\log n)$ time when we consider a new element.

This is a high level summary and now we discuss the details.

Best subproblems

Suppose we have already computed $L(i+1), L(i+2), \ldots, L(n)$ and now we want to compute $L(i)$.

For a given length k, consider the indices $i < i_1 < i_2 < \ldots < i_\ell$ so that $L(i_1) = L(i_2) = \ldots = L(i_\ell) = k$.

What is the best subproblem to keep for future computations of $L(i), L(i-1), \ldots, L(1)$?

Since we are extending these increasing subsequences using elements in $\{1, \ldots, i\}$, the starting positions $i_1, i_2, \ldots, i_\ell$ are not important.

What is important is the starting value.

If $L(i_1) = L(i_2) = k$ and $a_{i_1} > a_{i_2}$, then the subproblem $L(i_1)$ dominates the subproblem $L(i_2)$, because any increasing subsequence using numbers in $\{a_1, \ldots, a_{i_1}\}$ that can be extended by an increasing subsequence of length k starting at a_{i_2} can also be extended by an increasing subsequence of length k starting at a_{i_1}.

That is, among the increasing subsequences of length k, the one with largest starting value is easiest to be extended.

Therefore, we define $pos[k] = \underset{j>i}{\arg\max} \{ a_j \mid L(j) = k \}$, when $L(i)$ is the current subproblem to be computed.

Intuitively, $pos[k]$ is the best position to start an increasing subsequence of length k after the current index i.

Let $m = \max_{i<j\le n} \{L(j)\}$ be the length of a longest increasing subsequence we have computed so far.

By the reasoning above, when we compute $L(i)$, we just need to consider $L(pos[1]), L(pos(2)), \ldots, L(pos(m))$, as the other subproblems are dominated by these subproblems.

For example, given the sequence $2, 7, 6, 1, 4, 8, 5, 3$, when we compute $L(2)$, we have $L(3) = L(5) = 2$, $L(4) = 3$, and $L(6) = L(7) = L(8) = 1$, then we only keep $pos[1] = 6$ with $a_6 = 8$, $pos[2] = 3$ with $a_3 = 6$, $pos[3] = 4$ with $a_4 = 1$.

Monotonicity

Once we only keep the best subproblems, we have the following important monotone property.

Claim $a[pos[1]] > a[pos[2]] > \cdots > a[pos[m]]$ where $m = \max_{i<j\le n} \{L(j)\}$ and $L(i)$ is the current subproblem.

(Intuition: A longer subsequence should be more difficult to be extended, i.e. its starting value is smaller.)

Proof Suppose, by contradiction, that there exists j such that $a[pos[j]] \ge a[pos[j-1]]$.

Let an optimal increasing subsequence of length j be $a_{p_1} < a_{p_2} < \cdots < a_{p_j}$ where $p_1 = pos[j]$.

Then $a_{p_2} < \cdots < a_{p_j}$ is an increasing subsequence of length $j-1$ with $a_{p_2} > a_{p_1} = a[pos[j]] \ge a[pos[j-1]]$, contradicting that $pos[j-1]$ is the best position to start an increasing subsequence of length $j-1$. □

Updating the best subproblems using binary search

Suppose we have found the best subproblems $pos[m], pos[m-1], \ldots, pos[1]$ after processing the numbers $a_n, \ldots, a_{i+1}$.

Now, we process the number a_i, and would like to update the best subproblems for future computations.

We consider three cases.

(1). When $a_i < a[pos[m]]$.

This is the good case, as we can extend the longest increasing subsequence so far by one, by adding a_i in front of the increasing subsequence of length m starting at $pos[m]$.

So, we can increase m by 1, and set $pos[m] = i$.

(2) When $a[pos[j]] \le a_i < a[pos[j-1]]$.

Since $a_i > a[pos[j]]$, we cannot use a_i to form an increasing subsequence of length $j+1$.

But we can use a_i to form an increasing subsequence of length j, by adding a_i in front of the increasing subsequence of length $j-1$ starting at $pos[j-1]$.

Furthermore, this increasing subsequence of length j is better than the one starting at $pos[j]$, as $a_i > a[pos[j]]$.

So, in this case, we update $pos[j] = i$.

(3) When $a[pos[1]] \leq a_i$.

In this case, we cannot use a_i to extend any increasing subsequence because it is larger than all the starting values, but we can use it to update $pos[1] = i$.

Note that since $a[pos[m]] < a[pos[m-1]] < \ldots < a[pos[1]]$, we can use binary search to find the smallest j so that $a[pos[j]] \leq a_i$, and then we update by the above rules.

Fast Algorithm

```
m = 1, pos[1] = n.      // base case.
for i from n-1 downto 1 do
    if a_i < a[pos[m]], then set m ← m+1 and pos[m] = i.   // longer increasing subsequence
    else use binary search to find the smallest j so that a[pos[j]] ≤ a_i, then set pos[j] = i.
return m.
```

(The final algorithm is very simple, but may not be easy to come up with.)

Time complexity $O(n \log n)$.

Exercise Write the code to print a longest increasing subsequence.

Longest Common Subsequence [CLRS 15.3]

Input: Two strings $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, where each a_i, b_j is a symbol.

Output: The largest k such that there exist $i_1 < i_2 < \ldots < i_k$ and $j_1 < j_2 < \ldots < j_k$ s.t. $a_{i_\ell} = b_{j_\ell}$ for $1 \leq \ell \leq k$.

One example is that we are given two DNA sequences and want to identify common structures.

$S_1 = AAACCGTGAGTTATTCGTTCTAGAA$
$S_2 = CACCCCTAAGGTACCTTTGGTTC$
$\Rightarrow$ $ACCTAGTACTTTG$

Note that the longest subsequence problem (LIS) is a special case of the longest common subsequence problem (LCS).

3, 8, 7, 2, 6, 4, 12, 14, 9 (for LIS) $\Rightarrow$ reduce to 3, 8, 7, 2, 6, 4, 12, 14, 9 / 2, 3, 4, 6, 7, 8, 9, 12, 14 (for LCS)

Since the second sequence is sorted, it forces the solution of LCS to be an increasing subsequence.

Recurrence

Let $C(i,j)$ be the length of a longest common subsequence of $a_i, \ldots, a_n$ and $b_j, \ldots, b_m$.

Then the answer that we are looking for is $C(1,1)$.

The base cases are $C(n+1, j) = 0 \quad \forall\ 1 \le j \le m$, and $C(1, m+1) = 0 \quad \forall\ 1 \le i \le n$.

To compute $C(i,j)$, there are three cases, depending on whether a_i and b_j are used or not.

① (Use both a_i and b_j) If $a_i = b_j$, then we can put a_i and b_j in the beginning of a common subsequence, then the remaining subproblem is to find a longest common subsequence for $a_{i+1}, \ldots, a_n$ and $b_{j+1}, \ldots, b_m$.

So, let $SOL_1 = 1 + C(i+1, j+1)$ if $a_i = b_j$. otherwise $SOL_1 = 0$

② (Not use a_i) Then we find a longest common subsequence for $a_{i+1}, \ldots, a_n$ and $b_j, \ldots, b_m$.

So, let $SOL_2 = C(i+1, j)$.

③ (Not use b_j) Then we find a longest common subsequence for $a_i, \ldots, a_n$ and $b_{j+1}, \ldots, b_m$.

So, let $SOL_3 = C(i, j+1)$.

Then, we take the best out of these three possibilities. That is, $C(i,j) = \max\{SOL_1, SOL_2, SOL_3\}$.

Correctness All solutions for $C(i,j)$ fall into at least one of the above three cases.

We can then prove correctness by induction.

Time Complexity There are $n \cdot m$ subproblems. Each subproblem looks up three values.

Using top-down memorization, the total time complexity is $O(m \cdot n)$.

Tracing out solution We can either record some "parent" information when computing $C(i,j)$.

We can also compute it directly using $C(i,j)$ only, by recursively going to a subproblem that gives the maximum value for $C(i,j)$.

Bottom-up implementation

$C(i, m+1) = 0 \quad \forall\ 1 \le i \le n$, $C(n+1, j) = 0 \quad \forall\ 1 \le j \le m$. // base cases

for i from n downto 1 do

  for j from m downto 1 do

    if $a_i = b_j$, Set $SOL \leftarrow 1 + C(i+1, j+1)$, else $SOL \leftarrow 0$.

    $C(i,j) = \max\{SOL, C(i+1, j), C(i, j+1)\}$.

Edit Distance [DPV 6.3]

Input: Two strings $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, where each a_i, b_j is a symbol.

Output: The minimum k so that we can do k add / delete / change operations to transform $a_1, \ldots, a_n$ into $b_1, \ldots, b_m$.

For example, if the two input strings are SNOWY and SUNNY, the following are two ways:

S	–	N	O	W	Y	
S	U	N	N	–	Y	

Cost: 3

–	S	N	O	W	–	Y	
S	U	N	–	–	N	Y	

Cost: 5

In the first way, we match S, add U, match N, change O to N, delete W, and match Y.

This takes three add / delete / change operations to transform SNOWY to SUNNY.

The second way requires five add / delete / change operations to transform SNOWY to SUNNY.

We call the minimum number of operations to transform one string to another string the "edit distance" between the two strings.

It is a useful measure of the similarity of two strings, e.g. in a word processor.

Recurrence

The recurrence is similar to that in LCS.

Let $D(i,j)$ be the edit distance of the strings $a_i, \ldots, a_n$ and $b_j, \ldots, b_m$.

The answer that we want is $D(1,1)$.

The base case is $D(n+1, m+1) = 0$.

To compute $D(i,j)$, there are four possible operations to perform:

(Add) We add b_j to the current string, when $j \le m$. e.g. $\begin{matrix}\ldots & a & b & c \\ \ldots & d & e & f\end{matrix} \Rightarrow \begin{matrix}\ldots & - & a & b & c \\ \ldots & d & e & f\end{matrix}$

Then we match one more symbol of the target string and move on.

More precisely, if $j \le m$, $SOL_1 = 1 + D(i, j+1)$; else $SOL_1 = \infty$.

(Delete) We delete a_i from the current string, when $i \le n$. e.g. $\begin{matrix}\ldots & a & b & c \\ \ldots & d & e & f\end{matrix} \Rightarrow \begin{matrix}\ldots & a & b & c \\ \ldots & - & d & e & f\end{matrix}$

Then we move one symbol forward in the current string.

More precisely, if $i \le n$, $SOL_2 = 1 + D(i+1, j)$; else $SOL_2 = \infty$.

(Change) We change a_i to b_j, when $i \le n$ and $j \le m$.

Then we move one symbol forward in both strings. e.g. $\begin{matrix}\ldots & a & b & c \\ \ldots & d & e & f\end{matrix} \Rightarrow \begin{matrix}\ldots & a & b & c \\ \ldots & d & e & f\end{matrix}$

More precisely, if $i \le n$ and $j \le m$. $SOL_3 = 1 + D(i+1, j+1)$; else $SOL_3 = \infty$.

(Match) If $i \le n$ and $j \le m$ and $a_i = b_j$, then we match and move one symbol forward in both strings.

More precisely, if $i \le n$ and $j \le m$ and $a_i = b_j$, $SOL_4 = D(i+1, j+1)$; else $SOL_4 = \infty$.

Finally, we set $D(i,j) = \min \{ SOL_1, SOL_2, SOL_3, SOL_4 \}$.

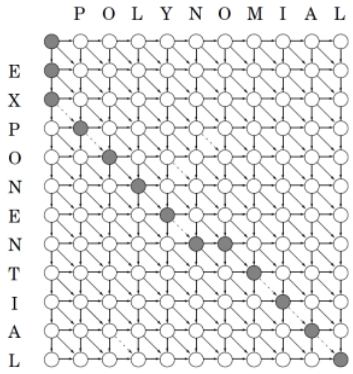
$\begin{matrix}\ldots & a & b & c \\ \ldots & a & a & c\end{matrix} \Rightarrow \begin{matrix}\ldots & a & b & c \\ \ldots & a & a & c\end{matrix}$

Correctness. Follows from the base case and an inductive argument. In the inductive step as discussed above, we have considered all the possibilities to transform one string to another string.

Time complexity There are $m \cdot n$ subproblems, each requiring a constant number of operations. Using top-down memorization, the time complexity is $O(n \cdot m)$.

Bottom-up implementation and returning a solution Similar to that in LCS. Leave as an important exercise.

Graph searching Once again, we would like to point out that dynamic programming can be thought of as finding a (shortest) path from the starting state to the target state in the state graph. This connection is even more transparent when we are using the state table to trace out a solution.



[DPV 6.3]

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 13: Dynamic programming on trees

We will see two examples to use dynamic programming to solve problems on trees.

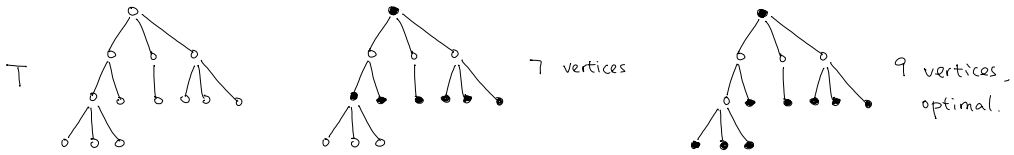
Independent Sets on Trees [DPV 6.7]

Given a graph $G=(V,E)$, a subset of vertices $S \subseteq V$ is called an independent set if $ij \notin E \ \forall i,j \in S$.

We will see that the problem of finding a maximum cardinality independent set is NP-Complete, but we can use dynamic programming to solve the problem in polynomial time on trees.

Input: A tree $T=(V,E)$

Output: An independent set $S \subseteq V$ of maximum cardinality.



Having a tree structure suggests a natural way to use dynamic programming.

We will define a subproblem for each subtree rooted at a vertex v.

The key point is that since there are no edges between different subtrees, we can solve the problem on each subtree separately and thus reduce to smaller subproblems.

Then we can write a recurrence relation between a parent and its children.

Actually, we have used this idea before.

When we compute the low[] array to compute all cut vertices, we have already used dynamic programming on trees.

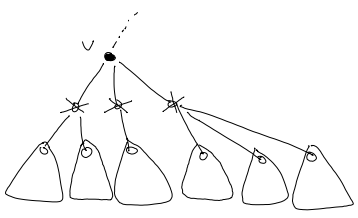
Dynamic Programming

Subproblems: Let $I(v)$ be the size of a maximum independent set in the subtree rooted at vertex v.

Answer: $I(root)$.

Base cases: $I(leaf)=1$ for all leaves in the tree.

Recurrence: To compute $I(v)$, we consider two possibilities.



① v is in the independent set.

Then all its children cannot be included in the independent set.

The optimal way to extend the current partial solution is to take a maximum independent set in each subtree rooted at its grandchildren.

So, in this case, the maximum size is $1 + \sum_{w: w \text{ grandchild of } v} I(w)$.

② v is not in the independent set.

The optimal way to extend the current partial solution is to take a maximum independent set in each subtree rooted at its children.

So, in this case, the maximum size is $\sum_{w: w \text{ child of } v} I(w)$.

Therefore, $I(v) = \max\left\{ 1 + \sum_{w: w \text{ grandchild of } v} I(w), \sum_{w: w \text{ child of } v} I(w) \right\}$.

Correctness follows from the explanation of the recurrence relation and induction.

Time complexity: There are n subproblems.

Each subproblem requires (# children + # grandchildren) lookups.

Note that $\sum_{v \in V} (\#\text{children} + \#\text{grandchildren}) = \sum_{v \in V} (\#\text{parent} + \#\text{grandparent}) \leq \sum_{v \in V} 2 = 2n$, by counting the sum in a different way (i.e. counting upward instead of counting downward).

So, using top-down memorization, the total time complexity is $O(n)$.

Bottom-up implementation and printing out solution: Exercises.

Alternative recurrence relation: We can also write a recurrence relation involving children only.

The idea is to use two subproblems on a vertex v.

Let $I^+(v)$ be the size of a maximum independent set with v included, and

$I^-(v)$ be the size of a maximum independent set with v excluded.

Then the answer is $\max\{ I^+(\text{root}), I^-(\text{root}) \}$.

The base cases are $I^+(\text{leaf}) = 1$ and $I^-(\text{leaf}) = 0$ for all leaves.

The recurrences are $I^+(v) = 1 + \sum_{w: w \text{ child of } v} I^-(w)$ and $I^-(v) = \sum_{w: w \text{ child of } v} \max\{ I^+(w), I^-(w) \}$.

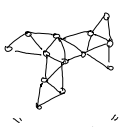
We leave the justification of these recurrences as an exercise.

Exercise: Extend the algorithm to solve the maximum weighted independent set problem on trees.

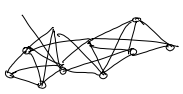
Dynamic Programming on "Tree-like" Graphs (optional) [KT 10.4, 10.5]

Many optimization problems are hard on graphs but easy on trees using dynamic programming.

Generalizing the ideas using dynamic programming on trees, it is possible to show that dynamic programming also works on "tree-like" graphs, e.g.



"treewidth" 2



"treewidth" 3

There is a way to define the "tree-width" of a graph so as to measure how "close" a graph is to a tree.

If the tree-width is small, then dynamic programming works faster, usually with runtime $\sim O(n^{\text{treewidth}})$.

This has become an important paradiam to deal with hard problems on graphs, at least in research papers.

Read [KT 10.4, 10.5] for an introduction to this approach.

Optimal Binary Search Tree [CLRS 15.5]

This problem is a bit similar to the Huffman coding problem, also about finding an optimal binary tree.

Imagine the scenario where there are n commonly searched strings, e.g. some French vocabularies for English meanings.

We would like to build a good data structure to support these queries effectively.

And somehow we have decided to use a binary search tree (say instead of using hashing).

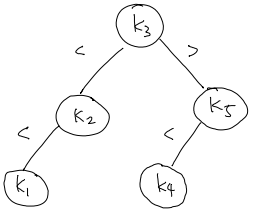
As there are n strings, we could build a balanced binary search tree to answer the queries in $O(\log_2 n)$ time.

As in Huffman coding, suppose we know the frequencies of the searched strings. Can we use this information to design a better binary search tree so as to minimize the average query time?

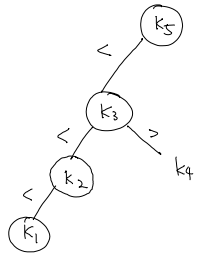
Input: n keys $k_1 < k_2 < \dots < k_n$, frequencies $f_1, f_2, \dots, f_n$ with $\sum_{i=1}^{n} f_i = 1$.

Output: a binary search tree T that minimizes the objective value $\sum_{i=1}^{n} f_i \cdot \text{depth}_T(k_i)$.

For example, given $f_1 = 0.1$ $f_2 = 0.2$ $f_3 = 0.25$ $f_4 = 0.05$ $f_5 = 0.4$,



objective value
$= 0.25 \times 1 + 0.2 \times 2 + 0.4 \times 2 + 0.1 \times 3 + 0.05 \times 3$
$= 1.9$



objective value
$= 0.4 \times 1 + 0.25 \times 2 + 0.2 \times 3 + 0.05 \times 3 + 0.1 \times 4$
$= 2.05$

In the above example, even though k_5 has the highest frequency, it is not necessarily optimal to put k_5 at the root for it to have the minimum depth.

So, this is unlike the prefix coding problems, in which keys with higher frequencies will have smaller depth.

This is because we have to maintain the binary search tree structure, such that smaller keys have to be put on the left subtree while larger keys have to be put on the right subtree.

This restriction turns out to be useful in setting up the recurrence relation.

Dynamic Programming

The subproblem structure is slightly different from those that we have seen before.

In the following, we let $F_{i,j} = \sum_{\ell=i}^{j} F_\ell$. To handle boundary cases, we let $F_{i,j} = 0$ for $i > j$.

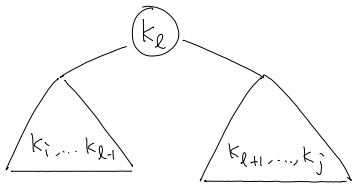
Subproblems: Let $C(i,j)$ be the objective value of an optimal binary search tree for keys $k_i < \dots < k_j$.

Answer: $C(1,n)$.

Base cases: $C(i,i) = f_i$ for $1 \le i \le n$. To handle boundary cases, we also set $C(i,i-1) = 0$ $\forall i$.

Recurrence: To compute $C(i,j)$, we try all the possible root of the binary search tree.

For $i \le \ell \le j$, if we set k_ℓ to be the root, then the keys $k_i, \dots, k_{\ell-1}$ must be on the left subtree of the root, while keys $k_{\ell+1}, \dots, k_j$ must be on the right subtree of the root.



The two subtrees can be computed independently of each other, as there are no more constraints between the two subtrees.

So, the best way is to find an optimal binary search tree for $k_i, \dots, k_{\ell-1}$ on the left, and an optimal binary search tree for $k_{\ell+1}, \dots, k_j$ on the right.

Therefore, $$C(i,j) = \min_{i \le \ell \le j} \left\{ \underbrace{f_\ell}_{\text{root}} + \underbrace{F_{i,\ell-1} + C(i,\ell-1)}_{\text{left subtree}} + \underbrace{F_{\ell+1,j} + C(\ell+1,j)}_{\text{right subtree}} \right\}$$

$$= \min_{i \le \ell \le j} \left\{ F_{i,j} + C(i,\ell-1) + C(\ell+1,j) \right\}$$

Note that the terms $F_{i,\ell-1}$ and $F_{\ell+1,j}$ are added because the keys $k_i, \dots, k_{\ell-1}$ and the keys $k_{\ell+1}, \dots, k_j$ are put one level lower, and so the two terms account for the increase in the objective value.

Correctness follows from the justification of the recurrence formula and by induction.

Time Complexity There are no more than n^2 subproblems

Each subproblem looks up no more than n values.

Using top-down memorization, the total time complexity is $O(n^3)$.

Bottom-up implementation This problem requires more care to write it correctly.

We will solve the subproblems with $j-i=1$ first, and then $j-i=2$, and so on.

```
C(i,i-1) = 0   for 1 ≤ i ≤ n          // boundary cases
Compute F_{i,j} for all 1 ≤ i,j ≤ n    // can be done in O(n^2) time using partial sums
For 1 ≤ width ≤ n-1 do                 // from short intervals to long intervals
    For i from 1 to (n-width) do
        j = i + width.   C(i,j) = ∞.
        For ℓ from i to j do
            C(i,j) ← min { C(i,j), F_{i,j} + C(i,ℓ-1) + C(ℓ+1,j) }.
```

It is clear the time complexity is $O(n^3)$ as there are three for-loops.

Tracing out solution : Exercise.

Faster Algorithm : With additional observations. Knuth used the same subproblems but showed how to solve the problem in $O(n^2)$ time!

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 14: Dynamic Programming on Graphs

We use dynamic programming to design algorithms for graphs, including shortest paths problems and TSP.

Single-Source Shortest Paths with Arbitrary Edge Lengths

Input: A directed graph $G=(V,E)$, an edge length l_e for each edge $e \in E$, and a vertex $s \in V$.

Output: The shortest path distances from s to every vertex $v \in V$.

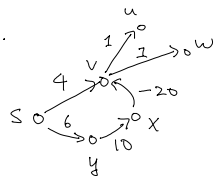
Dijkstra's Algorithm

This problem is solved using Dijkstra's algorithm in the special case where the edge lengths are non-negative, which runs in near-linear time.

It turns out that allowing negative edge lengths makes the problem considerably harder.

Let's first see why Dijkstra's algorithm does not work in this more general setting.

In Dijkstra's algorithm, we maintain a set $R \subseteq V$ so that dist[v] is computed correctly for all $v \in R$, and then we grow R greedily by adding a vertex $v \notin R$ closest to R into R.



With the presence of negative edge lengths, however, this greedy algorithm does not maintain this invariant anymore.

In the example, the vertex v is closest to s and is added to R first, but dist[v] $\neq 4$ as the path s,y,x,v is of length -4 because of the negative edge xv

The correctness of Dijkstra's algorithm crucially uses that the length of a prefix of a path cannot be shorter than the length of the path, and this doesn't hold anymore with the prescence of negative edges.

With this wrong start, Dijkstra's algorithm will add u and w to R.

Only after vertex x is added and explored later, we realize that there is a shorter path from s to v via x.

Then, we know that the distances to u and w are not computed correctly.

To fix it, we need to use the new distance to v to update the distance to u and w, but then we can no longer say that each vertex is only explored once.

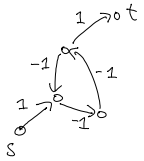
We can extend this example so that this update process needs to be done many times.

This is where we could not maintain the near-linear time complexity for solving this more general problem.

Negative Cycles

Another issue of having negative edges is that there may exist negative cycles.

In the example, from s to t, we can go around the negative cycle as many times as we want, and so the shortest path distance is not even well-defined.



In the following, we will study algorithms to solve the following problems.

① If G has no negative cycles, solve the single-source shortest paths problem.

② Given a directed graph G, check if there exists a negative cycle C, i.e. $\sum_{e \in C} l_e < 0$.

Bellman-Ford Algorithm [KT 6.8]

Intuition: Although Dijkstra's algorithm may not compute all distances correctly in one pass, it will compute the distances to some vertices correctly, e.g. the first vertex on a shortest path.

In the example above, dist[y] will be computed correctly.

Then, if we do the update on every edge again, then we would get dist[x] right for sure.

Then, with one more update phase on every edge, then we would get dist[v] correct and so on.

How many times we need to do?

If the graph has no negative cycles, then any shortest walk must be a simple path, which has at most $n-1$ edges.

So, by repeating the updating phases at most $n-1$ times, we should have computed all shortest path distances correctly, with time complexity about $O(nm)$.

This is basically the Bellman-Ford algorithm.

Dynamic Programming

To formalize the above idea, we design an algorithm using dynamic programming to compute the shortest path distance from s to every vertex $v \in V$ using at most i edges, from $i=1$ (base case) to $i=n-1$.

Then we will show that this is equivalent to the Bellman-Ford algorithm we see in textbooks.

Subproblems: Let $D(v,i)$ be the shortest path distance from s to v using at most i edges.

Answers: For each $v \in V$, $D(v, n-1)$ is the shortest path distance from s to v when the graph has no negative cycles.

Base cases: $D(s,0) = 0$ and $D(v,0) = \infty$ for all $v \in V-s$.

Recurrence: To compute $D(v, i+1)$, note that a path with at most $i+1$ edges from s to v must be coming from a path using at most i edges from s to u for an in-neighbor u of v.

Since we are to compute the shortest path distance from s to v using at most $i+1$ edges, we should use a shortest path using at most i edges from s to u.

We try all possibilities of u and get the recurrence relation

$$D(v, i+1) = \min \left\{ D(v,i), \min_{u: uv \in E} \{ D(u,i) + l_{uv} \} \right\}.$$

Time complexity Given $D(v,i)$ for all $v \in V$, it takes in-deg(w) time to compute $D(w,i+1)$.

So, the time to compute $D(w,i+1)$ for all $w \in V$ is $O\left(\sum_{w \in V} \text{in-deg}(w)\right) = O(m)$.

We do this for $1 \le i \le n-1$, thus the total time complexity is $O(nm)$.

Space Complexity A direct implementation requires $\Theta(n^2)$ space, to store all the values $D(v,i)$.

Note that to compute $D(w,i+1)$ $\forall w \in V$, we just need the values $D(v,i)$ $\forall v \in V$ but don't need $D(v,j)$ for $j \le i-1$, and so we can throw these away and only use $O(n)$ space.

Simple algorithm The algorithm can be made even simpler, matching the intuition that we mentioned in the beginning.

```
dist[s] = 0,   dist[v] = ∞   ∀ v ∈ V-s.
for i from 1 to n-1 do
    for each edge uv ∈ E do
        if dist[u] + l_uv < dist[v]
            dist[v] = dist[u] + l_uv   and   parent[v] = u.
```

This is the Bellman-Ford algorithm. The simplification is that we don't need to use two arrays.

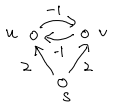
To see why we don't need two arrays, note that using one array could only have the intermediate distances smaller, and they remain to be upper bounds on the true distances, so using tighter upper bounds would not hurt (and may speed up in practice).

Shortest Path Tree

As in Dijkstra's algorithm, we would like to return a shortest path from s to v by following the edges (parent[v], v).

In Bellman-Ford, there are many iterations in the outerloop, and it is not clear whether these edges still form a tree.

Actually, it is possible to have a directed cycle in the edges (parent[v], v),



but the following lemma shows that these directed cycles must be negative cycles.

Lemma If there is a directed cycle C in the edges (parent[v], v),

then the cycle C must be a negative cycle, i.e. $\sum_{e \in C} l_e < 0$.

Proof Let the directed cycle C be $v_1, v_2, \ldots, v_k$, with $v_i v_{i+1} \in E$ $\forall\, 1 \le i \le k-1$ and $v_k v_1 \in E$.

Assume that $v_k v_1$ is the last edge in the cycle C formed in the algorithm, i.e.

the cycle C formed when v_k becomes the parent of v_1, while $\text{parent}[v_i] = v_{i-1}$ already for $2 \le i \le k$.

Consider the values $\text{dist}[v_i]$ right before v_k becomes the parent of v_1.

Since v_{i-1} is the parent of v_i, we have $\text{dist}[v_i] \ge \text{dist}[v_{i-1}] + l_{v_{i-1}v_i}$ for $2 \le i \le k$.

(Note that at the time when we set $parent[v_i] = v_{i-1}$, the inequality holds as an equality, but later $dist[v_{i-1}]$ could decrease and it may become an inequality.

Note also that we cannot have $dist[v_i] < dist[v_{i-1}] + \ell_{v_{i-1}v_i}$ as otherwise $parent[v_i]$ would be updated.)

Now, when we set $parent[v_1] = v_K$, it must be because $dist[v_1] > dist[v_K] + \ell_{v_K v_1}$ at that time.

Adding all these k inequalities, we have $\sum_{i=1}^{k} dist[v_i] > \sum_{i=1}^{k} dist[v_i] + \sum_{e \in C} \ell_e$, which implies that $\sum_{e \in C} \ell_e < 0$. □

The lemma implies that if there are no negative cycles, then there are no directed cycles in the edges $(parent[v], v)$.

Assuming that every vertex can be reached from vertex s, then every vertex has exactly one incoming edge in $(parent[v], v)$, and there are no directed edges by the lemma.

So, the edges $(parent[v], v)$ must form a directed tree, i.e. a tree with edges pointing away from s.

To conclude, when there are no negative edges, the edges $(parent[v], v)$ form a shortest path tree from s.

Negative Cycles [KT 6.10]

Ideas: We can extend the dynamic programming algorithm to identify a negative cycle if it exists.

Even with negative cycles, after k iterations of the dynamic programming algorithm, the same recurrence relation proves that we compute the shortest path distance from s to v using at most k edges $\forall v \in V$.

To solve the single-source shortest paths problem, we only used the assumption that there are no negative cycles to prove that the algorithm can stop after $n-1$ iterations and conclude that distances are computed correctly.

If there are are negative cycles, then we would expect that $D(v,k) \to -\infty$ as $k \to \infty$ for some $v \in V$.

For example, in the graph 

, we have $D(t,3) = -1$, $D(t,6) = -4$, $D(t,9) = -7$, and so on.

On the other hand, if there are no negative cycles, then we expect that $D(v,n) = D(v,n-1)$ for all $v \in V$, and this implies that $D(v,\infty) \not\to -\infty$ as $k \to \infty$ for all $v \in V$.

So, intuitively, by checking if $D(v,n) = D(v,n-1)$ $\forall v \in V$, we can determine if there is a negative cycle or not.

Assumption: In the following, we assume that every vertex can be reached from s.

For the problem of finding a negative cycle, this is without loss of generality since we can restrict our attention to strongly connected components and we learnt from L07 how to identify all SCCs in linear time.

Observations

We make the above ideas precise by the following claims.

Claim 1 If the graph has a negative cycle, then $D(v,k) \to -\infty$ as $k \to \infty$ for some $v \in V$.

Proof This follows from the definition of $D(v,k)$ and the assumption that every vertex can be reached from s. □

Claim 2 If the graph has no negative cycles, then $D(v,n) = D(v,n-1)$ for all $v \in V$.

Proof Any cycle is non-negative, so we can assume that any shortest walk from s to v has no cycles, and thus it is of length at most $n-1$. □

Claim 3 If $D(v,n) = D(v,n-1)$ for all $v \in V$, then the graph has no negative cycles.

Proof If $D(v,n) = D(v,n-1)$ for all $v \in V$, then $D(v,n+1) = D(v,n)$ for all $v \in V$, as the recurrences are the same.

More precisely, the recurrences are $D(v,n) = \min\left\{ D(v,n-1),\ \min_{u:uv \in E}\{ D(u,n-1) + l_{uv} \} \right\}$. (*).

So, $D(v,n+1) = \min\left\{ D(v,n),\ \min_{u:uv \in E}\{ D(v,n) + l_{uv} \} \right\}$

$= \min\left\{ D(v,n-1),\ \min_{u:uv \in E}\{ D(u,n-1) + l_{uv} \} \right\}$ (because $D(v,n) = D(v,n-1)$ $\forall v$ by assumption)

$= D(v,n)$ (because of (*)).

Hence, by induction, $D(v,k) = D(v,n-1)$ $\forall v$ $\forall k \geq n-1$, and thus $D(v,k)$ is finite when $k \to \infty$ for all $v \in V$.

Therefore, by claim 1, there are no negative cycles in the graph. □

Note that the same proof as in Claim 3 shows that as long as $D(v,k+1) = D(v,k)$ $\forall v \in V$, then $D(v,l) = D(v,k)$ $\forall v \in V$ $\forall l > k$, and so we can stop in that iteration with all distances computed correctly. This provides an early termination rule that is useful in practice (when shortest paths have few edges).

Algorithms

Checking: Claim 2 and 3 together imply that a graph has no negative cycles iff $D(v,n-1) = D(v,n)$ $\forall v \in V$.

Since we can compute $D(v,n)$ and $D(v,n-1)$ $\forall v \in V$ in $O(mn)$ time, this implies an $O(mn)$ time algorithm for checking.

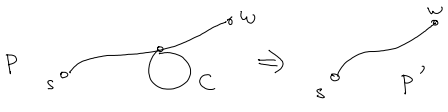
Finding: The next question is: if $D(w,n) < D(w,n-1)$ for some w, how do we find a negative cycle?

Here we assume that we use the $\Theta(n^2)$-space dynamic programming algorithm for computing $D(v,n)$, and that we have stored $parent(w,n) = u$ if $D(w,n) = D(u,n-1) + l_{uw}$.

First, since $D(w,n) < D(w,n-1)$, we know that the path P from s to w with total length $D(w,n)$ and at most n edges must have exactly n edges, as otherwise $D(w,n-1) = D(w,n)$.

A path of length n must have a repeated vertex, and thus a cycle C.

We claim that C must be a negative cycle.



Suppose not, that C is a non-negative cycle.

Then, we can skip the cycle C to get a path P' with fewer edges than that in P and $length(P') \leq length(P)$ since C is a non-negative cycle.

But that would imply that $D(w,n-1) \leq length(P')$ since P' has at most $n-1$ edges, and thus $D(w,n-1) \leq length(P') \leq length(P) = D(w,n)$, a contradiction.

So, the cycle C must be a negative cycle.

By tracing out the parents using the stored information, we can find P and thus the cycle C.

This gives an $O(mn)$-time algorithm to find a negative cycle, using $\Theta(n^2)$ space.

Space-efficient implementation There is also an $O(mn)$-time algorithm using only $O(n)$ space.

The details are more involved and we refer to [KT 6.10].

All-Pairs Shortest Paths [DPV 6.6]

Input: A directed graph $G=(V,E)$, an edge length l_e for $e \in E$.

Output: The shortest path length from s to t, for all $s,t \in V$.

We can solve this problem by running Bellman-Ford for each $s \in V$.

This would take $O(n^2m)$ time, which could be $\Theta(n^4)$ when $m=\Theta(n^2)$.

It is possible to solve the all-pairs shortest paths problem in $O(n^3)$ time, using a different recurrence.

Here we present the Floyd-Warshall algorithm.

Dynamic Programming

In the Floyd-Warshall algorithm, more subproblems are used to store information for each pair of vertices.

Subproblems: Let the vertex set V be $\{1,2,\ldots,n\}$.

Let $D(i,j,k)$ be the length of a shortest path from vertex i to vertex j using only vertices $\{1,\ldots,k\}$ as intermediate vertices in the path.

(Another perhaps more natural choice is $D'(i,j,k)$ which denotes the length of a shortest path from i to j using at most k edges similar to that in the Bellman-Ford algorithm.

We leave it as a question to think about $D'(i,j,k)$ doesn't work as well as the Floyd-Warshall subproblems.)

Answers: $D(i,j,n)$ $\forall i,j \in V$.

Base cases: $D(i,j,0) = l_{ij}$ if $ij \in E$ and $D(i,j,0) = \infty$ if $ij \notin E$.

This is because $D(i,j,0)$ is asking for the shortest path length from i to j without using intermediate vertices.

Recurrence: Assume $D(i,j,k)$ are computed correctly $\forall i,j \in V$ for some k.

We would like to compute $D(i,j,k+1)$ $\forall i,j \in V$.

The only difference between $D(i,j,k+1)$ and $D(i,j,k)$ is that $D(i,j,k+1)$ is allowed to use vertex $k+1$ as an intermediate vertex, while $D(i,j,k)$ is not allowed to do so.

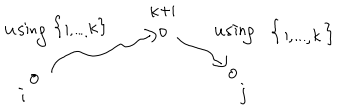
To use vertex $k+1$ as an intermediate vertex for a path between i and j, the path has to go from i to $k+1$ and then from $k+1$ to j.

What is the optimal way to do it using only vertices $\{1,2,\dots,k+1\}$ as intermediate vertices?

Of course, we should use a shortest path from i to $k+1$ using $\{1,2,\dots,k\}$ as intermediate vertices, and a shortest path from $k+1$ to j using $\{1,2,\dots,k\}$ as intermediate vertices.

Note that we don't need to use vertex $k+1$ more than once, as there are no negative cycles.

Therefore, $D(i,j,k+1) = \min\{ D(i,j,k),\ D(i,k+1,k) + D(k+1,j,k) \}$, where the first term considers the paths not going through $k+1$, while the second term consider paths that use vertex $k+1$.



Floyd-Warshall algorithm

$D(i,j,0) = \infty \quad \forall ij \notin E$. $\quad D(i,j,0) = l_{ij} \quad \forall ij \in E$. // base cases

for k from 1 to n do // allowing more and more intermediate vertices

  for i from 1 to n do } // going through all pairs

    for j from 1 to n do

      $D(i,j,k+1) = \min\{ D(i,j,k),\ D(i,k+1,k) + D(k+1,j,k) \}$.

Time complexity: It is clear that the runtime is $O(n^3)$.

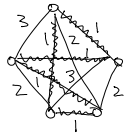
Exercise: Given $s,t \in V$, return a shortest path from s to t.

Open problem: It has been a long standing open problem whether there exists a truly sub-cubic time algorithm for computing all-pairs shortest paths (i.e. $O(n^{3-\varepsilon})$-time for some constant $\varepsilon > 0$, e.g. $O(n^{2.99})$).

Traveling Salesman Problem [DPV 6.6]

Input: An undirected graph $G=(V,E)$, with a non-negative edge length l_{ij} for all $ij \in V$.

Output: A cycle C that visits every vertex exactly once and minimizes $\sum_{e \in C} l_e$.



It is one of the most famous problem in combinatorial optimization.

As we will show in the last part of the course, this problem is NP-hard.

There is a naive algorithm for this problem, by enumerating all possible orderings to visit the vertices.

This will take $\Theta(n! \cdot n)$ time. It becomes too slow when $n=13$.

We present a dynamic programming solution that can probably work up to $n=30$.

Dynamic Programming

The difficulty of the problem is that it is not enough to remember only the shortest paths, but also what vertices that we have visited so that what other vertices yet to visit.

The recurrence is unlike everything that we have seen so far, as it has exponentially many subproblems!

Subproblems: Let $C(i,S)$ be the length of a shortest path to go from 1 to i, with vertices in S on the path.

(Note that we don't care about the ordering of vertices in S in the path, and this is where the speedup over the naive algorithm is coming from.)

Answers: $\min_{i \in V} \{ C(i,V) + l_{i1} \}$ - from 1 to i using all vertices in V once, then come back to 1.

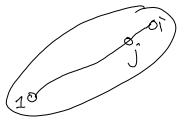
Base cases: $C(i, \{1,i\}) = l_{1i}$ for all $i \in V$.

Recurrence: Suppose we have computed $C(i,S)$ for all subsets S of size k.

We would like to use these to compute $C(i,S)$ for all subsets S of size $k+1$.

To compute $C(i,S)$ for S of size $k+1$, we try all possibilities of the second last vertex on the path.

Note that the second last vertex must be from S, and of course the best way to reach the second last vertex j is to use a shortest path from 1 to i that every vertex in $S-\{i\}$ exactly once.



Therefore, $C(i,S) = \min_{j \in S-\{i\}} \{ C(j, S-\{i\}) + l_{ji} \}$.

Algorithm: Exercise.

Time Complexity: There are $O(n \cdot 2^n)$ subproblems, each requiring $O(n)$ time to compute.

So, the total time complexity is $O(n^2 \cdot 2^n)$.

The main drawback of this algorithm is that the space complexity is $\Theta(n \cdot 2^n)$.

Concluding Remark: We have seen many examples and structures to design dynamic programming algorithms, from lines to trees to graphs.

With the help of homework problems and supplementary exercises, I hope that you will be familiar with this technique, with which you could solve a much larger class of interesting problems.

In 2022, there is a new paper "Negative-weight Single-Source Shortest-paths in near linear time", solving a long standing open question on this fundamental problem.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 15: Maximum Flow and Minimum Cut

We first introduce the concepts of flows and cuts in a directed graph.

Then we study the Ford-Fulkerson algorithm for finding a maximum flow and the celebrated max-flow-min-cut theorem.

Flows and Paths

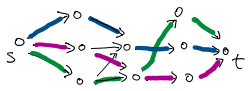
Using BFS / DFS, we can determine whether there is a path from vertex s to vertex t in linear time.

It is natural to consider the problem of finding 2 edge-disjoint paths from s to t, and more generally as many edge-disjoint paths from s to t as possible.

Edge-Disjoint Paths

Input: A directed graph $G=(V,E)$, and two vertices $s,t \in V$.

Output: A maximum number of edge-disjoint paths from s to t.



(Two paths P_1 and P_2 are edge-disjoint if they do not share an edge.)

This is basically the maximum flow problem, where the union of the paths defines a "flow" from s to t.

The maximum flow problem is defined on edge-capacitated graph, and the concept of flows is easier to work with than edge-disjoint paths.

s-t Flows

Let $G=(V,E)$ be a directed graph, where each edge $e \in E$ has an integer capacity $c(e) > 0$.

An s-t flow assigns a number $f(e)$ to each edge and satisfies the following two properties:

(i) Capacity constraints: $0 \leq f(e) \leq c(e)$ for each edge $e \in E$.

(ii) Flow conservation constraints: $f^{in}(v) = f^{out}(v)$ for each vertex $v \in V-s-t$ other than s and t, where $f^{in}(v) = \sum_{e \text{ into } v} f(e)$ is the total incoming flow into v and $f^{out}(v) = \sum_{e \text{ out of } v} f(e)$ is the total outgoing flow from v.

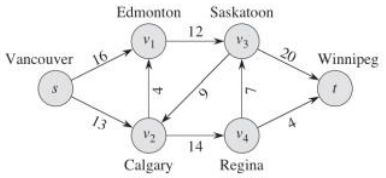
The value of the flow is defined as $\text{value}(f) = f^{out}(s) - f^{in}(s)$, the net flow going out of the source s.

(In our algorithms, there won't be incoming flows to s, and so value(f) would simply be $f^{out}(s)$.)

Maximum s-t Flow

Input: A directed graph $G=(V,E)$ with a capacity $c(e)$ on each edge $e \in E$, and two vertices $s,t \in V$.

Output: A flow from s to t with maximum value.



[CLRS]

As you may imagine, this problem is fundamental and useful for modeling problems such as liquids flowing through pipes, currents through electrical networks, and information through communication networks.

But it turns out that it is also useful for solving seemingly unrelated problems that we'll see next time.

In the above example for edge-disjoint paths where every edge is of capacity one, the maximum flow from s to t is 3, by setting the flow on each colored edge to be one.

The connection between flows and paths is that a flow can always be decomposed into "capacity-disjoint" paths.

Lemma (decomposition of s-t flow into capacity-disjoint s-t paths)

Let f be an s-t flow where $f(e)$ is an integer for each edge e, with $f^{in}(s)=0$ and $value(f)=k$.

Then there are s-t paths $P_1, P_2, \dots, P_k$ such that each edge e appears in $f(e)$ of these paths.

(For example, if $f(e)=0$, then the edge e will not appear in any of $P_1, \dots, P_k$, while if $f(e)=2$, then e will appear in exactly 2 of these k paths.)

Proof (Sketch) The proof is by a simple induction. The base case is when $k=1$, where the statement holds.

Consider the edges with non-zero flow value. There must exist an s-t path P in these edges (why?).

Decrease the flow of each edge in P by one. i.e. $f'(e) := f(e)-1$ if $e \in P$, otherwise $f'(e) := f(e)$.

Check that f' is a flow with value $k-1$, as each vertex other that s,t still satisfies conservation constraint.

By induction, there are s-t paths $P_1, P_2, \dots, P_{k-1}$ so that each edge e appears in $f'(e)$ of these $k-1$ paths.

Then, it should be clear that $P_1, P_2, \dots, P_{k-1}, P$ are k paths that the lemma guarantees. □

For example, when the flow value is k and $f(e) \in \{0,1\}$ $\forall e \in E$, then the above lemma says that the flow f can be decomposed into k edge-disjoint paths.

So, even our goal is to find k edge-disjoint paths, it would be easier for us to focus on finding a flow of value k with $f(e) \in \{0,1\}$ $\forall e \in E$ instead, so that we don't need to worry about which edges belong to which paths during the algorithm, and only decompose the flow into edge-disjoint paths in the end.

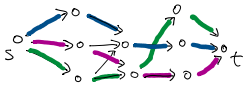
Exercise: Find a decomposition of the flow in the Canadian example above.

Flows and Cuts

What is a good upper bound on the value of a maximum flow? How do we know that a flow is maximum?

A trivial upper bound is the total capacity of all the edges, but it is almost never tight.

In the example for edge-disjoint paths, we know that there are at most 3 edge-disjoint s-t paths because the out-degree of s is 3.



We consider a generalization of this upper bound to a subset of vertices.

s-t Cut

Let $G=(V,E)$ be a directed graph, where each edge $e \in E$ has an integer capacity $c(e) > 0$.

A subset of vertices $\phi \neq S \subset V$ is an s-t cut if $s \in S$ and $t \notin S$, i.e. the partition $(S, V-S)$ separates s and t.

The capacity of an s-t cut S is defined as $c^{out}(S) := \sum_{e \in \delta^{out}(S)} c(e)$, the total capacity of the edges going out of S, where $\delta^{out}(S) := \{ uv \in E \mid u \in S, v \notin S \}$ is the set of directed edges going out of S.

Upper bounding max-s,t-flow by s-t cut

In the edge-disjoint paths problem where every edge is of capacity one, the capacity of an s-t cut S is simply the number of edges in $\delta^{out}(S)$. Note that the outdegree of s is the special case when $S = \{s\}$.

If an s-t cut S has at most k edges in $\delta^{out}(S)$, then it should be clear that there cannot be more than k edge-disjoint s-t paths, because each s-t path must have at least one edge in $\delta^{out}(S)$.

This idea can be generalized to maximum flow: If an s-t cut S has capacity k, then the value of a maximum s-t flow must be at most k.

One way to see this is through the decomposition of flow into paths in the lemma above. We leave it as an exercise.

We will give another proof of this upper bound later.

Minimum s-t Cut

To give the best upper bound on the value of a maximum s-t flow, we consider the minimum s-t cut problem.

Input: A directed graph $G=(V,E)$ with a capacity $c(e)$ on each edge $e \in E$, and two vertices $s, t \in V$.

Output: An s-t cut S with minimum capacity $c^{out}(S)$.

This is a natural and useful problem on its own, and we will see some interesting application next time.

Max-Flow Min-Cut Theorem

It turns out that having a small s-t cut is the only obstruction of having a large s-t flow, providing a concise and elegant proof of optimality for both problems.

Theorem The value of maximum s-t flow is equal to the capacity of a minimum s-t cut.

This is one of the most beautiful and important results in combinatorial optimization and graph theory.

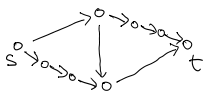
and has diverse applications in Computer Science and mathematics.

We will give an algorithmic proof of this theorem, that solves the maximum s-t flow problem and the minimum s-t cut problem at the same time.

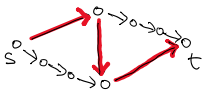
Ford-Fulkerson Algorithm

A natural strategy to find edge-disjoint s-t paths is to say finding a shortest s-t path (so that it uses the minimum number of edges), then finding another shortest s-t path in the remaining graph, and so on.

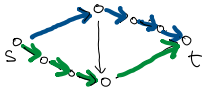
But this approach does not necessarily obtain an optimal solution.



input graph



greedy solution

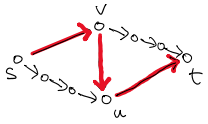


optimal solution

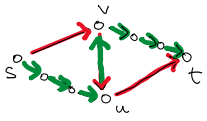
The difficulty of the greedy approach is to commit on a path that we have chosen, and there seems to be no good way to find an s-t path that belongs to an optimal solution.

The Ford-Fulkerson algorithm can be understood as a more general "local search" method, in which we may "undo" the decisions that we have made in earlier iterations, but only when we can improve the objective value of the solution (in this case the value of the flow).

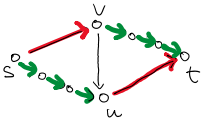
For the above example, we will find a path and "push back" some of the flow to obtain the optimal solution.



greedy solution



"augmenting path"



optimal solution

In the middle picture, we send a flow from s to u, "undo / push-back" the flow from v to u in the greedy solution, and send a flow from v to t to obtain the optimal solution.

To define the augmenting path more precisely, it will be more convenient to introduce an auxiliary graph called the "residual graph", so that we don't need to distinguish between "push-forward / push-back".

Residual Graph

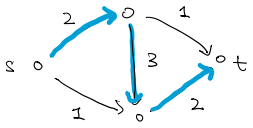
Given a directed graph G and a flow f on G, the residual graph G_f of G with respect to f is defined as follows:

- The vertex set of G_f is the same as the vertex set of G.
- For each edge e on which $f(e) < c(e)$, there are $c(e) - f(e)$ "leftover" units of capacity on which we can push flow forward. So we include the edge e in G_f, with a capacity $c(e) - f(e)$. We will call

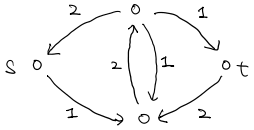
edges included this way the forward edges.

- For each edge $e = uv$ on which $f(e) > 0$, there are $f(e)$ units of flow that we can "undo" by pushing the flow backward. So we include the reverse edge $e' = vu$ in G_f, with a capacity of $f(e)$. We will call edges included this way the backward edges.

For an example,



a directed graph with a flow of 2 units



residual graph wrt the flow

Pushing Flow on an Augmenting Path

An augmenting path with respect to a flow f is just a simple s-t path in G_f.

Let P be an augmenting path wrt to f. Define bottleneck(P, f) as the minimum capacity of an edge on P in G_f.

The following is the subroutine to improve the flow f by using an augmenting path P.

```
augment (f, P)
    let  b = bottleneck (P, f)
    for each edge e = uv on P
        if e is a forward edge
            increase f(e) by b in G
        else if e is a backward edge
            let e' = vu be the reverse edge
            decrease f(e') by b in G
```

We show that the subroutine augment(f, P) will always improve the value of the current flow f.

Lemma Let f be a flow in G with $f^{in}(s) = 0$, and P be an augmenting path with respect to f. Let f' be the resulting flow after the subroutine augment(f, P) is called. Then f' is a flow with value(f') = value(f) + bottleneck(P, f) and $f'^{in}(s) = 0$.

Proof First, we check that f' is a flow, for which we need to check the capacity and conservation constraints.

It should be clear by construction that f' still satisfies the capacity constraint in G.

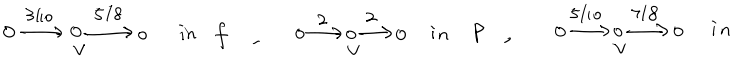
If e is a forward edge, then $0 \le f'(e) = f(e) + \text{bottleneck}(P, f) \le f(e) + (c(e) - f(e)) \le c(e)$;

while if e is a backward edge, then $c(e') \ge f'(e') = f(e') - \text{bottleneck}(P, f) \ge f(e') - f(e') \ge 0$,

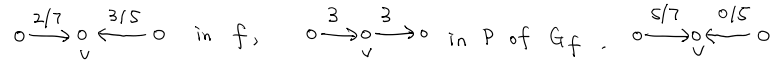
where e' is the reverse edge of e (i.e. if $e = uv$, then $e' = vu$).

For the conservation constraints, we verify that the change of the amount of flow entering v is equal to the change of the amount of flow leaving v, for any $v \in V-s-t$.

There are 4 cases to check, depending on whether the edges entering / exiting v are forward / backward edges.

Case 1: forward / forward 

In this case, both the incoming flow to v and the outgoing flow from v are increased by b.

Case 2: forward / backward 

In this case, both the incoming flow to v and the outgoing flow from v are unchanged.

Case 3: backward / forward, Similar to case 2 with both unchanged.

Case 4: backward / backward, Similar to case 1 with both decreased by b.

Since conservation constraints were satisfied in f, they continue to be satisfied in f'.

Finally, we check that $\text{value}(f') = \text{value}(f) + \text{bottleneck}(P, f)$.

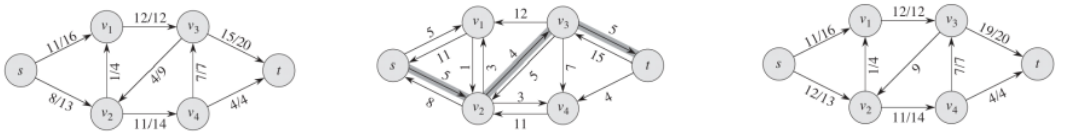
Since $f^{in}(s) = 0$, there are no backward edges incident on s in G_f.

Since P is a simple s-t path in G_f, the vertex s is only visited once with an outgoing forward edge.

Therefore, by augmenting f with P, we maintain the property that $f'^{in}(s) = 0$ and thus

$$\text{value}(f') = f'^{out}(s) - f'^{in}(s) = f'^{out}(s) = f^{out}(s) + b = f^{out}(s) - f^{in}(s) + b = \text{value}(f) + b. \quad \square$$

Example: [CLRS]



flow f | residual graph G_f and P | new flow $f' = \text{augment}(f, P)$

Remark: It is in the above lemma where it is easier to maintain a flow than to maintain a path decomposition, as for a flow we only need to check capacity and conservation constraints.

Ford-Fulkerson Algorithm

The above lemma shows that if we can find an augmenting path P in the residual graph G_f of the current flow f, then we can use it to obtain a flow f' with larger value.

The Ford-Fulkerson algorithm is simply to repeat this operation until we cannot do so.

```
initially, f(e) = 0 for all edges e in G
while there is an s-t path P in G_f do
    f ← augment(f, P) and update the residual graph G_f.
```

This is it. In the next section, we will prove that this "local search" method always gives us

a maximum flow, i.e. whenever there is no augmenting path, the current flow is a maximum flow.

Max-Flow-Min-Cut Theorem

Our strategy to prove that Ford-Fulkerson algorithm always finds a maximum flow is to show that when there is no augmenting path, there is a cut with capacity equal to the value of the current flow, thereby proving the max-flow min-cut theorem as well as solving the minimum s-t cut problem simultaneously.

To do so, we need the following claim about the flow of any s-t cut, from which the easy direction of the max-flow min-cut theorem (i.e. max flow $\leq$ min cut) follows immediately.

Claim Let f be any s-t flow, and $\phi \neq S \subset V$ be any s-t cut (i.e. $s \in S$ and $t \notin S$).

Then $f^{out}(S) - f^{in}(S) = value(f) = f^{out}(s) - f^{in}(s)$.

Proof The intuition is that only the source vertex s has a positive "net output", while all other vertices in $V-s-t$ satisfy the flow conservation constraints (intuitively just passing the flow along), and so the "net output" of an s-t cut S is simply equal to the "net output" of s.

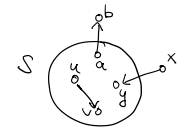
The proof is by a subtle manipulation:

$$value(f) = f^{out}(s) - f^{in}(s) = \sum_{v \in S} \left(f^{out}(v) - f^{in}(v) \right) \quad \text{// because } f^{out}(v) - f^{in}(v) = 0 \ \forall v \in V-s-t$$

$$= \sum_{v \in S} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \right) \quad \text{// by definition}$$

$$= \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(S)$$

$$= f^{out}(S) - f^{in}(S).$$



To see the second last equality, divide the edges involving in the sum into three types. Note that the edges involving in the sum in the second line must have at least one vertex in S.

(1) first type: $e = uv$, $u \in S$, $v \in S$. For this edge e, $f(e)$ appears positively in $f^{out}(u)$, and $f(e)$ appears negatively in $f^{in}(v)$, and so this edge contributes zero to the sum in the second line.

(2) second type: $e = ab$, $a \in S$, $b \notin S$, so ab is an outgoing edge of S. For this edge e, it only appears once positively in $f^{out}(a)$, and so contributes $f(e)$ to the sum.

(3) third type: $e = xy$, $x \notin S$, $y \in S$, so xy is an incoming edge of S. For this edge e, it only appears once negatively in $f^{in}(y)$, and so contributes $-f(e)$ to the sum.

This verifies the second last equality and thus completes the proof. □

Corollary Let f be any s-t flow, and $\phi \neq S \subset V$ be any s-t cut (i.e. $s \in S$ and $t \notin S$).

Then $value(f) = f^{out}(S) - f^{in}(S) \leq f^{out}(S) \leq c^{out}(S)$.

This is the easy direction of the max-flow min-cut theorem: the value of any s-t flow is at most

the capacity of any s-t cut, thus the value of a max s-t flow is at most the capacity of a min s-t cut.

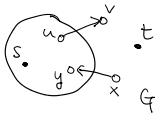
We are ready to prove the correctness of the Ford-Fulkerson algorithm and the max-flow-min-cut theorem.

Proposition. If f is an s-t flow such that there is no s-t path in the residual graph G_f, then there is an s-t cut S such that $\text{value}(f) = c^{out}(S)$.

Consequently, f has the maximum value of any s-t flow, and S has the minimum capacity of any s-t cut.

Proof As there is no path from s to t in G_f, by the result in BFS/DFS, there exists a subset S with $s \in S$, $t \notin S$ such that S has no outgoing edges in G_f. (Recall this?)

We claim that $c^{out}(S) = \text{value}(f)$. Consider two types of edges.



(1) Consider an edge $uv \in \delta^{out}(S)$ in G, with $u \in S$ and $v \notin S$.

Since S has no outgoing edge in G_f, we must have $uv \notin G_f$.

This implies that $f(uv) = c(uv)$, as otherwise uv would be a forward edge in G_f.

(2) Consider an edge $xy \in \delta^{in}(S)$ in G, with $x \notin S$ and $y \in S$.

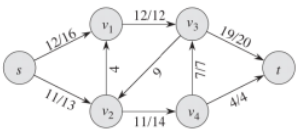
Since S has no outgoing edge in G_f, we must have $yx \notin G_f$.

This implies that $f(xy) = 0$, as otherwise yx would be a backward edge in G_f.

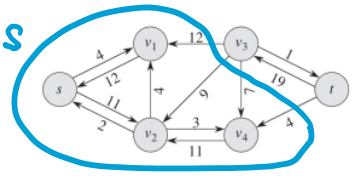
Putting together, it follows that $f^{out}(S) - f^{in}(S) = c^{out}(S) - 0 = c^{out}(S)$.

By the claim above, we conclude that $\text{value}(f) = f^{out}(S) - f^{in}(S) = c^{out}(S)$. □

Example: [CLRS]



maximum s-t flow f in G



residual graph G_f and minimum s-t cut

Note that this provides the correctness proof of the Ford-Fulkerson algorithm, as well as providing an algorithmic proof of the max-flow min-cut theorem.

Furthermore, this proof provides a linear time algorithm to compute a minimum s-t cut from a maximum s-t flow.

We will analyze the time complexity of the Ford-Fulkerson algorithm in the next section.

Time Complexity

It is not difficult to analyze the time complexity of the Ford-Fulkerson algorithm when all capacities are integers.

Claim Let $G = (V, E)$ be a directed graph where $c(e)$ is an integer for every $e \in E$.

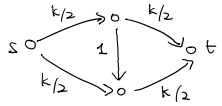
Let k be the value of a maximum s-t flow. The Ford-Fulkerson algorithm terminates in $O(k \cdot |E|)$ time.

Proof Each iteration can be implemented in $O(|E|)$ time, as searching an s-t path in G_f can be done by BFS/DFS.

If all capacities are integers, then bottleneck(P,f) is at least one.

This implies that the flow value is increased by at least one after each augmentation, and so there are at most k iterations in the algorithm. □

For the edge-disjoint s-t path problem in a simple directed graph, $k \le |V|$ and so the Ford-Fulkerson algorithm is a $O(|V| \cdot |E|)$-time algorithm for the problem.

The analysis of the above claim is tight, as there are examples where the algorithm takes k iterations. For example, in 

, there could be k iterations of augmentation (do you see why?).

So, in general, when k is large, the Ford-Fulkerson algorithm could be very slow.

Real-valued capacities: You may wonder why we made the assumption that all edge capacities are integers. Surprisingly, if we allow the edge capacities be real numbers, then there are pathological examples where the Ford-Fulkerson algorithm can run forever with pathological choices of augmenting paths.

Choosing good augmenting paths: So a natural question is whether we can choose augmenting paths in a good way that avoids these pathological examples.

Interestingly, perhaps the most natural implementation of the Ford-Fulkerson algorithm would not have any issue. Edmonds and Karp proved that if we always find a shortest s-t path in the residual graph G_f, then the resulting algorithm (which we call Edmonds-Karp algorithm) terminates in $O(|V||E|)$ iterations, even when the edge capacities are real-valued.

So, if we just implement the Ford-Fulkerson algorithm using BFS (which finds shortest s-t path), then the time complexity is at most $O(|V|^2 \cdot |E|)$.

Another natural strategy is to choose an augmenting path P that maximizes bottleneck(P,f), in a similar spirit to HW3 Q3.

If all edge capacities are integers, then it is not difficult to argue that there are at most $O(|E| \cdot \log k)$ iterations where k is the value of a maximum flow.

We won't go into details. You should be able to learn all these in the "Network Flows" course in C&O.

Concluding Remarks

There is a vast literature in designing faster algorithms for the maximum flow problem.

Very recently, in 2022, researchers finally found an almost linear-time algorithm for maximum flow (see the paper "Maximum flow and minimum cost flow in almost linear time").

Traditionally, maximum flow is solved using combinatorial techniques and sophisticated data structures.

In the past decade or so, however, researchers used a lot of tools and ideas from convex optimization to design faster algorithms for combinatorial problems.

The recent breakthrough is based on both convex optimization and sophisticated modern data structures.

As a related note, the maximum flow problem can be formulated as a linear programming problem.

Linear programming is a general framework to solve many combinatorial optimization problems.

The Ford-Fulkerson algorithm can be understood as a special instantiation of the simplex algorithm for solving linear programming problems.

The local search method is a general phenomenon for linear programming problems and more generally convex optimization problems, in which local minimums are global minimums.

You can take a look at chapter 7 of [DPV] for an introduction of this general framework, which is one of the most important paradigms in algorithm design.

References: [KT 7.1-7.3], [CLRS 26.1-26.2]

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 16: Applications of Max-Flow Min-Cut

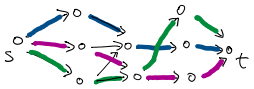
We see two basic applications and two fun applications of maximum flows and minimum cuts.

Disjoint Paths

As discussed in L15, the maximum edge-disjoint s-t path problem can be reduce to maximum s-t flow.

Input: A directed graph $G=(V,E)$, two vertices $s,t \in V$.

Output: A maximum-sized subset of edge-disjoint s-t paths



To recall the reduction, given a directed graph, simply set the capacity of every edge to be one.

Claim There are k edge-disjoint s-t paths if and only if there is an s-t flow of value k.

Proof On one hand, if $P_1, \ldots, P_k$ are edge-disjoint s-t paths, just set the flow on each edge on these paths to be one, and it is easy to check that this gives a flow of value k.

On the other hand, if f is an s-t flow of value k, then we can apply the flow decomposition lemma in L15 to obtain k edge-disjoint s-t paths. (This is also discussed in L15.) □

Check that the Ford-Fulkerson algorithm thus provides an $O(|V|\cdot|E|)$-time algorithm for edge-disjoint s-t paths.

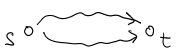
Also check that the max-flow min-cut theorem implies the following min-max theorem.

Theorem The maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal disconnects t from s (i.e. no paths from s to t after removal).

Vertex Disjoint Paths

What if we want s-t paths that are vertex disjoint?

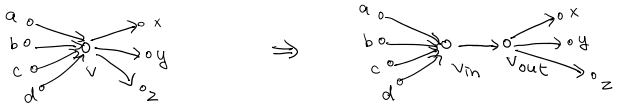
Two s-t paths P_1 and P_2 are called (internally) vertex-disjoint if they only share the vertices s and t, but not any other vertex.



It turns out that this problem can be reduced to maximum edge-disjoint s-t paths.

The idea is to create an edge to "imitate" a vertex.

Reduction: For each vertex $v \in V-s-t$, replace it by a two-vertex structure as follows:



Claim There are k vertex-disjoint s-t paths in the original graph if and only if there are k edge-disjoint s-t paths in the new graph.

This gives a $O(|V| \cdot |E|)$-time algorithm for the maximum vertex-disjoint s-t path problem, and the max-flow min-cut theorem implies the following min-max theorem.

Theorem The maximum number of vertex-disjoint s-t paths is equal to the minimum number of vertices whose removal disconnects t from s (i.e. no paths from s to t after removal).

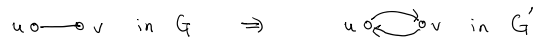
We leave the (straightforward) details to the reader.

Undirected Graphs

We could also ask the maximum edge-disjoint s-t path problem for undirected graphs.

This problem can also be reduced to the maximum edge disjoint s-t path problem for directed graphs.

The reduction is simple: for each edge uv in G, have both u→v and v→u in the directed graph G′.



Claim: There are k edge-disjoint s-t paths in G iff there are k edge-disjoint s-t paths in G′.

You may worry that both u→v and v→u are used in the edge disjoint paths in G′ and thus the corresponding undirected paths are not edge-disjoint, but we can argue that there is a solution in G′ such that no "anti-parallel" edges (i.e. u→v and v→u) are both used. We leave it to the reader to figure out.

So, we also have a polynomial time algorithm and a min-max theorem for undirected edge-disjoint s-t paths.

What about vertex-disjoint s-t paths in undirected graphs? We leave it as an exercise.

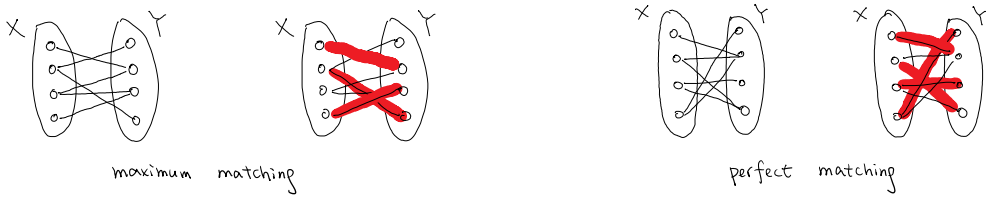
Exercise: Reduce maximum vertex-disjoint s-t paths in undirected graphs to a previous problem. Obtain a polynomial time algorithm and derive a min-max theorem for this problem.

Bipartite Matching

The bipartite matching problem is an important problem both in theory and in practice.

Input: A bipartite graph $G = (X, Y, E)$.

Output: A maximum cardinality subset of edges that are vertex disjoint.



A subset of edges $M \subseteq E$ is called a <u>matching</u> if edges in M are pairwise vertex disjoint, or in other words, no two edges in M share a vertex.

We are interested in finding a <u>maximum matching</u>, a matching with the maximum number of edges.

A matching is called a <u>perfect matching</u> if every vertex in the graph is matched.

Obviously, a perfect matching is the best that we can hope for for the maximum matching problem.

Reduction to Maximum Flow

The maximum bipartite matching problem can be reduced to the maximum flow problem.

Given a bipartite graph $G = (X, Y, E)$, we construct a directed graph G' as follows:

- The vertex set of G' is $\{s\} \cup X \cup Y \cup \{t\}$, where s and t are two new vertices.
- All the edges in E appear in G', but pointing from X to Y, with edge capacity being infinity.
  There is an edge from s to every vertex in X, with edge capacity one.
  There is an edge from every vertex in Y to t, with edge capacity one.

<u>Claim</u> There is a matching of size k in G iff there is an s-t flow of value k in G'.

<u>Proof</u> It should be clear from the picture that a matching of size k gives a flow of value k.

In the other direction, given an integer flow of value k (which is guaranteed by Ford-Fulkerson's algorithm), we apply the flow decomposition lemma in L15 to obtain k s-t paths of capacity one.

Note that each s-t path in G' has exactly 3 edges by our construction of G'.

Since each edge from s and to t has capacity one, these k paths must be (internally) vertex disjoint.

Removing the edges from s and the edges to t from these paths gives us k vertex disjoint edges, a matching of size k in G.

Through this reduction, the Ford-Fulkerson algorithm gives a $O(|V||E|)$-time algorithm for bipartite matching.

<u>Remark:</u> You may wonder why we assign edge capacity infinity to the original edges - but not edge capacity one. It will also work for this reduction, but infinity capacity will be more convenient later.

Minimum Vertex Cover

What should be the min-max theorem for bipartite matching? Consider the vertex cover problem.

Given a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is called a <u>vertex cover</u>

if for every edge $uv \in E$, $\{u,v\} \cap S \neq \phi$.

In words, S is a vertex cover if S intersects every edge.

Input: A bipartite graph $G=(V,E)$

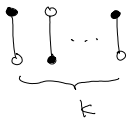
Output: A vertex cover of minimum cardinality.

Some examples: 

The vertex cover problem may seem to be unrelated to the matching problem, but they are actually "dual" of each other on bipartite graphs, in a precise and meaningful way.

To start to see their connection, first we observe that if there is a matching with k edges, then any vertex cover must have at least k vertices.

The reason is simple: Since the k matching edges are vertex disjoint, we must use k distinct vertices just to cover these k edges, and so any vertex cover must have at least k vertices.



Therefore, the size of a maximum matching is a lower bound on the size of a minimum vertex cover.

Surprisingly, this is always a tight lower bound on bipartite graphs.

Put it in another way, having a large matching is the only reason that we need a large vertex cover.

Theorem (König) In a bipartite graph, the maximum size of a matching is equal to the minimum size of a vertex cover.

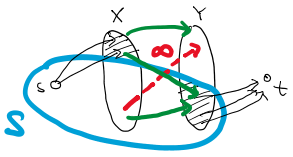
Proof We prove König's theorem using the max-flow min-cut theorem.

Let the maximum size of a bipartite matching in G be k. We would like to show a vertex cover of size k.

From the reduction above, the maximum s-t flow in G' above is of value k.

By the max-flow min-cut theorem, there is an s-t cut S in G' of capacity k, i.e. $c^{out}(S) = k$.

Consider such a set S, with $s \in S$ and $t \notin S$.



There can be at most k vertices in $(X-S) \cup (S \cap Y)$, because s has an edge of capacity one to each vertex in $X-S$, and every vertex in $Y \cap S$ has an edge of capacity one to t.

These edges are in $\delta^{out}(S)$ and so $k = c^{out}(S) \geq |X-S| + |S \cap Y|$.

We claim that the vertices in $(X-S) \cup (S \cap Y)$ is a vertex cover of G, with size at most k.

To prove this claim, we just need to show that there are no edges with one endpoint in $S \cap X$ and another endpoint in $Y-S$, but this is true because each such edge would be a directed edge in $\delta^{out}(S)$ in G' with infinity capacity, contradicting that $c^{out}(S) = k$.

another endpoint in $Y-S$, but this is true because each such edge would be a directed edge in $\delta^{out}(S)$ in G' with infinity capacity. Contradicting that $c^{out}(S) = k$.

This concludes that $(X-S) \cup (S \cap Y)$ is a vertex cover in G of size k, proving König's theorem. □

Note that this proof provides a $O(|V| \cdot |E|)$-time algorithm to solve the vertex cover problem on bipartite graphs.

Good Characterization (Discussions)

König's theorem is a nice example of a graph-theoretic characterization.

To see this, imagine that you work for a company and your boss asks you to find a maximum matching say to assign some jobs to the employees.

If your algorithm finds a perfect matching, then your boss would be happy.

But suppose there is no perfect matching in the graph. how could you convince your boss about the non-existen

Your boss may just think that you are incompetent and other smarter people could find a better assignment.

A convincing way is to show your boss a vertex cover of size $< \frac{n}{2}$, and explain that if there is a perfect matching such a vertex cover could not exist.

These min-max theorems are some of the most beautiful results in combinatorial optimization, providing succinct "proofs" for both the YES-case (a large matching) and the No-case (a small vertex cover).

They show the non-existence of a solution by the existence of a simple obstruction.

Remark: To put the above discussion into perspective, consider the dynamic programming algorithms.

Even though we could solve the problems (such as optimal binary search tree) in polynomial time, we don't have such a nice succinct characterization for the No-cases.

If your boss asks why there is no better solution, it would be quite difficult to explain (e.g. we search for all possible solutions using this recurrence and our is an optimal solution).

The "proof" is still succinct in the sense that it is a polynomial sized table. but it is not nearly as elegant as the proofs provided by the min-max theorems.

Hall's Theorem characterizes when a bipartite graph $G = (X, Y; E)$ has a perfect matching or not.

Without loss of generality, we assume that $|X| = |Y|$, as otherwise there is no perfect matching.

Theorem (Hall) A bipartite graph $G = (X, Y; E)$ with $|X| = |Y|$ has a perfect matching if and only if for every subset $S \subseteq X$, it holds that $|N(S)| \geq |S|$ where $N(S)$ is the neighbor set of S in Y.

(In words, every subset $S \subseteq X$ has at least $|S|$ neighbors in Y.)

Exercise : Derive Hall's theorem from König's theorem.

Hall's theorem is an important result and has many applications in graph theory.

We show one corollary, which may be used in homework.

Corollary Every d-regular bipartite graph $G=(X,Y;E)$ has a perfect matching.

Proof There are $d|X|$ edges in the graph. Note that $|X|=|Y|$ as G is regular.

Any subset of vertices S with $|S| \le |X|-1$ can cover at most $d(|X|-1)$ edges.

So, any vertex cover needs at least $|X|$ vertices.

By König's theorem, there is a matching of size at least $|X|=|Y|$, hence a perfect matching. □

Duality : **(Discussion)** Why vertex cover is the "dual" problem of matching? How could we come up with it?

Actually, there is a systematic way to define the dual problem of an optimization problem, through the use of linear programming.

Most combinatorial optimization problems can be solved in the general framework of linear programming.

This is one of the most, if not the most, powerful algorithmic framework for polynomial time computation.

The many beautiful min-max theorems can also be systematically derived from linear programming duality.

Unfortunately, we won't learn about linear programming in this course; see [KT, DPV, CLRS] for more.

There are also various courses in the C&O department on these topics.

Baseball / Basketball League Winner (optional) [KT 7.12]

Suppose you are a fan of Toronto Blue Jays or Toronto Raptors or Waterloo Warriors.

They are playing in the regular season. They are doing okay but not at the top of the table.

You wonder whether it is still (mathematically) possible for them to finish on top in the end of the season

Input: The current standing / table, and the remaining schedule.

Output: Whether it is possible that your team can win the league.

The feature of the baseball / basketball league is that there is only win or lose (i.e. no draw as in football).

Suppose your team has currently won W games.

First, it is obvious that we assume that they will win all their remaining games, for a total of W^* games.

Then, we want to know whether there is a scenario that every other team will win less than W^* games.

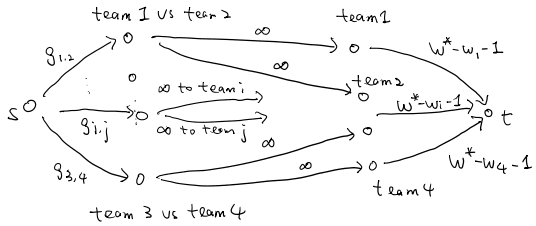
Let the other teams be $\{1,2,\ldots,n\}$ and currently team i has won w_i games.

We have the remaining schedule, showing that there are still g_{ij} games to be played between team i and team j.

How do we determine if there is still a scenario that your team can still win the league?

Reduction: This problem can be reduced to a maximum s-t flow problem.

The idea is to "assign" the winner of each remaining game so that no team will win more that $w^* - w_i - 1$ games.



The vertex set is $\{s\} \cup P \cup T \cup \{t\}$ where P has a vertex for each pair of teams and T has a vertex for each team.

There is an edge with capacity $g_{i,j}$ to each vertex $p_{i,j} \in P$, as there are still $g_{i,j}$ games between team i and j.

For $p_{i,j} \in P$, there is an edge from $p_{i,j}$ to $t_i \in T$ and from $p_{i,j}$ to $t_j \in T$, so that each game between team i and team j will be sent to either team i or team j, whoever is the winner of that game.

For each $t_i \in T$, there is an edge from t_i to the sink t with capacity $w^* - w_i - 1$, as we want team i to win at most $w^* - w_i - 1$ games.

Let $g = \sum_{i,j} g_{i,j}$. We want all these games to be played, with one winner for each game, so that team i wins at most $w^* - w_i - 1$ games for all i (and thus our favorite team with w^* wins is the champion).

With this intention in mind, it is not difficult to check the following claim.

Claim Our favorite team can still win the league iff there is an s-t flow of value g in the graph.

Remark: It is an NP-hard problem to determine if your favorite football team can still win the league (i.e. winning team gets 3 points, each team gets one point in a draw).

Project Selection (optional) [KT 7.11]

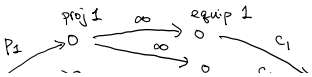
There are n projects available, completing each will give us profit p_i dollars.

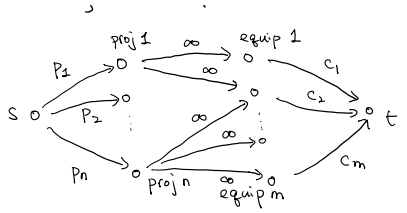
To do each project, however, we need to buy a subset of equipments, each will cost c_j dollars.

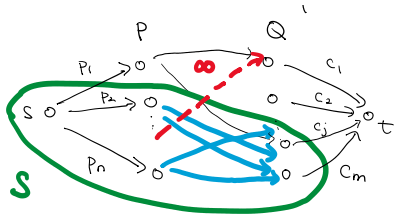
An equipment, once bought, can be used for multiple projects that require this equipment.

Now, the question is, what projects we should do and what equipments we should buy so that we can maximize the net profit, i.e. total profit from projects minus total costs from equipments.

Reduction: Interesting, this problem can be reduced to the minimum s-t cut problem.







The vertex set is $\{s\} \cup P \cup Q \cup \{t\}$. where P has a vertex for each project and Q has a vertex for each equipment.

There is an edge from s to project i with capacity p_i, and an edge from equipment j to t with capacity c_j.

Also, there is an edge with capacity infinity from project i to equipment j if it is required for project i.

Let $p = \sum_{i=1}^{n} p_i$. Clearly, this is the best possible net profit that we can hope to make, earning the profit of every project without buying any equipment.

Now, we want to minimize the difference to this ideal profit p, by minimizing the cost of the equipments that we need to buy and the profit of the projects that we need to miss.

Consider a minimum s-t cut S with $c^{out}(S) = k$.

We claim that it is possible to make a net profit of $p-k$.

Buy all the equipments in $S \cap Q$ and take all the projects in $S \cap P$.

The key point is that it is a feasible solution, as there are no edges from $S \cap P$ to $Q - S$, otherwise this is an outgoing edge of S with infinity capacity, contradicting $c^{out}(S) = k$.

So, all the equipments that the projects in $S \cap P$ required are in $S \cap Q$.

The net profit of this solution is $\sum_{i \in S \cap P} p_i - \sum_{j \in S \cap Q} c_j = p - \sum_{i \in P-S} p_i - \sum_{j \in S \cap Q} c_j = p - c^{out}(S) = p-k$.

In other words, the minimum s-t cut S tells us how to minimize the cost of the equipments that we need to buy (equipments in $S \cap Q$) plus the profit of the projects that we need to miss (projects in $P-S$).

On the other hand, a feasible solution with net profit $p-k$ gives us an s-t cut with capacity k, so we have the following conclusion.

Claim The maximum net profit is $p-k$ if and only if the capacity of a minimum s-t cut is k.

References: [KT 7.5, 7.6, 7.11, 7.12]

There are many more useful and interesting applications of max-flow min-cut.

You can learn more from the CSO network flow course.

You can also learn more about linear programing from CSO, which is the general underlying framework of the local search method as well as the duality theory behind the min-max theorems.

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 17 : Polynomial time reductions

We study polynomial time reductions, which would allow us to compare the difficulty of different problems.

Polynomial Time Reductions

Once we have learned more and more algorithms, they become our building blocks and we may not need to design algorithms for new problems from scratch every time.

So it becomes more and more important to be able to use existing algorithms to solve new problems.

We have already seen some reductions.

For instances, we have reduced subset-sum to knapsack, longest increasing subsequence to longest common subsequence, and baseball / basketball league winner to maximum flow, etc.

In general, if there is an efficient reduction from problem A to problem B and there is an efficient algorithm to solve problem B, then we have an efficient algorithm to solve problem A.

Decision Problems

To formalize the notion of a reduction, it is more convenient to restrict our attention to decision problems, for which the output is just YES or NO, so that every problem has the same output format.

For example, instead of finding a maximum matching, we consider the decision version of the problem "Does G have a matching of size at least k?".

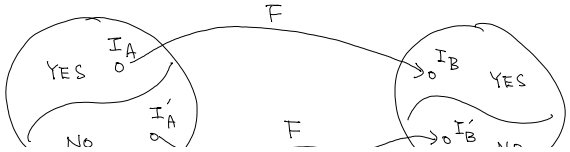
As we will discuss later, for all the problems that we will consider, if we know how to solve the decision version of our problem in polynomial time, then we can use the decision algorithm as a blackbox / subroutine to solve the search version of our problem in polynomial time.

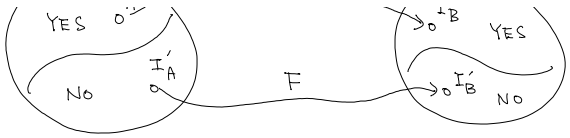
Definition (polynomial time reductions)

We say a decision problem A is polynomial time reducible to a decision problem B if there is a polynomial time algorithm F that maps / transforms any instance I_A of A into an instance I_B of B (that is, $F(I_A) = I_B$) such that

I_A is a YES instance of problem A if and only if I_B is a YES instance of problem B.

We use the notation $A \leq_p B$ to denote that such a reduction exists, intuitively saying that problem A is not more difficult than B in terms of polynomial time solvability. □





Now, suppose we have such a polynomial time reduction algorithm F and a polynomial time algorithm ALG_B to solve problem B, then we have the following polynomial time algorithm to solve problem A

Algorithm (solving problem A by reduction)

Input: an instance I_A of problem A

Output: whether I_A is a YES-instance

1. Use the reduction algorithm F to map/transform I_A into $I_B = F(I_A)$ of problem B.
2. Return $ALG_B(I_B)$.

Correctness follows from the property of the reduction algorithm F that I_A is a YES-instance of problem A iff I_B is a YES-instance of problem B, and the correctness of ALG_B to solve problem B.

Time complexity Suppose F has time complexity $p(n)$ for an instance I_A of size n where $p(n)$ is a polynomial in n, and ALG_B has time complexity $g(m)$ for an instance I_B of size m where $g(m)$ is a polynomial in m.

Then the above algorithm has time complexity $g(p(n))$, a polynomial in the input size n.

Proving Hardness using Reductions

So far it is all familiar: We reduce problem A to problem B efficiently, and use an efficient algorithm for problem B to obtain an efficient algorithm to solve problem A.

Now, we explore the other implication of the inequality $A \leq_p B$.

Suppose problem A is known to be impossible to be solved in polynomial time.

Then $A \leq_p B$ implies that B cannot be solved in polynomial time either, as otherwise we can solve problem A in polynomial time using the reduction algorithm proven above.

Therefore, if A is computationally hard and $A \leq_p B$, then B is also computationally hard.

By our current knowledge, however, we know almost nothing about proving a problem cannot be solved in polynomial time, and so we could not draw such a strong conclusion from $A \leq_p B$.

But suppose there is a problem, say the traveling salesman problem in L14, which is very famous and

has attracted many brilliant researchers to solve it in polynomial but without any success.

Now our boss gives us a problem C, and we couldn't solve it in polynomial time.

Instead of just saying that we couldn't solve it in polynomial time, it would be much more convincing if we could prove that $TSP \leq_p C$.

The following is a cartoon from the classical book about NP-completeness by Garey and Johnson.

"I can't find an efficient algorithm, I guess I'm just too dumb."

"I can't find an efficient algorithm, but neither can all these famous people."

This is what we will be doing in these few lectures!

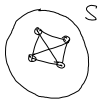
Simple Reductions

We will show that the following three problems are equivalent in terms of polynomial-time solvability. either they all can be solved in polynomial time or they all cannot be solved in polynomial time.

Maximum Clique (Clique) A subset of vertices $S \subseteq V$ is a clique if $uv \in E \quad \forall u,v \in S$.

Input: Graph $G = (V, E)$, an integer k.

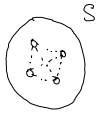
Output: Is there a clique in G with at least k vertices?



Maximum Independent Set (IS) A subset of vertices $S \subseteq V$ is an independent set if $uv \notin E \quad \forall u,v \in S$.

Input: Graph $G = (V, E)$, an integer k.

Output: Is there an independent set in G with at least k vertices?



Minimum Vertex Cover (VC) A subset of vertices $S \subseteq V$ is a vertex cover if $\{u,v\} \cap S \neq \emptyset \quad \forall uv \in E$.

Input: Graph $G = (V, E)$, an integer k.

Output: Is there a vertex cover in G with at most k vertices?

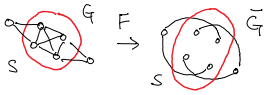
Proposition Clique $\leq_p$ IS and IS $\leq_p$ Clique.

Proof Clique and IS are very similar, one wants to find at least k vertices with all edges in between and one wants to find at least k vertices with no edges in between.

So, to reduce Clique to IS, we just need to change edges to non-edges and vice-versa.

More formally, to solve Clique on $G = (V, E)$, the reduction algorithm F would construct the

More formally, to solve Clique on $G=(V,E)$, the reduction algorithm F would construct the complement graph $\bar{G}=(V,\bar{E})$, where $uv \in E$ if and only if $uv \notin E$.



Clearly, the reduction algorithm runs in polynomial time, actually in linear time.

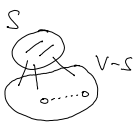
It should also be clear from the construction that S is a clique in G iff S is an independent set in $\bar{G}$.

Therefore, $\{G,k\}$ is an YES-instance for Clique iff $\{\bar{G},k\}$ is an YES-instance for IS. □

To see the connection between independent sets and vertex covers, we need the following observation.

Observation In $G=(V,E)$, $S \subseteq V$ is a vertex cover of G iff $V-S$ is an independent set in G.

Proof If S is a vertex cover, then $V-S$ is an independent set, because if there is an edge e between two vertices in $V-S$, then $S \cap e = \phi$ and S is not a vertex cover.



Similarly, if $V-S$ is an independent set, then all the edges in G have at least one endpoint in S, and thus S is a vertex cover. □

Proposition $VC \leq_p IS$ and $IS \leq_p VC$.

Proof The observation above states that G has a vertex cover of size at most k if and only if G has an independent set of size at least $n-k$.

So, the reduction algorithm simply maps $\{G,k\}$ for VC to $\{G,n-k\}$ for IS, and clearly it runs in polynomial time and it maps YES/NO instances for VC to YES/NO instances for IS respectively. □

Note that polynomial time reductions are transitive.

Lemma If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Proof If there exists a polynomial time algorithm F that maps instances of A to that of B, and a polytime algorithm H that maps B to C, then $H \circ F$ maps A to C in polynomial time.

Also, $H \circ F$ has the property that it maps YES/NO-instances of A to YES/NO-instances of C respectively. □

Therefore, Clique, IS and VC are equivalent in polynomial time solvability.

More Simple Reductions

Hamiltonian Cycle (HC) A cycle is a Hamiltonian cycle if it touches every vertex exactly once.

Input: Undirected graph $G=(V,E)$.

Output: Does G have a Hamiltonian cycle?

Hamiltonian Path (HP) A path is a Hamiltonian path if it touches every vertex exactly once.

Hamiltonian Path (HP) A path is a Hamiltonian path if it touches every vertex exactly once.

Input: Undirected graph $G = (V, E)$.

Output: Does G have a Hamiltonian path?

It is not surprising that these two problems are equivalent in terms of polynomial time computation, but it is a good exercise to work out the reductions.

Proposition $HP \leq_p HC$.

Proof Given $G = (V, E)$ for HP, we construct $G' = (V+s, E')$ by copying G and adding a new vertex s that connects to every vertex in V.

Clearly, the reduction runs in polynomial time.

We claim that G has a Hamiltonian path iff G' has a Hamiltonian cycle.

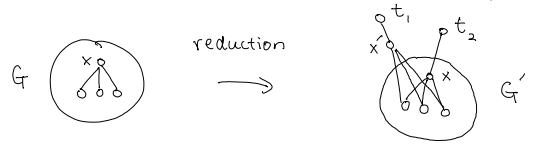
$\Rightarrow$) If G has a Hamiltonian path P with endpoints a and b, then $P+sa+sb$ is a Hamiltonian cycle in G'.

$\Leftarrow$) If G' has a Hamiltonian cycle C, then there are two edges incident on s, call them sa and sb, then $C-sa-sb$ is a Hamiltonian path in G. □

The other direction is slightly more interesting.

Proposition $HC \leq_p HP$.

Proof Given $G = (V, E)$ for HC, we pick an arbitrary vertex $x \in V$, and construct G' by adding a duplicate x' of x and two new degree one vertex t_1 and t_2 as shown in the picture

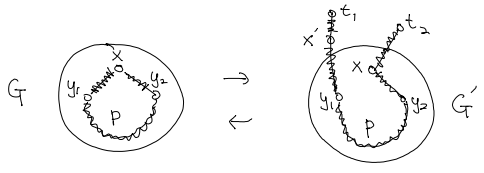


Clearly the reduction runs in polynomial time.

We need to prove that G has a Hamiltonian cycle iff G' has a Hamiltonian path.

$\Rightarrow$) If there is a Hamiltonian cycle in G, then there is a Hamiltonian path in G' by replacing xy_1 by $x'y_1$ and $x't_1$ and also adding xt_2.

$\Leftarrow$) If there is a Hamiltonian path in G', then the two endpoints must be t_1 and t_2 since they are degree one vertices.



Then, t_1x' must be in the path, and call the other neighbor of x' in the path be y_1.

Since x' is a duplicate of x, we know that x has an edge to y_1 in G.

We also know that t_2x must be in the Hamiltonian path, since t_2 is a degree one vertex.

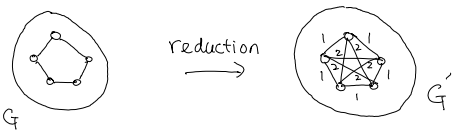
So, by replacing t_1x', $x'y_1$ and t_2x in G' by xy in G, we have a Hamiltonian cycle in G. □

A common technique to do reduction is to show that one problem is a special case of another problem. We call this technique specialization. The following is an example.

Proposition $HC \leq_p TSP$.

Proof Given $G=(V,E)$ for HC, we construct $G'=(V,E')$ for TSP such that if $uv \in E$, then we set its weight in G' to be 1, and if $uv \notin E$, then we set its weight in G' to be 2.

Then G has a Hamiltonian cycle if and only if G' has a TSP tour with cost n. □



An important problem and a nontrivial reduction

First, we introduce the 3-SAT problem, which will be important in the theory of NP-completeness.

In this problem, we are given n boolean variables $x_1, x_2, \ldots, x_n$, each can either be set to True or False.

We are also given a formula in conjunctive normal form (CNF), where it is an AND of the clauses, and each clause is an OR of some literals, where a literal is either x_i or $\overline{x_i}$.

For example, in $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee x_3)$, there are 4 clauses, each clause has at most 3 literals, and the formula has only 3 variables x_1, x_2, x_3.

3-Satisfiability (3-SAT)

Input: A CNF-formula in which each clause has at most three literals.

Output: Is there a truth assignment to the variables that satisfies all the clauses?

In the above example, setting $x_1=T$, $x_2=F$, $x_3=T$ will satisfy all the clauses, and hence the formula.

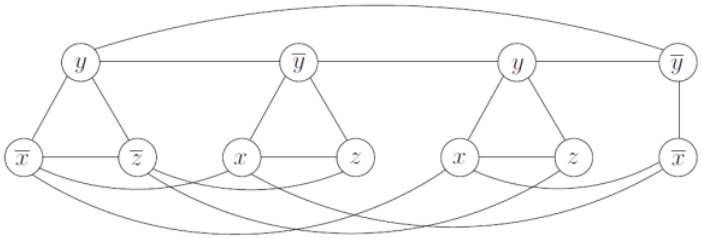
This problem looks quite different than other problems that we have seen.

Doing a reduction between two seemingly different problems usually requires some new ideas.

Theorem $3\text{-SAT} \leq_p IS$.

Proof: Given a 3SAT formula, we would like to construct a graph G so that the formula is satisfiable if and only if the graph G has an independent set of certain size.

Figure 8.8 The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$.



[DPV]

Reduction : For each literal in the formula, we create one vertex in the graph.

We would like a satisfying assignment of the formula corresponds to an independent set, and vice versa.

To satisfy the formula, we need to set at least one literal for each clause to be True.

It is easy to do so if we satisfy each clause separately.

The important point is to be consistent over the choices, i.e. if we set x to be True to satisfy some clause, then we can't set x to be False to satisfy some other clauses.

To enfore consistency, for each variable x_i, we add an edge between x_i and $\overline{x_i}$, for each appearance of x_i and $\overline{x_i}$, so that x_i and $\overline{x_i}$ won't both belong to an independent set.

We also add edges between literals of the same clause, to ensure that we only choose one literal in each clause to the independent set.

So, there are two types of edges.

One type are edges within each clause (i.e. one triangle for each clause of three literals).

Another type are edges between each appearance of x_i and $\overline{x_i}$ for $1 \le i \le n$.

Clearly, the construction of G can be done in polynomial time in the size of the formula.

The following claim will complete the proof of the theorem.

Claim Suppose the formula has k clauses. Then the formula is satisfiable if and only if there is an independent set of size k in the graph G.

Proof $\Rightarrow$) If there is a satisfying assignment, then we choose one literal that is set to True in each clause (e.g. if x_2=F in the satisfying assignment, then the literal $\overline{x_2}$ is True), and put the corresponding vertex to be in the independent set.

Since the assignment is satisfiable, there is at least one True literal in each clause, and so the set has at least k vertices.

These k vertices form an independent set because there are no edges of the first type between them as we choose only one literal vertex in each clause, and also there are no edges of

the second type as we won't choose both x_i and $\bar{x}_i$ as the satisfying assignment is consistent.

$\Leftarrow$) Suppose there is an independent set of size k in G.

Since there are edges (of the first type) between literals in the same clause, any independent set can only pick at most one vertex in each clause.

Since there are only k clauses, an independent set of size k must choose exactly one vertex from each clause.

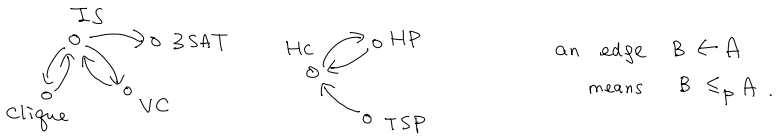
Also, because of the consistency edges (i.e. edges of the second type), each variable we choose at most one literal (note that we could choose none of the literals).

So, if we choose x_i in the independent set, then we set x_i to be True; otherwise, we set x_i to be False.

By the consistency edges, this assignment is well-defined, and since there is a vertex in each clause, this assignment satisfies every clause. □ □

Concluding Remarks

We have introduced the notion of a polynomial time reduction, and used it to establish relations between different problems, and so far we have



In principle, we can add new problems and relate to these problems, and slowly build a big web of all computational problems.

As $\leq_p$ is transitive, any strongly connected component in this web forms an equivalent class of problems in terms of polynomial time solvability.

Is there a better way to do it than to consider the problem one by one?

References: [KT 8.1, 8.2]

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 18 : NP-completeness

We study the class of NP problems and show that 3-SAT is NP-Complete.

The Class NP

As we discussed last time, we could do reductions between different problems and slowly build up a huge map showing the relations of all the problems.

But it would be exhausting to do so many reductions, or even just to look up the relations.

It would be nice if we could identify the "hardest" problem X of a large class.

Then, when we have a new problem Y, we just need to prove that $X \leq_p Y$ and then Y would also be at least as hard as all the problems in the large class.

This sounds very good, but how can we show that a problem is the hardest among a large class?

For this, we need a general and more abstract definition that captures a very large class of problems.

Short Proofs

A general feature of all the problems that we have seen is that although it may be difficult to determine whether an instance is a YES-instance or not, it is easy to verify that it is a YES-instance in the sense that there is a short proof for this fact.

For example, given a graph, we don't know how to determine if it has a Hamiltonian cycle efficiently, but we can easily verify that it is a YES-instance if someone tells us a Hamiltonian cycle and we just need to verify that all the edges in the cycle are really present in the graph to confirm.

As another example, given a 3-SAT formula, it is easy to confirm that it is a YES-instance if someone tells us a satisfying assignment.

In short, although it may be computationally difficult to find a solution for these problems, it is easy to verify that a solution is correct.

Informally, NP is the class of problems for which there is a "short proof" of its YES-instances that can be checked efficiently.

Definition (NP) For a problem X, each instance of X is represented by a binary string s.

A problem X is in NP if there is a polynomial time verification algorithm B_X such that the input s is a YES-instance if and only if there is a proof t which is a binary string of length $\text{poly}(|s|)$ so that $B_X(s,t)$ returns YES.

The key points are: B_X is a polynomial time algorithm and t is a short proof of length poly($|s|$).

In most problems, t is simply a solution and B_X is an efficient algorithm to check if t is indeed a solution.

This definition is a bit abstract, so let's see some concrete examples.

Example 1 (vertex cover): The problem X is whether a graph has a vertex cover of size at most k.

The input string s is a binary string representation of an input graph $G=(V,E)$

The proof string t is a binary string representation of a subset $S \subseteq V$ of at most k vertices.

The verification algorithm B_X will take s and t as input, go through all the edges in G and return YES if and only if S indeed covers all the edges in G.

Clearly, the proof is of length poly($|V|$) (short) and the verification algorithm runs in polytime (efficient).

Finally, the verification algorithm will only accept YES-instances, i.e. for a graph with no vertex cover of size at most k, there exists no proof that will make the verification algorithm to return YES.

So, the decision version of the vertex cover problem is in the class NP.

Example 2 (3-SAT): Given a 3-SAT formula, the verification algorithm expects the proof to be a satisfying assignment and will return YES if it indeed satisfies all the clauses. Clearly, the proof is short (polysize), the verification is efficient (polytime), and it only accepts YES instances.

Exercises Show that Clique, IS, HC, HP, Subset-Sum are all in the class NP.

It should be apparent that the class NP captures all the problems considered in this course, since they all have short solutions that are easy to verify.

Remark 1 (non-examples)

Suppose the problem is to determine whether a graph is non-Hamiltonian, i.e. no Hamiltonian cycles.

Then no one knows whether this problem is in NP, and the common belief is that it is not in NP.

In other words, we don't know how to efficiently check that a graph is a NO-instance of HC.

In fact, we don't know how to do much better than enumerating all potential Hamiltonian cycles and check.

Remark 2 (co-NP)

Contrast the above remark with the bipartite matching problem, for which we know how to efficiently check whether a graph is a NO-instance (i.e. no matching of size $>k$), by checking a vertex cover of size $\leq k$.

A problem with short and easy-to-check proofs for both YES and NO instances is in NP $\cap$ co-NP.

For example, a min-max theorem such as König's theorem would show that a problem is in NP $\cap$ co-NP.

But the common belief is that most problems in NP are not in NP ∩ co-NP (so, e.g. no mix-max theorem).

Remark 3 ($P \subseteq NP$)

Clearly, every polynomial time soluable decision problem is in NP: The verification algorithm is simply the algorithm to determine whether an input is a YES-instance, without the need of a proof string.

Let P denote the class of decision problem soluable in polynomial time. Then, $P \subseteq NP$.

Remark 4 (non-deterministic polynomial time)

NP is the class of problems solvable by a non-deterministic polytime algorithm, hence the name NP.

A non-deterministic machine, roughly speaking, has the power to correctly guess a solution or an accepting path. So, as long as there is a short solution, a non-deterministic machine will magically find it.

Remark 5 ($P = NP$?)

It is the most important open problem in theoretical computer science to answer whether $P = NP$ or $P \subset NP$.

It is now one of the seven open problems in mathematics posted by Clay Math Institute, with $1000000 award.

The common belief is that $P \neq NP$, and the intuition is that an efficient algorithm to verify a solution should not automatically implies an efficient algorithm to find a solution.

For examples, someone who can recognize good music doesn't imply that they are a good composer, and someone who can check a mathematical proof doesn't imply they could come up with the proof.

NP-completeness

Informally, we say a problem is NP-Complete if it is a hardest problem in NP.

Definition (NP-completeness) A problem $X \in NP$ is NP-Complete if $Y \leq_p X$ for all $Y \in NP$.

From the definition, we can formally show that an NP-Complete problem is a hardest problem in NP.

Proposition $P = NP$ if and only if an NP-Complete problem can be solved in polynomial time.

Cook and independently Levin formulated the class of NP problems and proved the following important theorem.

Theorem (Cook-Levin) 3-SAT is NP-Complete.

Since polynomial time reductions are transitive, if we can prove that $3\text{-SAT} \leq_p X \in NP$, then X is also NP-Complete.

For example, we have proved in L17 that $3\text{-SAT} \leq_p IS$, and so the independent set problem

is also a hardest problem in NP, a good reason that we couldn't solve it in polynomial time.

Then, it also follows from the results in L17 that VC and Clique are NP-Complete.

Proving NP-Completeness

To prove that a problem X is NP-Complete, first we show that $X \in NP$, then we find an an NP-Complete problem Y and prove that $Y \leq_p X$.

The reduction $Y \leq_p X$ is by a polynomial time algorithm, so interestingly we prove the hardness of a problem X by providing an efficient algorithm to use X to solve a hard problem Y.

But note that this is quite different from finding an efficient algorithm to solve X.

When we try to find an algorithm to solve X, we use the algorithms that we know (e.g. bipartite matching) and try to apply them to design an efficient algorithm for solving X.

When we try to prove X is NP-Complete, we assume that we know how to solve X, and we need to search for an NP-Complete problem Y and use X to design an efficient algorithm for Y.

This is a very different experience, and often we don't know what Y to start with.

We will do a few NP-Completeness proofs in the next lectures, many look quite magical, e.g. $3\text{-SAT} \leq_p IS$.

It will take a lot of practices to acquire the skill to prove NP-completeness result.

Cook-Levin Theorem

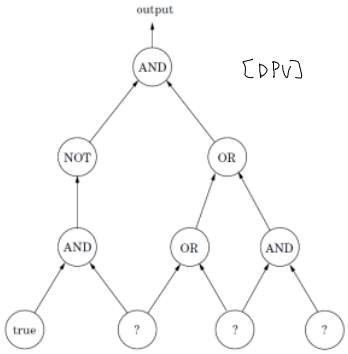
We would like to prove that 3-SAT is NP-Complete. The original proof directly does it.

Following the reference books, it would be easier to introduce some intermediate problems to do so.

Circuit-SAT

Input: a circuit with AND/OR/NOT gates, some known input gates, and some unknown input gates.

Output: is there a truth assignment to the unknown input gates so that the output is True?



[DPV]

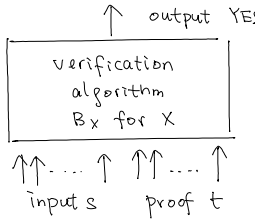
We can assume the input circuit is a directed acyclic graph, and each AND/OR gate has only two incoming edges.

Theorem Circuit-SAT is NP-Complete

Proof (sketch) To prove such a general statement, we need to start from the abstract definition of NP.

Our goal is to prove that for any $X \in NP$, $X \leq_p$ Circuit-SAT.

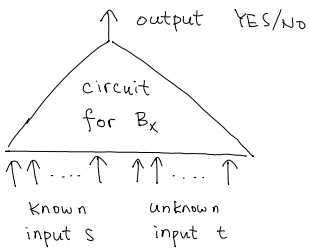
Since $X \in NP$, by definition, there is a polynomial time verification algorithm B_X such that if an instance s is a YES-instance, there exists a short proof t such that $B_X(s,t)$ returns YES; otherwise $B_X(s,t)$ returns NO for all t.



Let's think about what is an algorithm.

It can be written as a program and executed on a machine.

If the verification algorithm runs in poly(|s|) time, then there is a machine that executes it in poly(|s|) time, and so there is a circuit with only poly(|s|) size implementing the algorithm as it only requires poly(|s|) operations.



We are being sketchy about the reduction from an algorithm to a circuit. To do it precisely, we need a formal definition of a nondeterministic Turing machine and the details are quite tedious.

The main conceptual idea here is that a circuit is as general as an algorithm.

The reduction can be carried out in polynomial time (we can think of it as some kind of a compiler procedure that translates an algorithm into a circuit).

The original proofs of Cook and Levin directly transforms a non-deterministic Turing machine for the verification algorithm into a CNF formula (I may learn from CS 360 or CS 365).

Once we accept this reduction can be done in polynomial time, the rest is straightforward.

If s is a YES-instance, then there is a proof string t that will make the algorithm return YES, and hence make the circuit output True.

If s is a NO-instance, then no input t of the circuit will make it output True.

So, s is a YES-instance to problem X iff there is a satisfying assignment to the Circuit-SAT problem.

Therefore, $X \leq_p$ Circuit-SAT for any $X \in NP$. □

From circuit to formula

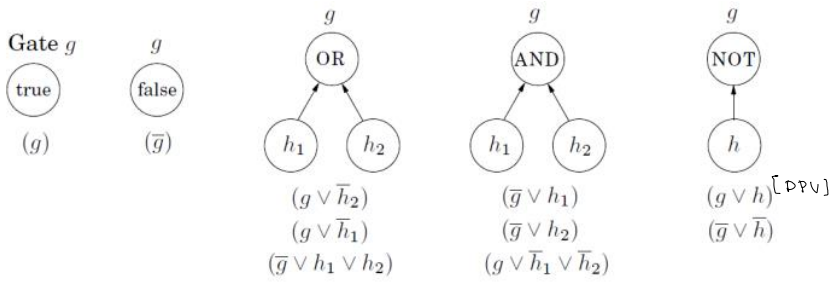
Now, we just need to show that a CNF formula has the same expressive power as a circuit.

Proposition Circuit-SAT $\leq_p$ 3-SAT.

Proof Given a circuit of n gates, we will construct a formula with $O(n)$ number of variables so that the circuit is satisfiable if and only if the 3-CNF formula is satisfiable.

For each gate in the circuit, we create one variable in the formula.

Then, we add clauses to the formula so as to simulate the function of the circuit as follows:



[DPV]

1. If there is a True gate g, we add a one literal clause (g) so that to satisfy the formula we must set g to be True, faithfully simulating the circuit.
2. Similarly, for a known False gate g, we add a clause $(\overline{g})$ to force g to be set to False.
3. For a NOT gate g, we want the value of g and its input h to be different, and we can enforce this by adding two clauses $\neg(g \wedge h) \wedge \neg(\overline{g} \wedge \overline{h}) = (\overline{g} \vee \overline{h}) \wedge (g \vee h)$
4. For an AND gate g, we want if the two inputs h_1 or h_2 is false, then g is false, so we add $(\overline{g} \vee h_1) \wedge (\overline{g} \vee h_2)$, and if h_1 and h_2 are true then g is true, so we add $(g \vee \overline{h}_1 \vee \overline{h}_2)$.
5. Similarly, for an OR gate g, we add $(g \vee \overline{h}_1) \wedge (g \vee \overline{h}_2)$ so that if h_1 or h_2 is true then g must be set to True to satisfy the formula, and we add $(\overline{g} \vee h_1 \vee h_2)$ so that if h_1 and h_2 are both False then g must also be set to False too.

Finally, to ensure that the output of the circuit is True, we add a variable o and add a clause (o) to ensure that o will be set to True.

It should be clear that this transformation from a circuit to a 3-CNF formula can be done in polynomial time, as we can just do "local replacement" for each gate by clauses.

Also, it should be clear that the circuit is satisfiable if and only if the formula is satisfiable, as there is a one-to-one correspondence between the variables and the gates, and the clauses enforce faithful simulation of the gates of the circuit. □

With the Cook-Levin theorem, we have a firm foundation to prove that a problem is NP-hard. We will grow our list of NP-complete problems in the next lecture.

References: [KT 8.3, 8.4], [DPV 8.3]

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 19: Hard graph problems

We show that two classical graph problems, Hamiltonian cycle and graph coloring, are NP-Complete.

Both reductions are from 3-SAT and require clever gadget design.

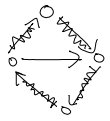
Hamiltonian Cycle [KT 8.5]

We will introduce an intermediate problem for the reduction.

Directed Hamiltonian Cycle (DHC)

Input: A directed graph $G=(V,E)$

Output: Does G have a directed cycle that touches every vertex exactly once?



We will show that 3-SAT $\leq_p$ DHC $\leq_p$ HC.

3-SAT and DHC are very different, so it requires some creativity to connect the two problems.

Theorem Directed Hamiltonian Cycle is NP-Complete.

Proof It is easy to see that DHC is in NP. To prove it is NP-complete, we will prove 3-SAT $\leq_p$ DHC.

Given a 3-SAT instances with n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$, we would like to construct a directed graph G so that the formula is satisfiable iff G has a Hamiltonian cycle.

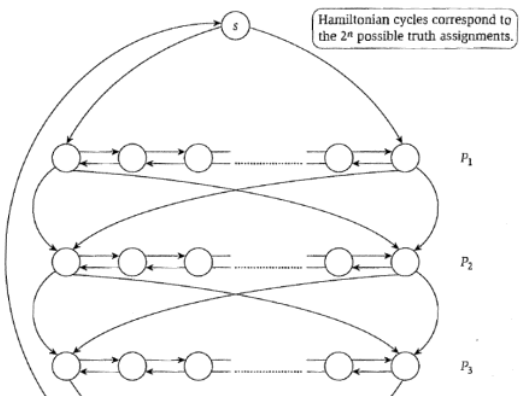
We need to create some graph structures for the variables and the truth assignment.

The idea is to associate a long "two-way" path to a variable and going the path in one way corresponds to setting the variable True, while going the other way corresponds to setting it to False.

The intention is that we go from left-to-right iff x_i is set to be True (i.e. right-to-left iff False).

In the following graph, there is a one-to-one correspondence between the 2^n truth assignment of the n variables and the directed Hamiltonian cycles in the graph.

[KT]

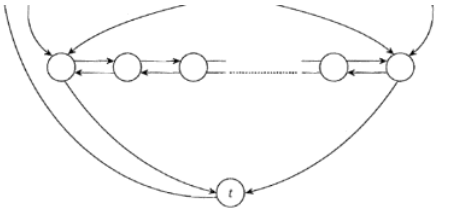


There are totally n two-way paths, one for each variable.

Each path is of length $3m$.

The two endpoints of the path P_i connect to the two endpoints of P_{i+1}.

There is a source s that connects to the two ends



There is a source s that connects to the two ends of P_1 and the two ends of P_n connects to the sink t, and t connects to s.

It should be clear that there are only 2^n possible Hamiltonian cycles in this directed graph, since we must use each path P_i either from left-to-right or from right-to-left. as the intermediate of the paths are not connected to anything else.

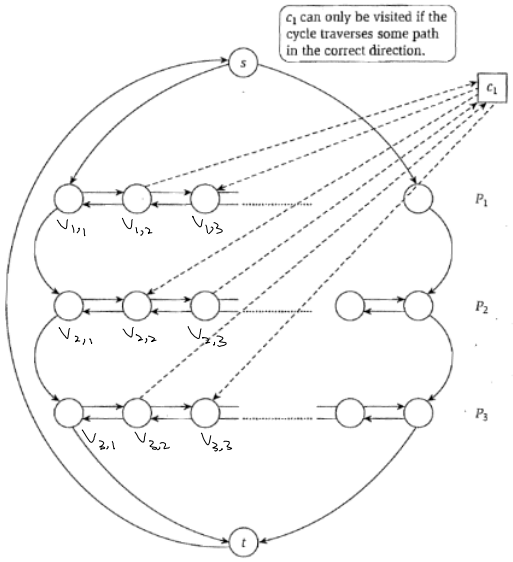
That's a good start, with a one-to-one correspondence between truth assignments and Hamiltonian cycles.

Next, we would like to add some clause structures to "kill" all the Hamiltonian cycles that do not correspond to satisfying assignments.

Recall that the two-way paths are of length 3m. The first three edges belong to the first clause, and in general the three edges between v_{3i+1}, v_{3i+2}, v_{3i+3}, $v_{3(i+1)+1}$ belong to the i-th clause.

Suppose there is a clause, say $x_1 \vee \bar{x}_2 \vee x_3$, then we want the Hamiltonian cycles to either (1) go from left-to-right in P_1, or (2) go from right-to-left in P_2, or (3) go from left-to-right in P_3 to satisfy the clause.

[KT]



To this end, we create one vertex c_j for each clause C_j.

Call the vertices on P_i be $v_{i,1}, v_{i,2}, \ldots, v_{i,3m+1}$.

If literal x_i appears in C_j, then we add the directed edges $v_{i,3j-1}\, c_j$ and $c_j\, v_{i,3j}$

Otherwise, if literal $\bar{x}_i$ appears in C_j, then we add directed edges $c_j\, v_{i,3j-1}$ and $v_{i,3j}\, c_j$.

We do this for every clause C_j.

Note that the edges for C_j and C_k don't share vertices, and that's why we created long paths for.

That's the whole construction. Clearly, it can be done in polynomial time.

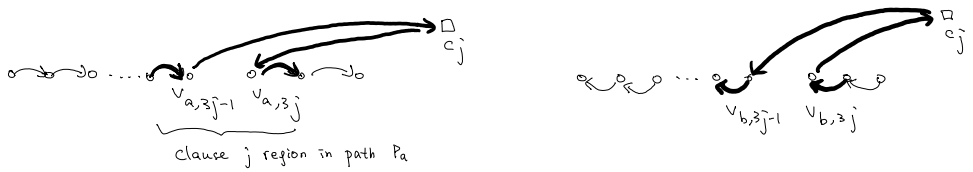
It remains to prove that the formula is satisfiable iff there is a Hamiltonian cycle in this graph.

⇒) Suppose there is a satisfying assignment.

If $x_i = T$, we visit P_i from left-to-right; otherwise if $x_i = F$ we visit P_i from right-to-left.

For a clause say $C_j = (x_a \vee \bar{x}_b \vee x_c)$, at least one literal is true in the satisfying assignment, say x_a.

Then, when we visit P_a from left-to-right, we "detour" to visit vertex c_j during the clause j region in the path P_a.



Similarly, if $x_b = F$, then we can also detour to visit c_j when we visit P_b from right-to-left.

Since every clause is satisfied in a satisfying assignment, we can visit all clause vertices following these directions and detours, and hence form a Hamiltonian cycle in the graph.

$\Leftarrow$) Suppose there is a Hamiltonian cycle. We would like to argue that it must look like those Hamiltonian cycles above that correspond to a satisfying assignment, but why must it be the case?

The crucial observation is that if we use the directed edge $v_{a,3j-1}\, c_j$, then we must use the edge $c_j v_{a,3j}$ immediately after it, as otherwise the vertex $v_{a,3j}$ will become a "dead-end" and there is no way to complete the cycle.

Since each clause vertex is visited in a Hamiltonian cycle, at least one variable path is going in the correct direction, and all the paths must go from left-to-right or right-to-left (as it must come back immediately after the "detour" to clause vertices), and so this corresponds to a satisfying assignment as we intended. □

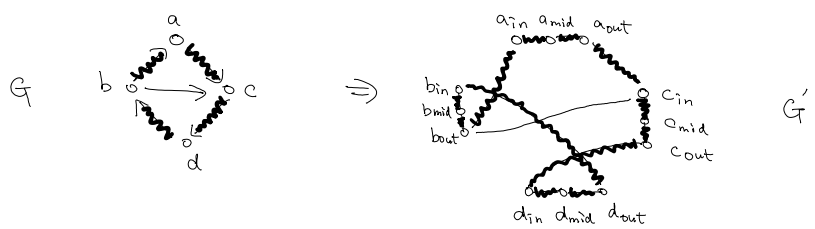
To prove DHC $\leq_p$ HC, the idea is to use a length 3 path to simulate a directed edge.

Proposition DHC $\leq_p$ HC.

Proof Given a directed graph $G = (V, E)$ for DHC, we construct an undirected graph G' in which we create three vertices v_{in}, v_{mid}, v_{out} for each vertex $v \in V(G)$.

Then, we create the edges in G' as follows: for every $v \in V(G)$, we add the edges (v_{in}, v_{mid}) and (v_{mid}, v_{out}).

Also, for each directed edge $uv \in E(G)$, we add an undirected edge (u_{out}, v_{in}) in G'



That's the whole construction, which can clearly be done in polynomial time.

Now we prove that G has a directed Hamiltonian cycle iff G' has an undirected Hamiltonian cycle.

$\Rightarrow$) One direction is immediate: if G has a Hamiltonian cycle, by following the cycle and replacing each directed edge uv by $u_{out}v_{in}$ and using the paths v_{in}-v_{mid}-v_{out} for all $v \in V(G)$, it is a Hamiltonian cycle in G'.

$\Leftarrow$) The other direction is slightly more interesting: in an undirected Hamiltonian cycle, start with a vertex v_{in}, then v_{mid} must be a neighbor of v_{in} in the Hamiltonian cycle, as otherwise v_{mid} will be a "dead-end" since it is of degree two, and then v_{out} must be a neighbor of v_{mid} in the Hamiltonian cycle. Then, from v_{out}, by construction, the cycle must go to w_{in} for some w, and then w_{mid} and w_{out} as argued above. So, following the undirected Hamiltonian cycle of G', it must be of the form as described in the previous direction, and hence it corresponds to a directed Hamiltonian cycle in G. □

Corollary TSP is NP-complete.

Graph Coloring [KT 8.7]

Graph coloring is one of the most classical problem in graph theory, e.g. 4-color theorem.

Input: An undirected graph $G=(V,E)$, and a positive integer k.

Output: Is it possible to use k colors to color all the vertices so that every vertex receives one color and any two adjacent vertices receive different colors?

When $k=1$, it is possible iff the graph has no edges.

When $k=2$, it is possible iff the graph is bipartite (why?).

When $k=3$, the problem becomes NP-complete.

The graph coloring problem is very useful in modeling resource allocation problems, e.g. interval coloring problem.

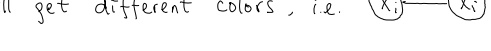
Theorem 3-coloring is NP-Complete.

Proof It is clear that 3-coloring is in NP (easy exercise). We prove 3-SAT $\leq_p$ 3-coloring.

Given a 3-SAT instances with n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$, we would like to construct a graph G so that the formula is satisfiable iff G is 3-colorable.

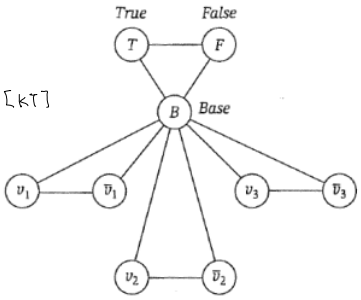
Quite naturally, we would like to associate one color to True and one color to False.

And we would like to ensure that the literals are colored consistently.

This is not difficult to do: we just create two vertices x_i and $\bar{x}_i$ for each variable and add an edge between them so that they will get different colors, i.e. 

Since there are three colors, to enforce that the two literals x_i and $\bar{x}_i$ get T/F colors,

we connect every literal vertex to a common vertex, called the base vertex.



Call the three colors T, F, B.

We create two vertices v_i and $\bar{v}_i$ for each variable x_i.

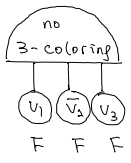
These edges enforce that either (v_i=T and $\bar{v}_i$ = F) or (v_i=F and $\bar{v}_i$ = T).

So, there is a one-to-one correspondence between truth assignments and the 3-colorings.

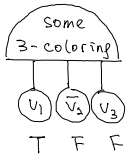
As in the NP-completeness proof of DHC, we would like to add some clause structures to "kill" the 3-colorings that don't correspond to satisfying assignments.

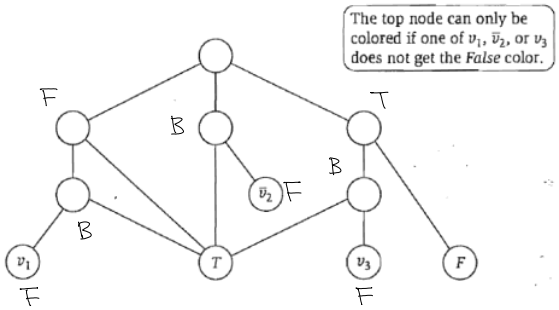
Say we have a clause $(x_1 \vee \bar{x}_2 \vee x_3)$, it would be great if there is a "gadget" to connect to the three variables $v_1, \bar{v}_2, v_3$ so that there is a 3-coloring for the gadget if and only if at least one of $v_1, \bar{v}_2, v_3$ gets the color T, e.g.

With some trial-and-error, it is possible to construct a gadget with exactly this "functionality".



and





← We need to verify this claim.

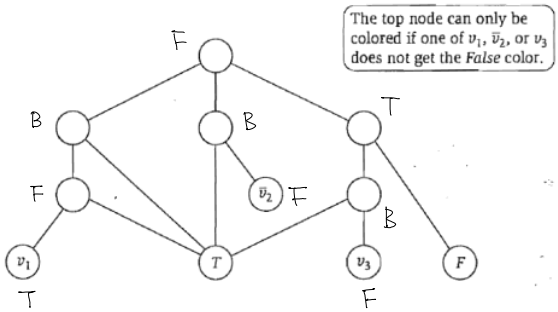
We create one such gadget for each clause.

All unlabeled vertices are new vertices created only for this clause.

First, suppose all $v_1, \bar{v}_2, v_3$ are colored F, then the top node in the gadget cannot be colored.

See the picture above; all the colors are forced, and there is no way to color the top node.

On the other hand, if at least one of $v_1, \bar{v}_2, v_3$ is colored T, then it is possible to color all nodes in the gadget.



If v_1 is colored T, then we can color all the nodes in the gadget using 3 colors.

We leave checking all other cases as coloring exercises.

It is clear that the reduction can be done in polynomial time.

With the claim about the gadget checked, it is not difficult to finish the proof by showing that

there is a satisfying assignment in the formula iff there is a 3-coloring in the graph.

⇒) If there is a satisfying assignment, for each variable x_i, if x_i = True, then we color $v_i = T$ and $\bar{v}_i = F$; otherwise if x_i = False, then we color $v_i = F$ and $\bar{v}_i = T$.

Clearly, it is a valid coloring for the initial base graph (without gadgets yet).

Since it is a satisfying assignment, each clause gadget has at least one of the three "inputs" set to be T, and thus it can be extended to a 3-coloring in the gadget by the gadget claim.

⇐) If there is a 3-coloring of the graph, then by the claim for each gadget, there is at least one "input" of the gadget with color T.

By the initial base graph, the coloring on $v_1, \bar{v}_1, v_2, \bar{v}_2, \ldots, v_n, \bar{v}_n$ must be consistent with a truth assignment.

So, following the coloring, we can define a truth assignment that satisfies all the clauses. □

CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 20: Hard partitioning problems

We will see that the 3-dimensional matching problem and the subset-sum problem are NP-complete. We then end with some discussions.

3-Dimensional Matching [KT 8.6]

This is a generalization of the bipartite matching problem.

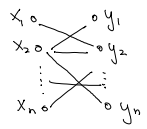
3-dimensional matching (3DM)

Input: Disjoint sets X, Y, Z each of size n, a set $T \subseteq X \times Y \times Z$ of triples.

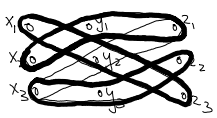
Output: Does there exist a subset of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of the triples?

The bipartite matching problem is a special case when we only have X, Y and pairs.

bipartite matching



3DM



Input: (x_1, y_2, z_3), (x_2, y_1, z_1), (x_3, y_2, z_1), (x_3, y_3, z_2)

Output: (x_1, y_2, z_3), (x_2, y_1, z_2), (x_3, y_3, z_2) is a perfect matching

A set of n triples is a perfect 3D-matching if every element is contained in exactly one of these triples.

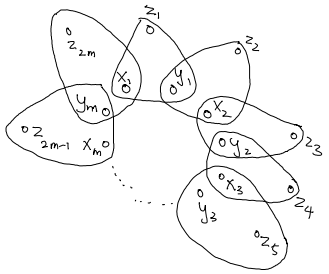
Theorem 3DM is NP-Complete.

Proof Clearly 3DM is in NP. We show that 3DM is NP-Complete by proving 3-SAT $\leq_p$ 3DM.

Given a 3-SAT instances with n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$, we would like to construct an 3DM instance so that the formula is satisfiable iff there is a perfect 3D-matching.

As in the Hamiltonian cycle problem, we would like to create some variable gadgets to capture the binary decisions of the variables.

For each variable v_i, we create the following gadget.

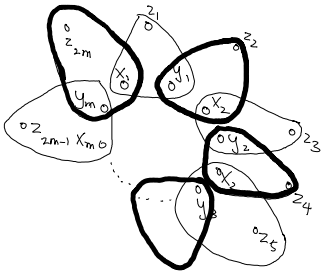


There are m vertices $x_1, \ldots, x_m$. m vertices $y_1, \ldots, y_m$, and $2m$ vertices $z_1, \ldots, z_{2m}$ created for the variable v_i.

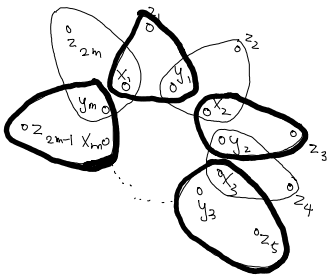
There is a triple x_i, y_i, z_{2i-1} for $1 \leq i \leq m$, and a triple x_{i+1}, y_i, z_{2i} for $1 \leq i \leq m-1$ and a triple x_1, y_m, z_{2m}.

In our construction, we will ensure that the vertices x_i, y_j are not contained in any other triples, besides the two triples in the variable gadget.

This implies that there are only two possibilities to choose the triples in the variable gadget:



Corresponds to Setting the variable to be True

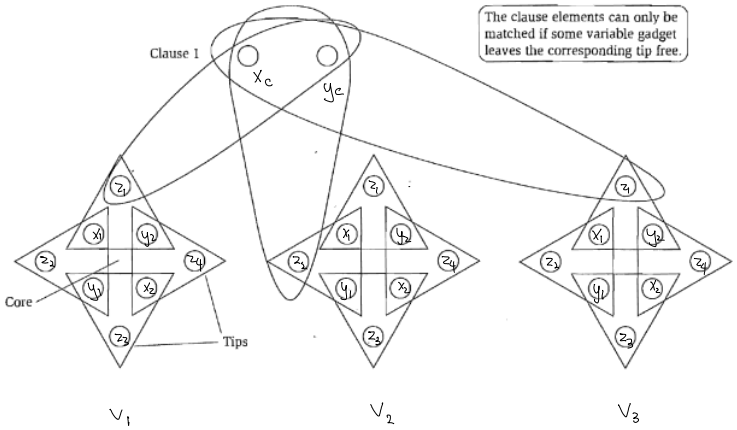


Corresponds to Setting the variable to be False

To cover y_1, either we choose x_1, y_1, z_1 or x_2, y_1, z_1. Once this is decided, the remaining decisions are forced.

This captures the binary decision for each variable, as we have one gadget for each variable.

It remains to add some clause structures to the 3DM-instance so that only satisfying assignments "survive".



Say we have a clause $C_j = (v_1 \vee \bar{v}_2 \vee v_3)$.

We create two new vertices x_c, y_c for C_j.

If a variable appears as v (but not $\bar{v}$),

we add a triple x_c, y_c, z_{2j-1}

Otherwise, if a variable appears as $\bar{v}$,

we add a triple x_c, y_c, z_{2j}.

We do the same for each clause. Note that the triples for different clauses are disjoint, because they use different "tips" in the variable gadgets.

Each clause can cover one tip. There will be $2nm - m = (2n-1)m$ uncover tips left over.

We will create $(2n-1)m$ pairs of "dummy" vertices x', y', and for each dummy pair, we add a triple (x', y', z) for every tip z in every variable gadget.

Totally, we will add $2(2n-1)m$ new dummy vertices and $(2n-1)m \cdot 2mn < 4m^2n^2$ dummy triples.

This is the construction, which is a little wasteful but certainly can be done in polynomial time.

It remains to prove that the formula is satisfiable iff there is a perfect 3D-matching.

$\Rightarrow$) Suppose there is a satisfying assignment.

If $v_i = T$, we cover the variable gadget using the even tips, leaving the odd tips free;

otherwise if $v_i = F$, we cover the variable gadget using the odd tips, leaving the even tips free.

Since this assignment is satisfying all the clauses, for each clause c, there is a literal v_i or $\bar{v}_i$ satisfied.

So, there will be a tip available in the variable gadget for v_i, so that we can choose that triple to cover x_c, y_c.

Finally, for the remaining $(2n-1)m$ uncovered tips, we use the dummy triples to cover them all.

This gives us a perfect 3D-matching.

$\Leftarrow$) Suppose there is a perfect 3D-matching.

For each variable gadget, there are only two ways to cover all the internal vertices x_i, y_j.

If these triples don't cover the odd tips, we set the variable to be True; otherwise False.

Since we can cover the clause vertices x_c, y_c, one of the three tips it connects to is free (i.e. not being used by the triples from the variable gadget).

By our construction, this means that the clause is satisfied by that variable.

Since all clause variables are covered, we have a satisfying assignment. □

Subset-Sum [KT 8.8]

Input: n positive integers $a_1, a_2, \ldots, a_n$, and an integer K.

Output: Does there exist a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$?

This is a problem about numbers, so some new ideas are needed to connect to previous combinatorial problems.

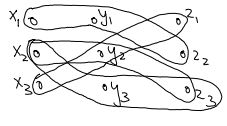
Theorem Subset-sum is NP-complete.

Proof It is clear that Subset-sum is in NP. We prove that it is NP-complete by proving 3DM $\leq_p$ Subset-sum.

Given an instance of 3DM, we will construct an instance of subset-sum so that there is a perfect 3D-matching iff there is a subset of certain sum K (whose value is to be determined later).

The idea is quite natural. We will first map a triple to a bit-vector, and then we will map a bit-vector to a number, and so eventually a triple will be mapped to a number.

We first show the reduction using an example and then describe it more formally.



3DM instance with 4 triples

(x_1, y_1, z_2), (x_2, y_2, z_3), (x_2, y_3, z_3), (x_3, y_2, z_1)

$\longrightarrow$

	V_1	V_2	V_3	V_4
x_1	1	0	0	0
x_2	0	1	1	0
x_3	0	0	0	1
y_1	0	0	0	0
y_2	1	0	0	1
y_3	0	1	0	0
z_1	0	0	1	0
z_2	0	0	0	1
z_3	0	1	0	0

Let $|X| = |Y| = |Z| = n$ and let m be the number of triples in the 3DM instance.

We will create a bit-vector for each triple. Each vector has $3n$ coordinates, one for each vertex.

For each triple (x_i, y_j, z_k), we have a bit-vector with 1 in the i-th, $(n+j)$-th, and $(2n+k)$-th coordinates.

From our construction, it is easy to verify that there is a perfect matching in the 3DM instance

if and only if there is a subset of vectors that sums to the all-one vector.

In the above example, the vectors v_1, v_3, v_4 sum up to the all-one vector, and the corresponding triples form a 3DM.

We leave the verification of this claim to the reader.

So, now, we have proved that the problem of choosing a subset of 0-1 vectors that sums to $\vec{1}$ is NP-complete.

We would like to reduce this problem to the subset-sum problem to show that it is NP-complete.

A very natural idea is to think of the 0-1 vector as the binary representation of a number.

That is, each triple (x_i, y_j, z_k) is mapped to the number $2^i + 2^{n+j} + 2^{2n+k}$.

With this mapping, if there is a subset of triples that form a perfect 3D-matching, their corresponding numbers would add up to $\sum_{i=1}^{3n} 2^i$, the number that corresponds to the all-one binary string.

However, if there is a subset of numbers that add up to $\sum_{i=1}^{3n} 2^i$, it may not correspond to a perfect 3D-matching.

The problem is that when we add up two numbers both with 1 in the j-th coordinate, the resulting number has a 1 in the $(j+1)$-th coordinate because of "carrying".

This is not our intention, which is to choose a vector with a one in the $(j+1)$-th coordinate.

There is a simple trick to get around this "carrying" problem, so that the above plan would work.

There are at most m numbers.

So, if we choose a large enough base $b = m+1$, there could be no carrying to the next "digit / position".

Final construction: Each triple (x_i, y_j, z_k) is mapped to the number $b^i + b^{n+j} + b^{2n+k}$.

Define $k = \sum_{i=1}^{3n} b^i$, the all-one number in base b (except for the lowest position)

It is clear that this can be done in polynomial time.

Claim There is a perfect 3D-matching iff there is a subset with sum $k = \sum_{i=1}^{3n} b^i$.

$\Rightarrow$) This direction is easy and has been explained in the bit-string setting.

$\Leftarrow$) Since there is no carrying, for each position l in the base-b representation, the subset sum records how many times we have covered the l-th vertex in the 3DM instance.

As the target number k has one in each position, the subset must correspond to a perfect matching.

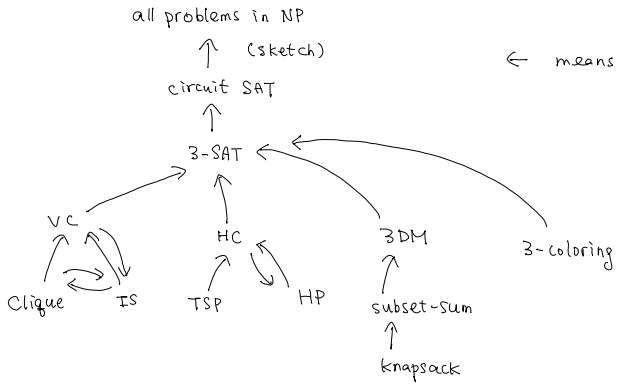
This claim finishes the proof of the theorem. □

Corollary Knapsack is NP-Complete.

Concluding Remarks

Concluding Remarks

First, we summarize what we have done so far.



Unlike what we have done in earlier topics, where we showed you some basic examples and asked you to similar or more advanced examples in homework, in NP-completeness, we have showed you the difficult reductions and will ask you to do simple reductions in homework.

Techniques of doing reductions

As we mentioned before, doing reduction requires a different way of thinking.

We need to find a hard problem Y and show that $Y \leq_p X$ for our problem X.

It requires practices to search for the right problem Y.

Once you know more NP-complete problems, it will be easier to find a similar problem to do the reduction, e.g. covering type uses VC, partitioning type uses 3DM, etc.

There are three common techniques in proving NP-completeness.

Specialization

Observe that a hard problem is a special case of your problem.

This is the easiest but also most useful technique, as most practical problems are often more complicated with different parameters, and often you may realize that restricting to a simple setting already captures an NP-complete problems.

Some examples that we have seen include TSP and knapsack.

You will see more in HW5 and supplementary exercises.

Local replacement

This is not as simple as specialization, but not as difficult as gadget design.

We replace each simple structure in problem Y by a simple structure in problem X.

Some examples that we have seen include Circuit-SAT $\leq_p$ 3-SAT where we replaced each logic gate by some simple clauses with the same functionality, DHC $\leq_p$ HC where we replaced each directed edge by an undirected path of length three and connect the paths accordingly, 3DM $\leq_p$ Subset-sum which is a more non-trivial example of this kind where we replaced each triple by a number.

You will see more in HW5 and supplementary exercises.

Gadget design

This requires creativity and good understanding to design gadgets and also the plan for the reduction.

We have seen 3SAT $\leq_p$ VC, 3SAT $\leq_p$ DHC, 3SAT $\leq_p$ 3DM, 3SAT $\leq_p$ 3-coloring are all of this kind.

We won't ask this type of questions in HW and in exam.

2 vs 3

It is an interesting phenomenon to observe that 2 is usually easy but 3 is usually hard.

For example, 2SAT is easy (see supplementary exercise list 2) but 3SAT is hard,

2-coloring is easy but 3-coloring is hard, 2D-matching is easy but 3D-matching is hard.

Usually, 2 is easy because once we make a decision then everything else is forced,

while 3 is difficult because after we made a decision we are still left with binary decisions.

Decision problems vs search problems

You may wonder whether restricting to decision problems will limit the scope of our problems, because we are usually interested in finding an optimal solution.

Many search problems can be easily reduced to decision problems.

For example, to find a Hamiltonian cycle, we can delete an edge and ask again whether the graph still has a Hamiltonian cycle, to remove all the edges until we are left with a Hamiltonian cycle.

You are encouraged to try this for other problems, e.g. VC, subset-sum, 3SAT, etc.

Actually, one can prove formally that any NP-Complete search problem can be reduced to polynomially many NP-complete decision problems, by reconstructing a solution bit by bit.

So, in terms of polynomial time solvability, this is without loss of generality.

Finally, to determine the optimal value (say for min-VC, max-IS), we can simply do binary search to figure out the optimal value by solving a logarithmic number of decision problems.