

IADS - Revision Semester 2

Antonio León Villares

May 2021

Contents

1	Week 1 - Introduction to Dynamic Programming	3
1.1	Motivation for Dynamic Programming	3
1.2	Principles of Dynamic Programming	4
1.2.1	Principles Of Dynamic Programming	4
1.2.2	Applying DP: The Coin Changing Problem	5
2	Week 1 - Seam-Carving	9
3	Week 1 - Edit Distance	14
4	Week 2 - All Pairs Shortest Paths	24
5	Week 2 - Probabilistic Finite State Machines and the Viterbi Algorithm	30
6	Week 3 - Context Free Languages and Grammars	30
6.1	Context Free Grammars and their Language	30
6.2	Syntax Trees	32
7	Week 3 - The CYK Algorithm	33
7.1	Chomsky Normal Form	33
7.2	The CYK Algorithm	34
7.3	Converting to Chomsky Normal Form	40
8	Week 4 - The LL(1) Algorithm and Predictive Parsing	43
8.1	Predictive Parsing	43
8.2	LL(1)	44
8.2.1	Parse Table	45
8.2.2	LL(1) Parsing	46
8.2.3	The LL(1) Algorithm	48
8.2.4	Further Remarks on LL(1)	48

9	Week 5 - P and NP	49
9.1	Decision Problems	49
9.2	Complexity Classes: P and NP	49
9.3	Reductions and NP-Completeness	50
9.3.1	SAT Problem	52
9.3.2	Independent Sets	52
9.4	Additional Resources	55
10	Week 6 - Approximation Algorithms for NP-Complete	55
10.1	Approximation Algorithms	55
10.2	Minimising Vertex Cover	56
10.3	Max 3-SAT	58
10.3.1	Randomised Max 3-SAT	58
10.3.2	De-Randomised Max 3-SAT	60
11	Week 7 - Recursive Backtracking for NP-Completeness	64
11.1	Dealing with NP-Completeness: Recursive Backtracking	64
11.2	Dealing with NP-Completeness: DPLL	71
12	Week 8 - Introduction to Computability Theory	73
12.1	Register Machines for Computability Theory	73
12.2	The Church-Turing Thesis: CT Computable Functions	77
12.3	Universal Machines	77
13	Week 9 - Unsolvable Problems	78
13.1	The Halting Problem	78
13.2	The World of Unsolvability	82
13.3	Food for Thought	82

1 Week 1 - Introduction to Dynamic Programming

1.1 Motivation for Dynamic Programming

- **Optimisation:** finding the best possible solution to a problem
 - we want to do so efficiently (polynomial time)
- **Efficiency in Divide and Conquer:** runtime heavily depends on the number of problems, how they are combined, etc ...
 - efficient for MergeSort, QuickSort
 - can be inefficient if we have to do the same calculation many times
- **Recursive Fibonacci:** calculating Fibonacci recursively has exponential runtime, as we are computing the same Fibonacci term an exponential number of times

Algorithm Dyn-Fib(n)

```

1.  $F[0] = 0$ 
2.  $F[1] = 1$ 
3. for  $i \leftarrow 2$  to  $n$  do
4.      $F[i] \leftarrow F[i-1] + F[i-2]$ 
5. return  $F[n]$ 

```

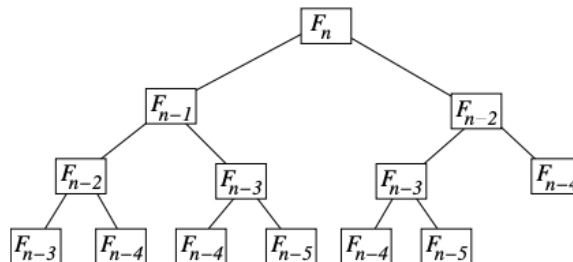


Figure 1: We are calculating many terms many times (for example, F_{n-4} needs to be recalculated 4 times). Such an algorithm has runtime $T(n) = T(n-1) + T(n-2) + \Theta(1) > 1.6^n$

- **Dynamic Programming Fibonacci:** instead of recalculating values, just store previously calculated values in an array, and use these. This allows $\Theta(n)$ time:

Algorithm Rec-Fib(n)

```
1. if  $n = 0$  then
2.     return 0
3. else if  $n = 1$  then
4.     return 1
5. else
6.     return REC-FIB( $n - 1$ ) + REC-FIB( $n - 2$ )
```

- this process is called **memoisation**
- it “turns recursion upside down”, building the recursion “from the bottom up”
- in Python, we can use memoisation as a decorator, and apply it with our standard recursive implementation

```
# The plain recursive implementation:
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

def memoize(f):
    memo = {}
    def check(s):
        if s not in memo:
            memo[s]=f(s)
        return memo[s]
    return check
```

Figure 2: The memoised function can then be run by `fib = memoize(fib)`

1.2 Principles of Dynamic Programming

1.2.1 Principles Of Dynamic Programming

1. optimal solutions can be achieved by solving smaller subproblems
2. solution of problem can be expressed as a recurrence
3. solutions of subproblems can be stored in polynomial space
4. smaller subproblems are solved before larger ones

1.2.2 Applying DP: The Coin Changing Problem

- **Problem Description:** given an arbitrary set of coin denominations, find a minimal collection of coins that add up to a value $v \in \mathbb{N}_0$
- **Problem Solution:** given a collection of k coins with values $[c_1, c_2, \dots, c_k]$, an array S of length k , with $S[i]$ indicating how many coins of value c_{i+1} are in the optimal solution
 - we also use $C(v)$ to denote the minimum number of coins required to add up to v
- **Developing Recurrence:** it is easy to see that, if we have an optimal solution, there exists a coin c_i that must be part of this solution. So it must be the case that:

$$C(v) = 1 + C(v - c_i)$$

Thus, for an optimal solution, we must have the following recurrence:

$$C(v) = \begin{cases} 1 & v = c_i, 1 \leq i \leq k \\ 1 + \min\{C(v - c_i) \mid 1 \leq i \leq k, c_i \leq v\} & \text{otherwise} \end{cases}$$

- **Finding an Optimal Solution:** from the above, if we can precompute $C(w) \forall w \in [1, v]$, then solving the recurrence is very easy. Thus, we get the following “plan”:
1. create array C of length $v+1$, such that $C[w]$ is the minimum number of coins to obtain a value of w
 2. we can compute solutions for $C[0], C[1], \dots, C[v]$ using our recurrence relation above
 3. to reconstruct the sequence of coins required for an optimal solution, store in an array P the value of a coin used to obtain the optimal solution for w (store at $P[w]$)
 4. to build the list of coins, we just do:
 - $P[v]$ is optimal coin required to achieve v
 - $P[v - P[v]]$ is optimal coin required to achieve $v - P[v]$
 - continue like this until we reach 0, at which point the sequence $P[v], P[v - P[v]], \dots$ is the optimal sequence of coins we were seeking

Looking online, most of the methods encountered use a matrix, but this is much shorter and elegant.

Algorithm Dyn-Coins($v; c_1, \dots, c_k$)

1. initialise array c of length k to hold the c_i values
2. initialise array S of length k (to 0s)
3. initialise arrays C, P of length $v + 1$ (to ∞)
4. $C[0] \leftarrow 0, C[1] \leftarrow 1$ //We assume $c_1 = 1$
5. **for** $w \leftarrow 2$ **to** v //We work "bottom-up"
6. **for** $i = 1$ **to** k //We try all coin values
7. **if** ($c[i] \leq w$) **and** ($C[w - c[i]] + 1 < C[w]$)
8. $C[w] \leftarrow 1 + C[w - c[i]]$
9. $P[w] \leftarrow i$
10. **while** $v > 0$ //Now we work back to build S
11. $i \leftarrow P[v]$
12. $S[i] \leftarrow S[i] + 1; v \leftarrow v - c[i]$
13. **return** $C[v]$ "is the number of coins. The solution is in array S ".

Figure 3: Lines 5 to 9 is the whole DP part, in which we compute the minimum number of coins for each w on the way to v . In lines 7 to 9, we are constantly rewriting the values of $C[w]$ and $P[w]$ until we reach the minimum value. In lines 10 to 12, we implement the rebuilding of the coin set required for the optimal solution. Mary's walkthrough is very thorough (Lecture 16, Part 4)

We consider $v = 11$, with coins 1,5,6 and 8.

$$C = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$P = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

1. $C[0] = 0, P[0] = 0$ trivially
2. $C[1] = 1, P[1] = 0$ trivially (the algorithm ignores these steps and sets them up beforehand)
3. consider $C[2]$
 - the only values of the coins which are less than or equal to 2 is the coin with value 1, so this will be the minimum value

Thus, $C[2] = 1 + C[2 - 1] = 1 + C[1] = 2$. Moreover, $P[2] = 1$, as 1 is the only coin that can be used:

$$C = [0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$P = [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

4. consider $C[3]$

- the only values of the coins which are less than or equal to 3 is the coin with value 1, so this will be the minimum value

Thus, $C[3] = 1 + C[3 - 1] = 1 + C[2] = 3$. Moreover, $P[3] = 1$, as 1 is the only coin that can be used:

$$C = [0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0]$$

$$P = [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]$$

5. consider $C[4]$

- the only values of the coins which are less than or equal to 4 is the coin with value 1, so this will be the minimum value

Thus, $C[4] = 1 + C[4 - 1] = 1 + C[3] = 4$. Moreover, $P[4] = 1$, as 1 is the only coin that can be used:

$$C = [0, 1, 2, 3, 4, 0, 0, 0, 0, 0, 0]$$

$$P = [0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]$$

6. consider $C[5]$

- we now have 2 choices with values less than or equal to 5: the coin of value 1, or the coin of value 5

Hence:

$$C[5] = 1 + \min\{C[5-1], C[5-5]\} = 1 + \min\{C[4], C[0]\} = 1 + \min\{4, 0\} = 1$$

Moreover, since we use the coin of value 5, $P[5] = 5$.

$$C = [0, 1, 2, 3, 4, 1, 0, 0, 0, 0, 0]$$

$$P = [0, 1, 1, 1, 1, 5, 0, 0, 0, 0, 0]$$

7. consider $C[6]$

- we now have the choice to use 3 coins (1,5,6)

Hence:

$$C[6] = 1 + \min\{C[6-1], C[6-5], C[6-6]\} = 1 + \min\{C[5], C[1], C[0]\} = 1 + \min\{1, 1, 0\} = 1$$

Moreover, since we use the coin of value 6, $P[6] = 6$.

$$C = [0, 1, 2, 3, 4, 1, 1, 0, 0, 0, 0]$$

$$P = [0, 1, 1, 1, 1, 5, 6, 0, 0, 0, 0]$$

8. consider $C[7]$

- we now have the choice to use 3 coins (1,5,6)

Hence:

$$C[7] = 1 + \min\{C[7-1], C[7-5], C[7-6]\} = 1 + \min\{C[6], C[2], C[1]\} = 1 + \min\{1, 2, 1\} = 2$$

By how the algorithm is constructed, the coin with value 6 would never pass the if statement, so we have $P[7] = 1$. Thus:

$$C = [0, 1, 2, 3, 4, 1, 1, 2, 0, 0, 0, 0]$$

$$P = [0, 1, 1, 1, 1, 5, 6, 1, 0, 0, 0, 0]$$

9. consider $C[8]$

- we now have the choice to use all 4 coins

Hence:

$$C[8] = 1 + \min\{C[8-1], C[8-5], C[8-6], C[8-8]\} = 1 + \min\{C[7], C[3], C[2], C[0]\} = 1 + \min\{2, 3, 2, 0\} = 3$$

Moreover, since we use the coin of value 8, $P[8] = 8$.

$$C = [0, 1, 2, 3, 4, 1, 1, 2, 1, 0, 0, 0]$$

$$P = [0, 1, 1, 1, 1, 5, 6, 1, 8, 0, 0, 0]$$

10. consider $C[9]$

- we now have the choice to use all 4 coins

Hence:

$$C[9] = 1 + \min\{C[9-1], C[9-5], C[9-6], C[9-8]\} = 1 + \min\{C[8], C[4], C[3], C[1]\} = 1 + \min\{3, 4, 3, 1\} = 4$$

By how the algorithm is constructed, the coin with value 8 would never pass the if statement, so we have $P[9] = 1$. Thus:

$$C = [0, 1, 2, 3, 4, 1, 1, 2, 1, 2, 0, 0]$$

$$P = [0, 1, 1, 1, 1, 5, 6, 1, 8, 1, 0, 0]$$

11. consider $C[10]$

- we now have the choice to use all 4 coins

Hence:

$$C[10] = 1 + \min\{C[10-1], C[10-5], C[10-6], C[10-8]\} = 1 + \min\{C[9], C[5], C[4], C[2]\} = 1 + \min\{4, 1, 4, 2\} = 5$$

Moreover, since we use the coin of value 5, $P[10] = 5$.

$$C = [0, 1, 2, 3, 4, 1, 1, 2, 1, 2, 2, 0]$$

$$P = [0, 1, 1, 1, 1, 5, 6, 1, 8, 1, 5, 0]$$

12. consider $C[11]$

- we now have the choice to use all 4 coins

Hence:

$$C[11] = 1 + \min\{C[11-1], C[11-5], C[11-6], C[11-8]\} = 1 + \min\{C[10], C[6], C[5], C[3]\} = 1 + \min\{2, 1, 1, 1\} = 2$$

By how the algorithm is constructed, the coin with value 6 would never pass the if statement, so we have $P[11] = 5$. Thus:

$$C = [0, 1, 2, 3, 4, 1, 1, 2, 1, 2, 2, 2]$$

$$P = [0, 1, 1, 1, 1, 5, 6, 1, 8, 1, 5, 5]$$

Thus, we have found that $C(11) = 2$. To find which coins are required:

1. to get to 11, we had to use coin $P[11] = 5$
2. to get to $11 - 5 = 6$, we had to use coin $P[6] = 6$
3. $6 - 6 = 0$, so we terminate, and the coins to use are 5, 6

In the actual algorithm we would get an array $S = [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]$, corresponding to using one 5 coin, and one 6 coin

- **Alternatives to DP:**

- a **recursive** implementation would get stuck in the same problem as Fibonacci, with redundant calculations occurring (for example, constantly finding the best way to get value 5)
- a **greedy** implementation, by which we use the coins of greatest value possible; a decent heuristic, but not optimal (for example, coins 1, 5, 7 and $v = 18$)

2 Week 1 - Seam-Carving

- **Seam Carving:** the process of removing/adding a “seam” of pixels to an image to resize it without distorting the proportions of the objects in the image
 - we select a seam with the lowest energy



- **Defining an Image:** define as a matrix with m rows and n columns. Objective is to turn an $m \times n$ image into an $m' \times n'$ image.
- **Vertical Seam:** a vertical collection of pixels in which any 2 consecutive pixels differ horizontally by at most 1 pixel, and vertically by exactly 1 pixel
 - if (i, j) is a pixel in the seam, then the only possible pixels below it must be $(i + 1, j - 1)$ (below left), $(i + 1, j)$ (below) or $(i + 1, j + 1)$ (below right)
 - can use j_i to denote the column of a pixel in a vertical seam going through row i
- **Horizontal Seam:** a horizontal collection of pixels in which any 2 consecutive pixels differ vertically by at most 1 pixel, and horizontally by exactly 1 pixel
 - if (i, j) is a pixel in the seam, then the only possible pixels to its right must be $(i - 1, j + 1)$ (right below), $(i, j + 1)$ (right) or $(i + 1, j + 1)$ (right above)
 - can use i_j to denote the row of a pixel in a horizontal seam going through column j
- **Energy of a Pixel:** if (i, j) is a pixel, denote its energy by:

$$e_I(i, j)$$

- a classical example is a Sobel Operator. For a pixel (i, j) ,

$$e_I(i, j) = \left| \frac{\partial}{\partial x} I \right|_{i,j} + \left| \frac{\partial}{\partial y} I \right|_{i,j}$$

where:

$$\frac{\partial}{\partial x} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

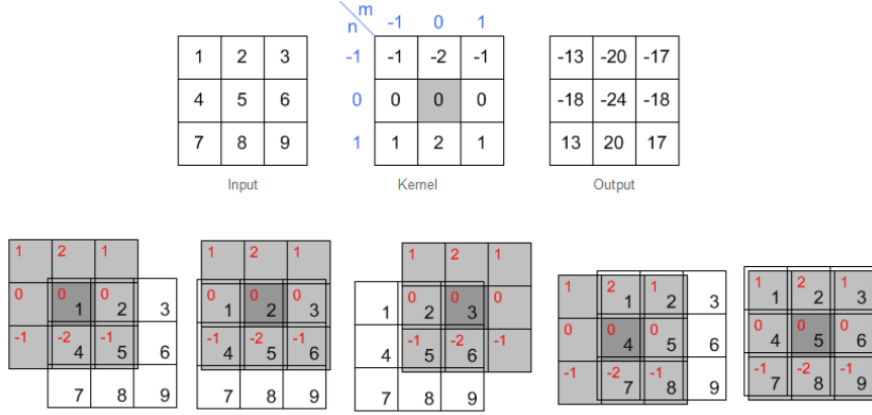


Figure 4: $\frac{\partial}{\partial y}|_{0,0} = (1 \times 1 - 1 \times 7) + 2(-1 \times 8 + 1 \times 2) + (1 \times 3 - 1 \times 9) = -24$

$$\frac{\partial}{\partial y} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{pmatrix}$$

These operators are applied by convolving them on a 3×3 square around (i, j) (we need to flip them on the axis of 0 to apply)

- **Energy of a Seam:** the sum of the energies of all pixels composing the seams
- **Recurrence for Optimal Seam:** notice that if an optimal, vertical seam ends at some pixel p , then all pixels above p must have conformed an optimal seam of length $m - 1$
 - if $p = (m, j_m)$, then the optimal seam of length $m - 1$ must have ended at $(m - 1, j_m)$ (directly above), $(m - 1, j_m - 1)$ (above to the left) or $(m - 1, j_m + 1)$
 - the above are the only possibilities which allow for an existence of a seam of ending at p that is optimal (if any of the 3 aren't ends to an optimal seam, the original seam of length m wouldn't be optimal)
 - but we have defined 3 new subproblems: what are the optimal seams of length $m - 1$ that end at each of these pixels?
 - in other words, an optimal seam of length m ending at a pixel (m, j_m) will have enrgy equal to the energy of this last pixel, plus the energy of an optimal seam of length $m - 1$.

Given the above, we can then define a recurrence for the optimal (vertical) seam energy, from somewhere in the first row ($m = 1$) to pixel (i, j) :

$$opt_I(i, j) = e_I(i, j) + \begin{cases} 0 & i = 1 \\ \min\{opt_I(i - 1, j - 1), opt_I(i - 1, j), opt_I(i - 1, j + 1)\} & otherwise \end{cases}$$

where $1 \leq i \leq m, 1 \leq j \leq n$

- we require $e_I(i, 1) = e_I(1, n) = \infty$, to ensure that the left/right sides are never selected as part of the seams

- **The *opt* Array:** the purpose of seam carving is to find **the** optimal path. This is the same as finding the optimal seam with the lowest energy ending at any of the pixels at row m . To do this, we create an $m \times n$ array, such that $opt[i, j]$ denotes the optimal value of any seam running from $m = 1$ to pixel (i, j) (so $opt[i, j] = opt_I(i, j)$).
- **The *e* Table:** will be used to store precomputed values of all energy values of any pixel (so $e[i, j] = e_I(i, j)$ and naturally $e[i, 1] = e[i, n] = \infty$ to avoid sides). This helps us readily access the available energies.
- **The *p* Table:** used to know which pixels are part of the seam. To do this:
 - $p[i, j] = -1$ if $opt_I(i, j)$ was computed via $opt_I(i - 1, j - 1)$
 - $p[i, j] = 0$ if $opt_I(i, j)$ was computed via $opt_I(i - 1, j)$
 - $p[i, j] = 1$ if $opt_I(i, j)$ was computed via $opt_I(i - 1, j + 1)$

We can then easily reconstruct the seam.

- **The Seam Carving Algorithm:** we use the above tables, alongside the recursion to develop the algorithm. It will go through each pixel, using previously calculated optimal path energies to fill in the *opt* and *p* tables. At the end, we find the pixel (call it p^*) in the last row with the lowest value. This means that there is a seam starting at $m = 1$ and ending at p^* , such that this seam is the seam of lowest energy in the whole image. We then use the *p* array to rebuild the seam (if $p^* = (m, j^*)$, then the pixel before it will be $(m - 1, j^* + p[m, j^*])$).

Algorithm Vertical-Seam(I, m, n)

```

1. for j ← 1 to n
2.   for i ← 1 to m
3.     e[i, j] ← "compute eI(i, j)"           //Θ(1) time
4.     opt[1, j] ← e[1, j], p[1, j] ← 0        //Base case
5. for i ← 1 to m
6.   for j ← 1 to n
7.     opt[i, j] ← opt[i - 1, j], p[i, j] ← 0  //default case
8.     if opt[i - 1, j - 1] < opt[i, j] then    O(mn)
9.       opt[i, j] ← opt[i - 1, j - 1], p[i, j] ← -1
10.    if opt[i - 1, j + 1] < opt[i, j] then
11.      opt[i, j] ← opt[i - 1, j + 1], p[i, j] ← +1
12.    opt[i, j] ← opt[i, j] + e[i, j]          //Always add e[i, j]
13. j* ← 2
14. for j ← 1 to n
15.   if opt[m, j] < opt[m, j*] then j* ← j    O(n)
16. Print("Best vertical seam ends at cell (m, j*)").

```

- **Lines 1 - 4:** initialise e (containing precomputed energies of each pixel); the optimal values for the first row are precisely the energies of the pixels in the first rows; initialise p array.
- **Lines 5 - 12:** execute the whole DP procedure; for each pixel (i, j) , initialise $opt[i, j]$ with the minimum from all optimal seam energies above it; then, add the energy at pixel (i, j) to obtain the energy of the seam ending at (i, j)
- **Lines 13 - 16:** go through all entries of opt in its last row, to find the last pixel of the optimal seam (that is, the value of j in $opt[m, j]$, such that $opt[m, j]$ is the smallest of all elements in row m)
- **Seam Carving Runtime:** since we iterate over all pixels doing constant time work, this is an $\mathcal{O}(mn)$ operation (we also do the same work in initialising e). In the last part, we do $\mathcal{O}(n)$ work, as we just need to iterate over 1 row. Thus, the total runtime for Seam Carving is:

$$\mathcal{O}(mn)$$

- A great video by 3Blue1Brown on seam carving

3 Week 1 - Edit Distance

- **Edit Distance:** the process of finding the minimum number of operations (out of insertion, deletion and substitution) required to transform a string into another

```

      I N T E * N T I O N
      | | | | | | | | |
      * E X E C U T I O N
      d s s   i s

```

Figure 5: The edit distance between the strings “intention” and “execution” is 5: 1 deletion (I → -), 1 insertion (- → C) and 3 substitutions (N → E, T → X, N → U)

- **Alignment of Sequences:** there are many ways to align 2 sequences of characters, such that the resulting alignment contains 2 strings of the same length, with no “-” in the same column:

```

      A C C G G T A T C C T A G G A C
      A C C T A T C T - - T A G G A C

      A C C G G T A T C C T A G G A C
      A C C - - T A T C T T A G G A C

```

Figure 6: Two possible alignments for a sequence of characters. We can add padding to make sure that both sequences have the same length, but have to ensure that padding isn’t present within the same column.

- **Score of an Alignment:** the total number of operations required to produce a valid alignment of 2 sequences. Let s and t be 2 sequences. An operation at index i (so at s_i and t_i) can be described as:
 - **insertion:** $s_i = -$, but t_i is a character
 - **deletion:** s_i is a character, but $t_i = -$
 - **substitution:** s_i and t_i are characters, but $s_i \neq t_i$
- **The Edit Distance:** the minimum number of operations required to produce a valid alignment of 2 sequences
- **Recurrence for Edit Distance:** if we have an optimal alignment of 2 strings (s of length m , and t of length n), then the last column must have been arranged in 1 of 4 ways:
 - both characters coincided, so nothing happened

- characters differed, so a substitution was made
- an insertion was made
- a deletion was made

Since the alignment was optimal, the edit distance must be the cost of the last operation performed (0, or 1 if we did an insertion/substitution/deletion) plus the minimum edit distance of the optimal alignment of one of:

- $s[1, \dots, m-1]$ and $t[1, \dots, n-1]$ (if s_m and t_n were both characters)
- $s[1, \dots, m]$ and $t[1, \dots, n-1]$ (if an insertion was made aka $- \rightarrow t_n$, in which case we still have the whole of s to compare)
- $s[1, \dots, m-1]$ and $t[1, \dots, n]$ (if a deletion was made aka $s_m \rightarrow -$, in which case we still have the whole of t to compare)

Thus, the edit distance for s and m can be described recursively by:

$$d(s[1, \dots, m], t[1, \dots, n]) = \begin{cases} m & n = 0 \text{ (} t \text{ has no characters left)} \\ n & m = 0 \text{ (} s \text{ has no characters left)} \\ d(s[1, \dots, m-1], t[1, \dots, n-1]) & s_m = t_n \\ 1 + \min\{d(s[1, \dots, m-1], t[1, \dots, n-1]), \\ d(s[1, \dots, m], t[1, \dots, n-1]), \\ d(s[1, \dots, m-1], t[1, \dots, n])\} & s_m \neq t_n \end{cases}$$

This is impossible to do recursively: the tree generated would have $3^{\min\{m,n\}-1}$ leaves, but we would only have $m \times n$ possible states

- **The d Table:** an $(m+1) \times (n+1)$ table used to store the edit distance of all possible substrings of s and m
 - $d[i, j] = d(s[1, \dots, i], t[1, \dots, j])$ corresponds to calculating the edit distance between $s[1, \dots, i]$ and $t[1, \dots, j]$
 - we have $d[i, 0] = i$ (a string of length i can only match a string of length 0 if it performs i deletions) and $d[0, j] = j$ (a string of length 0 can only match a string of length j if it performs j insertions). By ensuring these cases are covered, we ensure that we correctly define our recurrence for DP.
- **The a Array:** used to reconstruct the optimal alignment of the 2 strings. At each entry $a[i, j]$, stores a quaternary flag (0,1,2,3) to denote that the last character of the alignment of $s[1, \dots, i]$ and $t[1, \dots, j]$ was achieved via:
 - no action (0)
 - substitution (1)
 - insertion (2)

– deletion (3)

- The Edit Distance Algorithm:

Algorithm Edit-Distance($s[1 \dots m], t[1 \dots n]$)

```

1. for  $i \leftarrow 0$  to  $m$ 
2.      $d[i, 0] \leftarrow i, a[i, 0] \leftarrow 3$ 
3. for  $j \leftarrow 0$  to  $n$ 
4.      $d[0, j] \leftarrow j, a[0, j] \leftarrow 2$ 
5. for  $i \leftarrow 1$  to  $m$  do
6.     for  $j \leftarrow 1$  to  $n$  do
7.         if  $s_i = t_j$  then
8.              $d[i, j] \leftarrow d[i - 1, j - 1]$ 
9.              $a[i, j] \leftarrow 0$ 
10.        else
11.             $d[i, j] \leftarrow 1 + \min\{d[i, j - 1], d[i - 1, j], d[i - 1, j - 1]\}$ 
12.            if  $d[i, j] = d[i - 1, j - 1] + 1$  then  $a[i, j] \leftarrow 1$ 
13.            else if  $\min = d[i, j - 1] + 1$  then  $a[i, j] \leftarrow 2$ 
14.            else  $a[i, j] \leftarrow 3$ 

```

- **Lines 1-4:** initialise d when we are computing edit distance for empty strings; if the second string is empty, we perform deletions (3); if the first string is empty we perform insertions (2)
- **Lines 5-14:** iterates over all possible cases, for all possible substrings of s and t . The desired edit distance will be stored at $d[m, n]$. Notice that, for any $d[i, j]$ we only need to look at 3 other cells: $d[i - 1, j]$ (above), $d[i, j]$ (left upwards diagonal), $d[i, j - 1]$ (above)
- **Edit Distance Runtime:** we do $\mathcal{O}(m + n)$ work in lines 1 to 4, and we do $mn \mathcal{O}(1)$ actions for the rest. Thus, the total runtime is:

$$\mathcal{O}(mn)$$

- **Worked Example:** lets find the edit distance of $s = HYUNDAI$ and $t = HONDA$ (not sponsored). The initial table after executing lines 1 and 4:

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	-	-	-	-	-	-	-
O	2	-	-	-	-	-	-	-
N	3	-	-	-	-	-	-	-
D	4	-	-	-	-	-	-	-
A	5	-	-	-	-	-	-	-

1. Comparing H

- the human way of looking at this is noticing that “H” will match with any substring of “HYUNDAI” only once (with the initial “H”) so the edit distance is just going to be $len(substring) - 1$
- the algorithmic way of looking at this is by looking at the cells above, to the left, and to the left-top diagonal. If the last character matches, then the value is the diagonal cell. Alternatively, it is just the minimum of these 3 cells + 1
- for example, for “H” and “H” these 2 match. Looking at the diagonal ($d[0,0]$) it is 0, so $d[1,1] = 0$.
- for “HY” and “H”, “H” and “Y” don’t match, so

$$d[1,2] = 1 + \min\{d[0,2], d[1,1], d[0,1]\} = 1 + \min\{2, 0, 1\} = 1$$

- for “HYU” and “H”, “H” and “U” don’t match, so

$$d[1,3] = 1 + \min\{d[0,3], d[1,2], d[0,2]\} = 1 + \min\{3, 1, 2\} = 2$$

- if we continue we thus get

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	-	-	-	-	-	-	-
N	3	-	-	-	-	-	-	-
D	4	-	-	-	-	-	-	-
A	5	-	-	-	-	-	-	-

2. Comparing HO

- for “H” and “HO”, “H” and “O” don’t match, so

$$d[2,1] = 1 + \min\{d[1,1], d[2,0], d[1,0]\} = 1 + \min\{0, 2, 1\} = 1$$

this corresponds with the alignment “HO” “H-”

- again, continuing like this:

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	-	-	-	-	-	-	-
D	4	-	-	-	-	-	-	-
A	5	-	-	-	-	-	-	-

3. Comparing HON

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	-	-	-	-	-	-	-
A	5	-	-	-	-	-	-	-

- here, when comparing “HYUN” and “HON” (entry (3,4)), we get that $d[3,4] = 2$. This is because the “N” at the end of the substrings coincided, so for the edit distance we just consider the diagonal $d[2,3] = 2$ (aka we only need to match “YU” with “O”, which requires a substitution and a deletion)

4. Comparing HOND

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	-	-	-	-	-	-	-

5. Comparing HONDA

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Thus, the edit distance of “HONDA” and “HYUNDAI” is 3 (we need to turn “O” into “YU” and insert “I”)

6. **The a Table:** to get padded sequences back, we consider the a table, which is:

		H	Y	U	N	D	A	I
	2	2	2	2	2	2	2	2
H	3	0	2	2	2	2	2	2
O	3	3	1	2	2	2	2	2
N	3	3	3	1	0	2	2	2
D	3	3	3	3	1	0	2	2
A	3	3	3	3	3	1	0	2

To make sense of these numbers (normally these tables are filled in step by step, but it is helpful to understand the numbers)

- $a[3, 4] = 0$ because “HON” matches with “HYUN” (same ending letter N)
- $a[4, 4] = 1$ because “HOND” and “HYUN” have a substitution $D \rightarrow N$ available
- $a[2, 0] = 3$ because the only way to match the last character of “HO” with “—” is if we delete “O”; similarly $a[5, 2] = 3$ because the only way to match the last character of “HONDA” with “HY” is by deleting “A” (think of “HY” as “HY— —”; after deleting the “A” we get “HOND—”, which will match with “HY— —”)
- $a[2, 5] = 2$ because the only way to match the last character of “HO” with “HYUND” is by inserting a “D” (think of “HO” as “HO— —”; after inserting the “D” we get “HO—D” which will match with “HYUND”)

The algorithm for reconstructing the padded sequence is thus:

- to reconstruct the best alignment, we start at $a[m, n]$
- let b be an array to hold the padded s
- let c be an array to hold the padded t
- if:
 - * $a[i, j] = 0, 1$: insert s_i into b , and t_j into c . Then consider $i - 1$ and $j - 1$
 - * $a[i, j] = 2$: insert — into b , and t_j into c . Then consider i and $j - 1$ (if we put 2, then we chose an insertion, so the padded version of s must have nothing at index i)
 - * $a[i, j] = 3$: insert s_i into b , and — into c . Then consider $i - 1$ and j (if we put 3, then we chose a deletion, so the padded version of t must have nothing at index i)

- once either $i = 0$ or $j = 0$, we just have a run of insertion/deletions until $i = 0 = j$

7. **Reconstructing the Padded Sequences** Using the above, we can traverse the table. We indicate with colour the letters (i, j) which we consider. We start at $[m, n] = [5, 7]$

$$b = []$$

$$c = []$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[5, 7] = 2$, this means that we had to make an insertion. Thus, we go to cell $[5, 6]$. We need to add - to b and a t_7 to c (since HONDA has a space, which must turn into an I match the end of HYUNDAI)

$$b = [-]$$

$$c = [I]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[5, 6] = 0$, we have a match, so we can insert $s_5 = A$ into b and $t_6 = A$ into c . We also decrease, so that $i = 4, j = 5$:

$$b = [-, A]$$

$$c = [I, A]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[4, 5] = 0$, we have a match, so we can insert $s_4 = D$ into b and $t_5 = D$ into c . We also decrease, so that $i = 3, j = 4$:

$$b = [-, A, D]$$

$$c = [I, A, D]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[3, 4] = 0$, we have a match, so we can insert $s_3 = N$ into b and $t_4 = N$ into c . We also decrease, so that $i = 2, j = 3$:

$$b = [-, A, D, N]$$

$$c = [I, A, D, N]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[2, 3] = 2$, we must've had an insertion, so we add - to b , and insert $t_3 = U$ into c . We now have, after decreasing j , $i = 2, j = 2$:

$$b = [-, A, D, N, -]$$

$$c = [I, A, D, N, U]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[2, 2] = 1$, we must've had a substitution, so we insert $s_2 = O$ into b , and insert $t_2 = Y$ into c . Then, after decreasing, we have $i = 1, j = 1$

$$b = [-, A, D, N, -, O]$$

$$c = [I, A, D, N, U, Y]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Since $a[1, 1] = 0$, we must've had a match, so we insert $s_1 = H$ into b , and insert $t_1 = H$ into c . Then, after decreasing, we have $i = 0, j = 0$

$$b = [-, A, D, N, -, O, H]$$

$$c = [I, A, D, N, U, Y, H]$$

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Now, since $i = j = 0$, we terminate. Thus, our sequence, after reversing b and c , is:

H O - N D A -

H Y U N D A I

which clearly has edit distance 3 (1 substitution, 2 insertion), as we expected

- A short video on finding edit distance

4 Week 2 - All Pairs Shortest Paths

- **All Pairs Shortest Paths:** given a weighted digraph (edges have “weights”, corresponding to a “cost” of traversing through said edges), determine the minimum path cost between any 2 nodes $u, v \in V$
 - path cost is the sum of all weights for edges conforming the path
- **Expected Solution:** given a digraph with n nodes we want a matrix $D \in \mathbb{R}^{n \times n}$, we want $D[i, j]$ to be the shortest path between node i and j
- **Digraph Without Cycles:** if a digraph of n nodes has no cycle of negative cost, then the minimum cost path between any 2 nodes (assuming any path exists) will contain at most $n - 1$ nodes
 - if a path is a minimum cost path with more than n elements, then some element must have been repeated, implying that there must be a cycle. A cycle can only provide a minimum cost path if it reduces the total cost aka it is negative

- **The Floyd Warshall Algorithm: Idea:** consider finding the shortest path between 2 nodes i, j by using vertices for the path only in a restricted set:

$$V_k = \{0, 1, \dots, k - 1\}$$

For example, with V_0 , we can't use any vertex in V , so the shortest path will only exist if $\exists(i, j) \in E$. If V_1 , then our path is allowed to use the node 0.

- **The Matrix $D^{<k}$:** a matrix, such that $D^{<k}[i, j]$ contains the shortest path between the nodes i, j , by using only V_k to construct said path
- **Path Cost Conventions:**
 - the path cost between a node and itself is 0
 - if no path exists between 2 nodes, the path cost is ∞
 - * this includes when we restrict paths to only contain vertices in V_k
- **Floyd Warshall: Recurrence:** notice, to construct $D^{<k+1}$, we can do so using only D^k . The only difference is that, for $D^{<k+1}$, $k \in V_{k+1}$. Thus, there are two possibilities for the value of $D^{<k+1}[i, j]$:

- the minimum cost path between i and j does **not** contain k , in which case:

$$D^{<k+1}[i, j] = D^{<k}[i, j]$$

- the minimum cost path between i and j does contain k , in which case:

$$D^{<k+1}[i, j] = D^{<k}[i, k] + D^{<k}[k, j]$$

Thus, we obtain the following recursion:

$$D^{<k+1}[i, j] = \begin{cases} 0 & i = j \\ \min\{D^{<k}[i, j], D^{<k}[i, k] + D^{<k}[k, j]\} & \text{otherwise} \end{cases}$$

where we use the base case:

$$D^{<0}[i, j] = \begin{cases} 0 & i = j \\ w(i, j) & i \neq j, (i, j) \in E \\ \infty & i \neq j, (i, j) \notin E \end{cases}$$

- **The Floyd Warshall Algorithm for All Pairs Shortest Paths**

Algorithm FloydWarshall(G, w)

1. Initialise $D^{<0}$ using Base case details
2. **for** $k = 0$ **to** $n - 1$ **do**
3. **for** $i = 0$ **to** $n - 1$ **do**
4. **for** $j = 0$ **to** $n - 1$ **do**
5. $D^{<k+1}[i, j] \leftarrow D^{<k}[i, j]$ //Default option
6. **if** $j \neq i$ **and** $(D^{<k}[i, k] + D^{<k}[k, j]) < D^{<k+1}[i, j]$
7. $D^{<k+1}[i, j] \leftarrow D^{<k}[i, k] + D^{<k}[k, j]$
8. **return** $D^{<n+1}$

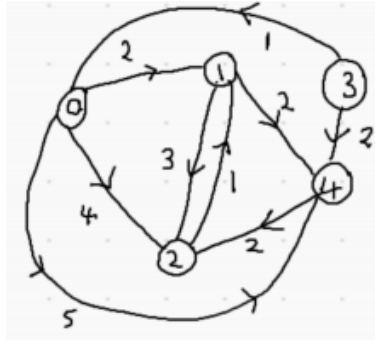
Figure 7: In practice, we only need 2 matrices: $D^{<k}$ and $D^{<k+1}$. The all pairs shortest paths matrix is $D^{<n+1}$. Moreover, given that constan time is done in 3 for-loops, runtime is $\mathcal{O}(n^3)$

- **Getting the Shortest Path:** to get the shortest path between 2 nodes, we create a matrix $\Pi^{<k}$, such that $\Pi^{<k}[i, j] = p$, where p is the node before j in the shortest path:

- if the shortest path doesn't include k , then $\Pi^{<k+1}[i, j] = \Pi^{<k}[i, j]$
- otherwise, $\Pi^{<k+1}[i, j] = \Pi^{<k}[k, j]$

5. $D^{<k+1}[i, j] \leftarrow D^{<k}[i, j]$ //Default option
- $\Pi^{<k+1}[i, j] \leftarrow \Pi^{<k}[i, j]$ //Copy "predecessor" (of j) too
6. **if** $j \neq i$ **and** $(D^{<k}[i, k] + D^{<k}[k, j]) < D^{<k+1}[i, j]$
7. $D^{<k+1}[i, j] \leftarrow D^{<k}[i, k] + D^{<k}[k, j]$
- $\Pi^{<k+1}[i, j] \leftarrow \Pi^{<k}[k, j]$
- // "Predecessor" (of j) is from the $D^{<k}[k, j]$ subpath

• Walkthrough: Tutorial 6, Question 2



– $k = 0$:

We initialise the matrix $D^{<0}$. This is basically an adjacency matrix, but instead of 0 or 1, we put in the weights:

$$\begin{array}{c} \begin{matrix} & 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 2 & 4 & \infty & 5 \\ \infty & 0 & 3 & \infty & 2 \\ \infty & 1 & 0 & \infty & \infty \\ 1 & \infty & \infty & 0 & 2 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix} \end{array}$$

– $k = 1$

Now, we allow paths to go through the vertex 0. It would be tedious to do it one by one, so we can use (Math Gods, please look away) “matrix sum”: given a column and a row vector, we can create a matrix given by a pairwise sum of the elements in each. For example:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} = \begin{bmatrix} 1.1 & 1.2 & 1.3 \\ 2.1 & 2.2 & 2.3 \\ 3.2 & 3.2 & 3.3 \end{bmatrix}$$

For $D^{<k+1}$, we use $D^{<k}$ by considering its k^{th} column (which includes all distances from every vertex to k), and its k^{th} row (which includes all distances from k to every vertex). Thus, the “matrix sum” of C_k and R_k will produce a matrix where the $[i, j]$ entry will be the value of $D^{<k}[i, k] + D^{<k}[k, j]$. Then, it is very easy to compare this matrix with the original $D^{<k}$ matrix to see whether including k actually provides an improvement. We must notice that in the “sum matrix”, we omit the k^{th} row and column, as they would be nonsensical (if $i = k$, we would be adding the distances of k to k , to the distance of k to j , which is just going to be the distance of k to j).

Thus, the sum matrix of $D^{<1}$ will be given by:

$$\begin{bmatrix} 0 \\ \infty \\ \infty \\ 1 \\ \infty \end{bmatrix} + \begin{bmatrix} 0 & 2 & 4 & \infty & 5 \end{bmatrix} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} - & - & - & - & - \\ - & \infty & \infty & \infty & \infty \\ - & \infty & \infty & \infty & \infty \\ - & 3 & 5 & \infty & 6 \\ - & \infty & \infty & \infty & \infty \end{bmatrix} \end{matrix}$$

And now we compare $D^{<0}$ to the sum matrix, to see if any “cell” has a lower path cost:

$$\begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 2 & 4 & \infty & 5 \\ \infty & 0 & 3 & \infty & 2 \\ \infty & 1 & 0 & \infty & \infty \\ 1 & \infty & \infty & 0 & 2 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix} \end{matrix} \quad \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} - & - & - & - & - \\ - & \infty & \infty & \infty & \infty \\ - & \infty & \infty & \infty & \infty \\ - & 3 & 5 & \infty & 6 \\ - & \infty & \infty & \infty & \infty \end{bmatrix} \end{matrix}$$

Some cells have changed, so some paths improve by including 0.
Thus:

$$D^{<1} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 2 & 4 & \infty & 5 \\ \infty & 0 & 3 & \infty & 2 \\ \infty & 1 & 0 & \infty & \infty \\ 1 & \textcolor{red}{3} & \textcolor{red}{5} & 0 & 2 \\ \infty & \infty & 2 & \infty & 0 \end{bmatrix} \end{matrix}$$

– **k = 2**

The sum matrix of $D^{<2}$ will be given by:

$$\begin{bmatrix} 2 \\ 0 \\ 1 \\ 3 \\ \infty \end{bmatrix} + \begin{bmatrix} \infty & 0 & 3 & \infty & 2 \end{bmatrix} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} \infty & - & 5 & \infty & 4 \\ - & - & - & - & - \\ \infty & - & 4 & \infty & 3 \\ \infty & - & 6 & \infty & 5 \\ \infty & - & \infty & \infty & \infty \end{bmatrix} \end{matrix}$$

And now we compare $D^{<1}$ to the sum matrix, to see if any “cell” has a lower path cost:

$$\begin{array}{c}
\begin{array}{ccccc}
& 0 & 1 & 2 & 3 & 4 \\
0 & \begin{bmatrix} 0 & 2 & 4 & \infty & 5 \end{bmatrix} \\
1 & \begin{bmatrix} \infty & 0 & 3 & \infty & 2 \end{bmatrix} \\
2 & \begin{bmatrix} \infty & 1 & 0 & \infty & \infty \end{bmatrix} \\
3 & \begin{bmatrix} 1 & 3 & 5 & 0 & 2 \end{bmatrix} \\
4 & \begin{bmatrix} \infty & \infty & 2 & \infty & 0 \end{bmatrix}
\end{array}
&
\begin{array}{ccccc}
& 0 & 1 & 2 & 3 & 4 \\
0 & \begin{bmatrix} \infty & - & 5 & \infty & 4 \end{bmatrix} \\
1 & \begin{bmatrix} - & - & - & - & - \end{bmatrix} \\
2 & \begin{bmatrix} \infty & - & 4 & \infty & 3 \end{bmatrix} \\
3 & \begin{bmatrix} \infty & - & 6 & \infty & 5 \end{bmatrix} \\
4 & \begin{bmatrix} \infty & - & \infty & \infty & \infty \end{bmatrix}
\end{array}
\end{array}$$

Some cells have changed, so some paths improve by including 1.
Thus:

$$D^{<2} = \begin{array}{c}
\begin{array}{ccccc}
& 0 & 1 & 2 & 3 & 4 \\
0 & \begin{bmatrix} 0 & 2 & 4 & \infty & \textcolor{red}{4} \end{bmatrix} \\
1 & \begin{bmatrix} \infty & 0 & 3 & \infty & 2 \end{bmatrix} \\
2 & \begin{bmatrix} \infty & 1 & 0 & \infty & \textcolor{red}{3} \end{bmatrix} \\
3 & \begin{bmatrix} 1 & 3 & 5 & 0 & 2 \end{bmatrix} \\
4 & \begin{bmatrix} \infty & \infty & 2 & \infty & 0 \end{bmatrix}
\end{array}
\end{array}$$

– **k = 3**

The sum matrix of $D^{<3}$ will be given by:

$$\begin{array}{c}
\begin{bmatrix} 4 \\ 3 \\ 0 \\ 5 \\ 2 \end{bmatrix} + \begin{bmatrix} \infty & 1 & 0 & \infty & 3 \end{bmatrix} = \begin{array}{c}
\begin{array}{ccccc}
& 0 & 1 & 2 & 3 & 4 \\
0 & \begin{bmatrix} \infty & 5 & - & \infty & 7 \end{bmatrix} \\
1 & \begin{bmatrix} \infty & 4 & - & \infty & 6 \end{bmatrix} \\
2 & \begin{bmatrix} - & - & - & - & - \end{bmatrix} \\
3 & \begin{bmatrix} \infty & 6 & - & \infty & 8 \end{bmatrix} \\
4 & \begin{bmatrix} \infty & 3 & - & \infty & 5 \end{bmatrix}
\end{array}
\end{array}$$

And now we compare $D^{<2}$ to the sum matrix, to see if any “cell” has a lower path cost:

$$\begin{array}{c}
\begin{array}{ccccc}
& 0 & 1 & 2 & 3 & 4 \\
0 & \begin{bmatrix} 0 & 2 & 4 & \infty & 4 \end{bmatrix} \\
1 & \begin{bmatrix} \infty & 0 & 3 & \infty & 2 \end{bmatrix} \\
2 & \begin{bmatrix} \infty & 1 & 0 & \infty & 3 \end{bmatrix} \\
3 & \begin{bmatrix} 1 & 3 & 5 & 0 & 2 \end{bmatrix} \\
4 & \begin{bmatrix} \infty & \infty & 2 & \infty & 0 \end{bmatrix}
\end{array}
&
\begin{array}{ccccc}
& 0 & 1 & 2 & 3 & 4 \\
0 & \begin{bmatrix} \infty & 5 & - & \infty & 7 \end{bmatrix} \\
1 & \begin{bmatrix} \infty & 4 & - & \infty & 6 \end{bmatrix} \\
2 & \begin{bmatrix} - & - & - & - & - \end{bmatrix} \\
3 & \begin{bmatrix} \infty & 6 & - & \infty & 8 \end{bmatrix} \\
4 & \begin{bmatrix} \infty & 3 & - & \infty & 5 \end{bmatrix}
\end{array}
\end{array}$$

Some cells have changed, so some paths improve by including 2.

Thus:

$$D^{<3} = \begin{array}{c} \begin{matrix} & 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 2 & 4 & \infty & 4 \\ \infty & 0 & 3 & \infty & 2 \\ \infty & 1 & 0 & \infty & 3 \\ 1 & 3 & 5 & 0 & 2 \\ \infty & \textcolor{red}{3} & 2 & \infty & 0 \end{bmatrix} \end{array}$$

– **k = 4**

Looking at the graph, no edge actually goes into node 3, so no path can actually go through 3, so:

$$D^{<4} = D^{<3}$$

– **k = 5**

(Last one!) The sum matrix of $D^{<5}$ will be given by:

$$\begin{bmatrix} 4 \\ 2 \\ 3 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} \infty & 2 & 2 & \infty & 0 \end{bmatrix} = \begin{array}{c} \begin{matrix} & 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} \infty & 6 & 6 & \infty & - \\ \infty & 4 & 4 & \infty & - \\ \infty & 5 & 5 & \infty & - \\ \infty & 4 & 4 & \infty & - \\ - & - & - & - & - \end{bmatrix} \end{array}$$

And now we compare $D^{<4}$ to the sum matrix, to see if any “cell” has a lower path cost:

$$\begin{array}{c} \begin{matrix} & 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 2 & 4 & \infty & 4 \\ \infty & 0 & 3 & \infty & 2 \\ \infty & 1 & 0 & \infty & 3 \\ 1 & 3 & 5 & 0 & 2 \\ \infty & 3 & 2 & \infty & 0 \end{bmatrix} \end{array} \quad \begin{array}{c} \begin{matrix} & 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} \infty & 6 & 6 & \infty & - \\ \infty & 4 & 4 & \infty & - \\ \infty & 5 & 5 & \infty & - \\ \infty & 4 & 4 & \infty & - \\ - & - & - & - & - \end{bmatrix} \end{array}$$

Some cells have changed, so some paths improve by including 4. Thus:

$$D^{<5} = \begin{array}{c} \begin{matrix} & 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 2 & 4 & \infty & 4 \\ \infty & 0 & 3 & \infty & 2 \\ \infty & 1 & 0 & \infty & 3 \\ 1 & 3 & \textcolor{red}{4} & 0 & 2 \\ \infty & 3 & 2 & \infty & 0 \end{bmatrix} \end{array}$$

and this is our final, shortest path array.

5 Week 2 - Probabilistic Finite State Machines and the Viterbi Algorithm

6 Week 3 - Context Free Languages and Grammars

6.1 Context Free Grammars and their Language

- **Context Free Grammar:** a CFG \mathcal{G} is composed of:
 - **terminals:** a set Σ of terminals represent concrete objects. For example, “cat”, or “2”. Allow us to form sentences with meaning.
 - **non-terminals:** a set N , disjoint from Σ , they provide a structure to the sentences we can form. For example, “2” is an instance of a *Number*, and “2+2” is an instance of an *Arithmetic-Expression*. We can build *Arithmetic-Expression* by using *Number*.
 - **productions:** a finite set P , which allow us to “map” non-terminals to non-terminals/terminals. That is:

$$X \rightarrow \alpha, \alpha \in (\Sigma \cup N)^*, X \in N$$

is a production, which says “ α is an instance of X ”

- **start symbol:** an element $S \in N$, such that any sentence of the grammar can be derived (by applying the **productions** of the grammar) from it

CFG are context-free because a production allow us to always derive a terminal, independently of any context. CFGs allow us to nest phrase structures as much as possible (see below)

Terminals: $+, *, (,), x, y, z, 0, \dots, 9$.

Non-terminals: Exp, Var, Num .

We designate the non-terminal Exp as the **start symbol**.

Rules:

$$\begin{aligned} Exp &\rightarrow Exp + Exp \\ Exp &\rightarrow Exp * Exp \\ Exp &\rightarrow Var \mid Num \mid (Exp) \\ Var &\rightarrow x \mid y \mid z \\ Num &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \\ &\quad 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Figure 8: A CFG for arithmetic expressions. Since $S = Exp$, this means that any sentence that we derive from the grammar can ultimately be expressed as an expression. We can derive sentences such as “ $0+(2*5)+y$ ”

- **Sentential Forms:** in essence the sentences that we can derive from the grammar. They are any sequence of terminals and non-terminals that can be obtained from applying the rules of the grammar. For example:

$$5 * (x + Exp)$$

is a sentential form of the above grammar

- **Sentences:** a sentential form consisting only of terminals
- **Set of Sentential Forms Derivable from a Grammar:** a set $\mathcal{S}(\mathcal{G})$ is the smallest subset of $(N \cup \Sigma)^*$ such that:

- the start symbol is in $\mathcal{S}(\mathcal{G})$:

$$S \in \mathcal{S}(\mathcal{G})$$

- if a sentential form containing a non-terminal is in $\mathcal{S}(\mathcal{G})$, then any sentence derived from it via a production must also be in $\mathcal{S}(\mathcal{G})$:

$$\alpha X \beta \in \mathcal{S}(\mathcal{G}), X \rightarrow \gamma \in P \implies \alpha \gamma \beta \in \mathcal{S}(\mathcal{G})$$

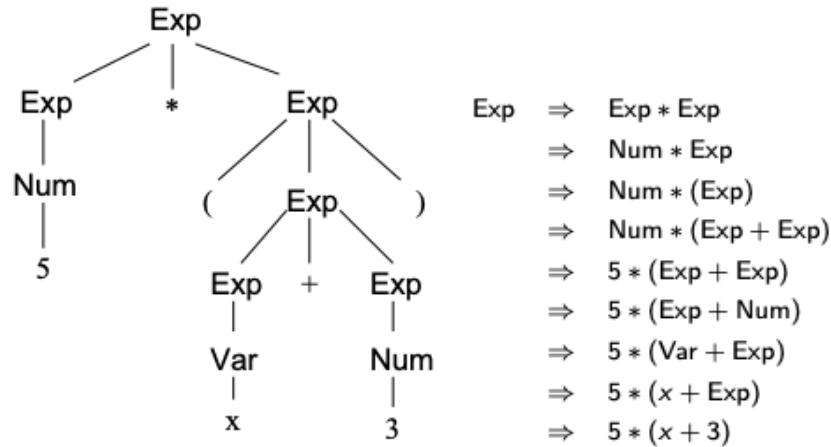
- **Language:** a language associated with a grammar is the set of sentential forms containing only terminals (aka the set of all sentences derivable from the grammar). Symbolically,

$$L(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$$

- **Context-Free Language:** a language which can be derived from a CFG

6.2 Syntax Trees

- **Syntax Trees:** allow us to break down a sentence into its atoms, in order to see if can be derived from a grammar. In essence, the grammar of language defines all the syntax trees that are possible



The above tell us that the sentence $5*(x+3)$ is a derivable expression from the grammar given above. Notice that the start symbol is what is used as part of the root of the tree. A sentence can also be derived by considering the sentential forms that can be derived from the start symbol.

- **Structural Ambiguity:** do to the structure of some grammars, some strings may be generate by more than one syntax tree (for example, $1+2+3$ can be thought of as $1 + (2 + 3)$ or $(1 + 2) + 3$). Because of this, we can make 2 types of derivations: left and right derivations

- **left derivation:** read string from left to right. For example, for $x + x * x$

$$\begin{aligned}
 Exp &\Rightarrow Exp + Exp \\
 &\Rightarrow Var + Exp \\
 &\Rightarrow x + Exp \\
 &\Rightarrow x + Exp * Exp \\
 &\Rightarrow x + Var * Exp \\
 &\Rightarrow x + Var * Var \\
 &\Rightarrow x + x * Exp \\
 &\Rightarrow x + x * x
 \end{aligned}$$

- **right derivation:** read string from right to left. For example, for

$x + x * x$

$$\begin{aligned}
 Exp &\Rightarrow Exp * Exp \\
 &\Rightarrow Exp * Var \\
 &\Rightarrow Exp * x \\
 &\Rightarrow Exp + Exp * x \\
 &\Rightarrow Exp + Var * x \\
 &\Rightarrow Var + Var * x \\
 &\Rightarrow Var + x * x \\
 &\Rightarrow x + x * x
 \end{aligned}$$

7 Week 3 - The CYK Algorithm

7.1 Chomsky Normal Form

- **The Parsing Problem:** going from a string of terminals to a syntax tree
- **Chomsky Normal Form:** if a grammar is CNF, then there is an algorithm (CYK) which allows us to parse any sentence.
 - any CFG can be turned in to CNF, so CYK can help parse any CFG
- **Structure of CNF:** CNF specifies what the right side of a production must be. Only allow RHS to have at most 2 symbols:
 - either 2 non terminals

$$X \rightarrow YZ$$

- or 1 terminal

$$X \rightarrow \alpha$$

Terminals: book, orange, heavy, my, very
Non-terminals: NP, Nom, AP, A, Det, Adv
Start symbol: NP

NP \rightarrow Det Nom
 Nom \rightarrow book | orange | AP Nom
 AP \rightarrow heavy | orange | Adv A
 A \rightarrow heavy | orange
 Det \rightarrow my
 Adv \rightarrow very

Figure 9: Notice that in making a grammar CNF, some redundancies might arise. For example, we could simply define $AP \rightarrow A|AdvA$, but this wouldn't be allowed by CNF.

7.2 The CYK Algorithm

- **CYK Algorithm:** a dynamic programming algorithm used to recognise whether a sentence can be derived from a grammar
- **Recurrence in CYK:** given a string, we can split it into substrings, by including markers between words:

“my very heavy orange book \rightarrow “0 my 1 very 2 heavy 3 orange 4 book 5”

For example $(0, 1)$ represents the substring “my”, and $(2, 5)$ represents the substring “heavy orange book”. The recurrence arises by acknowledging that any substring must be derived from others substrings, until we eventually reach terminals

- **The a Array:** we store our results in a table. If a sentence has n words, then a will be an $n \times n$ array. We index the entries by using $0 \leq i \leq n - 1$ and $1 \leq j \leq n$, such that $a[i, j]$ represents the non-terminal that would get mapped to the substring (i, j) . For example $(2, 5)$ (heavy orange book) can be parsed as a *Nom*, so:

$$a[2, 5] = \text{Nom}$$

If there are multiple interpretations, store both.

- notice that we will only populate the upper right part of the table, as we require $i < j$ (otherwise we would be considering null substring)
- entries for which $j = i + 1$ refer to parsings of single words
- entries above the main diagonal mean that we are parsing a substring containing at least 2 words

- **The CYK Algorithm:**

```

CYK (s,G):           # s=input string, G=CNF grammar
  n = length(s)
  allocate table[0,...,n-1][1,...,n]
  for j = 1 to n        # columns
    for (X → t) ∈ G
      if t = s[j-1]
        add X to table[j-1,j]  # diagonal cell
    for i = j-2 downto 0    # rows
      for k = i+1 to j-1    # possible splits
        for (X → YZ) ∈ G
          if Y ∈ table[i,k] and Z ∈ table[k,j]
            add X to table[i,j]  # non-diagonal cell
  return table

```

Figure 10: Notice the importance of CNF: it allows us to check only 2 cases: productions that end in terminals, and productions that end in 2 non-terminals

- **Lines 1-2:** create table a
- **Lines 3-6:** check, for each word in the sentence, whether there is a production that derives said word. If so, fill in the appropriate diagonal.
- **Lines 6-11:** now, we look at the rows above the diagonal, checking if there are non-terminals that derive the subexpression. For example, if we have “heavy orange book”, we would check to see if there is any non terminal that fits “heavy” + “orange book” or “heavy orange” + “book”. Naturally, these can be seen recursively from the table itself.
- **CYK runtime analysis:** due to the nested for loop, given a grammar of m elements, the runtime will be $\Theta(mn^2)$ ($\Theta(n^3)$ if grammar is fixed). Using CNF allows us to have a relatively low runtime (for example, if we allowed ternary productions, $X \rightarrow ABC$, this would increase the runtime to $\Theta(n^4)$)
- **Example Walkthrough:** want to know if

“my very heavy orange book → “0 my 1 very 2 heavy 3 orange 4 book 5”

is part of the grammar

Terminals: book, orange, heavy, my, very

Non-terminals: NP, Nom, AP, A, Det, Adv

Start symbol: NP

NP \rightarrow Det Nom

Nom \rightarrow book | orange | AP Nom

AP \rightarrow heavy | orange | Adv A

A \rightarrow heavy | orange

Det \rightarrow my

Adv \rightarrow very

Figure 11: See John's walkthrough

We prepare the table:

	j	1	2	3	4	5
i		my	very	heavy	orange	book
0	my	-	-	-	-	-
1	very		-	-	-	-
2	heavy			-	-	-
3	orange				-	-
4	book					-

We will go left to right, down to up, just because it is easier to read and understand

1. **i = 0**

- this is easy to fill in: we just need to consider $(0,1) = \text{"my"}$
- checking the grammar, we see that the only possibility is *Det*

	j	1	2	3	4	5
i		my	very	heavy	orange	book
0	my	Det	-	-	-	-
1	very		-	-	-	-
2	heavy			-	-	-
3	orange				-	-
4	book					-

2. **i = 1**

- we first check $(1,2) = \text{"very"}$. We see that this is a *Adv*
- next, we need to check whether $(0,2) = \text{"my very"}$ can be constructed from any other subsentence. Notice: "my" has been found to be a *Det* (see $(0,1)$), but "very" was found to be a *Adv* (see $(1,2)$). There is no structure in the grammar of the form *Det Adv*; hence, "my very" is not a valid sentence of the grammar

– thus we get:

i	j	1	2	3	4	5
		my	very	heavy	orange	book
0	my	Det		-	-	-
1	very		Adv	-	-	-
2	heavy			-	-	-
3	orange				-	-
4	book					-

3. $i = 2$

- we first check $(2, 3) = \text{“heavy”}$. We see that it can be 2 things: *AP* or *A*. We need to include both.
- next, we check whether $(1, 3) = \text{“very heavy”}$ can be constructed from any other subsentence. Notice: “very” has been found to be a *Adv* (see $(1, 2)$), but “heavy” was found to be a *AP* or *A* (see $(2, 3)$). Thus, we need to check whether *Adv AP* or *Adv A* are valid in our grammar. Indeed, we see that *Adv A* is of type *AP*, so “very heavy” is of type *AP*.
- lastly, we check whether $(0, 3) = \text{“my very heavy”}$ can be constructed from any other subsentence:
 - * consider “my very” + “heavy”. We saw “my very” is not valid in the grammar $((0, 2))$, so this won’t work
 - * consider “my” + “very heavy”. We saw “my” is a *Det* $((0, 1))$, and “very heavy” is *AP*. There is nothing in the grammar of type *Det AP*, so this isn’t valid either.
- thus we get:

i	j	1	2	3	4	5
		my	very	heavy	orange	book
0	my	Det			-	-
1	very		Adv	AP	-	-
2	heavy			A, AP	-	-
3	orange				-	-
4	book					-

4. $i = 3$

- we first check $(3, 4) = \text{“orange”}$. We see that it can be 3 things: *Nom* or *AP* or *A*. We need to include the three.
- next, we check whether $(2, 4) = \text{“heavy orange”}$ can be constructed from any other subsentence. Notice: “heavy” can be *A* or *AP* (see $(2, 3)$), but “orange” was found to be a *Nom* or *AP* or *A* (see $(3, 4)$). Thus, we need to check whether there is any

combination of these which are valid in our grammar. Indeed, we see that the only valid combination is *AP Nom*, of type *Nom*, so “heavy orange” is of type *Nom*

- next, we check whether $(1, 4) = \text{“very heavy orange”}$ can be constructed from any other subsentence:
 - * consider “very” + “heavy orange”. “very” is a *Adv* $((0, 2))$, whilst “heavy orange” is a *Nom*. There is nothing of type *Adv Nom*, so this is a no go
 - * consider “very heavy” + “orange”. “very heavy” is *AP* $((1, 3))$, whilst “orange” can be *Nom*, *AP* or *A*. Again, *AP Nom* works (only such combination), so “very heavy orange” is of type *Nom*
- lastly, we check whether $(0, 4) = \text{“my very heavy orange”}$ can be constructed from any other subsentence
 - * consider “my” + “very heavy orange”. “my” is *Det* $((0, 1))$, whilst “very heavy orange we have found to be *Nom* $((1, 4))$. Indeed, *Det Nom* is *NP* in our grammar, so this is valid
 - * consider “my very” + “heavy orange”. “my very” is not valid $((0, 2))$, so ignore
 - * consider “my very heavy” + “orange”. “my very heavy” is not valid $((0, 3))$, so ignore
- thus we get:

	j	1	2	3	4	5
i		my	very	heavy	orange	book
0	my	Det			NP	-
1	very		Adv	AP	Nom	-
2	heavy			A, AP	Nom	-
3	orange				Nom, AP, A	-
4	book					-

5. $i = 4$

- we first check $(4, 5) = \text{“book”}$, which can only be a *Nom*
- next, we check whether $(3, 5) = \text{“orange book”}$ can be constructed from any other subsentence. Notice: “orange” can be a *Nom* or *AP* or *A* (see $(3, 4)$), but “book” can only be a “*Nom*”. Thus, we need to check whether there is any combination of these which are valid in our grammar. Indeed, we see that the only valid combination is *AP Nom*, of type *Nom*, so “orange book” is of type *Nom*
- next, we check whether $(2, 5) = \text{“heavy orange book”}$ can be constructed from any other subsentence:

- * consider “heavy” + “orange book”. “heavy” is *A* or *AP* $((2, 3))$. “orange book” is *Nom* $((3, 4))$. Again, *AP Nom* is the only valid combination, so it can be of type *Nom*
- * consider “heavy orange” + “book”. “heavy orange” is a *Nom* $((2, 4))$. “book” is a *Nom* $((4, 5))$. Nothing is of type *Nom Nom*, so a no go
- next, we check whether $(1, 5) =$ “very heavy orange book” can be constructed from any other subsentence:
 - * consider “very” + “heavy orange book”. “very” is an *Adv* $((1, 2))$, whilst “heavy orange book” is *Nom* $((2, 5))$. There is nothing of type *Adv Nom*, so no go
 - * consider “very heavy” + “orange book”. “very heavy” is a *AP* $((1, 3))$. “orange book” is *Nom* $((3, 5))$. *AP Nom* is valid in the grammar, so the sentence can be a *Nom*
 - * consider “very heavy orange” + “book”. “very heavy orange” is a *Nom* $((1, 4))$, whilst “book” is a *Nom*. *Nom Nom* is not valid in the grammar so no go
- Thus, “very heavy orange book” is a *Nom*
- lastly, we check whether $(0, 5) =$ “my very heavy orange book” can be constructed from any other subsentence
 - * consider “my” + “very heavy orange book”. “my” is *Det* $((0, 1))$, whilst “very heavy orange book” was have found to be *Nom* $((1, 5))$. Indeed, *Det Nom* is *NP* in our grammar, so this is valid
 - * consider “my very” + “heavy orange book”. “my very” is not valid $((0, 2))$, so ignore
 - * consider “my very heavy” + “orange book”. “my very heavy” is not valid $((0, 3))$, so ignore
 - * consider “my very heavy orange” + “book”. “my very heavy orange” is *NP* $((0, 4))$. “book” is “*Nom*”. Nothing in the grammar is of type *NP Nom*, so a no go
- Thus, “my very heavy orange book” is *NP*
- thus we get:

	j	1	2	3	4	5
i		my	very	heavy	orange	book
0	my	Det			NP	NP
1	very		Adv	AP	Nom	Nom
2	heavy			A, AP	Nom	Nom
3	orange				Nom	Nom
4	book					Nom

Since $a[0, 5]$ is *NP*, which is the start symbol of the grammar, “my very heavy orange book” is valid in the grammar

- **Constructing Syntax Tree from CYK:** we know if a sentence is valid in a grammar, but not how to break it down. To do so, we just need to keep pointers to the substructures that allowed us to determine the value of any entry (i, j) . For example, to determine that $(1, 4) = \text{“very heavy orange”}$ was a *Nom*, we used the fact that $a[1, 2] = \text{“very”}$ is a *Adv*, and $a[2, 4] = \text{“heavy orange”}$ is a *Nom*. Thus, in a separate table b , we could let:

$$b[1, 4] = [(1, 2), (2, 4)]$$

which gives us pointers to the component elements of any subsentence.

i	j	1	2	3	4	5
		my	very	heavy	orange	book
0	my	Det			NP	NP
1	very		Adv	AP	Nom	Nom
2	heavy			A, AP	Nom	Nom
3	orange				Nom, A, AP	Nom
4	book					Nom

Figure 12: Notice, rotating your head 45° allows you to “see” the syntax tree. See John doing this.

By how the CYK executes, it will find **all possible** syntax trees for any given expression.

7.3 Converting to Chomsky Normal Form

- **Converting CFG to CNF:** if \mathcal{G} is a CFG, then we have an algorithm which allows us to develop a CNF equivalent \mathcal{G}'
 - it might be inequivalent only if \mathcal{G} allows the possibility of an empty string, but this can be easily added after the algorithm is done
- **The Algorithm:** we use an example grammar to illustrate the algorithm:

$$S \rightarrow TT \mid [S] \quad T \rightarrow \epsilon \mid (T)$$

where:

- $[,], (,)$ represent terminals
- T, S are non-terminals
- ϵ represents an empty string

1. Identify Non-Terminals Which can Derive the Empty String

- we store these in a set E

- these can be terminals which lead to ϵ directly (as is the case for T , as $T \rightarrow \epsilon$ is a valid production), or indirectly (as if the case of S , as $S \rightarrow TT$ is a valid production, and it can be the case that, if $T = \epsilon$, then $S \rightarrow \epsilon\epsilon = \epsilon$)
- thus, we would have:

$$E = \{S, T\}$$

2. Remove all ϵ Productions

- if there is an element in the RHS of a production, and said element is in E , then we can create a new production which doesn't include this element:

$$X \rightarrow \alpha Y \beta \text{ and } Y \in E \text{ then add } X \rightarrow \alpha \beta$$

- this makes sense, as those elements in E are those that lead to ϵ so they can be “ignored”
- for S :
 - * $S \rightarrow TT$: $T \in E$, so if we remove a T , we get the production:

$$S \rightarrow T$$

It would be non-sensical if both were ϵ , as then we would be adding a production $S \rightarrow \epsilon$ when what we are trying to do is remove ϵ

- * $S \rightarrow [S]$: $S \in E$, so we remove S from RHS, and get the production

$$S \rightarrow []$$

- for T :
 - * $T \rightarrow \epsilon$: this is exactly what we want to eliminate, so just remove this production
 - * $T \rightarrow (T)$: $T \in E$, so just remove the T from RHS to get the production:

$$T \rightarrow ()$$

Thus, our grammar now becomes:

$$S \rightarrow TT \mid T \mid [S] \mid [] \quad T \rightarrow (T) \mid ()$$

3. Remove Unit Production

- if $Y \rightarrow \alpha$, and $X \rightarrow Y$, then the production $Y \rightarrow \alpha$ is redundant, and can be included directly as part of X
- after removing these unit productions, all the RHS of the productions will either be a single terminal, or a bunch of symbols
- the only unit production is $S \rightarrow T$. Adding its effects to the RHS of S , the new grammar is:

$$S \rightarrow TT \mid (T) \mid () \mid [S] \mid [] \quad T \rightarrow (T) \mid ()$$

4. Remove Terminals

- we can create new productions/non-terminals whose only purpose is to produce terminals
- this makes it so that the RHS of productions only contain either non-terminals or terminals:

$$S \rightarrow TT \mid Z(TZ) \mid Z(Z) \mid Z[SZ] \mid Z[Z] \quad T \rightarrow Z(TZ) \mid Z(Z)$$

and we have added:

$$Z_{(} \rightarrow (\quad Z_{)} \rightarrow) \quad Z_{[} \rightarrow [\quad Z_{]} \rightarrow]$$

5. Replace Productions With More Than 3 Non-Terminals in RHS

- the last step is to ensure that every RHS of non-terminals only contain 2 non-terminals
- this can be done by “grouping” the non-terminals. If $X \rightarrow Y_1 Y_2 \cdots Y_n$, then create new non-terminals W_2, \dots, W_{n-1} , such that:

$$X \rightarrow Y_1 W_2, \dots, W_{n-1} \rightarrow Y_{n-1} Y_n$$

- for example, for the production $S \rightarrow Z[SZ] \mid Z(TZ)$, then we can define:

$$W \rightarrow TZ) \quad V \rightarrow TZ]$$

such that $S \rightarrow Z[SZ] \mid Z(TZ)$ becomes:

$$S \rightarrow Z[V \mid Z[W$$

• Worked Example:

S	→	NP VP	PP	→	Pre NP
S	→	I VP PP	V	→	ate
NP	→	Det N	Det	→	the a
VP	→	ate NP	N	→	fork salad
VP	→	V	Pre	→	with

We now convert the above CFG to a CNF. Since there are no ϵ , we don't have a set E , so we can omit steps 1 and 2

– Remove Unit Production

There is only 1 unit production, namely $VP \rightarrow V$. Since $V \rightarrow ate$, we introduce:

$$VP \rightarrow ate$$

– **Remove Terminals**

We need to create non-terminals for *I*, *ate*, *the*, *a*, *fork*, *salad*, *with*, which we do by capitalising their first letter:

$$I \rightarrow I \quad Ate \rightarrow ate \quad The \rightarrow the$$

$$A \rightarrow a \quad Fork \rightarrow fork \quad Salad \rightarrow salad \quad With \rightarrow with$$

We then introduce these new non-terminals into the RHS with more than 1 term:

$$S \rightarrow I VP PP \quad VP \rightarrow Ate NP$$

– **Replace Productions With too Many Non-Terminals**

We can see that *S* has 3 terms, so introduce a production :

$$X \rightarrow VP PP$$

such that:

$$S \rightarrow I X$$

Then, after removing unused values (such as *V*, and any non-terminal production for terminals), we get:

S	\rightarrow	NP VP	I	\rightarrow	<i>I</i>
PP	\rightarrow	Pre NP	Ate	\rightarrow	<i>ate</i>
S	\rightarrow	I X	Det	\rightarrow	<i>the</i> <i>a</i>
X	\rightarrow	VP PP	N	\rightarrow	<i>fork</i> <i>salad</i>
NP	\rightarrow	Det N	Pre	\rightarrow	<i>with</i>
VP	\rightarrow	Ate NP			
VP	\rightarrow	<i>ate</i>			

After this transformation we then need to transfer back to the toriginal CFG.

8 Week 4 - The LL(1) Algorithm and Predictive Parsing

8.1 Predictive Parsing

- **Token:** a character from an input string
- **Predictive Parsing:** a $\Theta(n)$ method for parsing a string of length n . Given a CFG, and a token from a string, it tries to derive a non-terminal that should be expanded in the next step of the parsing.

```

stmt    → if-stmt | while-stmt | begin-stmt | assg-stmt
if-stmt → if bool-expr then stmt else stmt
while-stmt → while bool-expr do stmt
begin-stmt → begin stmt-list end
stmt-list → stmt | stmt ; stmt-list
assg-stmt → var := arith-expr
bool-expr → arith-expr compare-op arith-expr
compare-op → < | > | <= | >= | == | !=

```

Figure 13: For the above grammar, if we get a token **begin** and we know that the start symbol is **stmt**, then we predict that we will be given a *begin-stmt*. Thus, we expect that the sentence we have been given is of the form **begin stmt-list end**. Thus, our non-terminal to expand will now be *stmt-list end*. Here we are being lax: what we are predicting is the production that we will use next. For example, we know that *stmt* \rightarrow *begin - stmt*, and given the **begin**, we know that *stmt* \rightarrow *beginstmt - listend*

8.2 LL(1)

- **LL(1) Grammar:** a grammar, such that given a current token, and a non-terminal to be expanded, we can **always** predict the next production that we must use
 - LL(1) stands for: build input from left, build leftmost derivation and look only **1** token ahead
 - the above grammar is not LL(1), as there is still some ambiguity. For example, *stmt-list* can have production *stmt - list* \rightarrow *stmt* or *stmt - list* \rightarrow *stmt ; stmt - list* no matter what token we read next.
 - this can be fixed by:

$$stmt - list \rightarrow stmt \ stmt - tail$$

$$stmt - tail \rightarrow \epsilon \mid ; stmt \ stmt - tail$$

which eliminates all redundancy

From the **if**, now see that the next two rules must be:

```
stmt-list → stmt stmt-tail
stmt      → if-stmt
if-stmt   → if bool-expr then stmt else stmt
```

The whole derivation so far:

```
stmt → begin-stmt
      → begin stmt-list end
      → begin stmt stmt-tail end
      → begin if-stmt stmt-tail end
      → begin if bool-expr then stmt else stmt stmt-tail end
```

Figure 14: With the LL(1) grammar, if the next token is an **if**, this is the derivation we can make

8.2.1 Parse Table

- **Parse Tables:** given a token and a non-terminal, allow us to determine which production to apply next
 - if a grammar is not LL(1), then such parse tables can't be built
 - if a sentence is not part of the grammar, then parsing via parse tables will fail
- **Example Grammar and Parse Table:** consider the grammar:

$$S \rightarrow \epsilon \mid TS \quad T \rightarrow (S)$$

This has parse table:

	()	\$
S	$S \rightarrow TS$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
T	$T \rightarrow (S)$		

Figure 15: The blank entries indicate situations that should never arise with legal input; if they arise, the sentence is invalid in the grammar

For example, if we get a token “(”, and our non-terminal symbol to expand is S , then the appropriate production for the grammar will be $S \rightarrow TS$; that is, we expect TS after the “(”.

- **End of Input Marker:** the \$ symbol allows us to know when the string ends

8.2.2 LL(1) Parsing

For parsing, we use a table with 3 columns:

- **Operation:** whether we use a token and a non-terminal to find the predicted production (from the parse table), or whether the token matches with what is predicted
- **Remaining Input:** the sentence to parse. Tokens get removed if there is a match with the predicted sentence. For example, if our sentence is *abc*, and we predict something of the form *aX*, we would seek to parse **bc**
- **Stack State:** used to store the predicted form for the remaining input. As we parse tokens, non-terminals get added (lookup) or removed (match)

This seems abstract, so we work through an example (see John’s walk-through). We consider the string *(())*, with start state *S*

Operation	Remaining Input	Stack State
-	(())\$	S

We have token “(” and non-terminal state *S*. We do a *Lookup* on the table, and we predict a production $S \rightarrow TS$. Thus, we add this information to the table:

Operation	Remaining Input	Stack State
-	(())\$	S
Lookup (, S	(())\$	TS

Table 1: Notice that we have changed the *S* by *TS*, as per the production. The input doesn’t change, as we haven’t been able to match.

Again, token “(”, but non-terminal *T*, which yields production $T \rightarrow (S)$:

Operation	Remaining Input	Stack State
-	(())\$	S
Lookup (, S	(())\$	TS
Lookup (, T	(())\$	(S)S

Now, we get token “(”, and a matching terminal from the stack “(”, so we can change the remaining input entry:

Operation	Remaining Input	Stack State
-	(())\$	S
Lookup (, S	(())\$	TS
Lookup (, T	(())\$	(S)S
Match (()\$	S)S

We continue as above:

Operation	Remaining Input	Stack State
-	(())\$	S
Lookup (, S	(())\$	TS
Lookup (, T	(())\$	(S)S
Match (()\$	S)S
Lookup (,S	()\$	TS)S
Lookup (,T	()\$	(S)S)S
Match ()\$	S)S)S
Lookup),S)\$)S)S
Match))\$	S)S
Lookup), S)\$)S
Match)	\$	S
Lookup \$, S	\$	-

Thus, since we reach an empty stack, $()$ is part of the grammar!

Notice, failure can arise:

- if the stack empties, but there is still input left (excluding \$)
- token and terminal can't be matched (i.e “(” is token, but we were expecting “)”)

8.2.3 The LL(1) Algorithm

```
LL1_Parse (table,S,input)
  pos = 0
  initialize stack with single entry S
  while stack not empty
    x = stack.peak()
    if x is non-terminal    # Lookup case
      case table[x,input[pos]] of
        blank: error
        rule  $x \rightarrow \beta$ :
          stack.pop()
          push symbols of  $\beta$  onto stack
            (backwards!)
      else                  # Match case
        if x = input[pos]
          stack.pop()
          pos += 1
        else error
  if input[pos] = $
    return Success
  else error
```

IANS Lecture

There is also an algorithm for building parse tables, but this is not covered in this course.

8.2.4 Further Remarks on LL(1)

- a top-down parser (build syntax tree from root)
 - CYK is bottom up (build syntax tree from its leaves aka terminals)
- LL(1) parser runs in $\Theta(n)$ for LL(1) grammar
- used for example for command languages, where CYK is impractical (programs can be very long, so $\Theta(n^3)$ not appropriate)
- for large scale languages use LR(1) (more flexible but complex than LL(1))
- in real world, parsers are built by defining a CFG, and then a parser generator is used

9 Week 5 - P and NP

9.1 Decision Problems

- **Polynomial Time Algorithms:** consider a problem (for example, finding the all pairs shortest paths of graphs). Let \mathcal{I} be a particular *instance* of the problem (for example, a graph of 5 nodes labelled 1,2,3,4,5, with a specific set of edges). An algorithm A solves the problem in *polynomial time*, if for any instance I , the algorithm runs in time at most:

$$\mathcal{O}(|\mathcal{I}|^r), \quad r \in \mathbb{R}$$

and provides a correct solution for said \mathcal{I}

- **Decision Problem:** a computational problem, which can be posed as a “yes-or-no question”, and whose output is a single boolean value
 - more formally, a problem Q defined in terms of set of possible solutions S to the problem
 - if an instance \mathcal{I} can be solved by a solution in S , then we say:

$$Q(\mathcal{I}) = 1$$

Otherwise, if no solution solves \mathcal{I} , then:

$$Q(\mathcal{I}) = 0$$

- decision problems can be described as *languages*, such that if L_Q is a language, an instance \mathcal{I} of Q is in L_Q if and only if \mathcal{I} has a solution:

$$\mathcal{I} \in L_Q \iff Q(\mathcal{I}) = 1$$

9.2 Complexity Classes: P and NP

- **Complexity Class:** a set of computational problems of related resource-based complexity (for example, time, or memory usage)
- **The Complexity Class P:** set of decision problems which can be solved with a polynomial time algorithm
 - informally may contain non-decision problems, such as sorting or edit distance
 - basically, problems for which there is a known efficient (polynomial) way of finding a solution to **any** instance of the problem
- **Verifier Algorithm:** an algorithm A , such that, if Q is a decision problem, and S is its set of solutions, A is a verifier for Q if and only if, for all instances \mathcal{I} of Q :

$$Q(\mathcal{I}) = 1 \iff \exists y \in S : A(\mathcal{I}, y) = 1$$

- if we are given a “guess solution” y , then A is a verifier if it can check whether y is a solution to a particular instance of a problem
- if the guess y turns out to be a solution, it is called a *certificate*
- **The Complexity Class NP:** set of decision problems for which there is a verifier which runs in polynomial time for any instance \mathcal{I} of the problem
 - basically, problems for which we can check whether something is a solution in polynomial time
 - every P problem is also NP , as we can always check whether some input is a valid in polynomial time (for example, solving the problem and comparing obtained solution with given solution)
- **Euler vs Hamilton; P vs NP:**
 - **Euler Tour:** is there a cycle in the graph which traverses every **edge exactly once**?
 - * this is a problem in P , as given any graph, we can apply DFS to check that the graph is connected, which runs in $\Theta(m + n)$ time
 - * Euler proved that a connected graph vertices of even degree have an Euler Tour
 - * we can check for evenness in polynomial time ($\mathcal{O}(m + n)$)
 - * thus, verifying if an Euler Tour exists can be done in polynomial time
 - **Hamilton Cycle:** is there a cycle in the graph which traverses every **edge exactly once**?
 - * this is a problem in NP , and in fact is *NP-Complete*
 - * guessing a solution would involve checking all $n!$ possible solutions; however, if we are given a solution, it is easy to check whether a given node is traversed less/more than once

9.3 Reductions and NP-Completeness

- **Reductions Between Decision Problems:** a problem A is reducible/reduces to B if there is a polynomial time computable function f which, for any instance \mathcal{I} of A :

$$A(\mathcal{I}) = 1 \iff B(f(\mathcal{I})) = 1$$

- we can “transform” any problem instance of A into an equivalent problem instance of B in polynomial time
- thus, solving A is no harder (in terms of runtime) than solving B
- thus, solving A is at most as hard (in terms of runtime) as solving B
- alternatively, solving B is at least as hard as solving A

- if A reduces to B , then:

$$A \leq_P B$$

Here the P subscript indicates that the reduction is done in polynomial time

- **NP-Complete Problems:** a decision problem Q such that:

1. Q is in NP
2. any other problem H in NP can be reduced to Q :

$$H \in NP \implies H \leq_P Q$$

In essence, if Q is *NP-Complete*, no other NP problem is harder than Q

- any problem A which satisfies condition 2 (aka any NP problem reduces to A) is said to be *NP-Hard*
- thus, an *NP-Complete* problem is any problem which is in NP , and is also *NP-Hard*

- **Consequences of Decision Problem Reductions:** assume we have 2 problems, such that $A \leq_P B$. Then:

- if B is in P , clearly A is no harder than B , so A must also be in P
 - * this follows from the fact that, given any problem instance \mathcal{I} of A , we can do polynomial work to turn it into $f(\mathcal{I})$, which will then be solved in B in polynomial time. This produces an answer in polynomial time
 - * thus, we can answer \mathcal{I} in polynomial time, so A must be in P
- if A is in *NP-Complete*, then clearly B is at least as hard as A . But A being *NP-Complete* means that it is the hardest possible problem, so B must also be *NP-Complete*
 - * consider any NP problem C . Since A is *NP-Complete*, we can find a function g such that C reduces to A :

$$C \leq_P A$$

- * since $A \leq_P B$, then there is a function f which reduces A to B
- * but then this means that the function $f \circ g$ reduces C to B ; in other words:

$$C \leq_P B$$

for any NP problem C

- * so by definition, since $f \circ g$ will be polynomial, B must be *NP-Complete*

9.3.1 SAT Problem

- **Satisfiability:** given a CNF (Conjunctive Normal Form), the problem of satisfiability is that of finding an assignment of literals which satisfies the CNF (makes it evaluate to true)

- a CNF is a conjunction of disjunctions of literals:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

where:

$$C_i = x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,k}$$

- if a CNF has n literals, then there are 2^n possible value assignments
- a CNF is true if at least one literal in a clause (disjunction) is true
- literals can be positive (x) or negative (\bar{x})
- **Cook-Levin Theorem:** showed that the SAT problem is *NP-Complete*
 - this had huge implications: it means that SAT must reduce to any other *NP-Complete* problem
 - in other words, if we can find a polynomial algorithm which turns any instance of SAT into any instance of some other problem, we can show that the other problem must be *NP-Complete*
- **Proving NP-Completeness:** if H is an *NP* problem, H can be shown to be *NP-Complete* if we can construct a CNF, such that whenever the CNF is satisfiable, there is a certificate for some instance of H
 - this is like turning the SAT verifier into a verifier for H via a polynomial function

9.3.2 Independent Sets

- **Independent Sets:** an independent set of a undirected graph is a set of vertices which share no edges
- **Decision Problem: Maximum Independent Set:** does a graph have an independent set of size at least t

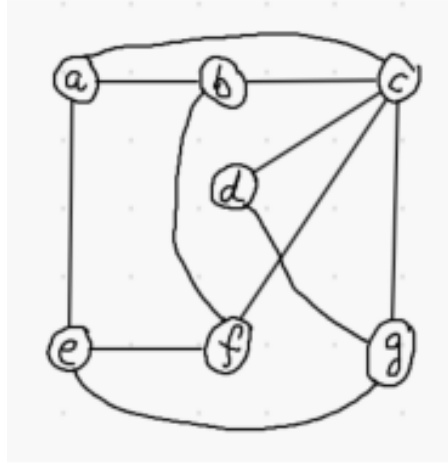
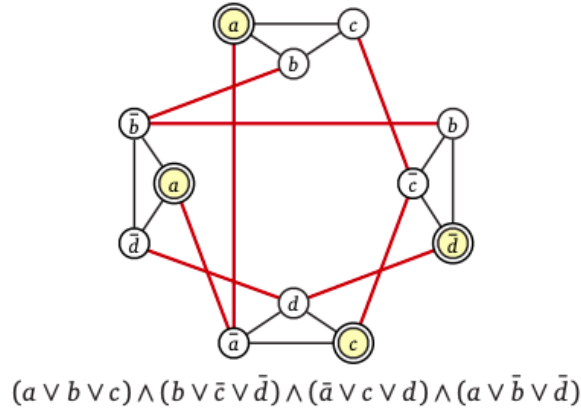


Figure 16: For the above graph, it is easy to see that the size of the max independent set is 3. One such example is $\{b, e, d\}$

- **3-SAT:** satisfiability problem for a 3-CNF. It is *NP-Complete*
 - a 3-CNF is a CNF in which each clause has exactly 3 literals
 - to show *NP-Complete*, we can show that **SAT** reduces to **3-SAT** (see here). The idea is that we can pad clauses with less than 3 literals, and combine literals in clauses with more than 3 literals.
- **Maximum Independent Set is NP-Complete:**
 1. **Maximum Independent Set is NP**
This is easy to see. If we are given a set of vertices S with n elements:
 - $\mathcal{O}(n^2)$ to check that no edge exists between each vertex (2 for loops)
 - verify that $n \geq t$ ($\mathcal{O}(n)$ time)
Thus, verifying a solution can be carried out in polynomial time, so the problem is NP
 2. **Maximum Independent Set is NP-Hard**
 - we show that **3-SAT** reduces to **MIS**. Since **3-SAT** is *NP-Complete*, it is *NP-Hard*, so if it reduces to **MIS**, then **MIS** is also *NP-Hard*
 - * technically, since **3-SAT** is *NP-Complete*, by work above, if it reduces to **MIS**, then **MIS** is *NP-Complete*, but this is more “rigorous”
 - to do this, we need 2 things: find a polynomial algorithm to turn any **3-SAT** problem into a **MIS** problem; show that the **3-SAT** problem is satisfiable if and only if the independent set of the resulting problem has at least m elements, where m is the number of literals in the **3-SAT**

– **3-SAT to MIS**

- * we give a procedure to convert any 3-CNF into a graph
- * let Φ be a CNF with k clauses
- * construct the graph representing literals as nodes, and connecting any 2 nodes if and only if: they represent literals in the same clause, or they represent complementary literals:



- * the resulting graph will have $3k$ nodes
- * this procedure runs in polynomial time. Let m be the number of literals in the 3-CNF (so $m = 3k$)
 - connecting nodes if in the same clause: $\mathcal{O}(m)$ (just iterate over each literal, and make connections when we finish reading clause)
 - connecting nodes if they represent complementary literals: $\mathcal{O}(m^2)$ (naively iterate over each literal using nested for loop, and produce edge where necessary)

Thus, constructing the graph can be done in polynomial time

– **Satisfiability in 3-SAT if and only if Maximum Independent Set of Cardinality at least m in MIS**

- * suppose Φ is satisfiable, and take any such satisfying assignment of literals. According to this assignment, at least one literal in every clause of Φ is **T**. Create a set S to contain the vertices corresponding to these true literals, but choosing only one from each clause. Thus, our set S will have m vertices.
 - no 2 vertices in S are joined by a triangle edge, since we are only taking 1 vertex from each triangle (aka 1 literal from each clause)
 - no 2 vertices in S are joined by a complementary edge, since if x was chosen as part of the satisfying literals, $x =$

T , so $\bar{x} = F$ and thus would never be chosen as a vertex to be added to S . Similarly, if $\bar{x} = T$, then $x = F$, and x wouldn't have been added to S

Thus, all vertices are independent. Moreover, S is of maximum size, as you can at most select 1 vertex from each triangle for an independent set. Thus, if Φ is satisfiable, the maximum independent set is of size m

- * suppose the graph has an independent set S of size m . Assign **T** to every literal corresponding to the vertices in S (this will be consistent, as S is an independent set, so complementary nodes won't be part of S). If a literal (or its complement) are not part of S , assign a random value. Then, since S is a maximum independent set, it must contain a vertex corresponding to a literal in every clause of a 3-CNF. By our assignment, each clause of the 3-CNF will contain at least 1 true literal, so the 3-CNF will be satisfied.

Thus, we have shown that:

$$\text{Independent Set of Size } m \iff \text{3-CNF is Satisfiable}$$

In particular, this means that we have an independent set of size at least m if and only if a CNF of m clauses is satisfiable, as required, so MIS is *NP-Complete*

9.4 Additional Resources

- How to Prove That a Problem Is NP-Complete?
- Mary explains how to derive NP-Completeness for MIS
- Short website on problem reduction
- A very funny book, giving more details on P, NP, proofs, etc ...

10 Week 6 - Approximation Algorithms for NP-Complete

10.1 Approximation Algorithms

- **Solving NP-Completeness:** we don't expect polynomial time algorithm for solving *NP-Complete* problems. Instead:
 - heuristics
 - algorithm for approximate solutions
 - brute-force (exponential)
 - recursive backtracking

- **Approximation Algorithm:** let OPT be an optimisation problem, in which an instance \mathcal{I} has optimal solution $OPT(\mathcal{I})$. An algorithm A is an α -approximation algorithm if:

$$A(\mathcal{I}) = \begin{cases} \leq \alpha \times OPT(\mathcal{I}) & OPT \text{ is a minimisation problem} \\ \geq \frac{1}{\alpha} \times OPT(\mathcal{I}) & OPT \text{ is a maximisation problem} \end{cases}$$

- either A gets a larger value than optimal in minimisation problem (approximate cost \$2 optimal cost \$1), or a lower value than optimal in maximisation problem (approximate sell price \$10, optimal sell price \$20)
- in either of the above cases, the algorithm would be 2-approximate
- α is known as an approximation ratio
- α is typically not preserved when we reduce an *NP – Complete* problem to another

10.2 Minimising Vertex Cover

- **Vertex Cover:** given an undirected graph, a vertex cover V' is a set of vertices, such that every edge of the graph has at least one endpoint in V'
- **Minimum Vertex Cover:** problem of finding the vertex cover of minimum cardinality for any graph
 - *NP-Complete* problem

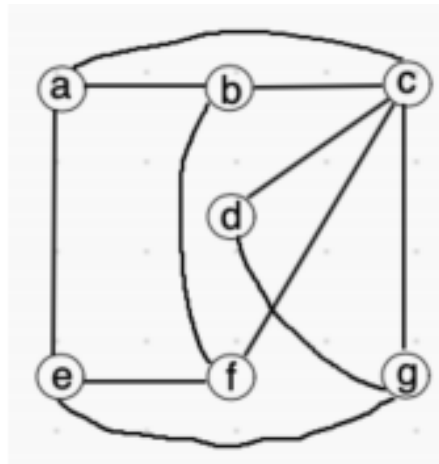


Figure 17: A minimal vertex cover for the above would be $\{c, d, a, f\}$, or $\{c, g, b, e\}$. Either way, the minimum vertex cover will have cardinality 4.

- **The Approximate Vertex Cover Algorithm:** we approximate the vertex cover by:

1. get an edge of the graph
2. add its 2 vertices u, v to the vertex cover set
3. remove any edge which has u or v as an endpoint
4. repeat until there are no edges left

Algorithm Approx-Vertex-Cover($G = (V, E)$)

1. $C \leftarrow \emptyset$
2. $E' \leftarrow E$
3. **while** $E' \neq \emptyset$
4. **do** take any edge $(u, v) \in E'$
5. $C \leftarrow C \cup \{u, v\}$ // add **both** u and v to the cover
6. Remove every edge g with u or v endpoint from E'
7. Print("There is a VC of size ", $|C|$)

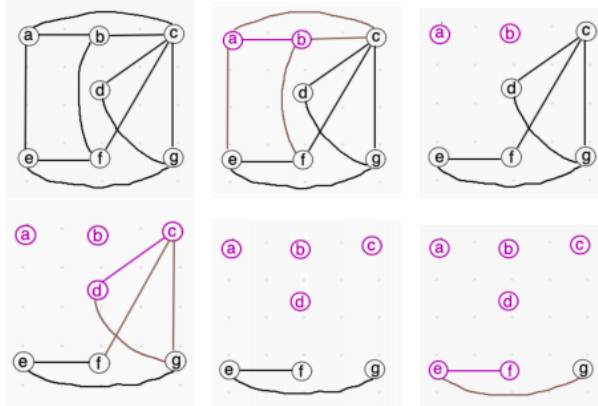


Figure 18: The pink edges are those selected by line 4, with the pink nodes representing the vertices for these edges. The algorithm approximates a vertex cover of cardinality 6.

- **Approximate Vertex Cover: Analysis:** Approx-Vertex-Cover is a 2-approximation
 - let F be the set of vertices selected by line 4
 - * in the above example, these would be the edges $\{(a,b), (c,d), (e,f)\}$
 - notice that these edges will share no endpoints
 - * after an edge is selected, all edges connected to any of the vertices is removed, so any edge that remains in the graph must not have been connected to these vertices
 - if C^* is a minimum vertex set, then C^* must contain at least 1 vertex from the endpoints in F

- * being a vertex cover, every edge must have at least one endpoint in C^*
- * F is just a subset of the edges, so every edge in F must have an endpoint in C^*

so it follows that:

$$|C^*| \geq |F|$$

- the guess for the vertex cover C contains all the vertices from the edges in F , so:

$$|C| = 2|F|$$

- but then

$$|C| = 2|F|, \quad |C^*| \geq |F| \implies |C^*| \geq \frac{|C|}{2} \implies 2|C^*| \geq |C|$$

- in other words, it is a 2-approximation

10.3 Max 3-SAT

- **Max 3-SAT:** given a 3-CNF, find the maximum number of clauses k , such that there is an assignment of binary variables which satisfies these k clauses
 - basically, given a CNF, find an assignment which satisfies the largest number of clauses
 - an *NP-Complete* problem (think of this as a generalised 3-SAT, for which $k = |\Phi|$ for a 3-CNF Φ)
- **A Search Problem:** for a 3-CNF with m clauses, we are **guaranteed** to be able to find an assignment that satisfies $\frac{7}{8}m$ of these clauses. We want to be able to find an algorithm which guarantees this bound
 - in other words, we want to prove that we have an algorithm which is a $\frac{8}{7}$ -approximation
 - if we are trying to maximise the number of clauses, we know we are guaranteed $\frac{7}{8}m$ to be satisfied, but we are more interested in finding the assignment that guarantees these number of clauses
 - we assume that in each clause, each variable is independent

10.3.1 Randomised Max 3-SAT

- **Expected Number of Satisfied Clauses:** consider a 3-CNF:

$$\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

Let Y_j be a random variable, such that $Y_j = 1$ whenever C_j is satisfied. The number of satisfied clauses is:

$$Y = \sum_{j=1}^m Y_j$$

By linearity of expectation, the *expected number of clauses* satisfied by an assignment of variables is:

$$E(Y) = E\left(\sum_{j=1}^m Y_j\right) = \sum_{j=1}^m E(Y_j)$$

But since each Y_j is identically distributed:

$$E(Y) = mE(Y_j), \quad j \in [1, m]$$

- **Expected Value of Satisfying a Clause:** since $E(Y)$ gets reduced to finding the expected value of satisfying a clause, consider a clause:

$$C_j = l_{j,1} \vee l_{j,2} \vee l_{j,3}$$

We can model each variable as a uniform random variable, with $P(l = 1) = 0.5$ (each variable has probability $\frac{1}{2}$ or being 1 or 0). The probability that C_j is satisfied is:

$$P(Y_j = 1) = 1 - P(Y_j = 0) = 1 - \frac{1}{2^3} = \frac{7}{8}$$

Thus,

$$E(Y_j) = P(Y_j = 0) \times 0 + P(Y_j = 1) \times 1 = \frac{7}{8}$$

- **Randomised Max 3-SAT:** from the above, the expected number of satisfied clauses will be:

$$E(Y) = mE(Y_j) = \frac{7}{8}m$$

The key is that, since the variables are uniformly distributed, the expected value is actually the average value over all possible assignment. In other words, over all possible assignments, there must exist at least one assignment which guarantees:

$$E(Y) \geq \frac{7}{8}m$$

- think that if all assignments satisfied less than $\frac{7}{8}m$ clauses, the average number of satisfied clauses could never be $\frac{7}{8}m$

Thus, a randomised algorithm to get an approximation to **Max 3-SAT** would involve generating random assignments to the variables until we achieve at least $\frac{7}{8}m$ satisfied clauses

- this is very naïve, producing many assignments that aren't even close to the approximation

10.3.2 De-Randomised Max 3-SAT

- **De-Randomising Max 3-SAT:** a key observation is that:

$$E(Y) = \frac{E(Y|x_1 = 0)}{2} + \frac{E(Y|x_1 = 1)}{2}$$

since in half of the assignments, $x_1 = 0$, so the expected value of satisfied clauses can be conditioned on x_1 (or any other variable)¹. Roughly, we are using the fact that $E(Y)$ is an average, and we can break it down as the average when $x_1 = 0$ and when $x_1 = 1$.

- since $E(Y) \geq \frac{7}{8}$, this must then mean that either $E(Y|x_1 = 0) \geq \frac{7}{8}$ or $E(Y|x_1 = 1) \geq \frac{7}{8}$
- otherwise, the average of the two could never be greater than $\frac{7}{8}$
- **Deterministic Max 3-SAT:** the above allows us to be deterministic about finding an assignment:
 - compute $E(Y|x_1 = 0)$ and $E(Y|x_1 = 1)$, and then create a partial assignment so that x_1 provides the maximum expectation
 - repeat, but this time by considering different x_2 , and with our previously calculate x_1 . Thus, we compute:

$$E(Y|x_1 = b_1, x_2 = 0)$$

$$E(Y|x_1 = b_1, x_2 = 1)$$

- for any x_i , we will have created a partial assignment b_1, b_2, \dots, b_{i-1} , and we will just have to consider:

$$Exp_0 = E(Y|x_1 = b_1, x_2 = b_2, \dots, x_i = 0)$$

$$Exp_1 = E(Y|x_1 = b_1, x_2 = b_2, \dots, x_i = 1)$$

Algorithm Greedy-3-SAT(ϕ, n, m)

1. **for** $i = 1, \dots, n$
2. Compute $Exp_0 \leftarrow E[Y \mid x_1 = b_1 \dots x_{i-1} = b_{i-1}, x_i = 0]$
3. Compute $Exp_1 \leftarrow E[Y \mid x_1 = b_1 \dots x_{i-1} = b_{i-1}, x_i = 1]$
4. **if** $Exp_0 \geq Exp_1$
5. **then** $b_i \leftarrow 0$; Update ϕ by fixing $x_i = 0$
6. **else** $b_i \leftarrow 1$; Update ϕ by fixing $x_i = 1$
7. **return** \mathbf{b}

Figure 19: By picking variables 1 by 1 in a deterministic manner, we are ensuring that the expected number of satisfied clauses will always be at least $\frac{7}{8}m$

¹Mary explains it more eloquently

- **Analysis of Greedy-3-SAT:** by definition, this will be a $\frac{8}{1}$ -approximation algorithm. Moreover, since we iterate over each of the n literals, and we do $\mathcal{O}(1)$ work at each of the m clauses, the runtime is:

$$\mathcal{O}(mn)$$

– if $P \neq NP$, this is the best possible approximation

- **Walkthrough and Actually Computing the Expectation:** a sort of important part of this algorithm is actually finding Exp_0 and Exp_1 . This is best done via an example, so consider the 3-CNF:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \\ \wedge (x_1 \vee x_2 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

– x_1

If $x_1 = 1$, there are clauses which get automatically satisfied if they contain x_1 . Otherwise, since \bar{x}_1 will never satisfy the clause, we will consider the probability that the remaining variables allow us to satisfy the clause. If x_1 or \bar{x}_1 don't appear, we compute a similar probability. The above CNF goes from:

$$(\textcolor{red}{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\textcolor{red}{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{\textcolor{red}{x}}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{\textcolor{red}{x}}_1 \vee x_2 \vee x_3) \\ \wedge (\textcolor{red}{x}_1 \vee x_2 \vee \bar{x}_4) \wedge (\textcolor{red}{x}_1 \vee \bar{x}_2 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

to:

$$(\bar{x}_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

* the probability of $(\bar{x}_2 \vee x_3)$ being satisfied is:

$$1 - \frac{1}{2^2} = \frac{3}{4}$$

* the probability of $(x_2 \vee x_3)$ being satisfied is:

$$1 - \frac{1}{2^2} = \frac{3}{4}$$

* the probability of $(x_2 \vee \bar{x}_3 \vee \bar{x}_4)$ being satisfied is:

$$1 - \frac{1}{2^3} = \frac{7}{8}$$

* the probability of the clauses containing x_1 of being satisfied is 1 (there are 4 of these)

Thus, and by using linearity of expectation:

$$Exp_1 = E(Y|x_1 = 1) = 4 \times 1 + 2 \times \frac{3}{4} + 1 \times \frac{7}{8} = \frac{51}{8}$$

Similarly $x_1 = 0$, we do the same, but “removing” the clauses which contain \bar{x}_1 , and removing x_1 whenever it appears. The CNF goes from:

$$(\textcolor{blue}{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\textcolor{blue}{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{\textcolor{blue}{x}}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{\textcolor{blue}{x}}_1 \vee x_2 \vee x_3) \\ \wedge (\textcolor{blue}{x}_1 \vee x_2 \vee \bar{x}_4) \wedge (\textcolor{blue}{x}_1 \vee \bar{x}_2 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

to:

$$(\bar{x}_2 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

* for the clauses with 2 literals, the probability of satisfiability is:

$$1 - \frac{1}{2^2} = \frac{3}{4}$$

there are 4 such clauses

* the probability of $(x_2 \vee \bar{x}_3 \vee \bar{x}_4)$ being satisfied is:

$$1 - \frac{1}{2^3} = \frac{7}{8}$$

* the probability of the clauses containing \bar{x}_1 of being satisfied is 1 (there are 2 of these)

Thus, and by using linearity of expectation:

$$Exp_0 = E(Y|x_1 = 0) = 2 \times 1 + 4 \times \frac{3}{4} + 1 \times \frac{7}{8} = \frac{47}{8}$$

Hence, since $Exp_1 > Exp_0$, we let $b_1 = 1$.

– **x_2**

Since now we have an assignment with $x_1 = 1$, we need only consider:

$$(\bar{x}_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

where we have removed those clauses with $x_1 = 1$, and reduced the size of clauses with \bar{x}_1 . (The clauses with x_1 must still be used for the expectation calculation, but we know that there were 4 clauses satisfied, so we can use this for the final expectation calculation.) If

$x_2 = 1$, the CNF goes from:

$$(\bar{\textcolor{red}{x}}_2 \vee x_3) \wedge (\textcolor{red}{x}_2 \vee x_3) \wedge (\textcolor{red}{x}_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

to:

$$(x_3)$$

* there are 4 clauses with $x_1 = 1$ (probability of satisfiability 1)

* the probability of the clauses containing x_2 of being satisfied is 1 (there were 2)

* the probability of (x_3) being satisfied is:

$$\frac{1}{2}$$

Thus, and by using linearity of expectation:

$$Exp_1 = E(Y|x_1 = 1, x_2 = 1) = 4 \times 1 + 2 \times 1 + \frac{1}{2} \times 1 = \frac{13}{2} = 6.5$$

If $x_2 = 0$, the CNF goes from:

$$(\bar{x}_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$$

to:

$$(x_3) \wedge (\bar{x}_3 \vee \bar{x}_4)$$

- * there are 4 clauses with $x_1 = 1$ (probability of satisfiability 1)
- * the probability of the clauses containing \bar{x}_2 of being satisfied is 1 (there was only 1)
- * the probability of (x_3) being satisfied is:

$$\frac{1}{2}$$

- * the probability of $(\bar{x}_3 \vee \bar{x}_4)$ being satisfied is:

$$1 - \frac{1}{2^2} = \frac{3}{4}$$

Thus, and by using linearity of expectation:

$$Exp_0 = E(Y|x_1 = 1, x_2 = 0) = 4 \times 1 + 1 \times 1 + \frac{1}{2} \times 1 + \frac{3}{4} \times 1 = \frac{25}{4} = 6.25$$

Hence, since $Exp_1 > Exp_0$, we let $b_2 = 1$

- **x_3** It is easy to see that, since we have $x_1 = 1, x_2 = 1$, our CNF reduces to:

$$(x_3)$$

since the other 6 clauses are satisfied by the assignment. This CNF is satisfiable with probability 1 if $x_3 = 1$, and satisfiable with probability 0 if $x_3 = 0$. In other words,

$$Exp_1 = E(Y|x_1 = 1, x_2 = 1, x_3 = 1) = 6 \times 1 + 1 \times 1 = 7$$

$$Exp_0 = E(Y|x_1 = 1, x_2 = 1, x_3 = 0) = 6 \times 1 + 1 \times 0 = 6$$

Hence, since $Exp_1 > Exp_0$, we let $b_3 = 1$.

- **x_4**

This is not necessary, as:

$$\{x_1 = 1, x_2 = 1, x_3 = 1\}$$

already satisfies the whole CNF, so we can assign any value to x_4 .

11 Week 7 - Recursive Backtracking for NP-Completeness

11.1 Dealing with NP-Completeness: Recursive Backtracking

- **Brute Force Method:** given an *NP-Complete* problem, we can try all possibilities, until we find a solution
 - this typically requires exponential runtime
 - for example, if we have a CNF, there are 2^n possible assignments for n literals
 - checking the validity of the whole CNF takes time $\mathcal{O}(|\Phi|)$ for each assignment
 - so finding satisfiability of CNF via brute force has runtime:

$$\mathcal{O}(2^n |\Phi|)$$

- **Recursive Backtracking for CNF:** we basically assign values to variables, recursively calling the function. If eventually some assignment leads to an empty clause, we go back and reassign values. The algorithm terminates if the whole CNF evaluates to true with the current assignment (namely every clause disappears as they are satisfied), or if eventually we just reach empty clauses (unsatisfiable)

Algorithm SAT-backtrack($\Phi = C_1 \wedge \dots \wedge C_m, \mathcal{I}, \mathbf{b}$)

1. **if** ($m = 0$) **then return** T
2. **else if** Φ contains an empty clause **then return** F
3. **else** choose an unassigned variable x_i ($i \in [n] \setminus \mathcal{I}$) how?
4. $\Phi' \leftarrow \Phi(x_i \leftarrow 0)$
 (simplifying Φ' based on this new assignment)
5. **if** SAT-backtrack($\Phi', \mathcal{I} \cup \{i\}, \mathbf{b} \cdot 0$)
6. **then return** T
7. **else** $\Phi' \leftarrow \Phi(x_i \leftarrow 1)$
 (simplifying Φ' based on this new assignment)
8. **return** SAT-backtrack($\Phi', \mathcal{I} \cup \{i\}, \mathbf{b} \cdot 1$)

Figure 20: \mathbf{b} is the current partial assignment, given in reference to the instance \mathcal{I} (for every index i in the instance, \mathbf{b} contains the assignment b_i corresponding to variable x_i). If the initial assignment ($x_i \leftarrow 0$) leads to unsatisfiability (Line 5), the next assignment is considered (Line 7). Satisfiability is reached when the number of remaining clauses m is 0.

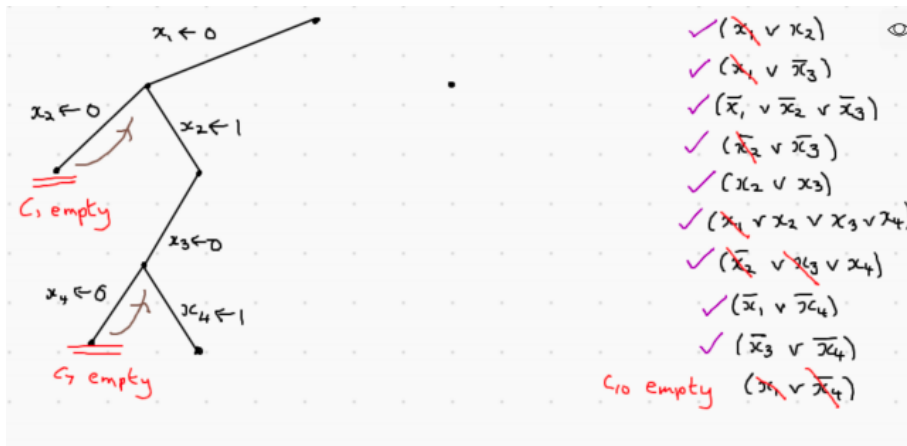
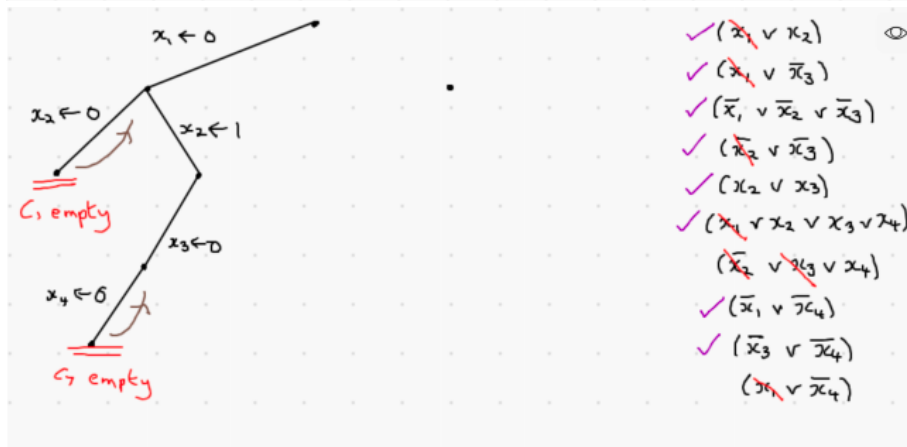
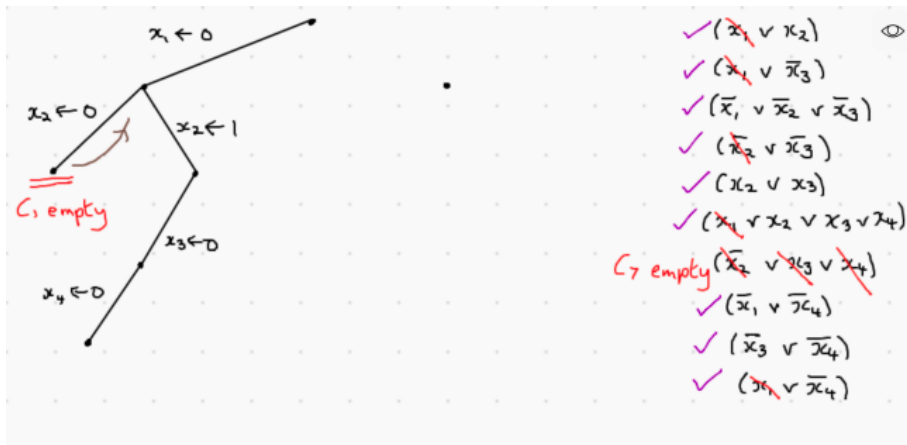
$$\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (x_2 \vee x_3) \\ \wedge (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4) \wedge (\bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_4)$$

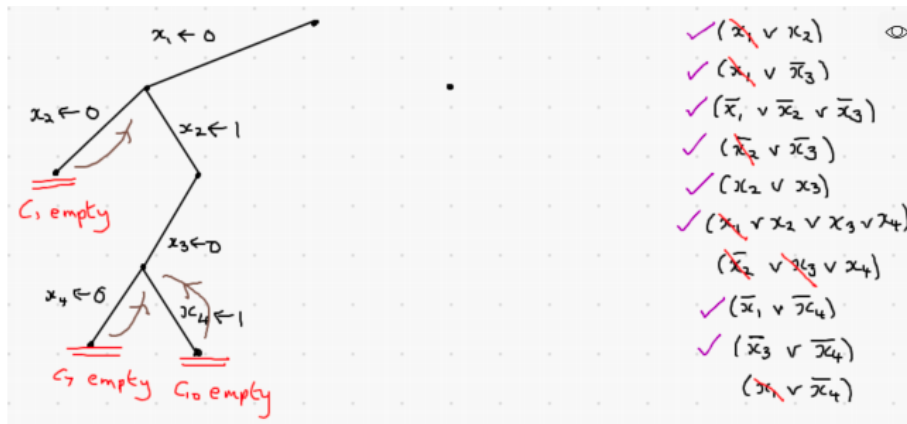
	$(x_1 \vee x_2)$ $(x_1 \vee \bar{x}_3)$ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ $(\bar{x}_2 \vee \bar{x}_3)$ $(x_2 \vee x_3)$ $(x_1 \vee x_2 \vee x_3 \vee x_4)$ $(\bar{x}_2 \vee x_3 \vee x_4)$ $(\bar{x}_1 \vee \bar{x}_4)$ $(\bar{x}_3 \vee \bar{x}_4)$ $(x_1 \vee \bar{x}_4)$
	$(x_1 \vee x_2)$ $(x_1 \vee \bar{x}_3)$ $\checkmark (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ $(\bar{x}_2 \vee \bar{x}_3)$ $(x_2 \vee x_3)$ $(x_1 \vee x_2 \vee x_3 \vee x_4)$ $(\bar{x}_2 \vee x_3 \vee x_4)$ $\checkmark (\bar{x}_1 \vee \bar{x}_4)$ $(\bar{x}_3 \vee \bar{x}_4)$ $(x_1 \vee \bar{x}_4)$
	C_1 empty $(x_1 \vee x_2)$ $(x_1 \vee \bar{x}_3)$ $\checkmark (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ $\checkmark (\bar{x}_2 \vee \bar{x}_3)$ $(x_2 \vee x_3)$ $(x_1 \vee x_2 \vee x_3 \vee x_4)$ $\checkmark (\bar{x}_2 \vee x_3 \vee x_4)$ $\checkmark (\bar{x}_1 \vee \bar{x}_4)$ $(\bar{x}_3 \vee \bar{x}_4)$ $(x_1 \vee \bar{x}_4)$

- ~~$(\bar{x}_1 \vee x_2)$~~ ①
- ~~$(\bar{x}_1 \vee \bar{x}_3)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- $(\bar{x}_2 \vee \bar{x}_3)$
- $(x_2 \vee x_3)$
- ~~$(\bar{x}_1 \vee x_2 \vee x_3 \vee x_4)$~~
- $(\bar{x}_2 \vee x_3 \vee x_4)$
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
- $(\bar{x}_3 \vee \bar{x}_4)$
- ~~$(x_1 \vee \bar{x}_4)$~~

- ✓ ~~$(\bar{x}_1 \vee x_2)$~~ ①
- ~~$(\bar{x}_1 \vee \bar{x}_3)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ~~$(\bar{x}_2 \vee \bar{x}_3)$~~
- ✓ $(x_2 \vee x_3)$
- ✓ ~~$(\bar{x}_1 \vee x_2 \vee x_3 \vee x_4)$~~
- ~~$(\bar{x}_2 \vee x_3 \vee x_4)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
- $(\bar{x}_3 \vee \bar{x}_4)$
- ~~$(x_1 \vee \bar{x}_4)$~~

- ✓ ~~$(\bar{x}_1 \vee x_2)$~~ ①
- ✓ ~~$(\bar{x}_1 \vee \bar{x}_3)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- ✓ ~~$(\bar{x}_2 \vee \bar{x}_3)$~~
- ✓ $(x_2 \vee x_3)$
- ✓ ~~$(\bar{x}_1 \vee x_2 \vee x_3 \vee x_4)$~~
- ~~$(\bar{x}_2 \vee \bar{x}_3 \vee x_4)$~~
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
- ✓ $(\bar{x}_3 \vee \bar{x}_4)$
- ~~$(x_1 \vee \bar{x}_4)$~~

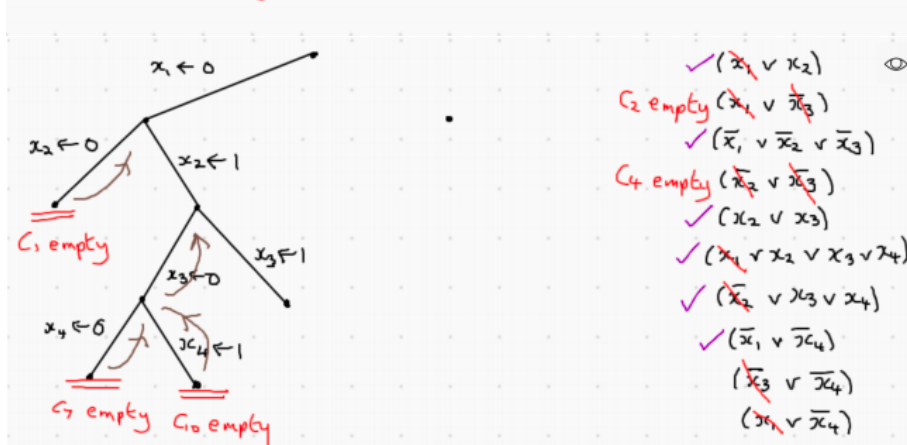




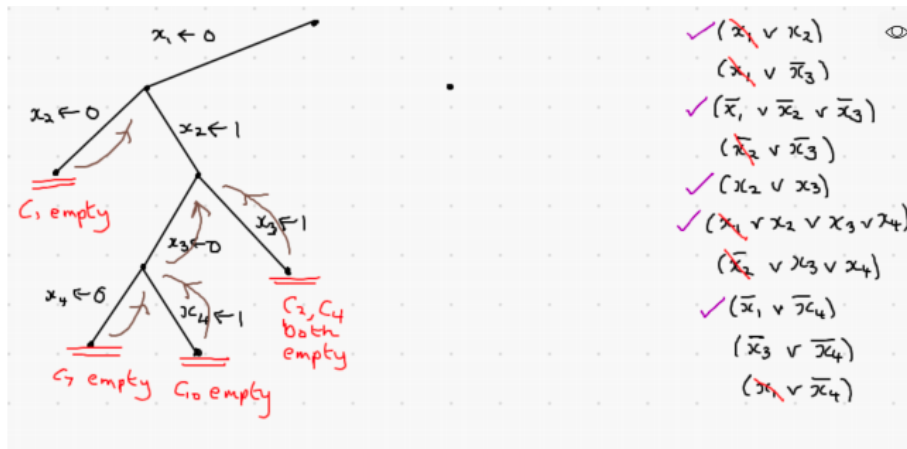
- ✓ (~~x_1~~ \vee x_2)
- ✓ (~~x_1~~ \vee \bar{x}_3)
- ✓ (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)
- ✓ (~~\bar{x}_2~~ \vee \bar{x}_3)
- ✓ (x_2 \vee x_3)
- ✓ (~~x_1~~ \vee x_2 \vee x_3 \vee x_4)
- (~~\bar{x}_2~~ \vee ~~x_3~~ \vee x_4)
- ✓ (\bar{x}_1 \vee \bar{x}_4)
- ✓ (\bar{x}_3 \vee \bar{x}_4)
- (~~x_1~~ \vee \bar{x}_4)



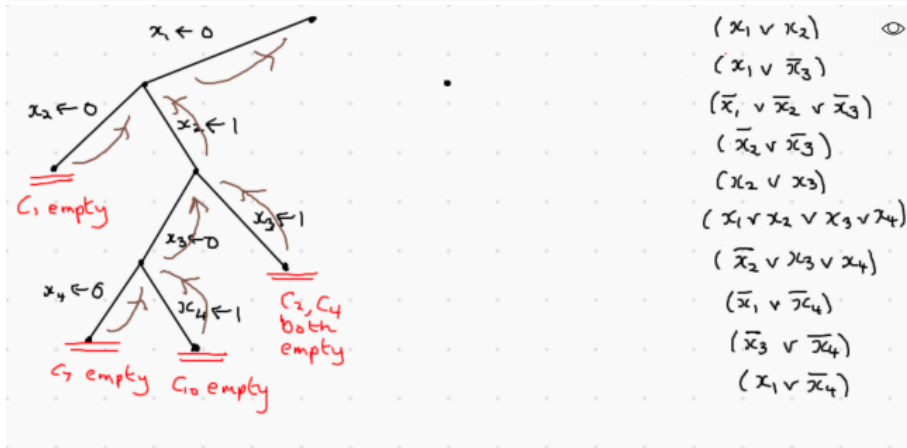
- ✓ (~~x_1~~ \vee x_2)
- (~~x_1~~ \vee \bar{x}_3)
- ✓ (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)
- (~~\bar{x}_2~~ \vee \bar{x}_3)
- ✓ (x_2 \vee x_3)
- ✓ (~~x_1~~ \vee x_2 \vee x_3 \vee x_4)
- (~~\bar{x}_2~~ \vee x_3 \vee x_4)
- ✓ (\bar{x}_1 \vee \bar{x}_4)
- (\bar{x}_3 \vee \bar{x}_4)
- (~~x_1~~ \vee \bar{x}_4)



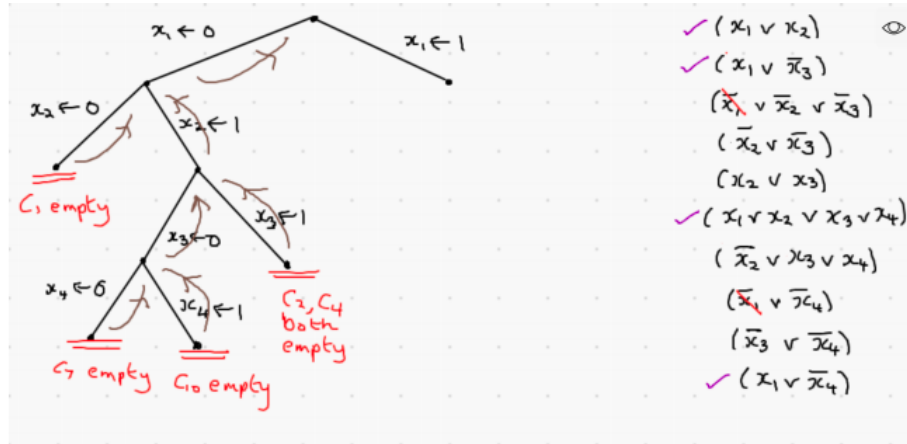
- ✓ (~~x_1~~ \vee x_2)
- C2 empty (~~x_1~~ \vee ~~\bar{x}_3~~)
- ✓ (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)
- C4 empty (~~\bar{x}_2~~ \vee ~~\bar{x}_3~~)
- ✓ (x_2 \vee x_3)
- ✓ (~~x_1~~ \vee x_2 \vee x_3 \vee x_4)
- ✓ (~~\bar{x}_2~~ \vee x_3 \vee x_4)
- ✓ (\bar{x}_1 \vee \bar{x}_4)
- (~~\bar{x}_3~~ \vee \bar{x}_4)
- (~~x_1~~ \vee \bar{x}_4)



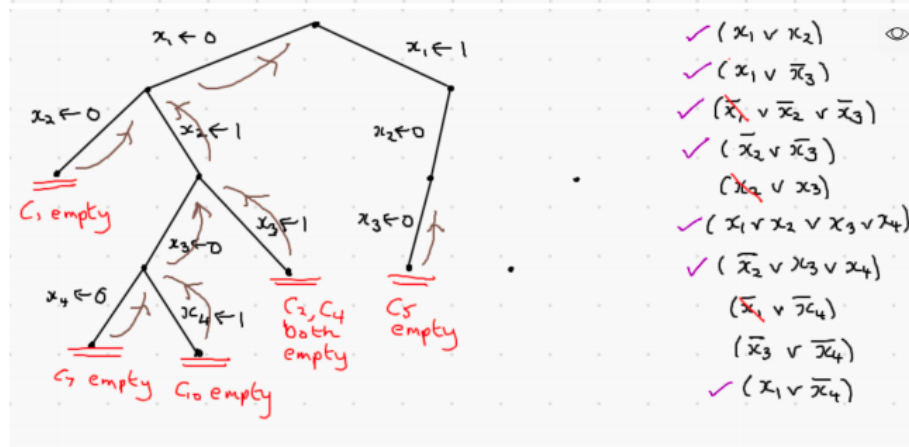
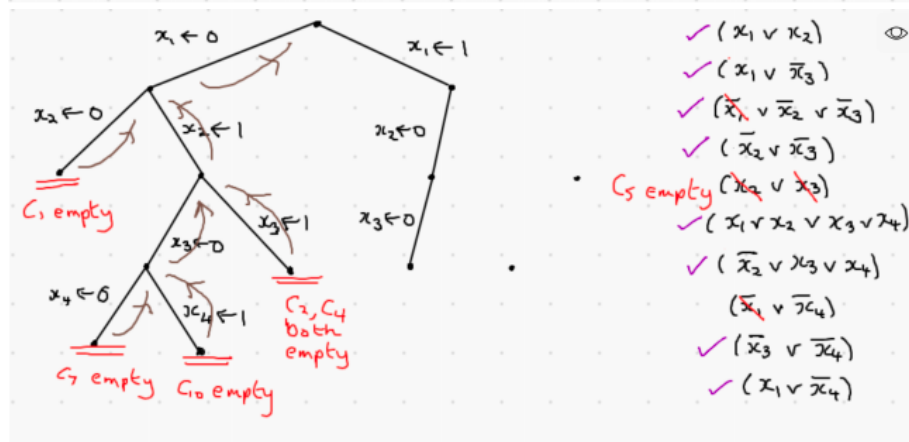
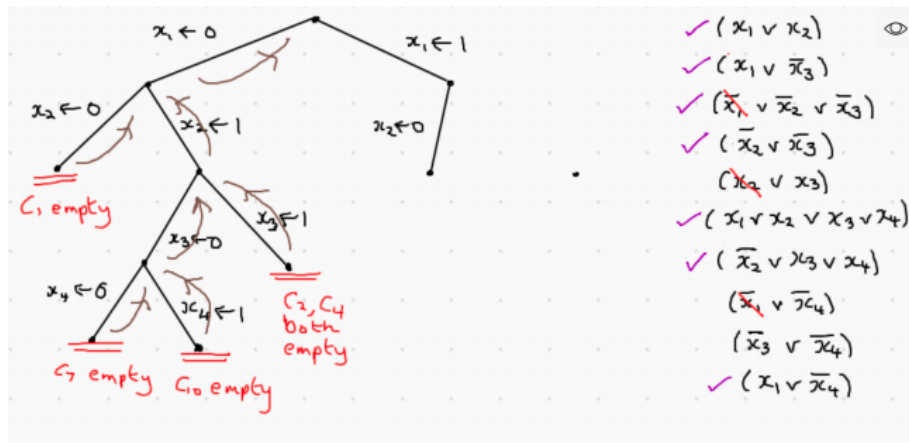
- ✓ $(x_1 \vee x_2)$ \odot
- $(x_1 \vee \bar{x}_3)$
- ✓ $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- $(\bar{x}_2 \vee \bar{x}_3)$
- ✓ $(x_2 \vee x_3)$
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- $(\bar{x}_2 \vee x_3 \vee x_4)$
- ✓ $(\bar{x}_1 \vee \bar{x}_4)$
- $(\bar{x}_3 \vee \bar{x}_4)$
- $(x_1 \vee \bar{x}_4)$

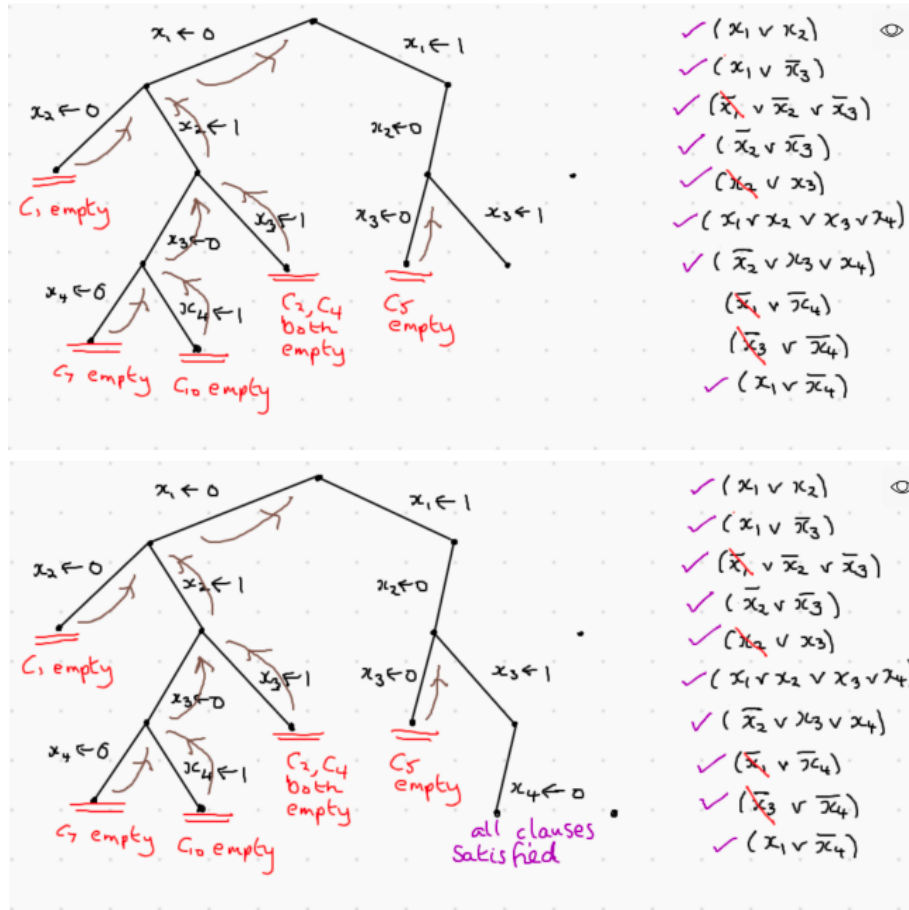


- $(x_1 \vee x_2)$ \odot
- $(x_1 \vee \bar{x}_3)$
- $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- $(\bar{x}_2 \vee \bar{x}_3)$
- $(x_2 \vee x_3)$
- $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- $(\bar{x}_2 \vee x_3 \vee x_4)$
- $(\bar{x}_1 \vee \bar{x}_4)$
- $(\bar{x}_3 \vee \bar{x}_4)$
- $(x_1 \vee \bar{x}_4)$



- ✓ $(x_1 \vee x_2)$ \odot
- ✓ $(x_1 \vee \bar{x}_3)$
- $(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
- $(\bar{x}_2 \vee \bar{x}_3)$
- $(x_2 \vee x_3)$
- ✓ $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- $(\bar{x}_2 \vee x_3 \vee x_4)$
- $(\bar{x}_1 \vee \bar{x}_4)$
- $(\bar{x}_3 \vee \bar{x}_4)$
- ✓ $(x_1 \vee \bar{x}_4)$





11.2 Dealing with NP-Completeness: DPLL

- **Unit Clause:** a clause containing a single literal
- **Pure Literal:** a literal which is always positive/negative throughout the CNF
- **PDDL:** a method to determine satisfiability, by refining backtracking search. Uses 2 heuristics:
 - **Unit Clause Heuristic:** if there is a unit clause, then make its literal positive ($x_i = 1$ or $\bar{x}_i = 0$)
 - **Pure Literal Heuristic:** set pure literal to its polarity ($x_i = 1$ or $\bar{x}_i = 0$)

By applying these heuristics, we can greatly improve the practical runtime of backtracking search:

Algorithm DPLL($\Phi = C_1 \wedge \dots \wedge C_m$)

1. **if** every literal in Φ is “pure” **then return** T
2. **else if** Φ contains an empty clause **then return** F
3. **else**
4. **while** we have some “unit clause” (ℓ) in Φ
5. **if** (ℓ is x_i) **then** $\Phi \leftarrow \Phi(x_i \leftarrow 1)$
6. **else if** (ℓ is \bar{x}_i) **then** $\Phi \leftarrow \Phi(x_i \leftarrow 0)$
7. **while** we have some “pure literal” ℓ in Φ
8. **if** (ℓ is x_i) **then** $\Phi \leftarrow \Phi(x_i \leftarrow 1)$
9. **else if** (ℓ is \bar{x}_i) **then** $\Phi \leftarrow \Phi(x_i \leftarrow 0)$
10. Choose a undetermined variable x_i of Φ how?
11. **return** (DPLL($\Phi(x_i \leftarrow 0)$) **or** DPLL($\Phi(x_i \leftarrow 1)$))

Figure 21: Notice in line 11, we are considering both possible assignments of x_i . This is called splitting.

- **Choosing the Next Variable:** to choose the next variable, we use heuristics, which work better depending on the problem:
 - any variable in an unsatisfied clause
 - variable appearing the most
 - variable which occurs in one polarity the most
 - literal in the shortest clause
 - variable with highest weighted sum of clause size
- **Runtime of DPLL:** runtime is stil $\in^{\setminus} |\oplus|$, but in practice is much better than this
 - depending on which heuristic, behaviour of DPLL varies, so can be considered as a collection of algorithms

$(p \vee q \vee r \vee s) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg q \vee \neg r \vee s) \wedge (p \vee \neg q \vee r \vee s)$
 $\wedge (q \vee \neg r \vee \neg s) \wedge (\neg p \vee \neg r \vee s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$

- (1) No unit or pure elimination possible.
- (2) We have four s -Literals, and two $\neg s$: Split using s
 - Remove clauses: $(\neg p \vee q \vee \neg r) \wedge (q \vee \neg r \vee \neg s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$
 - Remove $\neg s$: $(\neg p \vee q \vee \neg r) \wedge (q \vee \neg r) \wedge (\neg p) \wedge (p \vee \neg q)$
- (3) **Unit propagation** with $\neg p$: $(q \vee \neg r) \wedge (\neg q)$
 - Unit propagation** with $\neg q$: $(\neg r)$
 - Unit propagation** with $\neg r$ gives the empty formula, return **true**

(1) No unit or pure elimination possible.
 (2) Split using $\neg r$

- Remove clauses: $(p \vee q \vee r \vee s) \wedge (p \vee \neg q \vee r \vee s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$
- Remove r : $(p \vee q \vee s) \wedge (p \vee \neg q \vee s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$

- (3) No **Unit propagation** possible
 - No pure literal
- (4) Split using $\neg q$
 - Remove clauses: $(p \vee q \vee s) \wedge (\neg p \vee \neg s)$
 - Remove q : $(p \vee s) \wedge (\neg p \vee \neg s)$
- (5) No **Unit propagation** possible
 - No pure literal
- (6) Split using p
 - Remove clauses: $(\neg p \vee \neg s)$
 - Remove $\neg p$: $(\neg s)$
- (7) No **Unit propagation** $\neg s$ gives the empty formula, return **true**

Figure 22: Taken from Cornell notes. Notice that the choice of literal heavy affects the runtime.

12 Week 8 - Introduction to Computability Theory

12.1 Register Machines for Computability Theory

- **Computability Theory**: seeks to understand:
 1. what does it mean for a function to be computable?
 2. if a function is non-computable, can we place it within a hierarchy of non-computable functions?

We seek to shed light on these concepts

- **Computability Theory Motivation:**

- if we use comparisons to sort an array, we can never do better than $\Theta(n \lg(n))$
- we believe that any *NP-Complete* problem can't be solved in $\mathcal{O}(n^d)$, $d \in \mathbb{R}$. However, this is reliant on $P \neq NP$: for all we know, it could be possible to solve 3-SAT in $\mathcal{O}(n)$ time
- is it possible for a problem to exist, such that no algorithm **ever** solves it, independent of time or space requirements? (i.e if we had infinite time, infinite memory, is there a problem that could never be solved)

- **Register Machines:** credited to Marvin Minsky, these are simple machines which can be composed to create more complex machines. They contain **registers**, which can hold natural numbers.

- we can think of registers as being in a table or a space in memory
- register machines have basic actions available:
 - * add 1 to a given register
 - * subtract 1 from a register (except if the register is 0)
 - * test whether the contents of a register are 0
- plugging these trivial components, more complex machines are built
- see here for a representation of register machines

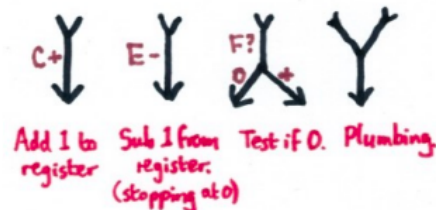


Figure 23: The basic building blocks to create more complex register machines

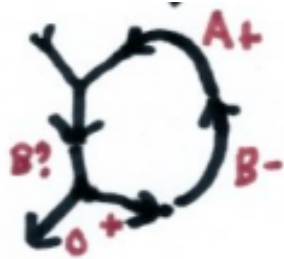


Figure 24: Adds the contents of register B to register A, in the process making B contain 0. At each cycle, checks if B is 0. If it is, then the machine terminates. If it isn't, it reduces B by 1, and increases A by 1.

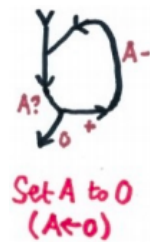


Figure 25: By checking if A is 0 at each cycle, it reduces A to 0 by decrementing the register.

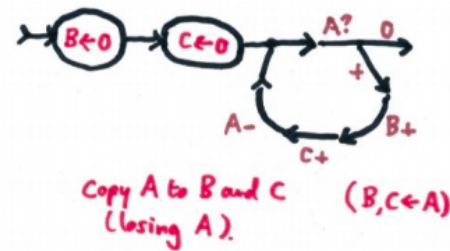


Figure 26: Creates 2 empty registers B, C. Then it cycles until termination. If A is 0, the machine terminates. Otherwise, increases B and C, and decrements A.

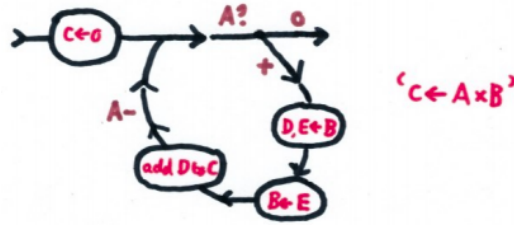


Figure 27: Multiplies the contents of register A and B. Create an empty register C. Then it cycles until termination. If A is 0, the machine terminates. Otherwise, use the machine above to populate 2 registers D,E with the value of B. B is now 0. Repopulate B, by using the first machine, to transfer the contents of E to B. Now, B and D have the value of the original B, whilst D and E are empty. Add D to C. Now, B and C contain the original value of B, whilst D and E are empty. Decrement A. In the next iteration, $C = 2B$, $B = B$, $C = D = 0$. After that, $C = 3B$, $B = B$, $C = D = 0$. By the time the machine terminates, $C = AB$, which is what we wanted.

- **Register Machines as Functions:** notice, the above register machines can be considered as mathematical functions: given a set of natural numbers (stored in the registers), it returns a set of natural numbers at some registers
- **Functions Computable by Register Machines:** we make the above statement more precise. We say a register machine M computes a partial function:

$$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

if $\forall m, n \in \mathbb{N}$, we set up registers $A = m, B = n, C = D = \dots = 0$, and then run M on these registers:

- the computation terminates if and only if $f(m, n)$ is defined
- if the function is defined, then the value stored at register A will correspond with the value of $f(m, n)$
- **RM-Computability:** a function $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is **RM-Computable** (register machine computable) **if and only if** there exists a register machine which can compute f
 - from the above RMs, we can see that addition and multiplication are RM-Computable
 - this also applies to functions that take one argument, but in the context of computability, it is most useful to consider functions of 2 arguments

12.2 The Church-Turing Thesis: CT Computable Functions

- **RM-Computable Functions are not Alone:** the class of RM-Computable functions is *equivalent* to:

- functions definable in λ -calculus
- functions computable by Turing Machines
- functions in any programming language

In fact, any of these formalisms can be used to simulate any of the others (for example, building an interpreter for RMs in Python).

- **CT-Computable Functions:** refers to the class of functions defined above (since they are all the same). Stands for “Church-Turing Computable Functions”
- **Church-Turing Thesis:** given any function on the natural numbers, the class of CT-Computable functions coincides with the class of algorithm-computable functions
 - a function on the natural numbers can be calculated via an algorithm if and only if it can be computed via RMs/Turing Machines/Lambda Calculus (the original paper only uses Turing Machines)
 - what we mean with an algorithm here is an *effective method*: that if we are given a problem and a procedure, then given an infinite amount of paper and time, we can mechanically follow the procedure and solve the problem
- **Feasibility of Church-Turing Thesis:** not proven (not amenable to mathematical proof), but points in favour include:
 1. no mechanical algorithm has been created which has been outside of this class
 2. many ways of defining “computable functions”, but they always ends up referring to the CT-Computable Functions - suggests this is the true class of computable functions
 3. Turing Machines can model a human calculator (and human calculators will solve a problem by following an algorithm)

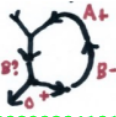
12.3 Universal Machines

- **Everything is a Number:** since register machines operate on natural numbers, if we can encode structures as natural numbers, we could construct register machines which operate on these structures:

- a set of registers can be encoded perfectly as a natural number

$A = 23$ $B = 05$ $C = 00$ $D = 00$ $E = 00$
 $F = 00$ $G = 00$ $H = 00$ $I = 00$
 might be coded as **200000000350000000**.

- a register machine itself can be coded up as a natural number

E.g. our adding machine  might be coded as
10220400401103003050320200204101 (details unimportant).

- **Universal Machine:** create a machine, which given 2 natural numbers m, n :

- reconstructs a machine M encoded in m
- reconstructs the register values R on which M is applied using n

Then, the machine executes by simulating the execution of M on R . Since the reconstruction of M and R , alongside their execution, is purely algorithmic, we can construct such a machine!

- this basically means that any machine can be simulated by only 1 machine
- this is the essence of modern computers: we write a piece of software, and our computer executes it

- **Beyond Computability:**

- by the above, our current computers bear no hope in breaking the computability barrier
- **quantum computers** offer improvements in practice (better performance, break really hard problems), but in terms of computability they are equally limited
- **black hole computers:** a crazy idea, use the gravity of black holes, to essentially break time, and solve computation problem

13 Week 9 - Unsolvable Problems

13.1 The Halting Problem

- **Defining Functions:** we have talked about functions being RM-Computable, but we haven't really specified *what* we mean with a function:

1. for CS oriented, a function can be defined using keywords like, `def` or `public static int`, followed by code
2. during the 18th century, a function was thought as a formula which accepted and spitted out numbers ($f(x) = x^2 + 2x - 7$)
3. after the 18th century, Dirichlet coined the use of functions as an assignment of values (for example, a lookup table), without the need for an explicit formula

Our notion of computable functions is better adapted to work with Dirichlet's view

- **Computation Halting:** a RM computation is said to **halt** if it eventually terminates
- **The Halting Problem:** is there any way (aka does a machine exist) such that for an arbitrary machine M , and any input n we know exactly if M will halt when given the input n ?
 - for example, a while loop of the form $\text{while}(\text{True}) \rightarrow \text{print}(\text{"Neverhalting!"})$ will never halt, no matter what
 - however, the function $f(x) \rightarrow x^2$ will eventually halt, no matter how big the input x
- **Unsolvability of Halting Problem:** long story short, *no halting tester can exist*. That is, there can be no register machine H , such that given 2 inputs m, n (m encodes another machine, n is the input for the machine), H determines if m halts on n . Thus, **the Halting Problem is RM-Unsolvable**
 - in other words, no function:

$$h(m, n) = \begin{cases} 0 & m \text{ halts on } n \\ 1 & m \text{ doesn't halt on } n \end{cases}$$

can exist, so h is **not** RM-Computable

- **Proof of the Unsolvability of Halting Problem:** lets assume that a halting tester exists. Call it H . H takes an encoded machine m and an input n . H tells us whether m will halt when applied to n . The proof relies on 4 keys:

1. Applying a Machine to Itself

- machines can be encoded as natural numbers; machines also accept natural numbers as inputs. Thus, machines can accept other machines and inputs ... even themselves
- informally, if we think of a machine X as a function, applying X to itself is given by the result of $X(X)$

- in terms of register machines, if we have a machine encoded as m , we can think of applying m to itself as applying m on an initial state given by registers:

$$A = m, B = C = \dots = 0$$

2. The Self-App Machine

- the **SELF-APP** machine sets us up for applying a machine to itself. **SELF-APP** is applied on registers:

$$A = m, B = C = \dots = 0$$

where m is an encoded machine. **SELF-APP** then returns the following register states:

$$A = m, B = n, C = D = \dots = 0$$

where n corresponds to an encoding of the initial state of the registers of **SELF-APP**

$$n = \text{encode}(A = m, B = C = \dots = 0)$$

In other words, **SELF-APP** gets m , and returns m , alongside a complete description of itself (namely n)

3. **The P Machine:** this is a machine designed to determine whether a machine will halt on itself, given itself as an input (bear with me!). We construct P as follows.

- at the top level, we have **SELF-APP**, which takes a machine m as input. As output, we get both m , and a complete (encoded) description of m , called n .
- at the bottom level, we have a modified version of H . Recall, H takes a machine and an input (registers), and determines whether the machine halts or not on the input. Our modified H (call it H') is such that it will halt if the machine will never halt on the current input (so H' halts if the inputs to H get stuck in an infinite loop). Alternatively, if the machine halts on the input, then H' will run on an infinite loop, never halting
- the output of **SELF-APP** (m and n) are then passed as the inputs of H'

Overall, what P does is to determine whether a machine m will halt when it is applied to itself. P halts whenever m applied to itself gets stuck. P gets stuck whenever m applied to itself halts. In a way, P reverse H

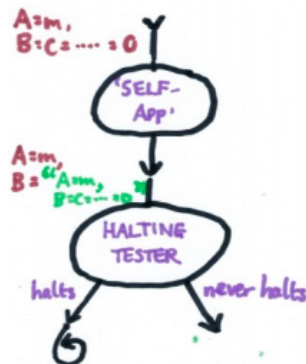


Figure 28: The machine P

4. A Contradiction

- now, consider applying P to itself (so $P(P)$)
- as inputs to H' , we get P and its description
- assume that H' determines that P halts when running itself. Notice that P running on itself is our original call. If H' finds that P applied on itself halts, then it will tell P to get stuck on an infinite loop. But this contradicts the fact that H' knew that P would halt.
- alternatively, if H' determines that P gets stuck, then H' tells P to halt. But this contradicts the fact that H' knew that P would get stuck when run on itself
- we reach a contradiction no matter what. Thus, our assumption that a machine such as H existed must have been false, so no register machine can solve the halting problem

All of the above is quite mind boggling. These 2 resources really helped me:

- a very nice, short video. It isn't as "formal", but it is a lot easier to understand
- a very nice explanation, which uses code to better understand how the contradiction arises

Note that the Halting Problem is the OG of undecidability. It is analogous to SAT for *NP-Hard/Complete* problems.

- **Russel's Paradox:** R is the set of all sets that don't contain themselves. Is it the case that $R \in R$?
 - if $R \in R$, then R contains itself, so R can't be in itself
 - if R is not part of itself, it satisfies the criteria of R , so we must have $R \in R$

- analogy: the village barber is a man who only shaves men in the village who don't shave themselves. Does the barber shave himself? If he did shave himself, the barber wouldn't shave him, but he is the barber, so he wouldn't shave himself. If he didn't shave himself, he would be a man in the village who doesn't shave himself, so he would have to be shaved by the barber, who is himself.

13.2 The World of Unsolvability

- **Diophantine Equations:** unsolvable; no computer program can take a Diophantine Equation and determine if it has a solution
 - recall, a Diophantine Equation is a multivariable, polynomial equation with integer coefficients for which we require integer solutions:

$$x^2 + y^2 + z = 26$$

$$x^2y - 20z^5 + zxy^3 - v = 12$$

- **Post's Word Problem:** given 2 sets of strings S, T , decide whether there is a string that can be formed by:
 - concatenating elements in S
 - concatenating elements in T

Again, unsolvable.

$$S = \{a, ab, bba\}, \quad T = \{baa, aa, bb\}$$

Then the answer is **YES**, because:

$$bba.ab.bba.a = bbaabbbbaa = bb.aa.bb.baa$$

13.3 Food for Thought

- **Gödel's Dichotomy:** there are undecidable problems, so there are problems which machines can't give a yes/no. Since machines can imitate human thought, does this mean that there must be mathematical questions which we humans can't answer? This is the essence of **Gödel's Dichotomy**:

Either the human mind can infinitely surpass the power of finite machines in mathematics, or there exists

- Gödel's Incompleteness Theorem showed that any system for formal mathematical proof will leave questions unresolved
- Gödel believed that human reason was unlimited, so no mathematical questions were absolutely unsolvable

- If a question is absolutely unsolvable, can we say that the question has a definite answer?
- Which mathematical statements have a definite meaning? How much is just human convention? answer to