AA Mod 4

- **Branch and Bound** BFS

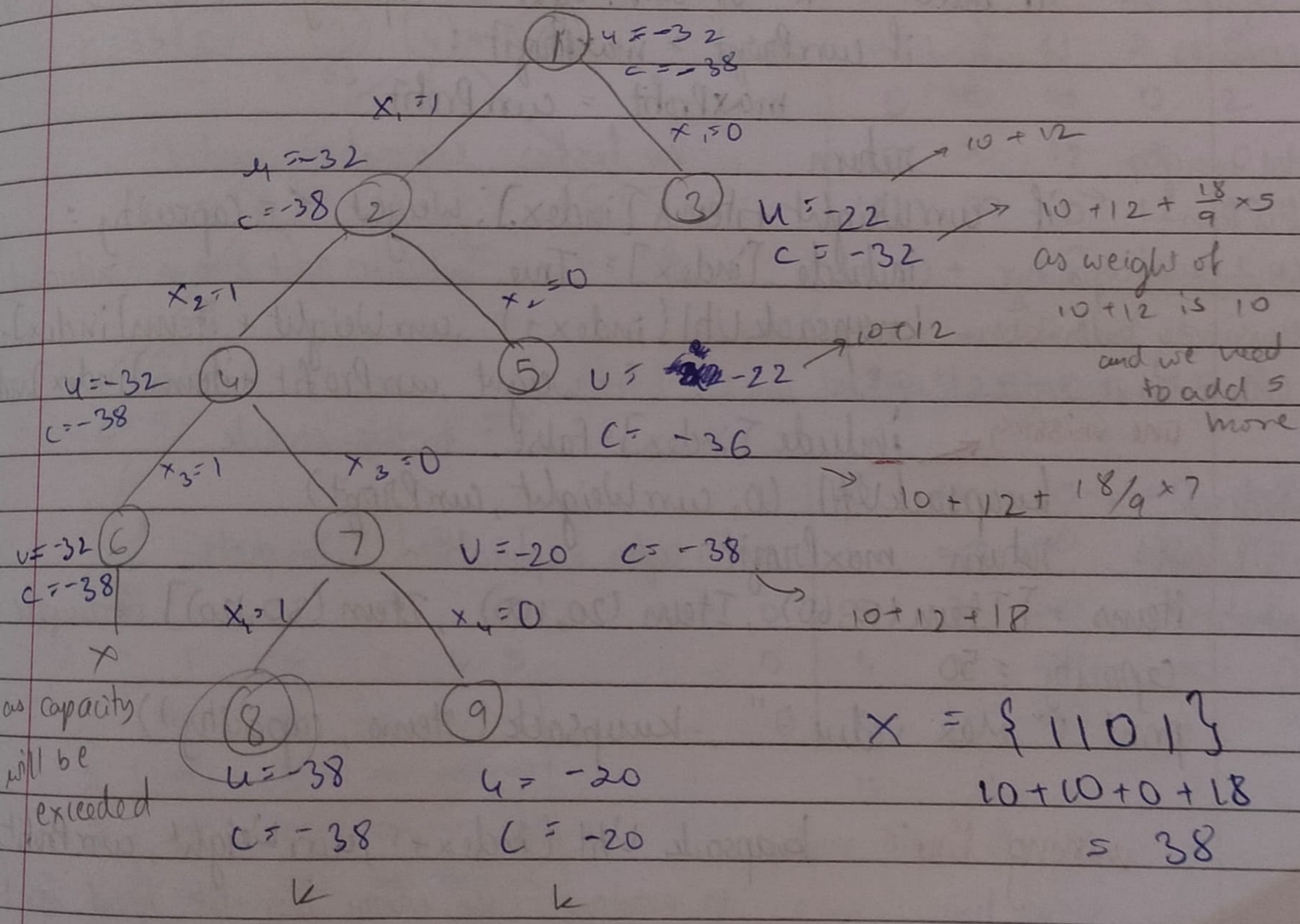
It works by dividing the problem into sub problems, or branches, then eliminating certain branches based on bounds of optimal solutions. This process continues until best solution is found or all branches are explored.

- **0/1 Knapsack Using Branch & Bound**

	1	2	3	4
Profit :	10	10	12	18
Weight :	2	4	6	9

$m = 15 \rightarrow$ weight
$n = 4$

upper bound $u = \sum_{i=1}^{n} P_i x_i$ cost $c = \sum_{i=1}^{n} P_i x_i$ with fraction



$X = \{1101\}$
$10 + 10 + 0 + 18$
$= 38$

Time Complexity : $O(2^n)$ Space : $O(N)$

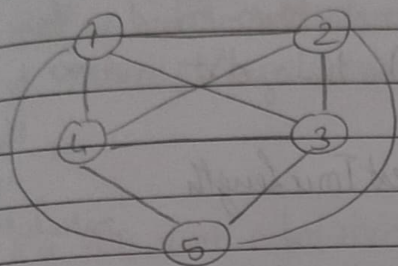
Page No.
Date

```
class Item : represents each item
    def __init__(self, weight, value):
constructor for <  self.weight = weight
initialization      self.value = value
def knapsack(items, capacity):
    n = len(items)  no. of items in the list
    maxProfit = 0    currWeight = 0    currProfit = 0
    include = [False] * n   indicates whether each item is included or not
items sorted <- items.sort(key = lambda x: x.value / x.weight,
based on λ = value/weight descending   reverse = True)
    def knapsackUtil(index, currWeight, currProfit):
recursive fn  nonlocal maxProfit and upload maxm Profit
to explore all  if index == n or currWeight == capacity: knapsack is full
combos   of current item   if currProfit > maxProfit:
                    maxProfit = currProfit
            return    no further exploration needed
total weight
after item  <- if currWeight + items[index].weight <= capacity:
doesn't exceed capacity   include[index] = True indicates that item is included
recursively calls it <- knapsackUtil(index + 1, currWeight + items[index].
with next index & updated curr Wt & Profit  weight, currProfit + items[index].value)
line missing <- include[index] = False to backtrack
initializes, <- knapsackUtil(0, currWeight, currProfit)
sorts & starts   return maxProfit after exploration is complete
exploration items = [Item(10,60), Item(20,100), Item(30,120)] Example
    capacity = 50
    print("Max value:", knapsack(items, capacity))

missing line:  knapsackUtil(index+1, currWeight, currProfit)
                   to explore case where current item isn't
                   included in the knapsack
```

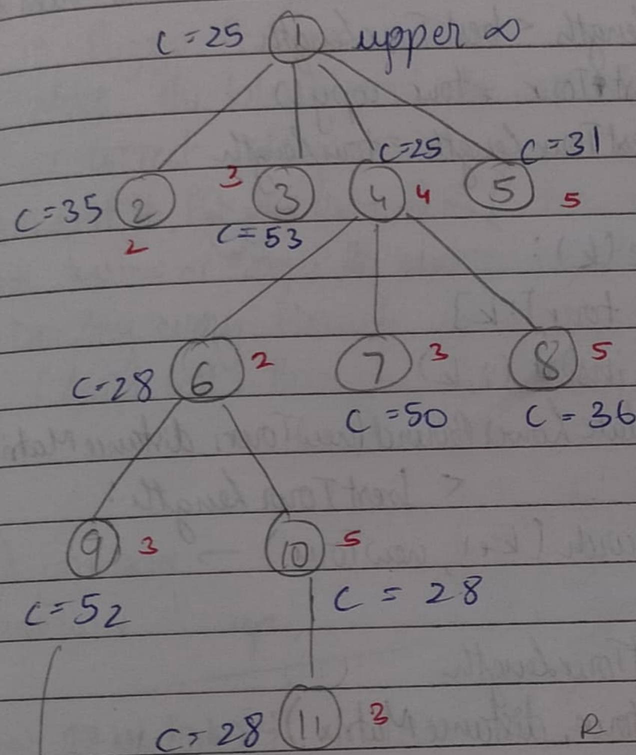
Travelling Salesman Problem using Branch & Bound

Given a set of cities & dist b/w every pair of cities, find the shortest path that visits every city exactly once & returns to starting point.
↳ vertices



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 10 & 17 & 0 & 1 \\ 2 & 12 & \infty & 11 & 2 & 0 \\ 3 & 0 & 3 & \infty & 0 & 2 \\ 4 & 15 & 3 & 12 & \infty & 0 \\ 5 & 11 & 0 & 0 & 12 & \infty \end{array}$$

Reduced Cost = 25



R C
1-2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

$C(1,2) + r + \hat{r}$ → reductions done
cost 1 to 2, reduced cost 25

$10 + 25 + 0 = 35$

R C
1-3

$$\begin{array}{c|ccccc} 0 & \infty & \infty & \infty & \infty & \infty \\ 0 & 1 & \infty & \infty & 2 & 0 \\ 0 & \infty & 3 & \infty & 0 & 2 \\ 0 & 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & 0 & \infty & 12 & \infty \\ \hline & 11 & 0 & 0 & 0 & 0 \end{array}$$

Here are 0s in every row, and every column except c_1.
We replace 1-2, c_3, 3-1
Subtract 11 from c_1
↳ smallest = $\hat{r}$

2-3

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & \\ 2 & \infty & \infty & \infty & \infty & \infty & \\ 3 & \infty & \infty & \infty & \infty & 2 & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty & \\ 5 & 11 & \infty & \infty & \infty & \infty & 11 \end{array}$$

$C(2,3) + C(6) + 2 + 11$
$= 52$

Page No.
Date

Time Complexity : $O(n!)$
Space Complexity : $O(n^2)$

```
def TSP(n, distanceMatrix):
    bestTour = []          list to find best tour found so far
    bestTourLength = float('int') initialized to +ve ∞ to ensure shorter length
    recursive, def search (k, tour):
    explores all      nonlocal bestTour, bestTourLength
    tours starting    if k == n : all cities visited   → calculate current TL
    from city 0           tourLength = calculateTL (tour, distanceMatrix)
                          if tourLength < bestTourLength :
    if current tour length      bestTour = tour.copy()
    < best, we update it ←      bestTourLength = tourLength
                          return   → for each city k, it tries inserting
                      for i in range(k):   into all i posⁿ in current tour
    copies current tour   ← newTour = tour[:k]
    inserts city k at i   ← newTour.insert (i, k)
    before exploring further, it ← if calculateLowerBound(newTour, distanceMatrix)
    checks if lower bound of tour < best TL,   < bestTourLength :
    if not, the branch is pruned      search (k+1, newTour) → next city & new tour
        search (1, [0])
        return bestTour, bestTourLength
    def calculateTourLength TL (tour, distanceMatrix): calculates TL considering
    iterates over city index tourLength = 0       distance from last to starting city
    i to tour, except last city  for i in range (len (tour) -1):
    for each city, it adds dist from tourLength += distanceMatrix [tour[i]][tour[i+1]]
    current to next city   tourLength += distanceMatrix [tour[-1]][tour[0]]
    in TL                  return tourLength
                                                → for remaining TL
    def calculateLowerBound (tour, distanceMatrix): by summing minᵐ
    after loop, dist   tourLength = calculateTL (tour, distanceMatrix) distances from
    from last is       missingCities = set(range(len(distanceMatrix))) - set(tour)
    added to 1st         ↳ identifies set of cities not included      each unvisited city
                           in tour                                    to nearest unvisited city
```

it finds the nearest city to cities excluded

lower Bound = tour Length → for each missing city
for city in missing Cities :
min Distance = min (distance Matrix[city][other City] for other city in missing Cities if other City != city)
lower Bound += min Distance
return Lower Bound → min^m additional distance required to complete tour assuming shortest paths are taken b/w missing cities.

it finds the min from that city to another city in the set of missing cities excluding itself

There's a very short algo in mam's PPT

- 15-Puzzle Problem

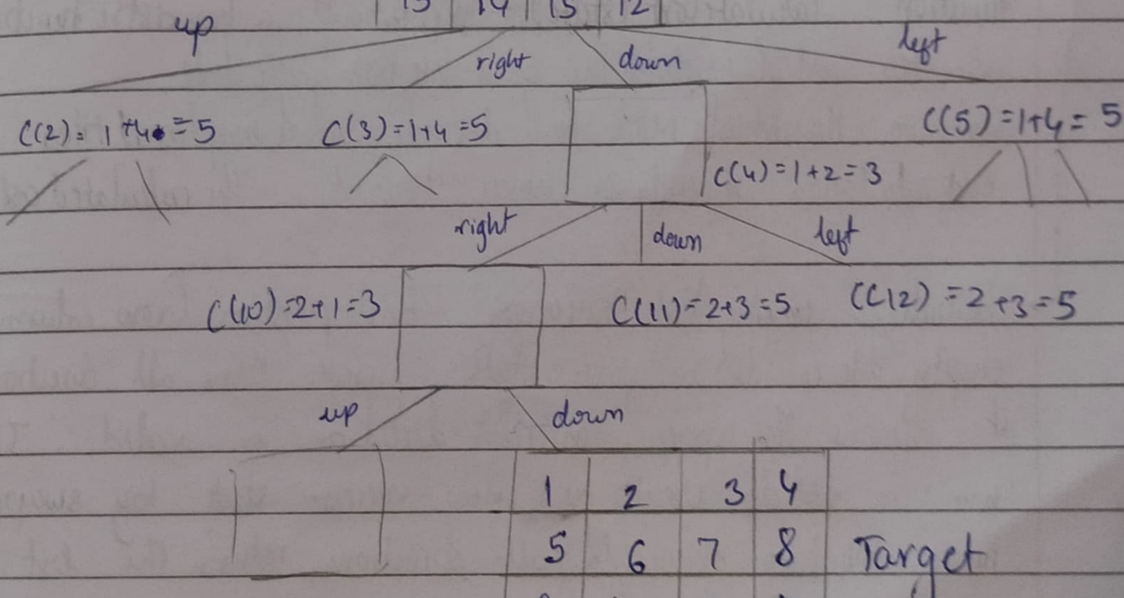
Move the blank piece one step each time to reach the final or correct stage.

Cost = f(x) + g(x) → how many pieces are away from final/correct position

f(x) → no. of moves made

Drawing every branch is hard, so we can follow LC-BB (Least Cost Branch & Bound)

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Target

function solve15Puzzle (initialState):
openList = PriorityQueue() to store partial solⁿ
openList.put(Node(initialState, 0)) Enqueue initial state
while not openList.empty(): from openList
currentNode = openList.get() Dequeue lowest cost node
check if current state is goal node ← if isGoalState(currentNode.state):
return currentNode.state return goal state
Generate successor states ← successors = generateSuccessors (currentNode.state)
for successor in successors:
Calculate successor state cost ← successorCost = calculateCost (successor)
enqueue successor state ← openList.put(Node(successor, successor))
return None No solution found

class Node: to represent each state of the puzzle
def __init__(self, state, cost):
self.state = state state of the puzzle (board config)
self.cost = cost cost of reaching this state

function isGoalState (state): iterate through each tile in state & check if each tile is in correct position according to goal, if they are then true else false

function generateSuccessors (state):

function calculateCost (state): implements a heuristic function that estimates cost of reaching the goal state from given state.
Common Heuristics: Manhattan distance, no. of misplaced tiles
Cost calculation depends on chosen heuristic. The calculated cost is returned

→ initialize empty list 'successors'. Find position (row column) of empty tile is (0) in given state, iterate thru all directions, check if moving the space in that direction is valid. If the move is valid, create a new successor state by swapping. After iterating through all directions, return the list of successors

Time Complexity: $O(n^2)$ Space Complexity: $O(n)$