SRM UNIVERSITY AP
Andhra Pradesh

Design & Analysis of Algorithms

Semester 3 Project

Topic - 7: Approximation Algorithms, Backtracking Algorithm

Group: 18

Group Members:

Susrita Voleti – AP21110010185

Adi Vishnu Avula – AP21110010186

Krishna Varshita Borra – AP21110010187

Under the Guidance of:

Nitul Dutta Sir

Surya Samantha Ma’am

Problem Statement

(a) Implement Approximation algorithms for
    1. Vertex cover problem
    2. Set covering problem

(b) Write program to generate as many moves as possible (not required to generate all moves) for a Knight (Horse) on a chess board starting from any arbitrary position. Knight cannot touch any square (place) second time.

Description

(a) Approximation Algorithms:

An approach to NP-completeness for an optimization problem is an approximation algorithm. A set of decision problems known as NP can be resolved in polynomial time using an algorithm that consistently selects the correct option from the available options. The optimal answer is not always ensured by this method. The approximation algorithm aims to reach the ideal answer in polynomial time as close as possible. Such algorithms are called approximation algorithms.
Let's say we are working on an optimization problem where each possible solution has a price. A legal answer is produced using an approximate algorithm, although the cost may not be ideal.

Key features of an Approximation Algorithm:

- Although it does not provide the best outcome, an approximation method guaranteed to execute in polynomial time.
- An algorithm for approximations promises to look for highly accurate and excellent solutions (let's say within 1% of the optimum).
- So, with the use of approximation techniques, an optimization issue may be solved in polynomial time with a result that is close to the (optimal) solution.

We can use the approximation algorithms to solve vertex cover problem, and set covering problem. Finding the vertex cover with the fewest vertices is the vertex cover problem's optimization problem, while finding the vertex cover with few vertices is the vertex cover problem's approximation problem. In the case of set covering problem, the optimization problem simulates a variety of issues where resources must be apportioned. A logarithmic approximation ratio is applied in this case.

1) Vertex cover problem:

Whenever an edge of a graph G has at least one of the vertices in the set V_c as an endpoint, that edge is said to have a vertex cover of that graph. This indicates that at least one edge is touched by each vertex in the graph. Vertex cover is an NP problem because any solution may be checked in polynomial time by checking all of the edges' endpoints to see if the suggested vertex cover includes them.

Vertex covers can be determined using a variety of techniques. Finding an exact solution to the NP-complete vertex cover problem is highly challenging and time intensive. Approximation techniques are rather. These operate much more quickly than accurate algorithms but might result in a less than ideal solution.

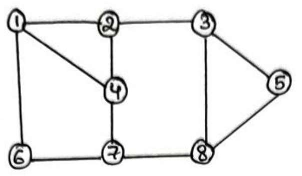
Algorithm for Vertex Cover:

```
ApproximateVertexCover (G = (V, E))
Input: Graph with vertices(V) and edges €
Output: Vertex Cover array
    Step 1. Start
    Step 2. Set C = {} //empty-set
    Step 3. Set E'= E
    Step 4. While E' is NOT empty do
            Step 4.1 Let (u, v) be any edge in E'
            Step 4.2 Add u and v to C
            Step 4.3 Remove from E' all edges incident to u or v
    Step 5. Return C
    Step 6. Stop
```

The technique basically operates by locating a maximum matching in G and adding at least one endpoint of each edge to the collection of vertices C that it is covering. A substandard covering might have both endpoints from each edge, resulting in a covering set C that is up to twice as large as the ideal response. The best answer will contain one vertex from each edge in the matching.

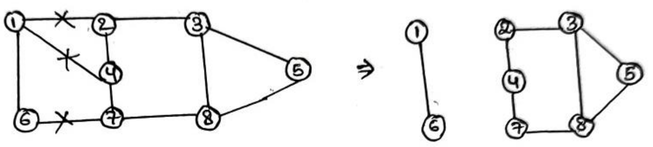
Example

The set of edges of given graph be -
{{(1,6), (1,2), (1,4), (2,3), (2,4), (6,7), (4,7), (7,8), (3,8), (3,5), (8,5)}}



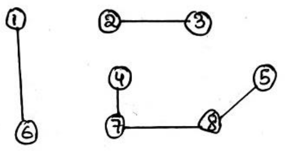
Now, we start by selecting an arbitary edge, say (1,6), we eliminate all the edges which are incident to vertex 1 or 6 and add (1,6) to cover.

Edges removed:-
(1,2)
(1,4)
(6,7)



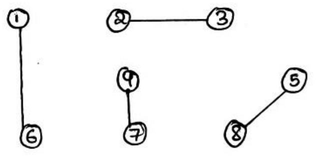
In the next step, we have chosen another edge (2,3) arbitarily
This results in ⇒

Edges removed:-
(2, 4)
(3,5)
(3, 8)

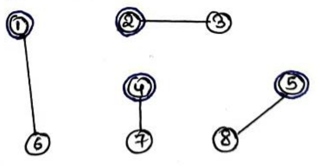


Next, we select the edge (4,7)

Edges removed:
(7, 8)



Now e select the edge (8,5)



Now, since all the edges are covered, this is the vertex cover graph for the given graph.

Hence, the output V_c = {1, 2, 4, 5}

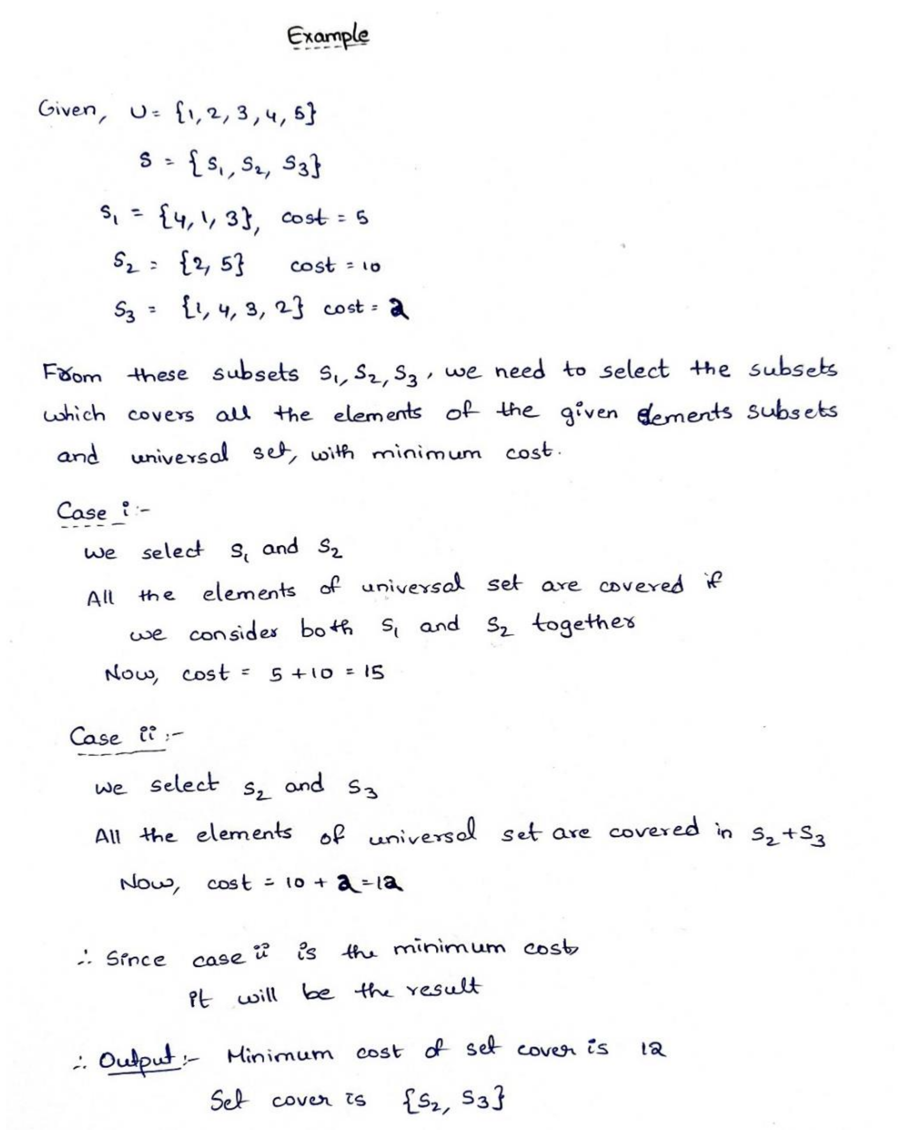
2) Set Covering Problem:

In the set covering problem our input is a set U and a collection of subsets S1, ..., Sm, where each element Si has a non-negative weight wi. A solution is a subcollection of the set whose union is the entire set U. The score of a solution is the total weight of the sets used, and our goal is to minimize this score.

For example, given an undirected graph G we can let U be the set of edges E and let S_v for every vertex v be the set of edges incident on v. If we give each S_v a weight of 1, the minimum weight set cover is just the minimum size vertex cover of G.

Algorithm for Set Cover:

```
SetCover(G, S)
Input: Universal set, set of subsets
Output: Set cover array
    Step 1. Start
    Step 2. Set U = G
    Step 3. Set C = {} //empty-set
    Step 4. While U is NOT empty, do
            Step 4.1. Select S[i] in S that covers the most elements of U
            Step 4.2. Add i to C
            Step 4.3. Remove the elements of S[i] from U
    Step 5. Return C
    Step 6. Stop
```



Example

Given, $U = \{1, 2, 3, 4, 5\}$

$S = \{S_1, S_2, S_3\}$

$S_1 = \{4, 1, 3\}$, cost = 5

$S_2 = \{2, 5\}$ cost = 10

$S_3 = \{1, 4, 3, 2\}$ cost = 2

From these subsets S_1, S_2, S_3, we need to select the subsets which covers all the elements of the given elements subsets and universal set, with minimum cost.

Case i :-

we select S_1 and S_2

All the elements of universal set are covered if we consider both S_1 and S_2 together

Now, cost = 5 + 10 = 15

Case ii :-

we select S_2 and S_3

All the elements of universal set are covered in $S_2 + S_3$

Now, cost = 10 + 2 = 12

∴ Since case ii is the minimum cost it will be the result

∴ Output :- Minimum cost of set cover is 12

Set cover is $\{S_2, S_3\}$

b) Backtracking – Knight’s Tour Problem:

Backtracking employs the brute force method of problem solving, which states that we attempt to create all feasible solutions for the given issue before selecting the best one among all the candidates. When we need all of the possible answers and have many ones, we use backtracking. Backtracking implies that we are moving back and forward; if the condition is satisfied, we return with success; otherwise, we go back.

Now, coming to the Knight’s tour problem: A knight's tour is a series of knight moves on a chessboard where each square is visited exactly once by the knight. The tour is closed if the knight stops on a tile that is one knight's move from the starting square; otherwise, it is open. So, given a N*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

We solve the Knight’s tour problem using the following approach:

1. If all squares are visited
   a) print the solution
2. Else
   a) Add one of the next moves to solution set and recursively check if this move leads to a solution. (A Knight can make maximum eight alternative moves. We choose one of the 8 moves).
   b) If the move chosen in the above step doesn't lead to a solution, then remove this move from the solution vector and try other alternative moves.
   c) If none of the alternative moves are valid, remove the previously added item in recursion and if false is returned by the very first call of recursion, then print "no solution exists"

Algorithm for checking if move is valid or not:

```
Let N be the number of squares on one edge of the chess board

checkValidity(x, y, sol)
Input: X-position, y-position, solution array
Output: Returns true is the move is valid
    Step 1. Start
    Step 2. If (0 ≤ x ≤ N) AND (0 ≤ y ≤ N) AND (x, y) is empty, then
            Step 2.1. Return true
    Step 3. Stop
```

Algorithm for Knight's Tour Problem:

```
knightTour(x, y, move, sol, xMove, yMove)
Input: X-position, Y-position, move number, solution array, movement to X-
position, movement to Y-position
Output: Final solution array containing all the cell positions
    Step 1. Start
    Step 2. if move = N * N, then //when all squares are visited
            Step 2.1Return true
    Step 3. For k = 0 to number of possible xMove or yMove
            Step3.1. xNext = x + xMove[k]
            Step 3.2. yNext = y + yMove[k]
            Step 3.3. if checkValidity(xNext, yNext, sol) is true, then
                    Step 3.3.1. sol[xNext, yMext] = move
                    Step 3.3.2. if knightTour(xNext,yNext,move+1,sol,xMove, yMove)
                            Step 3.3.2.1. return true
                    Step 3.3.3. else
                            Step 3.3.3.1. Remove move from the sol[xNext, yNext]
    Step 4. return false
    Step 5. Stop
```