CSE 624/424 HOMEWORK

ONE DIMENSIONAL CIRCLE PACKING PROBLEM

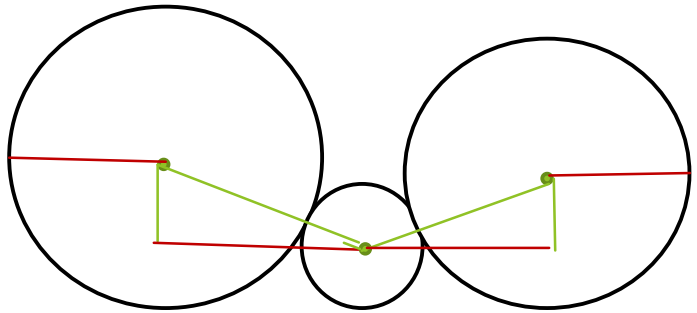
Sevgi BAYANSALDUZ

PROBLEM DEFINITION

- **Problem Instance**: A list of the varius radiuses size of the circles. A[1…n]
- **Problem Definition:** One dimensional circle packing where the circles are packed in a twodimensional box such that each circle is tangent to the bottom of the box. The problem is to find the box with minimum width and the arrangement of the circles.
- **Feasible Solution:** A width of the box, all elements of the circles list are tanget to the bottom of the box.
- **Optimum Solution : Minimum width of the box**

PROBLEM DEFINITON

- **Objective function :**Sum of the first and last circles radiuses, and the distance between of the adjacent circles (distance is calculated with Pythagorean Theorem).
- $r_1 + r_n + \sum_{i=1}^{n-1} \sqrt{(r_i + r_{i+1})^2 - (r_i - r_{i+1})^2}$



BRUTE FORCE

- Brute Force calculates all permutations for given circle list. If we assume the size of the circle list is n, the complexity of the brute force algortihm is O(n!).

- Example :
  - Given list : [1,2,3]
  - Brute force calculates below permutations:
    - [1,2,3] , [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]

BRUTE FORCE

- **Results:**

Input list	size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	615.856371553563	0.77	40, 60, 20, 80, 70, 10, 70, 30, 50
[50, 2, 4, 10, 30,80,40,15,90]	9	382.12984529746024	9.89	30, 15, 50, 4, 90, 2, 80, 10, 40
[100,35,40,50,45,70,80,90,85, 120]	10	1329.2853929163145	103.17	70, 85, 45, 100, 35, 120, 40, 90, 50, 80
[15,25,40,98,54,36,21,49,100, 140,7]	11	864.5793029931995	1757.11	40,49,25,98,15,140,7,100,21,54,36

GREEDY

- **Greedy Choice :** Select the circle with the minumum distance.
- **Greedy Algorithm:**

```
Greedy Algorithm ( a [ 1 .. N ] )
{
        distanceMatrix ← createDistanceMatrix(a)
        bestSolution ← None
        for i ← 1 to n
                solution ← {a[i]}
                x ← constructSolution (solution, a)
                if fitness(x) < fitness (bestSolution)
                        solution ← solution U {x}
        return bestSolution
}
```

- **Complexity : O(n^3)**

GREEDY

- It works faster than brute force but usually does not return solutions that are close to optimal solution

Input list	List size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	681.7266854699864	0	80, 10, 20, 30, 40, 50, 60, 70
[50, 2, 4, 10, 30,80,40,15,90]	9	567.2759427286699	0	90, 2, 4, 10, 15, 30, 40, 50, 80
[100,35,40,50,45,70,80,90,85,120]	10	1401.7464378641914	0	120,35,40,45,50,70,80,85,90,100
[15,25,40,98,54,36,21,49,100,140,7]	11	1075.2252146437377	0	140,7,15,21,25,36,40,49,54,98,100

LOCAL SEARCH

- **Neighborhood structure:** Swap of two circles.

This algorithm firstly takes an initial solution from greedy solution as a parameter. Its stopping conditon is L itearations, after L iteration algortihm loop will be stop.

- **Algortihm:**

```
Local Search Algorithm ( a [ 1 .. N ] , L)
{
    initialSolution ← greedySolution(a)
    t ← 0
    while t <L
        x ← bestNeighborhood(initialSolution)
        x ← constructSolution (solution, a)
        if fitness(x) < fitness (bestSolution)
            solution ← x
            t ← t+1
        else
            stop the iteration
    return initialSolution
}
```

- **Complexity: O(L.n^2)**

LOCAL SEARCH

- It works faster than the brute force and gives better results than the greedy algorithm.

Input list	List size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	626.7287127460996	0.01	60, 10, 80, 30, 40, 70, 20, 50
[50, 2, 4, 10, 30,80,40,15,90]	9	404.9264898629956	0.01	40, 4, 90, 10, 30, 50, 2, 80, 15
[100,35,40,50,45,70,80,90,85,120]	10	1334.6997472869564	0.01	85, 40, 120, 35, 90, 70, 50, 100, 45, 80
[15,25,40,98,54,36,21,49,100,140,7]	11	864.9301923525259	0.06	49,25,98,15,140,7,100,21,54,36,40

VARIABLE NEIGBORHOOD SEARCH

- **Neighborhood structure:**
  - Consider an arangement given by a permutation $\alpha = (\alpha 1, \alpha 2, \cdots, \alpha n)$ of $\{1, 2, \cdots, n\}$.
  - First order neighborhood $N1(\alpha)$, as $\{\alpha(11), \alpha(12), \cdots, \alpha(1n)\}$, of a solution $\alpha = (\alpha 1, \alpha 2, \alpha 3, \alpha 4 \cdots, \alpha n\text{-}1, \alpha n)$:

    $\alpha(11) = (\alpha 2, \alpha 1, \alpha 3, \ldots, \alpha n-1, \alpha n)$,

    $\alpha(12) = (\alpha 1, \alpha 3, \alpha 2, \ldots, \alpha n-1, \alpha n)$,

    .....

    $\alpha(1\ n-1) = (\alpha 1, \alpha 2, \alpha 3, \ldots, \alpha n, \alpha n-1)$,

    $\alpha(1\ n) = (\alpha n, \alpha 2, \alpha 3, \ldots, \alpha n-1, \alpha 1)$.
  - The kth order neighborhood of an solution α is constructed by exchanging αi and $\alpha i+k \pmod n$
  - $Ni(\alpha) = Nn-i(\alpha)$ so that maximum value of k is $kmax = [n/2]$

VARIABLE NEIGBORHOOD SEARCH

- This algorithm firstly takes an initial solution from greedy solver. Its stopping conditon is M itearations, after M iteration algortihm will be stop.
- **Algortihm:**

```
VNS( a [ 1 .. N ] ,M){
        initialSolution ← greedySolution(a)
        t ← 0
        while t < M
        {
            k ← 1
            whilek ≤kmaxdo
                    s'← Shake(initialSolution,k)  #Generate a point x1  at random from the kth neighborhood of x
            (x1 ∈ Nk(x))))
                    s"←LocalSearch(s')
                    if fitness(s") < fitness (initialSolution)
                     initialSolution ← s"
                     k ← 1
                        else
                            k  ←k+1
            t  ← t+1
        }
        return initialSolution
}
```

- **Complexity:**O(M.K.L.n^2)

VARIABLE NEIGBORHOOD SEARCH

- It works slower than the Greedy and local search algorithms, but it gives a solution which is the optimal solution or very close to the optimal solution.

Input list	List size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	615.856371553563	1.8	40, 60, 20, 80, 10, 70, 30, 50
[50, 2, 4, 10, 30,80,40,15,90]	9	382.12984529746024	2.78	40, 10, 80, 2, 90, 4, 50, 15, 30
[100,35,40,50,45,70,80,90,85,120]	10	1329.4851658451025	4.9	80, 45, 100, 35, 120, 40, 90, 50, 85, 70
[15,25,40,98,54,36,21,49,100,140,7]	11	864.9301923525259	5.64	49,25,98,15,140,7,100,21,54,36,40

TABU SEARCH

- **Initial solution:** It generates the initial solution with random permutation.
- **Neighbors:** The neighbors of a solution is swap of two circles.
- **Cost of a solution (objective function):** width of the box.
  - $r_1 + r_n + \sum_{i=1}^{n-1} \sqrt{(r_i + r_{i+1})^2 - (r_i - r_{i+1})^2}$
- **Use of memory:** Recency-based memory is used. It maintains the number of remaining iterations for which a given swap stays on the tabulist. Swap is used for key and its value is assigned by the iteration value.
- **Tabu Tenure:** 5 iteration.
- **Termination Criteria:** number of iteration to stop
- **Aspiration Criteria:** If there is a tabu solution that is better than a global solution, it can be choose for solution and its value of the tabu list changed with the tenure.

TABU SEARCH

- **Algorithm**:MAX_ITER, ITER, tenure and the list of the circles (the size of the circle is n) is given
  - Create recency-based memory. (Time complexity of this step is O(n^2))
  - Create an initial solution with random permutation. Assign the global solution with the inital solution. (Time complexity of this step is O(n))
  - The following steps were performed during MAX_ITER iteration
    - Assign the best local solution with the global solution.
    - The following steps were performed during ITER iteration
      - Identify 2 – interchange moves to the set N ,and select the best admisssable move from the set N.(Time complexity of this step is O(n^2)
      - Update Tabulist and other variables.(Time complexity of this step is O(n^2)
      - If the new move is better than the best local solution,then update the best local solution with the new move.
    - If the best local solution is better than the global solution, then update the global solution with the best local solution.
- **Complexity :** O (n^2* MAX_ITER* ITER)

TABU SEARCH

- **Parameters**: Tabu tenure :5 MAXITER:7 ITER=100
- Tabu gives better result than LocalSearch, and work faster than VNS.

Input list	List size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	615.856371553563	0.57	40, 60, 20, 80, 10, 70, 30, 50
[50, 2, 4, 10, 30,80,40,15,90]	9	382.12984529746024	2.78	30, 15, 50, 4, 90, 2, 80, 10, 40
[100,35,40,50,45,70,80,90,85,120]	10	1333.8867633093216	1.2	70, 80, 50, 90, 40, 100, 35, 120, 40, 85
[15,25,40,98,54,36,21,49,100,140,7]	11	864.5793029931995	5.64	40, 49, 25, 98, 15,140, 7,100,21, 54,36

GENETIC ALGORITHM

- **Population:** Created with the random permutation
- **Parent Selection:** Rouleette Wheel Selection is used for the parent selection
- **Crossover**: Order 1 Crossover is used for crossover
- **Mutation**: Scramble Mutation is uses for mutation. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.
- **Survivor Selection**: Fitness Based Selection used for survivor selection. In this fitness based selection, the children tend to replace the least fit individuals in the population.
- **Temination Criteria**: number of iteration to stop

GENETIC ALGORITHM

- **Algorithm:** ITER (iteration size), population size (m), and list of the circles (the size of the circle is n) is given
  - Create the population. Time complexity of this step is O(m.n)
  - The following steps were performed during ITER iteration
    - Select two parents. Time complexity of this step is O(m)
    - Croosover the parents and create two children. Time complexity of this step is O(n)
    - Mutate the children. Time complexity of this step is O(n)
    - Select the survivors. Time complexity of this step is O(nlogn)
- **Complexity:** O(ITER.n.logn)

GENETIC ALGORITHM

- GA works faster than VNS, but result of VNS better than the GA. Tabu Works slower than the GA but gives better result than the GA.s
- **Parameters**: size of population :10 MAXITER:400

Input list	Size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	623.4221972653426	0.07	60, 20, 80, 10, 70, 30, 50, 40
[50, 2, 4, 10, 30,80,40,15,90]	9	386.347384402607	0.07	50, 4, 90, 2, 80, 10, 40, 15, 30
[100,35,40,50,45,70,80,90,85, 120]	10	1329.48516584510	0.09	80, 45, 100, 35, 120, 40, 90, 50, 85, 70
[15,25,40,98,54,36,21,49,100, 140,7]	11	869.714162314763	0.07	21,100, 7, 140, 15, 98, 25,54,36,49, 40

PARTICLE SWARM OPTIMIZATION

- **Particle:** permutation of the given circle list .
- **Swarm:** Created with the random permutation.
- **Fitness function:** $r_1 + r_n + \sum_{i=1}^{n-1} \sqrt{(r_i + r_{i+1})^2 - (r_i - r_{i+1})^2}$
- Each particle has a position vector X , velocity vector V, best experience of own P.
- Updating velocity with: $v_{i,d}(t+1) = \omega \cdot v_{i,d}(t) + c_1 \cdot r_1 \cdot \left(p_{i,d}(t) - x_{i,d}(t)\right) + c_2 \cdot r_2 \cdot \left(p_{g,d}(t) - x_{i,d}(t)\right)$
- Updating position with: $x_{i,d}(t+1) = x_{i,d}(t) + v_{i,d}(t+1)$
- **Termination Criteria:** number of iteration to stop

PARTICLE SWARM OPTIMIZATION

- **Algorithm** PSO (a [1 .. N] , MAX_ITER, m :population size)

```
swarm ← create population (a, m)
global_best ← minimum particle
t ← 0
While t < MAX_ITER
{
        For each particles in swarm
        {
                Update velocity vector
                Calculate position vector
                Calculate position vector
                Update position
                Find fitness function value of particle
                Update local best position of particle
                Update global best position
        }
}
```

PARTICLE SWARM OPTIMIZATION

- **Updating of position:**

  Position array: [6.8, 7.2, 1.06, 4.11, 3.53, 2.32, 3.75, 5.53]

  Position array after update : [6, 7, 0, 4, 2, 1, 3, 5]

- **Calculation of Fitness Value:**

  Position array: [4, 6, 2, 3, 5, 0, 7, 1]

  Original array: [10, 20, 30, 40, 50, 60, 70, 80]

  Permutation array: [60, 80, 30, 40, 10, 50, 20, 70]

PARTICLE SWARM OPTIMIZATION

- Because these method uses random parts, it doesnt always get close to optimal solution. It gives better result than the greedy. But Vns gives better results.
- **Parameters**: size of population =60, MAXITER=200, w=0.8, c1=0.9, c2=0.9

Input list	Size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	623.4221972653426	1.28	50, 60, 20, 80, 10, 70, 40, 30
[50, 2, 4, 10, 30,80,40,15,90]	9	408.51851111049336	1.52	40, 15, 50, 4, 80, 2, 90, 30, 10
[100,35,40,50,45,70,80,90, 85,120]	10	1329.2853929163145	2.12	70, 85, 45, 100, 35, 120, 40, 90, 50, 80
[15,25,40,98,54,36,21,49, 100,140,7]	11	908.9004661718175	1.89	25, 98, 7, 54, 21, 49, 40, 100, 15, 140, 36

Ant Colony Optimization (Ant System)

- **Problem Instance:** A list of the varius radiuses size of the circles. A[1…n]
- **Initial pheromone value:** $\boldsymbol{\tau_0 = \frac{1}{n * n}}$
- **Static information** to direct the choice of the ants towards circle with minimum distance :

$$\eta_{ij} = \frac{1}{d_{ij}} \quad , \quad d_{ij} = \sqrt{(i+j)^2 - (i-j)^2}$$

- **Selecting of a circle:** Random proportional transition rule is used accordig to Ant System.
- **Pheromone increase:** $\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{if } (i,j) \in T^k(t) \\ 0 & \text{if } (i,j) \notin T^k(t) \end{cases}$,
  - $\boldsymbol{L^k(t)}$: *the width of the so*lution

Ant Colony Optimization (Ant System)

For $t = 1, \ldots, t_{max}$

For each ant $k = 1, \ldots, m$

Choose a city randomly

For each non visited city i

Choose a city j, from the list J_i^k of remaining cities, according to

$$p_{ij}^k(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha \cdot (\eta_{ij})^\beta}{\sum_{l \in J_i^k} (\tau_{il}(t))^\alpha \cdot (\eta_{il})^\beta} & \text{if } j \in J_i^k \\ 0 & \text{if } i \notin J_i^k \end{cases}$$

End For

Deposit a trail $\Delta\tau_{ij}^k(t)$ on the path $T^k(t)$ in accordance with

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{if } (i,j) \in T^k(t) \\ 0 & \text{if } (i,j) \notin T^k(t) \end{cases}$$

End For

Evaporate trails according to

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t)$$

End For

Ant Colony Optimization (Ant System)

- **Parameters**:ant size = number of circles, tmax=400,alpha=4,beta=0.25,rho=0.6,Q=1

Input list	Size	Width	Time	Order
[10,20,30,40,50,60,70,80]	8	618.5340181951838	1.6	[40, 70, 20, 80, 10, 60, 30, 50]
[50, 2, 4, 10, 30,80,40,15,90]	9	390.4721026231761	2	30, 50, 2, 80, 4, 90, 10, 40, 15
[100,35,40,50,45,70,80,90, 85,120]	10	1333.2193194215236	2.76	80, 45, 120, 40, 100, 50, 90, 35, 85, 70
[15,25,40,98,54,36,21,49, 100,140,7]	11	889.4948152617609	3.52	25, 98, 7, 54, 21, 49, 40, 100, 15, 140, 36

COMPARISON OF ALGORITHMS

- **Input list:** [10,20,30,40,50,60,70,80], **size:** 8

Algorithm	Width	Time
Brute Force	615.856371553563	0.77
Greedy	681.7266854699864	0.01
Local Search	626.7287127460996	0.01
Variable Neigborhood Seacrh	615.856371553563	1.8
Tabu Search	615.856371553563	0.57
Genetic Algorithm	623.4221972653426	0.07
Particle Swarm Optimization	623.4221972653426	1.28
Ant Colony Optimization	618.5340181951838	1.6

COMPARISON OF ALGORITHMS

- **Input list:**[50, 2, 4, 10, 30,80,40,15,90], **size: 9**

Algorithm	Width	Time
Brute Force	382.12984529746024	9.89
Greedy	567.2759427286699	0
Local Search	404.9264898629956	0.01
Variable Neigborhood Seacrh	382.12984529746024	2.78
Tabu Search	382.12984529746024	2.78
Genetic Algorithm	386.347384402607	0.07
Particle Swarm Optimization	408.51851111049336	1.52
Ant Colony Optimization	390.4721026231761	2

COMPARISON OF ALGORITHMS

- **Input list:** [100,35,40,50,45,70,80,90,85,120], **size: 10**

Algorithm	Width	Time
Brute Force	1329.2853929163145	103.17
Greedy	1401.7464378641914	0
Local Search	1334.6997472869564	0.01
Variable Neigborhood Seacrh	1329.4851658451025	4.9
Tabu Search	1333.8867633093216	1.2
Genetic Algorithm	1329.48516584510	0.09
Particle Swarm Optimization	1329.2853929163145	2.12
Ant Colony Optimization	1333.2193194215236	2.76

COMPARISON OF ALGORITHMS

- **Input list:**[15,25,40,98,54,36,21,49, 100,140,7] , **size: 11**

Algorithm	Width	Time
Brute Force	864.5793029931995	1757.11
Greedy	1401.7464378641914	0
Local Search	864.9301923525259	0.06
Variable Neigborhood Seacrh	864.9301923525259	5.64
Tabu Search	869.714162314763	0.07
Genetic Algorithm	869.714162314763	0.07
Particle Swarm Optimization	908.9004661718175	1.89
Ant Colony Optimization	889.4948152617609	3.52