Billy Saysavath

Duy Do

Phung Tran

CPSC 335 – Section 3

Fall 2015

Assignment #3

# **Exhaustive Optimization Algorithm:**

* 1. Pseudocode:

#Clock starts at this point

Dist = farthest(n,P)

bestDist [:n]= Dist

A [n]

for i in range(0, n-1):

A[i] = i;

def print\_perm( n, A[], sizeA, P, bestSet, bestDist ):

i, dist

if (n == 1):

dist = sqrt( pow( P[A[i]].x – P[A[sizeA-1]].x, 2 ) + pow( P[A[i]].y – P[A[sizeA-1]].y, 2 ))

if( disk < bestDist ):

bestDist = dist

else

for i in range (0, sizeA):

print\_perm(n – 1, A, sizeA, P, bestSet, bestDist)

if ( n % 2 == 0 ):

A[i], A[n-1] = A[n-1], A[i]

else:

A[0], A[n-1] = A[n-1], A[0]

print\_perm(n – 1, A, sizeA, P, bestSet, bestDist)

def farthest(n, P):

max\_dist=0;

i, j;

dist;

for i in range(0, n):

for j in range(0, n):

dist = (P[i].x - P[j].x)\*(P[i].x - P[j].x) + (P[i].y - P[j].y)\*(P[i].y - P[j].y)

if (max\_dist < dist):

max\_dist = dist

return sqrt(max\_dist)

* 1. Analyze:

- Line 2 calls farthest function, and it takes O(n2) steps.

- Line 3 takes n steps. Line 4 takes a constant step, so we say 1.

- Line 6 takes n steps to fill the values in a permutation array A.

- Then, the program calls print\_perm. In print\_perm, line 10 takes 2 steps.

- In an if-else statement, line 12 takes 1 step. Line 13 takes 1 + max(1, 0) step.

- Line 16, the for-loop will take sizeA steps, which equals to number of points m.

- Multiply the recursive call to print\_perm at line 17 which, as we learned in a class, takes O(n!), and 1 + max(1, 1) for if-else statement from line 18 to 21.

- Lastly, we add another O(n!) for a recursive call to print\_perm. We have the following:

We have, where m = n; therefore,

1. **Approximation algorithms**
   1. Pseudocode:
2. // allocate space for the INNA set of indices of the points
3. M = new int[n];
4. // set the best set to be the list of indices, starting at 0
5. for( i=0 ; i<n ; i++)
6. M[i] = i;
8. // Start the chronograph to time the execution of the algorithm at this point
10. // allocate space for the Visited array of Boolean values
11. Visited = new bool[n];
12. // set it all to False
13. for( i = 0; i< n; i++)
14. Visited[i] = false;
15. // calculate the starting vertex A
16. A = farthest\_point(n,P);
17. // add it to the path
18. I = 0;
19. M[i] = A;
21. // set it as visited
22. Visited[A] = true;
24. for(i=1; i<n; i++) {
25. // calculate the nearest unvisited neighbor from node A
26. B = nearest(n, P, A, Visited);
27. // node B becomes the new node A
28. A = B;
29. // add it to the path
30. M[i] = A;
31. Visited[A] = true;
32. }
34. // calculate the length of the Hamiltonian cycle
35. dist = 0;
36. for (i=0; i < n-1; i++)
37. dist += sqrt((P[M[i]].x - P[M[i+1]].x)\*(P[M[i]].x - P[M[i+1]].x) +
38. (P[M[i]].y - P[M[i+1]].y)\*(P[M[i]].y - P[M[i+1]].y));
39. dist += sqrt((P[M[0]].x - P[M[n-1]].x)\*(P[M[0]].x - P[M[n-1]].x) +
40. (P[M[0]].y - P[M[n-1]].y)\*(P[M[0]].y - P[M[n-1]].y));
42. // End the chronograph to time the loop at this point
44. def farthest\_point(n, P):
45. farthest\_point = 0
46. max\_dist, dist
47. i, j
48. max\_dist = sqrt( (P[0].x - P[n-1].x)\*(P[0].x - P[n-1].x) + (P[0].y - P[n-1].y)\*(P[0].y - P[n-1].y) )
49. for i in range[0, n-1]:
50. for j in range[0, n-1]:
51. dist = sqrt( (P[i].x - P[j].x)\*(P[i].x - P[j].x) + (P[i].y - P[j].y)\*(P[i].y - P[j].y) )
52. if (max\_dist < dist) :
53. max\_dist = dist;
54. farthest\_point = i
55. return farthest\_point
56. def nearest(n, P, A, Visited):
57. min\_dist, dist
58. nearest, i
59. for i in range[0, n]:
60. if ( !Visited[i] ):
61. min\_dist = sqrt( (P[A].x - P[i].x)\*(P[A].x - P[i].x) + (P[A].y - P[i].y)\*(P[A].y - P[i].y) )
62. nearest = i
63. for i in range[0, n]:
64. if ( !Visited[i] ):
65. dist = sqrt( (P[A].x - P[i].x)\*(P[A].x - P[i].x) + (P[A].y - P[i].y)\*(P[A].y - P[i].y) );
66. if (min\_dist > dist):
67. min\_dist = dist
68. nearest = i
70. return nearest;
    1. Analyze:

- Line 2 takes 1 step

- Line 4-5, the for-loop takes n steps

- Line 10 takes 1 step

- Line 12-13 take n steps

- Line 16 calls farthest\_point() which takes n2

* farthest\_point() begins at line 45
* Lines 46-48, and line 57 each takes 5 steps
* Line 51, outer loop, iterates i from 0 to n-1 which takes n steps
  + Line 52, inner loop, iterates i from 0 to n-1 which steps n steps
    - Line 53 takes 1 step
    - Line 54-56 takes 1 + max(2, 0) which equals 3 steps

- Line 23 iterates i from 1 to n-1 steps which takes n-2 steps. Multiply to a time calling nearest() we have n2-n

* Line 10 calls nearest which takes n time
* nearest() begins at line 33
* Lines 60, 61, and 73 each takes 1 step = 3 steps
* Line 62 iterates from 0 to n-1 takes n steps
  + Lines 63-65 take 1 + max(2,0) which is 3 steps
* Line 66 iterates from 0 to n-1 takes n steps
  + Lines 67-71 takes 1 + max(1+1+max(2,0),0) which takes 5 steps

- Lines 27-31 each take 3 steps,

- Line 35 takes 1 step.

- Line 36-38 iterates from 0 to n – 2, so it takes n-1 steps

- Line 40 takes 1 step

Altogether, we have:

Therefore,