CS 341 — Assignment 4

Kevin Carruthers (20463098) — CA002

Winter 2015

Question 1

Part A

Subproblems: Given $i = [0, \ldots, n], j = [0, \ldots, W], k = [0, \ldots, W]$, our subproblems are

$$C[i, j, k] = \max\left(\sum_{L \in A} v_L + \sum_{L \in B} v_L\right) \text{ such that } \sum_{L \in A} w_L \leq j \text{ and } \sum_{L \in B} w_L \leq k$$

Our answer, then, is $C[n, W, W]$.

Part B

Base Cases:

- $C[0, j, k] = 0, \forall j, \forall k$
- $C[i, 0, 0] = 0, \forall i$

Part C

Recursive Formula:

$$C[i,j,k] = \begin{cases} \max\left(C[i-1,j,k], C[i-1,j-w_i,k]+v_i, C[i-1,j,k-w_i]+v_i\right) & \text{if } j \geq w_i \text{ and } k \geq w_i \\ \max\left(C[i-1,j,k], C[i-1,j-w_i,k]+v_i\right) & \text{if } j \geq w_i \\ \max\left(C[i-1,j,k], C[i-1,j,k-w_i]+v_i\right) & \text{if } k \geq w_i \\ C[i-1,j,k] & \text{otherwise} \end{cases}$$

Justification: if we have enough space in either knapsack, we can recurse with the maximum between either putting the current item in knapsack one or two, or by placing it in neither. If only one of the knapsakcs will fit this item, we simply compare putting that item in the knapsack or not. If the item would not fit in either, we simply ignore it. For each of these options, we simply take the recursion that gives the largest overall value.

Part D

```
1.  for j = 0 to W:
2.    for k = 0 to W:
3.      C[0, j, k] = 0
4.  for i = 0 to n:
5.    C[i, 0, 0] = 0
6.  for i = 1 to n:
7.    for j = 1 to W:
8.      for k = 1 to W:
9.        if j >= w_i and k >= w_i:
10.         C[i, j, k] = max(C[i-1, j, k], C[i-1, j - w_i, k] + v_i, C[i-1, j, k - w_i] + v_i)
11.       else if j >= w_i:
12.         C[i, j, k] = max(C[i-1, j, k], C[i-1, j - w_i, k] + v_i)
13.       else if k >= w_i:
14.         C[i, j, k] = max(C[i-1, j, k], C[i-1, j, k - w_i] + v_i)
15.       else:
16.         C[i, j, k] = C[i-1, j, k]
17. return C[n, W, W]
```

Part E

Note that we define $\texttt{max}\big(\texttt{(A, B, c), (D, E, f)}\big)$ where A, B, D, E are sets and c, f are integers as returning (X, Y, z) such that z is the largest integer in the list (eg. returns (A, B, c) if $c > f$ and (D, E, f) otherwise).

```
1.  for j = 0 to W:
2.    for k = 0 to W:
3.      C[0, j, k] = ({}, {}, 0)
4.  for i = 0 to n:
5.    C[i, 0, 0] = ({}, {}, 0)
6.  for i = 1 to n:
7.    for j = 1 to W:
8.      for k = 1 to W:
9.        if j >= w_i and k >= w_i:
10.         C[i, j, k] = max(C[i-1, j, k], C[i-1, j - w_i, k] + v_i, C[i-1, j, k - w_i] + v_i)
11.         if C[i-1, j - w_i, k] + v_i == C[i, j, k]:
12.           C[i, j, k][0].add(i)
13.         else if C[i-1, j, k - w_i] + v_i == C[i, j, k]:
14.           C[i, j, k][1].add(i)
15.       else if j >= w_i:
16.         C[i, j, k] = max(C[i-1, j, k], C[i-1, j - w_i, k] + v_i)
17.         if C[i-1, j - w_i, k] + v_i == C[i, j, k]:
18.           C[i, j, k][0].add(i)
19.       else if k >= w_i:
20.         C[i, j, k] = max(C[i-1, j, k], C[i-1, j, k - w_i] + v_i)
21.         if C[i-1, j, k - w_i] + v_i == C[i, j, k]:
22.           C[i, j, k][1].add(i)
23.       else:
24.         C[i, j, k] = C[i-1, j, k]
25. return (C[n, W, W][0], C[n, W, W][1])
```

Part F

- Lines 1-3: $\theta(W^2)$
- Lines 4-5: $\theta(n)$
- Lines 6-24: $\theta(nW^2)$
- Line 25: $\theta(1)$
- Overall: $\theta(W^2 + n + nW^2 + 1) = \theta(nW^2)$

The space complexity is bounded by $O(n^2W^2)$, since the nW^2 items may contain subsets of up to n (combined) items.

Question 2

Part A

Subproblems: Given $i = [0, \ldots, n], j = [0, \ldots, k]$, our subproblems are

$$C[i, j] = \min\Big(C[i-1, j-1], C[i-1, j] + AreaChange\Big)$$

where $AreaChange$ is the amount of area increase which is caused by adding our new point to the rightmost rectangle. Note that our input must be sorted.

Our answer, then, is $C[n, k]$.

Part B

Base Case:

- $C[i, j] = 0, \forall i <= j$

Part C

Recursive Formula:

$$C[i, j] = \begin{cases} 0 & \text{if } j >= i \\ \min\Big(C[i-1, j-1], C[i-1, j] + AreaChange\Big) & \text{otherwise} \end{cases}$$

Justification: if j is larger than, or equal to, i we have enough rectangles to cover our points with "thin" rectangles of area zero. Otherwise, we can recurse with either

- using the solution for $C[i-1, j-1]$, assume that we have removed one rectangle from the solution and used it to create a "thin" rectangle for our new point, or

- using the solution for $C[i-1, j]$, expand the rightmost rectangle to cover our new point

In either case, we recurse and take the minimum of these options.

Part D

```
1.  for j = 0 to k:
2.    for i = 0 to n:
3.      if i <= j:
4.        C[i, j] = ({(i.x, 0), (i.x, i.y)}, 0)
5.  for i = 0 to n:
6.    for j = 0 to k:
7.      if j < i:
8.        newRectangle = C[i-1, j][0]
9.        newRectangle[1][1] = i.y
10.       if newRectangle[1][0] > i.x:
11.         newRectangle[1][0] = i.x
12.       areaChange = area(newRectangle) - area(C[i-1, j][0])
13.       if C[i-1, j-1][1] <= C[i-1, j][1] + areaChange:
14.         C[i, j] = ({(i.x, 0), (i.x, i.y)}, C[i, j][1])
15.       else:
16.         C[i, j] = (newRectangle, C[i-1, j][1] + areaChange)
17. return C[n, k][1]
```

Part E

```
1.  for i = 0 to n:
2.    for j = 0 to k:
3.      if j = 0:
4.        C[i, j] = ({}, 0)
5.      else if i <= j:
6.        C[i, j] = (C[i-1, j][0] + {(i.x, 0), (i.x, i.y)}, 0)
7.  for i = 0 to n:
8.    for j = 0 to k:
9.      if j < i:
10.       newRectangle = C[i-1, j][0]
11.       newRectangle[1][1] = i.y
12.       if newRectangle[1][0] > i.x:
13.         newRectangle[1][0] = i.x
14.       areaChange = area(newRectangle) - area(C[i-1, j][0])
15.       if C[i-1, j-1][1] <= C[i-1, j][1] + areaChange:
16.         C[i, j] = (C[i, j][0] + {(i.x, 0), (i.x, i.y)}, C[i, j][1])
17.       else:
18.         C[i, j] = (C[i, j][0] + newRectangle, C[i-1, j][1] + areaChange)
19. return C[n, k][0]
```

Part F

- Lines 1-6: $\theta(nk)$
- Lines 7-18: $\theta(nk)$
- Line 19: $\theta(1)$
- Overall: $\theta(nk + nk + 1) = \theta(nk)$

The space complexity is bounded by $O(nk^2)$, since there are nk items with at most k elements in their array.

Question 3

We can create rectangles by expanding outwards from every intersection between vertical and horizontal lines until either another line is reached or another rectangle is reached. For example: if we have horizontal lines $\{(0,0),(10,0)\}$, $\{(0,5),(5,5)\}$, and $\{(0,10),(10,10)\}$, as well as vertical line $\{(5,2.5),(5,5)\}$, we can imagine rectangles designated by opposite corners $\{(0,0),(5,5)\}$, $\{(5,0),(10,10)\}$, and $\{(0,5),(5,10)\}$. There must also exist rectangles on the "outside" of our graph, eg. $\{(-\infty,-\infty),(0,\infty)\}$, $\{(0,10),(10,\infty)\}$, $\{(0,-\infty),(10,0)\}$, and $\{(10,-\infty),(\infty,\infty)\}$. While creating these rectangles, we connect them to adjacent rectangles if and only if there is no line entirely between the two rectangles (ie. in the above example, $\{(0,0),(5,5)\}$ and $\{(5,0),(10,10)\}$ are considered adjacent despite the half-line between them) In this way, we can ensure we have at most n^2 (since there are n horizontal and vertical lines, and the rectangles cannot overlap) rectangles. Note that these rectangles are not necessarily unique, but any set of possible rectangles will be correct for our purposes.

Using these rectangles as if they were graph nodes, we can perform a search as follows: beginning at the node containing point s, search downward until we either find a node containing point t or have no more nodes to search. If we find t, clearly there is a path connecting the two points, otherwise there must not be. This search must be bounded by $O(n^2)$ time, since we only have at most n^2 rectangles (ie. nodes) to search.

Thus we have found the presence or lack of a path connecting s and t in $O(n^2)$ time.

Question 4

My code is as follows:

```
#include <cmath>
#include <iostream>
#include <queue>
#include <vector>


#define SUCCESS 1
#define FAIL 0

#define DEFAULT_PARTITION -1


struct Disk {
    int x;
    int y;
    int radius;

    int partition;
};
```

```
int inRange(const Disk left, const Disk right) {
    double xDist = pow(left.x - right.x, 2);
    double yDist = pow(left.y - right.y, 2);
    double range = pow(left.radius + right.radius, 2);

    return xDist + yDist <= range;
}

int BFS(std::vector<Disk> *disks, int idx) {
    std::queue<int> q;
    q.push(idx);

    disks->at(q.front()).partition = 1;
    while (!q.empty()) {
        int curr = q.front();
        q.pop();

        for (int i = 0; i < disks->size(); ++i) {
            if (i == curr) {
                continue;
            } else if (inRange(disks->at(i), disks->at(curr))) {
                if (disks->at(i).partition == DEFAULT_PARTITION) {
                    disks->at(i).partition = disks->at(curr).partition == 1 ? 2 : 1;
                    q.push(i);
                } else if (disks->at(i).partition == disks->at(curr).partition) {
                    return FAIL;
                }
            }
        }
    }

    return SUCCESS;
}

int partition(std::vector<Disk> *disks) {
    for (int i = 0; i < disks->size(); ++i) {
        if (disks->at(i).partition == DEFAULT_PARTITION) {
            int retValue = BFS(disks, i);
            if (!retValue) {
                return FAIL;
            }
        }
    }

    return SUCCESS;
}
```

```
int main(int argc, char *argv[]) {
    int n;

    std::vector<Disk> *disks = new std::vector<Disk>();
    for (std::cin >> n; n > 0; --n) {
        Disk d;

        std::cin >> d.x;
        std::cin >> d.y;
        std::cin >> d.radius;
        d.partition = DEFAULT_PARTITION;

        disks->push_back(d);
    }

    int exists = partition(disks);
    if (!exists) {
        std::cout << "0" << std::endl;
    } else {
        for (int i = 0; i < disks->size(); ++i) {
            std::cout << disks->at(i).partition;
            if (i < disks->size() - 1) {
                std::cout << " ";
            }
        }
        std::cout << std::endl;
    }

    return 0;
}
```