Relatório do Problema 1

Equipa: CDI

1. Algorithm Description

O algoritmo implementado começa sempre por colocar o primeiro device. Após o posicionamento do primeiro device numa coordenada inicia-se a função recursiva (*recursive)*. Esta função recebe como argumentos o índice do "real device", o número de intersecções no actual momento da recursão e o número de linhas criadas entre os colliders até ao momento. Este *real_devices* é um array dos dispositivos que se encontram realmente em utilização, sendo que existe um pré processamento dos mesmos no momento da leitura do input. O caso base consistirá então na colocação do primeiro "real device", explicado anteriormente em todas as posições possíveis para os mesmos. Esta posição é marcada como ocupada, e é então chamada a função *recursive(1,0,0)*, que recebe 3 parâmetros, onde o primeiro corresponde ao índice do primeiro device colocado, o segundo é referente ao número de intersecções actuais, onde no caso base será 0, dado que ainda só existe um dispositivo colocado, assim como o número de linhas criadas entre os dispositivos colocados, sendo este o terceiro parâmetro da função. Relativamente à função recursiva em si, identificaram-se as seguintes condições de rejeição:

A. Se o número de intersecções até ao momento já for superior ao *best_case* encontrado.
B. Se o número de intersecções até ao momento + o número de intersecções do caso em questão for maior do que o *best_case,* então já não é testado o device actual nesse respectivo place.

Funcionamento da recursiva:

```
1   RECURSIVE(INDEXRDEV,NINT,NLINES)
2       if NINT >= best_case
3           return
4       //all placed
5       if INDEXRDEV = NREALDEV          2
6           if NINT < best_case
7               best_case = NINT
8           return
9       dev_atual = real_devs[INDEXRDEV]
10      for i=1 to ncoord
11          if coord_used[i] = false
12              coord_used[i] = true
13              place_of_dev[dev_atual] = i
14
15              count_int = 0
16              aux_lines = NLINES
17              //looking for the adjacencies of the device
18              for a = 0 to size_of_adjacencies
19                  adj_dev = adjacencies[dev_atual]
20                  if place_of_dev[adj_dev] != 0
21                      //create line with the id's
22                      source = coords[place_of_dev[dev_atual]][2]
23                      target = coords[adj_dev][2]
24                      count_int += COUNTINTERSECTIONS(SOURCE,TARGET,NLINES)
25                      //bigger number to best case dont try this place
26                      if NINT + count_int >= best_case    3
27                          flag = false
28                          break
29                      lines[aux_lines][0] = source
30                      lines[aux_lines][1] = target
31                      aux_lines +=1
32              if flag == true
33                  RECURSIVE(INDEXRDEV+1, NINT + count_int, aux_lines)
34              place_of_dev[dev_atual] = 0
35              coord_used[i] = false
```

O device actual é associado a um par ordenado que não se encontre ocupado. De seguida, são percorridas as adjacências desse device, e criadas as ligações entre o device actual e cada um dos seus adjacentes. A cada linha criada, é verificado quantas das linhas criadas até aquele momento da recursão esta intersecta, actualizado o valor do *count_int.* Quando todos os devices tiverem sido colocados, é verificado se o número obtido é melhor do que o *best_case* actual. Caso essa condição se verifique, o *best_case* é actualizado. (condição de paragem 2). De maneira a tentar optimizar esta implementação, para além das condições de rejeição acima mencionadas, existe um vector (*intersections_saved*) que serve para guardar os cálculos

efectuados. Assim, em vez de ser continuamente calculado se existe uma interseção entre duas linhas, se este já tiver sido calculado anteriormente, será apenas necessário aceder ao seu valor, e não realizar o cálculo a cada linha criada.

2. Intersection Procedure

Para determinar as intersecções utilizou-se a seguinte abordagem:
Como função principal, tem-se a função *doIntersect*. Este abordagem é implementada de acordo com o seguinte princípio: dois segmentos de recta (sendo o segmento 1 p1,q1 e o segmento 2 p2,q2) intersectam-se se e só se obedecerem a apenas uma das seguintes condições:

A. Caso Geral

A. (p1,q1,p2) e (p1,q1,q2) têm orientações diferentes e (p2,q2,p1) e (p2,q2,q1) têm orientações diferentes.

Para verificar o caso geral, é utilizada a função *orientation,* que encontra a orientação de um triplo ordenado (clockwise, counterclockwise ou colinear).

A. Caso Especial

A. (p1,q1,p2), (p1,q1,q2), (p2,q2,p1) e (p2,q2,q1) são todos colineares.
B. (p1x,q1x) intersecta com (p2x,q2x).
C. (p1y,q1y) intersecta com (p2y,q2y).

Para a verificação dos casos especiais é utilizada a função *onSegment,* que com recurso a 3 pontos colineares (p,q,r) confirma se um ponto “q” se encontra ou não no segmento de recta “qr”.
Por fim, de maneira a cumprir com os requisitos do cálculo de interseções, existe uma condição na função *countIntersections* que verifica se os colisores partilham ou não dispositivos. Se o fizerem, então essa interseção não é “contabilizada”.
Este procedimento foi baseado na primeira referência apresentada na secção 6.

3. Data Structures

No início desta implementação, optou-se por implementar a estrutura ponto, que continha a sua coordenada x, a sua coordenada y, e o seu id, e a estrutura linha, constituída por 2 pontos, uma source e um target. Posteriormente, e de forma a tentar uma abordagem mais leve, optou-se por trabalhar apenas com arrays e com um vector. De modo a controlar os dispositivos e coordenadas em utilização, foram criados dois arrays de formato booleano, respectivamente *device_to_use* e *coordinate_used.*

Para além disso, também são utilizados os seguintes arrays de inteiros:

1. *coordinates:* guarda as coordenadas dos dispositivos. Índice 0 corresponde ao x, índice 1 ao y, e índice 2 ao seu id.
2. *real_devices:* guarda os id’s dos dispositivos que se encontram realmente em utilização.
3. *place_of_device:* utilizado para guardar as associações de dispositivos aos pontos da sala.
4. *lines:* contém os colliders, ou seja, os pares de pontos *source* e *target,* onde a source é o índice 0 e o target o índice 1.
5. *intersections_saved:* guarda o valor do cálculo das intersecções, para evitar o cálculo constante das mesmas (referência à secção 1).

Existe ainda o vector de inteiros *adjacencies,* que guarda as associações entre os sources e os targets dos colliders.

4. Correctness

Como abordagem inicial para este problema, implementou-se uma solução brute-force, onde era feito o cálculo de todas as combinações possíveis para colocar todos os devices nos vários pontos da sala, e posteriormente calculado o número de interseções de cada combinação encontrada. No entanto, foi rapidamente perceptível que esta não seria uma solução viável, dado que não corria em tempo útil em casos de teste com elevada complexidade.

Portanto, concluiu-se que a abordagem correcta seria uma abordagem recursiva, com recurso a backtracking, dado que, utilizando condições de rejeição e de paragem, é possível cortar-se na realização de muitos passos, diminuindo bastante a complexidade temporal do algoritmo. Para este problema, identificaram-se situações para as quais seria dispensável que a recursiva continuasse, tais como:

A. o best_case ser 0, visto não existir nenhum caso melhor.

B. continuar a executar a recursiva se o número de interseções já for superior ao best_case encontrado.

O processamento das interseções num array de 4 dimensões, em vez do contínuo cálculo das mesmas, fez com que este algoritmo diminuísse bastante o seu tempo de execução, tornando-o numa abordagem muito mais eficiente.

Por fim, e de maneira a tornar este algoritmo ainda mais a eficiente, foi implementada uma função que calculava um *best_case* inicial, onde o objectivo seria eventualmente apanhar logo um valor muito favorável ao problema, de forma a diminuir significativamente o número de iterações da recursiva. Esta função melhorou pouco significativamente a complexidade temporal do algoritmo, no entanto, obteve melhor performance do que o original submetido, para os casos que tinham um máximo de 2 segundos.

Para efeitos de verificação da solução foi criada a função *print_best,* cujos resultados são apresentados na seguinte tabela:

Caso de teste	Verificação
4 / 1 2 / 2 2 / 3 2 / 4 2 / 4 6 / 1 2 / 1 3 / 1 4 / 2 3 / 2 4 / 3 4	device: 1 at place 1 device: 2 at place 2 device: 3 at place 3 device: 4 at place 4
9 / 3 2 / 4 2 / 5 2 /3 3 / 4 3 / 5 3 / 3 4 / 4 4 / 5 4 / 5 10 / 1 2 / 1 3 / 1 4 / 1 5 / 2 3 / 2 4 / 2 5 / 3 4 / 3 5 /4 5	device: 1 at place 1 device: 2 at place 2 device: 3 at place 5 device: 4 at place 6 device: 5 at place 7

14 / 0 1 / 2 1 / 3 1 / 0 2 / 1 2 / 2 2 / 3 2 / 0 3 / 1 3 / 2 3 / 3 3 / 0 4 / 2 4 / 3 4 / 8 16 / 1 5 / 1 6 / 1 7 / 1 8 / 2 5 / 2 6 / 2 7 / 2 8 / 3 5 / 3 6 / 3 7 / 3 8 / 4 5 / 4 6 / 4 7 / 4 8	device: 1 at place 1 device: 2 at place 5 device: 3 at place 10 device: 4 at place 14 device: 5 at place 3 device: 6 at place 6 device: 7 at place 9 device: 8 at place 12

5. Algorithm Analysis

A função *recursive* tem uma complexidade, no pior caso, de:

$$ncolliders * ncoordenadas$$

Onde ncolliders é o número de colliders adjacentes ao device actual, e ncoordenadas o número de pontos da sala.
Para efeitos de teste imprimiu-se o número de vezes que cada caso de entrou na condição de rejeição, **if NINT + count_int >= best_case,** resultando na seguinte tabela:

Caso de teste	Número de vezes que entrou na condição de rejeição
4 / 1 2 / 2 2 / 3 2 / 4 2 / 4 6 / 1 2 / 1 3 / 1 4 / 2 3 / 2 4 / 3 4	23
9 /3 2 /4 2 / 5 2 /3 3 / 4 3 / 5 3 / 3 4 / 4 4 / 5 4 / 5 10 / 1 2 / 1 3 / 1 4 / 1 5 / 2 3 / 2 4 / 2 5 / 3 4 / 3 5 /4 5	15117
14 / 0 1 / 2 1 / 3 1 / 0 2 / 1 2 / 2 2 / 3 2 / 0 3 / 1 3 / 2 3 / 3 3 / 0 4 / 2 4 / 3 4 / 8 16 / 1 5 / 1 6 / 1 7 / 1 8 / 2 5 / 2 6 / 2 7 / 2 8 / 3 5 / 3 6 / 3 7 / 3 8 / 4 5 / 4 6 / 4 7 / 4 8	11716652

Com isto conseguimos verificar que, muito provavelmente, o worst case tem uma probabilidade baixíssima de se verificar.
Fora da função recursiva, o caso base deste algoritmo tem no, no pior caso, complexidade de:

$$npontos + ndevices$$

6. References

https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
S.Skiena and M.Revilla, Progamming Challenges, 2003.T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., 2009.

Membros da equipa que implementaram esta abordagem:

Número de Estudante: 2015231975 Nome: Diogo Alexandre Santos Amores
Número de Estudante: 2015233281 Nome: Maria Inês António Roseiro
Número de Estudante: 2015262771 Nome: Pedro Miguel Gonçalves Chicória