Report for Tron.py programme
URL: https://www.hackerrank.com/challenges/tron

In the Tron problem, each of the two players walks on a grid of size 15*15 (with 1-unit each side as the boundary) like a snake, or a motorbike which leaves a wall behind as he/she moves. The objective is to cut off the walking path of the opponent.

Three solutions to the problem

This report contains the information to three solutions to the problem, namely basic, improved and advanced. The three solutions have incremental sophistication and performance. The basic version was finished in 65 minutes. The improved version and advanced version were developed together, the components in the improved and advanced version took around 2 hours and 8 hours of extra work. Running the three versions in hackerrank.com returned a rank of 121, 30 and 23 respectively.

Basic version

There are four possible moves for oneself and for the opponent.
  [UP, LEFT, DOWN, RIGHT]
making 4*4=16-combinations in the next step.

The design of the code looks at all 16 cases. For each case, a Voronoi diagram is generated to partition the 15*15 grid into the **self** part and the **opponent** part. Two centroids represent the locations of **self** and **opponent**. The Voronoi diagram is a partitioning of the grid into regions based on the distance to these two centroids. The goal is to choose an optimal next move such that the **self** partition of the Voronoi diagram will be maximised, the **opponent** partition will be minimised (and in the advanced version, the inaccessible partition due to wall blockade will be minimised).

The basic version contains two core components, and is represented by the pseudo-code below. A textual summary will follow.

1. Derivation of the Voronoi diagram

```
gen_voronoi(self_i, self_j, opponent_i, opponent_j) {
   for every position i,j {
      if  (i,j) is closer to (self_i,self_j) {
         v[i][j] = 1;
      } else if (i,j) is closer to (opponent_i,opponent_j) {
         v[i][j] = -1;
      } else {
         v[i][j] = 0;
   }}
   return v;
}
```

* N=dimension of grid=15

2. Reward calculation on the Voronoi diagrams in 16 possible cases

```
for self.nextmove = [UP,LEFT,DOWN,RIGHT] {
    if self.nextmove steps on a wall or boundary {
        s=Assign minimum output score (-999) to forbid this move
    } else {
        for opponent.nextmove = [UP,LEFT,DOWN,RIGHT] {
            Sum the N*N v matrix to get a voronoi-sum
        }
        s=Sum the voronoi-sum over 4 opponent directions
    }
}
Decide DIRECTION where s[DIRECTION] gives the maximum sum
```

The inputs of the gen_voronoi() function are positions of **self** and **opponent**. The positions can be the current position, or hypothetic position in future moves. The output of gen_voronoi() is a 2-D array of sized 15*15. It has value 1 when the point is closer to **self** , value -1 vice versa, and value 0 for an equidistant point.

In the reward calculation stage, for each of the 16 **self** and **opponent** move-combinations a Voronoi diagram was generated. The reward of the move was then computed by summing the value of the whole Voronoi diagram. When a move was known to be forbidden (moving into a wall and backtracing), the Voronoi diagram would not be generated, and a default move penalty -999 was set.

The final decision of next move for **self** was decided by summing the reward of all possible **opponent** moves given the target **self** move.

[Results (Basic version)]
The code was tested on hackerrank.com platform and the results are shown in the following table:-

URL:
https://www.hackerrank.com/challenges/tron/submissions/game/16612130
Score: 37.23
Won: 16
Tie: 11
Lost: 29
Rank: 121/249

[Schedule (Basic version)]

Problem reading and understanding	20 mins
Coding	45 mins
Test and debug	20 mins
(Total)	65 mins

Additional challenges

The following recorded the additional work to improve the performance of the code. The improved and advanced version were developed together, the components in improved solution took approximately 2 hours of extra work. In the *improved* version, an N-step look-ahead algorithm was implemented in the search of optimal move direction. It returned a better performance in the hackerrank submission (with rank raised from 121 to 30). In the *advanced* version, approximately 8 hours of extra work was committed to establish a refined zone from which the Voronoi diagram was constructed. Two trials were run on hackerrank and the final ranks were 21 and 23 respectively.

Improved version

In the basic version, the direction of next move depends on a score derived by the Voronoi diagram of one and only one next move. In the improved version, the algorithm looks further ahead to model **self** and **opponent** moves S steps ahead (S set to 3 in the current implementation).

In terms of algorithm, the characteristics include:

- A Data class [Node] to represent the Tree of height S to model the next S moves (S is set to 3 in this simulation)
- Members of [Node]:
    - s_i, s_j, o_i, o_j: Position of **self** and **opponent** as a result of the hypothesised move
    - vor: Reward of the hypothesised move
    - link: An ordered array of links to the child node to represent next move. From the order one can deduce the move direction of **self** and **opponent** in the next move
- Methods of [Node]:
    - genneighbour():
      Given the move direction of **self** and **opponent**, and the move reward (vor), construct a child node and a link to be referenced by the parent
    - gen_voronoi_sum()
      Compute the move reward by summing the Voronoi diagram
    - genneighbour()
      Initialise a leave with the current node, and move direction
    - stepadvance():
      Advance one move in all direction for all currently tracked move combinations. Compute the move reward with function gen_voronoi_sum() and extend the tree by adding new leaves to the current leaf with function genneighbour()
    - totalvor()
      Use a depth-first approach to compute the maximum path reward of a subtree

The core of the algorithm was stepadvance(), which grew the height of search tree by 1. It is represented by the pseudocode below:

```
function stepadvance(node) {
   if node is leave {
      for (all move = [all possible **self** and **opponent** next moves]) {
         gen_voronoi_sum(node,move)      # compute move reward
         leaf = genneighbour(node,move)          # generate neighbour
         node.links.append(leaf)
   } } else  {                         # node is not leave, depth-first transversal
      for (link = [all node.links]) {
         stepadvance(link)
} } }
```

[Results (Improved version)]

```
URL:
https://www.hackerrank.com/challenges/tron/submissions/game/1661323
4
Score: 47.78
Won: 9
Tie: 41
Lost: 6
Rank: 30/249
```

Advanced version

In the basic and improved version, the Veronoi diagram was constructed based on the assumption that all locations were valid except the 4 sides of the grid. As the players moved and built walls in the grid, this assumption became invalid. The further extension is covered in limited details in this report. Any further enquiries on the detailed implementation are welcomed.

In the advanced version, a 2-D array called zone was constructed. This array mapped every location in the grid to a zone index. Any two points which were connected without the barrier of any walls were assigned to the same zone.

```
zone[self.i][self.j] == zone[opponent.i][opponent.j]
➔ self and opponent in the same zone
zone[i][j] != zone[self.i][self.j]
➔ a hypothesised point (i,j) is not connected to self  (blocked by a wall)
```

In the computation of Voronoi diagram, a different path reward strategy was used. With the zone array, the first component of deriving Voronoi diagram in the basic version was modified, and shown below. The extra conditions and assignments were bolded and underlined.

```
gen_voronoi_wblocks(zone,self_i, self_j, opponent_i, opponent_j) {
  for every position i,j {
    if (i,j) and self in different zones {
       v[i][j] = -1
    } else if (i,j), self and opponent are in the same zone {
       if  (i,j) is closer to (self_i,self_j) {
          v[i][j] = 1;
       } else if (i,j) is closer to (opponent_i,opponent_j) {
          v[i][j] = -1;
       } else {
         v[i][j] = 0;
    } else { v[i][j] = 0;  }
```

Challenge: The challenge lies on constructing the zone index with low time complexity. This was implemented by the function drawzones() in the Zones() class. A bottom-up clustering strategy was adopted. First the algorithm drew a zone for every point in the grid. Then pairwise zone comparison is done to merge contagious zones. When a zone has more than one neighbour and the neighbours contain different zone indices, the corresponding zone indices are enqueued in a todo queue, where a second-pass zone merging algorithm, implemented by a mergezones() function, will merge the zones.

[Results (Advanced version: 2 trials)]

URL:
https://www.hackerrank.com/challenges/tron/submissions/game/16609338,
https://www.hackerrank.com/challenges/tron/submissions/game/16614454
Score: 49.89, 49.42
Won: 13,13
Tie: 42, 42
Lost: 1, 1
Rank: 21/249, 23/249

Further improvements

- S can be increased to look ahead more steps in the future
- Depth first search in Node.totalvor() and Node.stepadvance() functions can be changed to breadth first search and early stopping criteria can be added (e.g. stop the search if the voronoi sum falls under the threshold) to speech up the search
- Adapting the path reward computation equation [gen_voronoi_wblocks()]
- Voronoi diagram can be updated dynamically by updating the forbidden locations when the hypothesised move advances. A test implementation was tried, but it happened that the Voronoi sum computed in different move directions have different zone sizes and the max() function in totalvor() failed to pick the meaningful link, resulting a 3-4 point degradation in hackerrank competition.