The Deferred: CMIMC AI Round

Source code available at iCalculated/CMIMC2021AI

Overview:

- Bet: Submits auctioned card plus one (and 14→2) in the first round. Keeps a log of the threshold to win a card and which cards had ties, then exploits in the following round assuming that opponents make the same bids (wrt card auctioned).
- Scotty: Both algos construct a heatmap, Scotty surfs the gradient, favoring open paths, and Trapper primarily works from the edges, leaving exits open until the last second, but will block adjacent squares when convenient.
- Spaceships: Goes in a circle, dances about using increasingly specific heuristics to avoid destruction.

Bet: Try Your Best, or Give Up.

Our first few algorithms did things such as mirror the card being auctioned or add one to it and they performed rather well to start. That didn't last long, so we moved on to an iterated strategy.

Information Stored

Our algo tracked several things, as can be seen in the `__init__` function:

```
def __init__(self):
    # Tracking
    self.past = defaultdict(list)     # dictionary of cards played with auctioned cards as keys
    self.other_hands = []             # list of opponent's hands
    self.previous_auction = -1        # the previous card auctioned

    # Processing
    self.win = defaultdict(list)      # dict of values needed to win by card
    self.tie = defaultdict(list)      # dict of values to match the base winner (0 if already tied)
    self.bids = defaultdict(int)      # plan for the round, dict[auction] = bid
```

Sad Strategy and Recording

During the first round, there is no prior information, so the algorithm returns

```
2 if card == 14 else card + 1
```

bidding one higher than the card and giving up on 14. Over the course of the round, it keeps track of its opponents' bids.

```
past_cards = []
if len(hand) != 13:
    for curr, prev in zip(others, self.other_hands[-1]):
        past_cards.extend(set(prev).difference(set(curr)))
    self.past[self.previous_auction].append(past_cards)
if len(hand) == 1:
    self.past[card].append([x for sub in others for x in sub])
self.previous_auction = card
self.other_hands.append(copy.deepcopy(others))
```

Processing

The magic in the strategy is processing the information once it has a complete `past` dictionary. The processing has two stages, the first of which is threshold calculation: the algorithm figures out what it would have needed to win or tie in retrospect, with an edge case that if its opponents already tie then it just needs to submit 0.

```
def calculate_thresholds(self):
    for card, bids in self.past.items():
        self.win[card].append(max(bids[-1]) + 1)
        self.tie[card].append(max(bids[-1]) if bids[-1][0] != bids[-1][1] else 0)
```

Once it has the thresholds, it calculates bids for each auction, iterating down from the highest cards. It figures out if it can win the round, then bids the highest remaining card[1] if it can and its lowest card if it can't.

[1]This is rather naive, it could have potentially won more rounds if it saved higher cards for where it needed them, but it won so much that we didn't bother changing it.

Trap the Scotty Dog: Geneva Suggestion

Prior to any actual strategy, our greatest victory was implementing the ability to add and subtract tuples:[2]

```
Point = namedtuple('Point','x y')
Point.__add__ = lambda a, b: Point(a[0] + b[0], a[1] + b[1])
Point.__sub__ = lambda a, b: Point(a[0] - b[0], a[1] - b[1])
```

"Dijkstra" heatmap

Both algorithms used a BFS heatmap stored in a helper class clumsily named `dijkstra`:

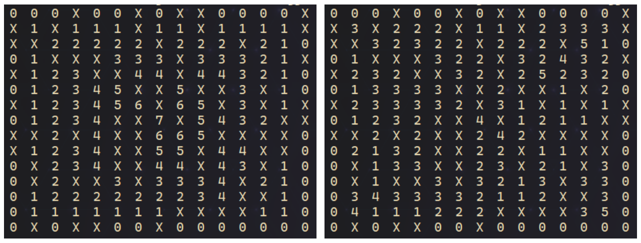
```
class dijkstra:
    def valid(self, x, y):                 # checks if a square is blocked
    def neighbors(self, point):            # finds valid neighbors of any square
    def generate_heatmap(self, start):     # flood fills from Scotty (DEPRECATED!)
    def generate_backmap(self):            # flood fills form exits
    def generate_tree(self, start):        # generated a tree of all possible routes (FAILURE!)
    def generate_node_rank(self):          # counts square neighbors that are along a path
    def find_path(self, start):            # surfs the gradient
    def find_ultimate_path(self, start):   # tries multiple surfing directions
```

Originally, we attempted to generate a heatmap from the dog. Obviously that failed, any movement moves further from the dog. I have not yet refactored, so all references to `heatmap` actually use the result of `generate_backmap()` which checks how far each square is from an exit.[3]

Maintenance

Both algorithms start with maintenance, updating `self.pathing` which is a `dijkstra` object. We locate scotty, update the board, then generate the heatmap and check how many "useful neighbors"[4] each square has, which becomes its `node_rank`. The `node_rank` system was a big decision for us, we were trying to decide between something more computational such as determining which nodes in a tree of paths are the most visited and a rough heuristic such as "how open is this path?" The tree was far too slow, so that decision was made for us.

```
x,y = self.find_scotty(board)
self.pos = Point(x,y)
self.pathing.board = board
self.pathing.heatmap = self.pathing.generate_backmap()
self.pathing.generate_node_rank()
```



```
0 0 0 X 0 0 X 0 X X 0 0 0 0 X
X 1 X 1 1 1 X 1 1 X 1 1 1 1 X
X X 2 2 2 2 2 X 2 2 2 X 2 1 0
0 1 X X X 3 3 3 X 3 3 3 2 1 X
X 1 2 3 X X 4 4 X 4 4 3 2 1 0
0 1 2 3 4 5 X X 5 X X 3 X 1 0
X 1 2 3 4 5 6 X 6 5 X 3 X 1 X
0 1 2 3 4 X X 7 X 5 4 3 2 X X
X X 2 X 4 X X 6 6 5 X X X X 0
X 1 2 3 4 X X 5 5 X 4 4 X X 0
0 X 2 3 4 X X 4 4 X 4 3 X 1 0
0 X 2 X X 3 X 3 3 3 4 X 2 1 0
0 1 2 2 2 2 2 2 2 3 4 X X 1 0
0 1 1 1 1 1 1 1 X X X X 1 1 0
0 X 0 X X 0 0 X 0 0 0 0 0 0 0
```

```
0 0 0 X 0 0 X 0 X X 0 0 0 0 X
X 3 X 2 2 2 X 1 1 X 2 3 3 3 X
X X 3 2 3 2 2 X 2 2 2 X 5 1 0
0 1 X X X 3 2 2 X 3 2 4 3 2 X
X 2 3 2 X X 3 2 X 2 5 2 3 2 0
0 1 3 3 3 3 X X 2 X X 1 X 2 0
X 2 3 3 3 3 2 X 3 1 X 1 X 1 X
0 1 2 3 2 X X 4 X 1 2 1 1 X X
X X 2 X 2 X X 2 4 2 X X X X 0
0 2 1 3 2 X X 2 2 X 1 1 X X 0
0 X 1 3 3 X X 2 3 X 2 1 X 3 0
0 X 1 X X 3 X 3 2 1 3 X 3 3 0
0 3 4 3 3 3 3 2 1 1 2 X X 3 0
0 4 1 1 1 2 2 2 X X X X 3 5 0
0 X 0 X X 0 0 X 0 0 0 0 0 0 0
```

Figure 1: The backmap (left) and the node ranks (right). Note how many useful neighbors corner nodes have.

[2]Knowing this might save you some confusion if for whatever reason you try to read my code.
[3]I'm so sorry.
[4]Those that progress along a path.

Scotty

Scotty is expected to either win or lose quickly, so the algorithm uses `find_ultimate_path` which casts out multiple paths with varying initial directions then evaluates each of them. The evaluation is a weighted sum of the ranks of all squares visited in the path, we chose to weigh the final points higher since many Trappers focused on the edges of the arena and did not have much time to test other values. This should find the most open path by finding the path with the most variations along the route to sidestep obstacles.

```
def find_ultimate_path(self, start):
  def dist(point): return self.heatmap[point[1]][point[0]]
  def find_dir(start,end): return [end[0] - start[0], end[1] - start[1]]
  dist_traveled = dist(start)
  good_choices = list(filter(lambda x: dist(x) < dist_traveled, self.neighbors(start)))
  super_set = []
  for node in good_choices:
    steps.append(Point(*find_dir(start,node)))
    steps = self.find_path(node)
    evaluation = 0
    curr = start
    scalar = 1
    for step in steps:
      curr += step
      evaluation += scalar * self.node_rank[curr]
      scalar *= 1.2
    super_set.append((steps, evaluation))
  if super_set == []:
    return []
  return max(super_set, key=lambda x: x[1])[0]
```

Last-Ditch Attempts

If Scotty reaches an edge he steps off, if he finds there is no path then he goes on a random walk, which was hilariously effective at throwing off trapper (though they were *usually* victorious in the end). Not much more to it.

Trapper Structure

The Trapper is a bit more complicated, despite using pathing similarly:

1. If path exists: keeps track of Scotty's escape square, the final one in the path.
   - If there is a square that can be blocked that would lengthen Scotty's primary path: block it.
   - Else if Scotty can reach the escape square: block the escape square.
   - Else: continue letting him believe, block backup routes.
2. Else: fill tiles adjacent to Scotty randomly.

Time Concerns

Since Trapper wins generally take a lot longer, we found that using `find_ultimate_path()` took too long, especially when Scotty stayed near the middle. While our Scotty algorithm could abide by Escape or ~~Bust~~ TLE, it made sense to use it there but since `find_ultimate_path()` function was a last second addition we didn't have time to do anything sophisticated for Trapper Thus, Trapper uses the boring `find_path()` method.

Adjacent Interference

First, we check if we blocking Scotty's next square makes his life more difficult by comparing path lengths with and without the blockage.[5] If blocking the next square gains time for the trapper we go for it.

```
next_n = self.pos + steps[0]
self.pathing.board[next_n.y][next_n.x] = 2
self.pathing.heatmap = self.pathing.generate_backmap()
one_block = self.pathing.find_path(self.pos)
newt = len(one_block)
if newt - s > 1 or newt == 0:
    return next_n
```

Violating the Geneva Convention

Otherwise, we get into our torturous practice of letting Scotty retain hope until the last possible second. The trapper finds the escape square and then virtually blocks it in future consideration, choosing other squares to block. When Scotty thinks he is about to be free, the trapper sadistically blocks his escape.

```
end_delta = sum(steps,Point(0,0))
self.escape = self.pos + end_delta

if len(steps) == 1:
    return self.escape
else:
    self.pathing.board[self.escape.y][self.escape.x] = 2
    self.pathing.heatmap = self.pathing.generate_backmap()
    steps = self.pathing.find_path(self.pos)
    if len(steps) == 0:
        return self.escape
    end_delta = sum(steps,Point(0,0))
    self.pathing.board[self.escape.y][self.escape.x] = 2
    #print(f"current: {self.pos + end_delta}")
    return self.pos + end_delta
```

I've never seen a replay where we manage to bait Scotty then lose, it usually leaves Scotty in a tough position moving along a wall of blocks.

[5] Added this minutes before the deadline, thus "newt."

Spaceships: Try, Try, Run-into-another-spaceship

Spaceship was the problem in which we iterated on our strategy the most. In the end, we ended up with an algorithm that lost rating if it scored less than 20 points in a match, so I think that went pretty well. At the beginning of our development process, we had the *brilliant* idea that we could write a bot that just did its think that left the avoiding to everyone else. After having a match list or two full of deaths before reaching a single revolution, we realized that we couldn't trust others to do the avoidance for us so we wrote our own.

Investigation

We wrote over ten simple strategies for testing and learning purposes, I'll outline major advancements below:

- `class random_bot`: first bot to run!
- `class safe_random_bot`: first bot to exceed the intelligence of a moth and *not* crash into the sun![6]
- `class naive_circle_bot`: first bot with an objective! created a perpendicular vector then found the move most aligned with it to orbit the sun.
- `class dist_circle_bot`: first bot to fulfill an objective efficiently! Added another term to the move evaluation to prioritize getting near the sun.
- `class dist_square_bot`: first bot to realize that the sun is not a circle (wrt finding optimal direction)!
- `class survivor_bot`: first bot to predict and avoid others!

While we made a lot of progress with perpendicular vectors and dotting to ascertain alignment most of this progress was made moot when we borrowed the objective function used in the game loop, checking the angular position delta. That would have been a much better starting point. Only our "gravity" advancement to get closer to the sun was still useful.

Code Structure

Our final spaceship was modular, with the following core loop:

```python
self.others_archive.append(others)
self.update_wreckage()

x, y, self.direction, score = ship
self.pos = Point(x, y)
self.others = others

self.round += 1
for strat in [predict_all_moves, predict_directional_moves, predict_optimal_move]:
    self.avoidance_strategy = strat
    self.check_moves()

    choice = self.choose_move()
    if choice:
        self.priors.append(choice)
        return choice
return random.sample(self.moves.difference(self.illegal_moves), 1)[0]
```

[6] Note: I'm not claiming that moths crash into the sun, but that they would given the chance.

Wreckage

It took an embarrassingly long time to realize that wreckage destroyed ships. Once we knew that, we used the `others` array to find bots on the same square then add them to wreckage. That had the additional benefit of preventing us from predicting the moves of destroyed spaceships.

```python
def update_wreckage(self):
    locations = set()
    for x, y, _, _ in self.others_archive[-1]:
        location = Point(x,y)
        if location in locations:
            self.wreckage.add(location)
        else:
            locations.add(location)
```

Move Evaluation

Afterwards, it does various maintenance then gets into the core decision loop. Despite bouncing around ideas about genetic algorithms and *n*-move prediction, our bot looked ahead just a single move. The `check_move()` function creates two classes of moves, `unsafe_moves`, where something bad *might* happen, and `illegal_moves`, where something bad *will* happen (i.e. getting toasted by the sun or joining a wreckage pile). First, we create a list of dangerous positions, `danger_nodes`, which is wreckage plus anywhere that we expect a ship might move. That "might" is according to the avoidance strategy. The final condition prevents the bot from getting into loops, since its self-preservation instinct is stronger than its desire to score points and we can't have that.

```python
def check_moves(self):
    danger_nodes = copy.deepcopy(self.wreckage)
    def is_threat(ship, other):
        return dist2(ship, other) < 25 and not (other[0], other[1]) in self.wreckage

    for s in filter(lambda l: is_threat(self.pos,l), self.others):
        danger_nodes.update(danger_zone(s,self.avoidance_strategy(s)))

    self.unsafe_moves = self.find_unsafe_moves(danger_nodes)
    self.illegal_moves = self.find_illegal_moves()
    if len(self.priors) >= 4 \
            and self.priors[-2] not in self.illegal_moves \
            and self.priors[-1] - self.priors[-3] == (0,0) \
            and self.priors[-2] - self.priors[-4] == (0,0) \
            and self.priors[-1] + self.priors[-2] == (0,0):
        self.illegal_moves.append(self.priors[-2])
```

In the choose_move() function, it evaluates moves that are not illegal or unsafe, then choose the best of them. Often, it can't do that and returns nothing.

```
def choose_move(self):
    val = -math.inf
    best = None
    for move in self.moves.difference(self.illegal_moves):
        if not move in self.unsafe_moves:
            evaluation = evaluate_move(move, self.pos, self.direction)
            if evaluation > val:
                val = evaluation
                best = move
    return best
```

If it successfully makes a choice, then it submits it. More often than not, though, it has to iterate over the array of avoidance strategies. Each is less paranoid than the one before it, until it (hopefully) gets to the point where it can make a move. In the (never before seen) case that it doesn't have a move after trying everything, it chooses a random legal (but unsafe) move.

Avoidance Strategies

The avoidance strategies are as follows:

- `predict_all_moves()`: it's in the name, it returns every move nearby ships can make. There is *no* chance of destruction if the algorithm finds a move with this as the avoidance function.
- `predict_directional_moves()`: makes the daring assumption that ships won't move backwards, even when avoiding they generally sidestep forwards. This is usually enough of a relaxation for the algorithm to find a move.
- `predict_optimal_move()`: asks "what would we do?" except minus self-preservation. It's a last resort.

Nebulous Ideas

With more time, we would have further explored the genetic algorithm option and implemented our heuristic lookup, `avoidance_lookup()`. The lookup checked if the ship had every seen other ships in the exact position they are currently in then assumes that they will make the same move. It didn't make it into production because it failed to account for context.