How to Cheat in Puzzles and Dragons

Amanda Chow, Ted Li, and David Zeng

I. Introduction

Puzzles and Dragons (PAD) is one of the most popular games on both Android and iOS. It takes a unique approach to mobile gaming, combining aspects of both RPGs and puzzle games to create a game where you pick a team of monsters and level them up by battling evil monsters. During the battles, you play a matching puzzle game, which will be focus of our project. The puzzles in the game present you with randomized puzzle board (shown above) with 5 rows and 6 columns. Each of these spaces contains one of six orbs: Fire, Water, Grass, Light, Dark, and Heart. To begin solving the puzzle, you select an orb and drag it. As you drag it through other orbs, the other orbs will swap locations with the orb you are dragging, effectively pushing the orbs in your path behind you. The goal of dragging the orb around is to position other orbs so that you form groups of orbs of the same color. Once you let go of your selected orb, or reach a 4 second limit, your turn is completed. The game then finds adjacent groups of orbs of the same color and gives you a score based on the size of these groups and the number of groups.

II. Gameplay Model

In order to implement our various AI, we created a model for our game that we could easily work with. In our model, we represented our board as a 5×6 matrix, where each entry in the matrix could take on a value of 0 to 5, correlating to the six possible orb colors in the game.

A. States and Actions

We defined our states as any representation of the current board, where each of the 30 entries has a value of 0 to 6. In each of our states, we not only have the board matrix representation, but we also have the coordinates to our current orb. Actions are defined as any direction, `U`, `D`, `L`, and `R`, and which correspond to moving our current orb $(0, 1), (0, -1), (-1, 0)$, or $(1, 0)$. When we perform an action, we update our state by swapping the value of our current location with the value of our next location, which is our current orb's location plus the move. We then define the location of our current orb as the new location.

B. Model Restrictions

Each turn gives the player four seconds to drag around an orb. Newer players can typically accurately drag an orb about 20 spaces in this time, while for a good player, this usually equates to around 40 space. Thus, we limit our AI to 20 to 40 moves, depending on the algorithm. In addition, the game permits diagonal movements which in turn, allow players much more flexibility in their moves. However, this is difficult for most human players to perform given their time limit and small screen, so we will limit our possible moves to up, down, left, and right.

C. Scoring Methods

To properly evaluate the performance of our AI, we need a metric to evaluate the PAD puzzle board. Our full evaluation metric, which we will call `PADScore` performs the following algorithm.

1) Find all continuous groups of 3 or more orbs in single rows and columns.
2) Combine groups of the same color that either intersect or are adjacent.
3) Each group is given a score of $1 + \frac{1}{4}(\text{size} - 3)$.
4) Sum the scores for each group.
5) Multiply by the sum by the combo multiplier, given by $1 + \frac{1}{4}(\text{\# of groups} - 1)$.

The actual PAD game also takes into account orb types and how they interact with the offensive attributes of your team and the defenses of the actual monsters, but the pure mathematical score of the board is computed in an identical manner.

To speed up the scoring computation, we also define a score heuristic, which we call `GroupCount`. This is given by

1) Count the number of continuous groups of 3 or more orbs in single rows and columns.

This captures a good estimate of the full score, without having to run the intersection logic between all the groups.

III. Oracle

We were unable to find a guaranteed oracle for PAD, and it is not feasible for us to do an exhaustive search for 40 moves. However, we did find a very popular PAD web application, pndopt (Puzzles and Dragons Optimizer) [1], that runs a search algorithm with greedy heuristics. Evaluating their outputs with our scoring metric, over 100 randomized trials, pndopt averaged a `PADScore` of 15.52 with an average path length of 28.57, which roughly corresponds to aligning 6 groups of three orbs. Upon further searching, most of the popular PAD solvers use greedy search algorithms, so we wanted to try a few different approaches.

IV. Baseline AI Implementation

Our baseline AI implementation is based on a player making random moves. It performs the following algorithm.

```
1: for $i = 1$ to 1000 do
2:     reset the board to original state
3:     pick a random orb to start
4:     $S_i :=$ score of board
5:     $P_i :=$ empty path
6:     for $j = 1$ to 40 do
7:         drag the orb in a random direction
8:         $S :=$ score of board
9:         if $S > S_i$ then
10:            $S_i := S$
11:            $P_i :=$ path so far
12:        end if
13:    end for
14: end for
15: return $\max_i\{S_i\}$ and corresponding path $P_i$
```

We generated 100 boards at random and ran our baseline algorithm on the boards and recorded its performance. The baseline achieved a `PADScore` of 7.38 using an average path length of 25.31. This roughly corresponds to the baseline algorithm being able to align 4 groups of three orbs each.

V. Greedy Player - Hill Climbing

After the baseline, we started with a greedy player implemented using hill climbing. The idea was to only take moves that increase or maintain the current score. We explored several different variations of hill climbing, with varying levels of greediness, and the major variations are reported below.

A. Steepest Ascent Hill Climbing

To set a baseline for hill climbing, our first hill climbing player was the greediest - not only did it only choose moves that increase or maintain score, it actually chose the move that increased the score the most. In the case where multiple moves can result in the same maximum possible immediate score, the player chooses one of those moves at random. If all possible moves decrease the score, then the player has reached a local maximum score, and stops. The player would take these moves as described to increase the immediate score as much as possible, thus tending toward local maxima. The algorithm is performed as follows:

```
1: input: board $B$
2: pick a random orb to start
3: $P :=$ empty path
4: for $j = 1$ to 40 do
5:     $M :=$ moves with best immediate score
6:     if $M$ is empty then
7:         break
8:     end if
9:     $A :=$ random move from $M$
10:    Take move $A$
11:    $P :=$ path so far
12: end for
13: return board score of $b$ and path $P$
```

This player averaged a `PADScore` of 3.32 with an average path length of 40.7 over 1000 random starting boards. This is roughly equivalent to aligning 2 groups of three orbs each.

As expected, this player is too immediately greedy and does not perform very well. Examining several games in detail, we noticed that the player often gets stuck at local maxima, stepping back and forth endlessly between states that have equal scores until it runs out of moves - hence the path length of 40.7. Aside from the states explored during the hill climb, the player unfortunately never has the chance to explore more of the state space - which is generally advantageous for most AI's and increases the likelihood of finding higher local maxima.

B. Simulated Annealing

In order to balance the trade-off between greediness and exploration of state space, we then decided to incorporate simulated annealing into our hill climbing player. Rather than always choosing moves that increase or maintain the score as much as possible, at each state, there is a probability p that the player will chose any random move that *decreases* the score. (If there are no moves that decrease the score, then the player randomly chooses any move). The rationale behind this choice is that it gives the player a chance to explore the state space more, and increases the likelihood of stumbling on a state that leads to better scores. In other words, it occasionally strays off the hill climb and makes sacrifices the score in hopes of discovering new, taller ”hills” to climb.

To give the player a chance to explore the state space, we let it play 1000 times from the same starting board. Each play, it generates a random starting position and makes moves from there. Over the 1000 different plays, we take the best score and return its corresponding path. Given a randomly generated starting board B and a probability p of taking worse moves, the algorithm works as follows:

```
1: input: board $B$, probability $p$
2: $S := 0$
3: for $i = 1$ to 1000 do
4:     Start at original board $B$
5:     pick a random orb to start
```

```
6:      $P_i$ := empty path
7:      for $j = 1$ to $40$ do
8:          Pick random real $x$ in [0, 1]
9:          if $x < p$ then
10:             $M$ := moves that decrease score
11:             $A$ := random move from M
12:         end if
13:         if $x > p$ or $M$ was empty then
14:             $M$ := moves with best immediate score
15:             $A$ := random move from M
16:         end if
17:         Take move $A$
18:         $P_i$ := path so far
19:     end for
20:     $S_i$ := score of board
21:     if $S_i > S$ then
22:         $S := S_i$
23:         $P := P_i$
24:     end if
25: end for
26: return score $S$ and path $P$.
```

The choice for the value of p depends on how much exploration of the state space we want. The first greedy hill climber essentially operated with $p = 0$, always opting for the best immediate score, but hardly exploring the state space. We started with a probability of 0.2, and had the player find solutions to 100 randomly generated starting boards. Over these 100 boards, the player averaged a `PADScore` of 7.9, and an average path length of 38.57. This corresponds to roughly 4 groups of three orbs each.

This player was less greedy and performed significantly better than the greediest hill climber. However, it was still a long ways from the oracle, so we set out to find the best value for p. After trial and error, we found the best value for p to be approximately $p = 0.08$. A player with $p = 0.08$ could produce an average score of 8.5 with an average path length of 40.8 over 100 boards. This corresponds to roughly 2 groups of three orbs each and 2 groups of 4 orbs each.

This was the best value we were able to achieve using hill climbing with simulated annealing. This player performs barely better than our general baseline (the random player) due to its nearsightedness, in that it only aims to maximize the immediate score after one move. Even with simulated annealing, the amount of exploration of the state space is very limited - the state space is simply too large to explore fully. Given the limitations of hill climbing in such a large state space, our next step was to employ search techniques with local optimizations.

VI. Genetic Algorithms

Genetic algorithms are based on the principle of natural selection. We encode members of our search space as if they were DNA strings, and emulate selection, reproduction, and mutation of a population over time, in hopes that the population grows stronger and leads us towards better solutions. The motivation to try genetic algorithms for PAD comes from two sources:

1) The goal of solving PAD is to find an optimal path of some maximum length. Searching over this state space easily translates to searching over string representations of paths, which is in line with the language of genetics.
2) The human approach to PAD involves using small sequences of moves to line up orbs one group at a time. The small sequences represent "genes" of the path, and solution essentially comprises of stringing together the right genes.

A. *Genomic Path Representation*

The first step in making a genetic algorithm is to design a genomic string representation of the search space. A path in PAD essentially comprises of first picking a starting orb, and then making up, down, left and right movements. For a given starting orb, all paths from that orb can simply be encoded as strings of four characters, each representing a directional action: `U`, `D`, `L`, and `R`. Genetic algorithms work best when all genomes of the population have the same length. In order to allow variation in path length while still keeping the genomic length constant, we also introduced a stop action `S`.

B. *PAD Genetic Algorithm*

Our current genetic algorithm for PAD works as follows:

```
1: input: board $B$, starting location $(r, c)$
2: parameters: population size $S$, genome length $L$,
   number of generations $G$
3: $P$ := list of $S$ random genomes of length $L$
4: for $i = 1$ to $G$ do
5:     assign fitness score to each member of $P$
6:     $C$ := empty list for offspring
7:     for $j = 1$ to $S$ do
8:         selection: pick members of $P$ with probability
           proportional to fitness score
9:         reproduction: generate 1 offspring based on the
           members selected
10:        mutation: randomly modify the offspring
11:        add the offspring to $C$
12:    end for
13:    $P$ := $C$
14: end for
15: return path with highest score in $P$
```

There are many schemes to try for selection, reproduction, and mutation. Currently, for our basic algorithm, we pick 2 parents during selection, reproduce using single point crossover, and do not perform any mutations on the children. Single point crossover is defined as the following:

1) Given two parents, pick a split point at random
2) Take the genetic info of one parent before the split, and one parent after the split

```
parent 1:  UDLRU|DLRUDLRUDLR
parent 2:  LRDUL|RDULRDULRDU
---------------------------
offspring: UDLRU|RDULRDULRDU
```

There are also many ways we could go about evaluating the fitness of the genomes in our population. We currently simply trace through the path on our board, and then run `GroupCount` on the board. Paths that produce more groups are favored in selection. In particular, paths that run off the board will receive a score of 0 and will not be viable for selection.

C. Initial Results

With our current implementation of our genetic algorithm we achieve an average `PADScore` of 9.94 with an average path length of 28.80 over 100 random boards using a randomized starting location for each board. For this run, we use a population size of 10000 genomes, each of length 30 (instead of 40), and iterate the genetic algorithm for 50 generations. This corresponds to our algorithm producing 5 groups of three orbs each, and improvement of one group over the baseline method.

We noticed that runs of our algorithm that started in the center of the board as opposed to edges produced higher scores in the end. If we only start our genetic algorithm on orbs that are not in the outer ring, we achieved a `PADScore` of 10.73 over our random boards, with all other parameters held constant. This is likely to do with the fact that paths that start on the edge are more likely to run off the board and produce illegal paths, which have a fitness of 0 under our scheme.

D. Ideas for Improvement

Between now and the final due date, there are a multitude of things to explore to improve upon our genetic algorithm. Here are some ideas we have

1) Changing fitness evaluation. Currently, paths that run off the board achieve a fitness of 0. It might be better to evaluate the path up to the point where it falls off and still give it some level of fitness, so that potential good genes in the path can carry on.
2) Play around with the genome length. We found that increasing the genome length to 40 did not actually help our algorithm as is. It's possible 40 moves is not necessary to achieve a good score, given that the oracle requires under 30 moves on average. Paths of length 40 increase the search space and increase the chance of producing illegal paths. It's possible paths of length 30 might even be too high, and that more efficient solutions can be found.
3) Introduce more variability in the population. We found that after 50 iterations of our genetic algorithm on the population, usually the population was very homogeneous. On some runs the entire population had converged to one genome. Although the convergent population usually performed decently well, more variation is needed for the population incorporate more of the search space.
   a) Introduce mutation. We tried a basic form of mutation, which was to occasionally toggle one move in an offspring to another. This had no detectable effect on our performance. For this problem, it might be more appropriate to modify entire chunks of the path, or favor modifying the ends of the paths to explore uncharted territory while still preserving some initial moves that are known to be good.
   b) Change the reproductive behavior. We also tried using two point crossovers when producing offspring, but this actually worsened our algorithm. Again, we probably want to take an approach where we are more conservative with the start of the path, and introduce more variability on the tail end. We may also want to try mixing genetic material from more than 2 parents when producing offspring.

VII. Conclusions

Overall, we found that of our two approaches, genetic algorithms performed much better. Hill climbing would often get stuck in local maxima before exploring most of the states of the board, even with annealing. On the other hand, our genetic algorithm AI was able to get better scores, but still struggled with a lack of diversity in its population. Moving forward, we will primarily focus on improving our genetic algorithm through the aforementioned changes in hopes to get on par with or better than existing PAD solvers.

References

[1] pndopt, http://kennytm.github.io/pndopt/pndopt.html