Flaippy Bird: Project White Paper

Aron Szanto, Caetano Hanta-Davis, Joseph Kahn

January 24, 2017

1 Introduction

This project was a technical study of the game "Flappy Bird" (FB) through the lens of artificial intelligence. Using both deterministic and randomized paths, we represented FB as both an informed search problem and a Reinforcement Learning process. Adopting the obvious end of maximizing the game's score as our agent's learning goal, we approached this problem in three phases. First, we adapted an open source version of this project[1] into an interface conducive to AI work, redefining the game to be comprised of familiar objects like states, goals, rewards, successors, and the other necessary underlying factors. Second, we devised an informed search solution to the problem. This involved requiring the agent to solve a game with deterministic pipes, i.e., to figure out an action set that guides the agent through each pipe successfully. We used the A* algorithm in this phase, with a carefully designed heuristic that served effectively to break the larger problem into subproblems, with the agent only needing to find a path between one pair of pipes at a time. The final phase of the project was to relax the constraint of determinism in the pipes and to require the agent to solve a game "as it comes". This involved a shift in paradigm, reforming the game as an off-policy learning problem rather than as a search problem. Our results were uniformly positive, with the AI-representation of the game serving us well, the informed search algorithms able to solve large generated games in quasilinear time, and the policy derived from Q-learning successful outperforming any reasonable human.

This project was a true capstone to our study in this course. From proofs of heuristic admissibility in informed search to extensive optimization of the policy derivation algorithm in the Q-learning section, we engaged deeply with the core tenets of the course throughout our work.

[1] https://github.com/sourabhv/FlapPyBird

2 Background and Related Work

Flappy Bird is an addictive smartphone game released in May 2013 by Dong Nguyen. It immediately went viral, garnering 50 million downloads by January 2014 and generating 50,000 of advertising income for Nguyen each day. It was so popular that the developer took the game down in February 2014 due to its addictive nature, his unwanted popularity, and what he thought was the games overuse.
It is a notoriously hard game for humans to succeed in. It is an infinite game in which a bird navigates its way through a set of pipes, choosing at each timestep whether to flap or fall. These actions serve to change the bird's vertical velocity and position. After each pipe that the bird agent passes, one point is added to your score, and the game stops once the bird hits any of the pipe pixels. Most people cannot make it past about ten or so pipes without many days and longer hours of practice.

Because of the physical dynamics involved in the game, it is conducive to informed search and reinforcement learning. Extracting these properties and changing the framework to work for AI was a large engineering challenge. To complicate matters, FB represents a very large state space. Taking into account the number of pixels on a standard iPhone (i.e. the number of locations for the pipes and agent), the possible set of velocities and accelerations, and the other factors that make up a game state, the number of states in the space grows very quickly to around 10^{42} in the 500-pipe problem.
A successful Informed Search (IS) agent will solve any fixed maze of pipes in front of it, hopefully in time that is subexponential in the number of pipes.
A successful reinforcement learning (RL) agent will solve a set of randomly generated mazes and get far without unreasonable training time.

3 Problem Specification

We will first present an overview of our system's representation of the FB environment, before specifying the problem formally for both informed search and RL.

3.1 Parameters of the Flappy Bird World

The FB world is fully specified by the following:

1. The location (in pixel coordinates) of the bird
2. The locations of the pipes that the bird (or human player) can see
3. The velocity and acceleration vectors of the bird
4. The current score of the bird
5. *(If deterministic)* The offset in the pipe list that is next going to be generated

3.2 Informed Search

In general terms, informed search is applied to FB by representing the FB world as a sequence of nodes, each with a full specification of the world's state, as outlined above. We define a goal test to ask whether a node's current score is equal to the score desired (i.e., the size of the game currently being solved). We also define a function to return the successors of a given node. Combined with functionality to simulate the game's progress through time given the physical attributes of each state, the FB world is represented well as an informed search problem. Formally, the game is comprised of:

State nodes, each with:

1. A location $x \in \mathbb{R}^2$
2. A velocity $v \in \mathbb{R}^2$
3. An acceleration $a \in \mathbb{R}^2$
4. A score $c \in \mathbb{Z}$
5. A set of pipe locations $P \in (\mathbb{R}^2)^n$, where n is the number of pipes the bird can see

The functions, each operating on a node s, with attributes subscripted as above

1. $G(s) = \mathbb{1}(s_c = C)$, a goal test that returns true if and only if a node's score is equal to the goal score C.
2. $A(s)$, a function returning the actions available to the agent at a state. These will always be a member of $P(\mathbb{A})$, where $\mathbb{A} = \{\text{FLAP}, \text{FALL}\}$, and P denotes the power set operator.
3. $N(s)$, a function that returns a set of neighbors of the node, along with the action required to get to each.
4. $D(s)$, a function returning true if and only if the state represents the "game over" state (after a crash).

And the constants C, the "goal number" of nodes that codes the complexity of the game, and various more mundane constants that denote the number of pixels on the screen and such. It should be noted that for cleanliness, we abstracted these away into the functions G, D, A, N so that the agent doesn't ever need to interact with them.

3.3 Reinforcement Learning

Instead of having a fixed number of pipes that are deterministically mapped out, our second problem involves solving FB with randomly generated pipes. This more closely models how the game is experienced and played by human users. We chose to approach

this using reinforcement learning techniques, specifically Q-Learning. We have the following parameters and functions:

1. State space s, we discretize the state space into a $\mathbb{Z}_{10} \times \mathbb{Z}_{10}$ grid (with ten horizontal and ten vertical bins)
2. Actions a: The actions are still the same. At each state we can either FLAP(humans users can do this by pressing the spacebar or up-arrow) or FALL.
3. Reward $R(s, a)$: To get a good Q-learning result, a consistent reward function for all cases is important. We chose to give +1 for every step the bird stayed alive and -1000 for if the bird died

Parameters in Q-Learning:

1. α - the learning rate
2. ϵ - the exploration rate
3. γ - the discount factor
4. λ - trace decay parameter in TD-Lambda Learning Algorithm

4 Approach

4.1 Informed Search

We used the well-known A* algorithm for the informed search part of our problem. Generating a large number of deterministic pipes for the agent to test on, we encoded world information into state-nodes, as specified above. We then defined the functions as specified above to allow to agent to "move" between states through time. This was an interesting procedure, as the agent does not directly control its fate, as Pac-Man did. Rather, it has control over the second- and third- order physical attributes of its state: its velocity and acceleration. In this way, the agent's path to the goal is not so much a sequence of choices of of physical locations, but rather a sequence of mechanical vectors through space and time. Rather than rewriting the A* algorithm here, the foregoing formal specification defined the key parts of the algorithm in terms of the FB game. For example, the canonical Get-Neighbors function is implemented as N, which returns the next state nodes, operating on the current one using the physical transitions that we encoded in addition to taking into account the range of actions that the agent has available. It is here that we were able to merge the algorithm with the physicality of the game to find an efficient solution.

4.1.1 Heuristic

The crux of the informed search section of this problem was in the choice of heuristic. As specified by the A* algorithm, the priority of a node in the frontier is given by the function $f = g + h$, where g is the foregoing cost to get to the node and h is the value of the heuristic function that estimates the future cost to the goal. We experimented extensively with heuristic functions, finding that there is high variance in the results as a function of the quality of the heuristic. We will present our final heuristic function, then prove its admissibility. Define $m(s)$ to be the Manhattan distance between the current state's location and the *midpoint* of the next pipe that the agent will encounter. This is the coordinate at the x- and y-averages of the pipe's left and right corners, and upper and lower pipes, respectively. Further recall that for a node s, the score of the agent at that state is given by s_c. Then let our heuristic $h(s) = m(s) - 1000s_c$, with 1000 an arbitrary number greater than the maximum Manhattan distance between a state and its next pipe. We will now show certain properties of this heuristic. But first, note an important rule in this game's system: the x velocity of the agent is constant, which means that every successive node represents a movement of constant distance in the direction of the goal. The implication is that every correct solution to a problem will have exactly the same cost (specifically, the number of pixels traveled to the goal in the x direction, divided by the velocity in the x direction).

Admissibility is trivial to show, since the heuristic gives the Manhattan distance to the next pipe. Because the distance between pipes is greater than the vertical range of pipe midpoints, $m(s)$ is necessarily less than or equal to the true distance to the goal. Adding on the large negative constant relative to the score of the node, there is no way for the algorithm to overestimate the true goal cost. Interestingly, we do not require consistency, even though this is a graph search rather than a tree search. The reason for this is that we guarantee that every correct solution has equivalent cost, implying that optimality is trivially shown given a correct solution.
The empirical results for the heuristic will be discussed in the results section.

4.1.2 Implementation Notes

The encoding of states into graph search nodes can be found in `node_util.py`, while the A* implementation can be found in `algs.py`. Both implementations were written independently and internally, without consultation of outside sources for either. **We respectfully recommend that the reader inspect these files, as they are commented extensively and should be considered part of our report.**

4.2 Reinforcement Learning

For our reinforcement learning algorithms, we discretized the state into a $\mathbb{Z}_{10}$ x $\mathbb{Z}_{10}$ grid, as mentioned above, by first extracting the following three features: the relative hori-

zontal distance from the bird's RHS to the LHS of the lower pipe, the relative vertical distance from the bird's midpoint to the pipe gap's midpoint, and the vertical velocity. Then, to discretize the relative horizontal and vertical distances into ten bins, we used the following piecewise function:

$$x_{offset} = \begin{cases} x_{\text{offset}} \pmod{10} & \text{if } x_{\text{offset}} \leq 100 \\ x_{\text{offset}} \pmod{100} & \text{otherwise} \end{cases}$$

$$y_{offset} = \begin{cases} y_{\text{offset}} \pmod{10} & \text{if } |y_{\text{offset}}| \leq 100 \\ y_{\text{offset}} \pmod{100} & \text{otherwise} \end{cases}$$

Our algorithm is adapted from a TD(λ) approach, with a classical Q-learning update, and is not reproduced here, as it is extensively documented (and covered in lecture). In our implementation we experimented with using either an ϵ-greedy or pure greedy policy to choose the actions in different states.

To get a good Q-learning result, a consistent reward function is important. We chose to give +1 for every step the bird stayed alive and -1000 for if the bird died to highly penalize the agent when it flaps into a pipe or into the ground.

The TD(λ) algorithm involves a λ trace decay parameter that determines what proportion of the reward is given to states that are closer or further away. That is, if λ is zero, we update predictions based on only the state right before the current one, whereas a λ value of 1 translates into updating our predictions weighting all the preceding states equally. In this way, the algorithm learns more quickly which nodes are "responsible" for high or low rewards.

Our discount factor γ is fixed at 1, as our rewards do not decay even if the agent stays alive longer.

In experimenting, we tried using both a pure greedy and an ϵ-greedy policy to choose actions in different states.

In addition, we varied our ϵ value over time, such that the exploration rate was much higher in the first several episodes and decayed after more episodes had passed as such: $\epsilon = \max(0.05/(N + 1.0), 0.005)$ $where \text{N} is the total number of episodes passed$.

5 Experiments and Results

We present our experimentation and results jointly, as our project is more accurately described as experimental than as comparative.

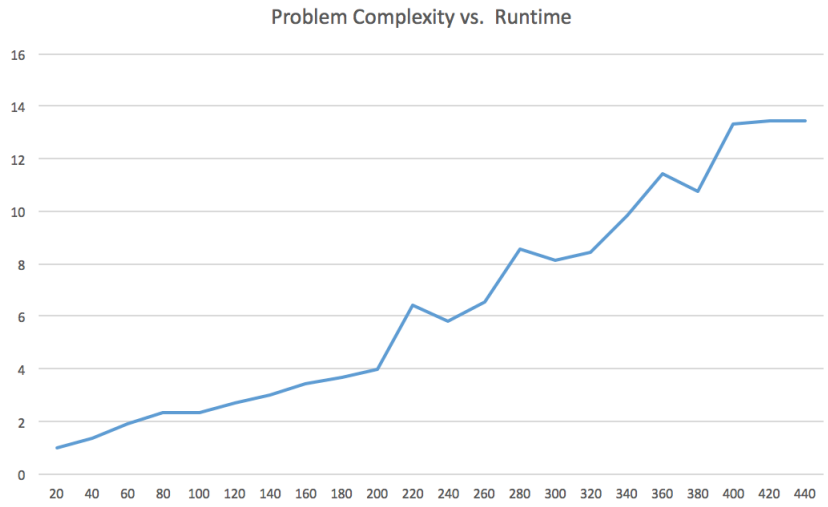
5.1 Informed Search

It might seem prudent to first provide a comparison of two search processes, with one using the defined heuristic and the other using a naive heuristic. We'll spoil the surprise by reporting that a search process using a naive heuristic won't find a solution even to the

one pipe problem, even after expending hundreds of thousands of nodes. On the other hand, the heuristic as defined successfully solves arbitrarily- sized problems with ease and efficiency. Our experimentation and results are grouped neatly into the categories of speed and of expansion efficiency.

5.1.1 Speed Performance

We were impressed by the efficiency of the algorithm. It was quick to choose the best nodes to expand and found solutions to increasingly difficult problems with little resistance. Below, we present an experimental graph that relates the complexity of the problem (measured by the number of pipes it needed to pass to win) with the time needed to find a solution. We note that if we took more time and computational power to report each y value as an average of several runs of the same x, our graph might be much smoother, but we trust that the reader can discern the pattern.



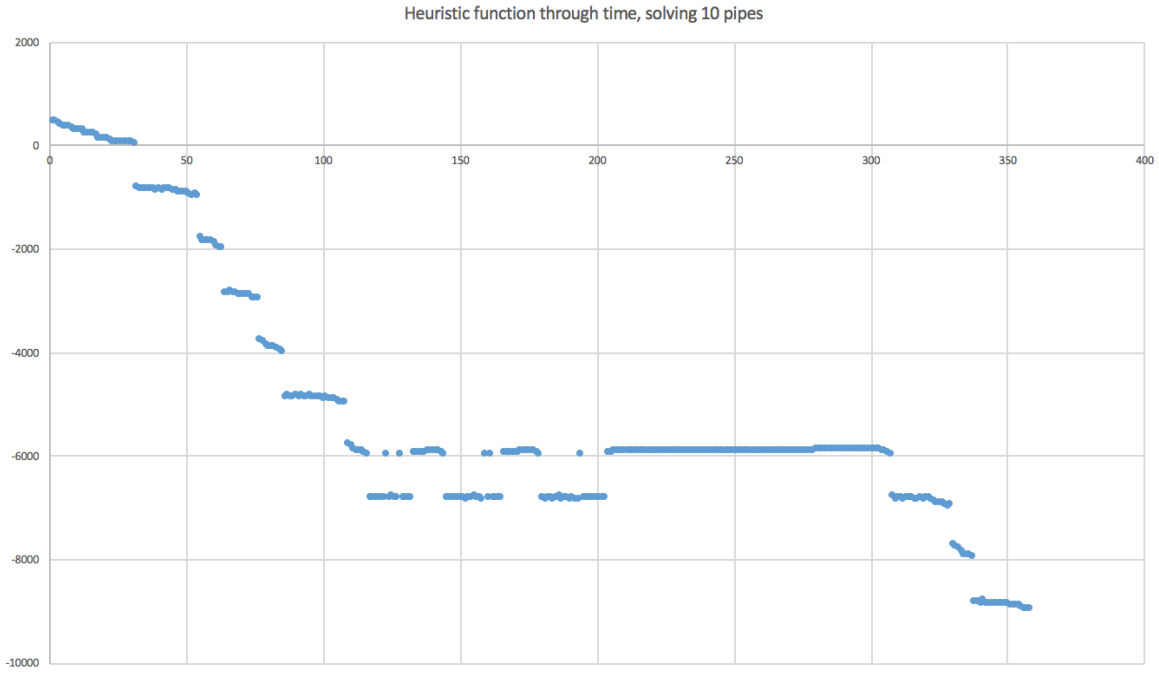
It is evident that the runtime increases in the number of pipes, but what is perhaps most striking is that the runtime increases far below exponentially, despite the state space growing exponentially. We attribute this to the well-designed heuristic function, especially in light of the fact that a process without the heuristic will churn nodes ad infinitum without finding a single path through one pipe, let alone close to 500.

5.1.2 Heuristic Analysis

The heuristic function is designed carefully to break up the large problem into smaller instances. We took a fairly economic route to this heuristic function; it is designed to model lexicographic preferences[2] over the (ordered) spaces of state score and closeness to the next pipe's midpoint. Adopting more language from economics, we require that the

[2]Lexicographic preferences refer to a preference ordering that weights categories hierarchally. If one is choosing between baskets of blue and red balls, and has lexicographic preferences over blue balls and red

nodes in the frontier compete against each other for expansion, with the "fittest" nodes being expanded first. We can approximate this behavior with the heuristic as defined; $h(s_1) < h(s_2)$ for all s_1, s_2 with $s_{1_c} > s_{2_c}$. However, with two nodes that have passed the same number of pipes, we choose the one that is closest to the next score, i.e., closest to the next pipe's midpoint. In this way, nodes are expanded within one pipe range in an order that is most efficient to find the path to the next pipe, and we expand those nodes first overall that are farthest along (closest to the ultimate goal). We show this heuristic's progress graphically by plotting the value of the heuristic for every node that is expanded off the frontier:



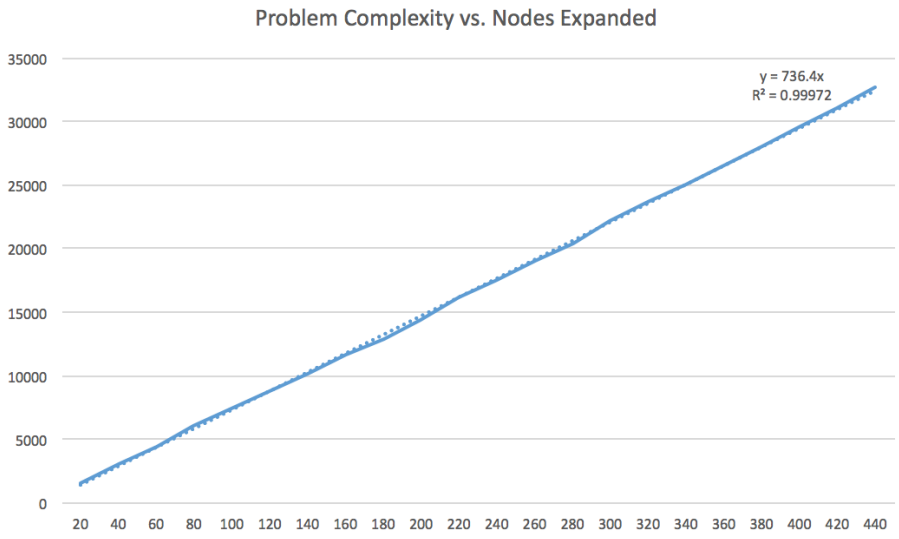
At each "jump", a node that solved the next pipe was found. Usually, after such a jump, the heuristic value will decrease steadily in that range, identifying the algorithm's progress towards the following pipe. However, behavior like that between pipes 7 and 8 are to be expected, with nodes that have passed 7 pipes having been popped off the frontier but not having physical attributes conducive to being a part of a path to the 8th pipe (for example, consider a node encoding a state where the agent passes the 7th pipe by a small margin, but is going too quickly downwards to recover, and has no other choice but to crash. Then the algorithm will backtrack (perhaps a few times) to find a node that reaches the 7th pipe but is stable enough to be a part of the path to the 8th). We also note that this behavior is likely the cause of the superlinear runtime of the algorithm; finding nodes that solve a pipe but that are functionally dead ends cause a bit of churn that is

balls, then a basket with more blue balls than another will be preferred no matter the number of red balls. However, between baskets with identical numbers of blue balls, the one with the most red balls is preferred. In this way, one is "infinitely" more preferential over categories that descend in priority.

reflected in the time it takes to solve a problem. On the whole, however, this heuristic function worked incredibly well.

5.1.3 Node Expansion

A good metric of efficiency is how many nodes an algorithm will expand with respect to the total number of nodes in the final solution. Find below a graph of the number of nodes it expands as a function of the problem complexity. The trend is about as perfectly linear as one will ever find in the sciences, with the smoothness empirical evidence for the heuristic's quality.



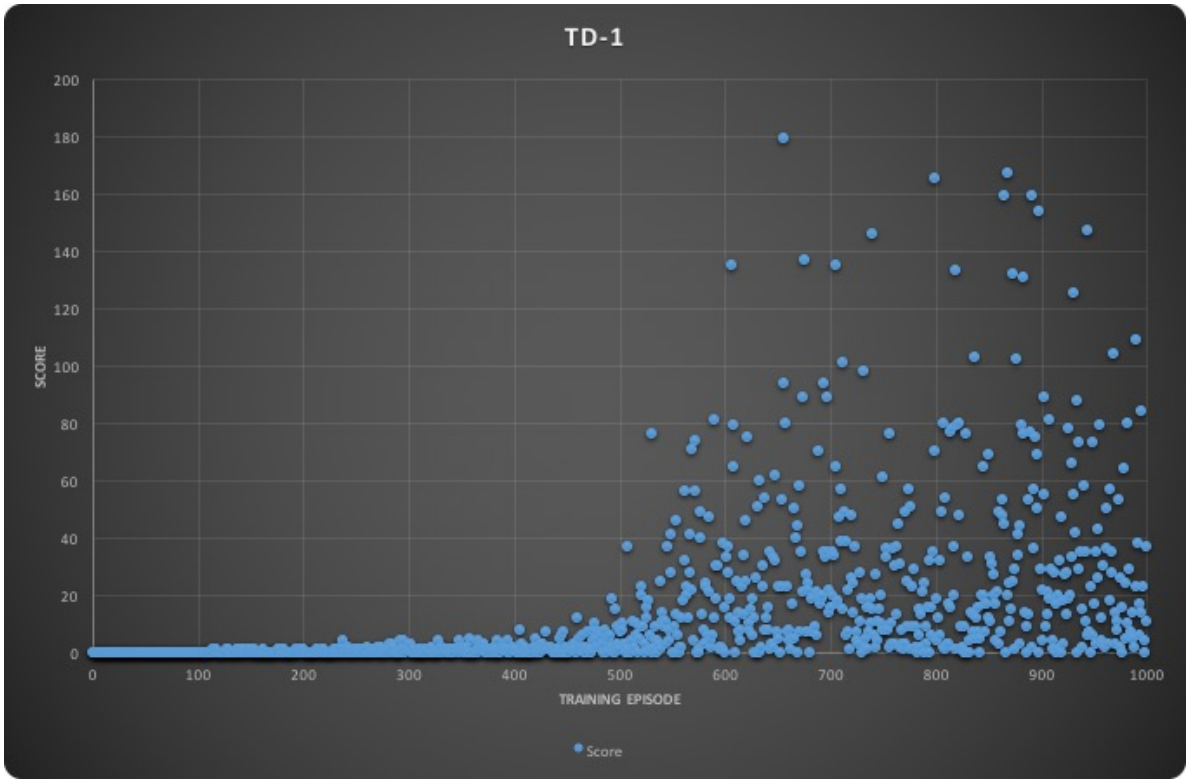
Most importantly, the algorithm expands 1.98 nodes per eventual action in the solution, meaning that its performance is within a factor of two of the "oracle" agent, which immediately knows the solution upon looking at the problem. This is an important finding that lends more credence to the heuristic's efficacy.

5.2 Reinforcement Learning

Tuning the parameters to get the best results was a considerable challenge. The parameters we focused on tuning here are the learning rate α, the exploration rate ϵ and the trace decay parameter λ.

5.2.1 Best Result To Date

We achieved our best high score to date of 179 using a pure greedy strategy and setting λ equal to 1 and α to 0.7.



5.2.2 Tuning α

Holding all other variables we want to tune constant, after running trials with 1000 episodes we discovered that the best value for α was 0.7, a higher learning rate that would make the agent consider more of the most recent information.[3]

α	High Score
0.01	10
0.1	23
0.3	31
0.5	14
0.7	46
0.9	14
0.99	7

5.2.3 Tuning ϵ

In the equation for calculating our ϵ exploration rate there is a lower bound, which is reached rather quickly, as it is a decaying function in terms of the number of episodes

[3]Note that these alpha trials are significantly lower then our final results, as when tuning we first worked with alpha without knowing the optimal values for the other parameters. This also applies for the other parameters.

elapsed. Here are experimental results after 1000 episodes in each trial changing the lower of bound of the epsilon value. Clearly, having a very low exploration rate with less random movements later on is the best move. It is still important, however, to have an agent that has a higher exploration rate earlier on, as there is a danger that the agent might converge to sub-optimally committing suicide if it can not explore and find the states that get it through the pipes.

Lowest ϵ	High Score
0.0001	91
0.0005	16
0.01	14
0.1	3
0.2	1

5.2.4 Tuning λ

Upon Consideration of our data after 1000 episodes, the TD(λ) agent appears to work best when lambda is set to 1, which if we are in state n, applies the reward to state n and n-1.

λ	High Score
0	73
1	89
2	27

5.2.5 Tuning: Epsilon Greedy vs. Pure Greedy

In addition, using a pure greedy policy over an epsilon greedy policy seemed to work best for our data and state space.

Type of Greedy	High Score
Pure	179
Epsilon	136

Our flappy bird reinforcement learning agent performs very well overall. After only 1500 episodes, it is able to get high scores of over 100, which would take any human many days to achieve. Over a long time horizon, this agent will only improve.

A major challenge in this reinforcement learning problem is the question of how to treat states the agent has not seen before and dealing with going through the gap. Flapping through the gap between pipes must be done with a very particular set of dynamical factors, and there exists a danger of an agent that fails so often trying to flap through a gap in the first several episodes using an ϵ-greedy policy that the learning algorithm converges to a sub-optimal policy where it does not attempt to succeed at all, instead flapping into

the ceiling or the ground in those situations.

We noticed that varying the α learning rate value affected how often this occurred, which would intuitively makes sense. Although we might want to set the alpha value closer to 1 to make it more difficult to un-learn, that approach may converge to a sub-optimal policy if it is over-fitting based on failing trying to get through the gap. This happened a few times as we were first developing our algorithm and tuning, and we originally considered different approaches including reducing the number of bins in the state space and changing the reward functions. It may be worth considering in future studies and projects whether if defining a reward function that does not have any negative values and only assigns positive rewards to flapping through a pipe yields different results.

6 Discussion

This project focused on two key facets of artificial intelligence: search and learning. Through the lens of the iPhone game Flappy Bird, we were able to implement a program with a complex physical engine as an AI problem and solve it using both deterministic and off-policy approaches. In the case of search, we devised an effective heuristic that allowed the agent to find a solution very quickly, proving both mathematically and empirically that the formulation was efficient. Relaxing the assumption of determinism, we also solved the random-pipe problem using Q-learning techniques, achieving a high rate of success far surpassing any normal human player.

Our main takeaways from this project were twofold. First, there's no limit to the sorts of domains that AI is applicable to. Second, clear thinking and good mathematical planning (of heuristic functions in particular!) are essential to efficient problem solving.

As per our original specification, future iterations could include a dimensionality reduction of the state to generalize to features of the game; this would involve some sort of gradient descent to optimize a policy given the salient dimensions of the game and their evolving weights. Though it's likely a subject for a thesis or dissertation, we still think the idea of training a convolutional neural network to take in the RGB values of the pixels and learn important features of the game to the extent that the network could play the game just as a human would- with visual stimulus only.

We declare no conflicts of interest, affirm that the work presented here is our own, and thank Scott Kuindersma and Curren Iyer for their help along the way.

A System Description

1. Ensure that the system has PyGame 1.9+ installed and is running Python 2.7+. It may also be necessary to update your the system's package of libpng.
2. To run the informed search agent, run python flappy.py -s n (where n is the number of pipes ¡450 you'd like it to solve. To start with, try -s 20)
3. To run the TD-lambda game using trained and stored Q-value weights, run python flappy.py -lw
4. To run the TD-lambda agent, run python flappy.py -l
5. To run the demo TD-lambda agent using the weights we found gave us the high score, run python flappy.py -d.
6. Results for graphing episodes and scores are then stored in stores.csv, which we used to generate our graphs

B Group Makeup

Aron Szanto- Adapting Pygame environment for AI use, informed search implementation, heuristic derivation and testing, results and graphics for informed search, write-up introduction, formal specification for IS, and conclusion
Caetano Hanta-Davis- reinforcement learning, tuning, results for RL, write-up background, system description
Joseph Kahn- reinforcement learning, tuning, adapting AI interface to RL-specific functionality, results for RL