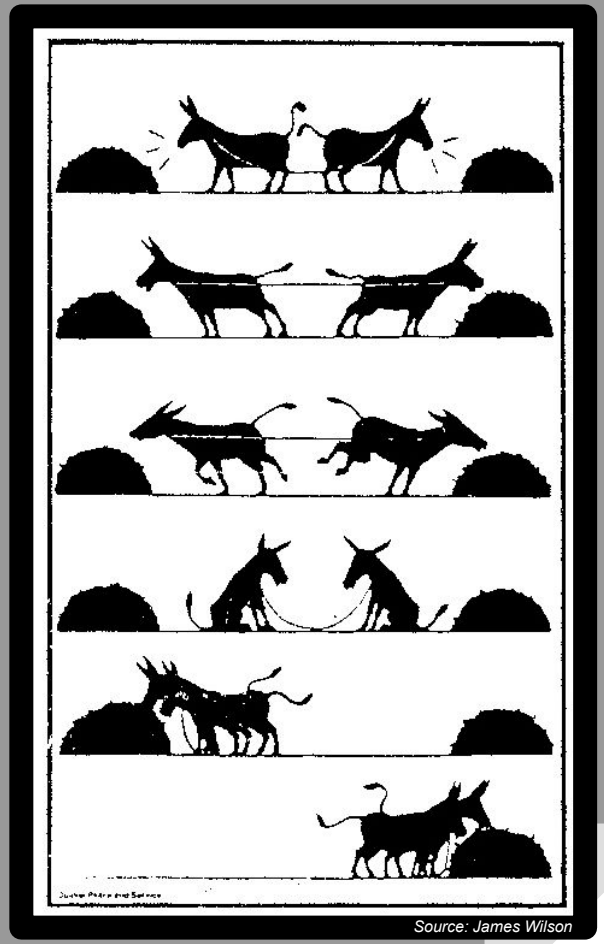
Prisoner’s Dilemma

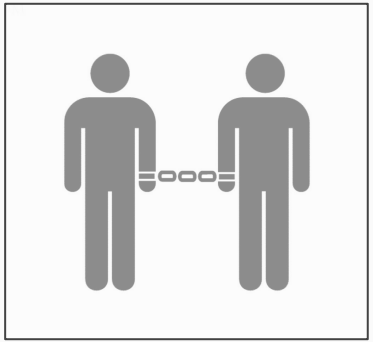
Stefanie Bodnar
Evian Kassab
Tyler Johnson
Zach Shaver



Source: James Wilson

Introduction

- Problem Introduction
- Strategies & Implementation
- Hill-Climbing
- Genetic Algorithm
- Conclusion



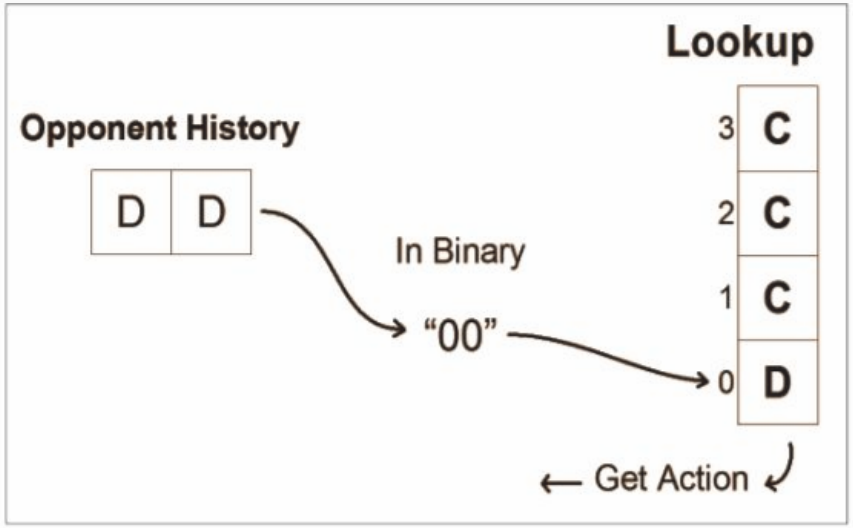
Problem Introduction

- Game & Real World Model
- Three Components
  - Player
  - Actions
  - Payoff Function
- Rationality & Irrationality
- Objective
  - Performance of Search Algorithm
  - Testing
  - Multiple Strategies
  - Compare & Contrast

		Bob	
		cooperate	defect
Ann	cooperate	–1, –1	–20, 0
	defect	0, –20	–10, –10

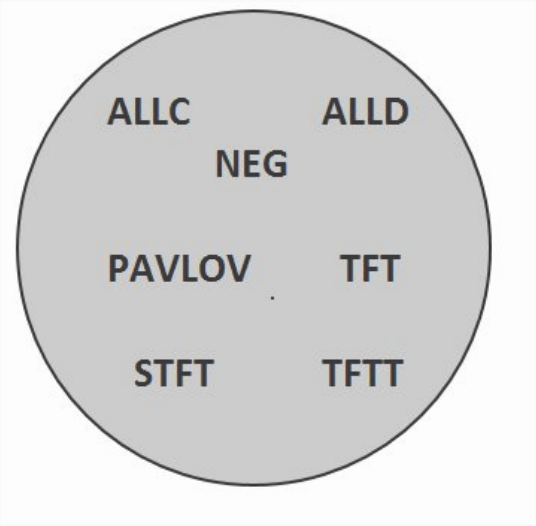
Strategies & Implementation

- Technology Used
  - IPDX Java Library
    - Provides basic programming interface for playing the game
- History of Moves
  - Stored as an integer, to be used as index in lookup table.
- Array Lookups
  - 1D - Opponent History
  - 2D - Opponent & Player History
- Evaluation Function
  - Generates heuristic by playing a set number of games against a training set and returning the average score



Strategies & Implementation

- Iterated Prisoner's Dilemma
  - Memory of Previous Moves
  - Goal: Best Lookup
  - Train algorithms to compete against specific, and general strategies.
  - Payoff Matrix
- Data Collection
  - Storing results in CSV to visualize with charts



	B:Cooperate	B:Defect
A: Cooperate	3,3	0,5
A: Defect	5,0	1,1

Hill-Climbing - Implementation

```
Input: Max Restarts, Max Sideways
Output : Lookup that is a Local Maximum

Lookup = new Random Lookup.

Repeat: {
        Neighbour = getBestNeighbour().
        if ( Neighbour's Score > Lookup's Score)
                Store Neighbour as Lookup
        else if ( Neighbour's Score < Lookup's Score OR Out of Restarts)
                return Lookup
        else
                if ( Sideways Moves Available )
                        Do a Sideways Move
                else
                        Reset Sideways Moves
                        Do a Restart
}
```

- A local optimization approach
  - Start with a random lookup,
  - Search local neighbourhood for higher scoring tables,
  - Replace lookup with the best neighbour,
  - Repeat until no more are found.

- Operator Function - *getBestNeighbour()*
  - Gets the best neighbour by:
    - Flipping each action in current lookup and returning the neighbour with the best score.

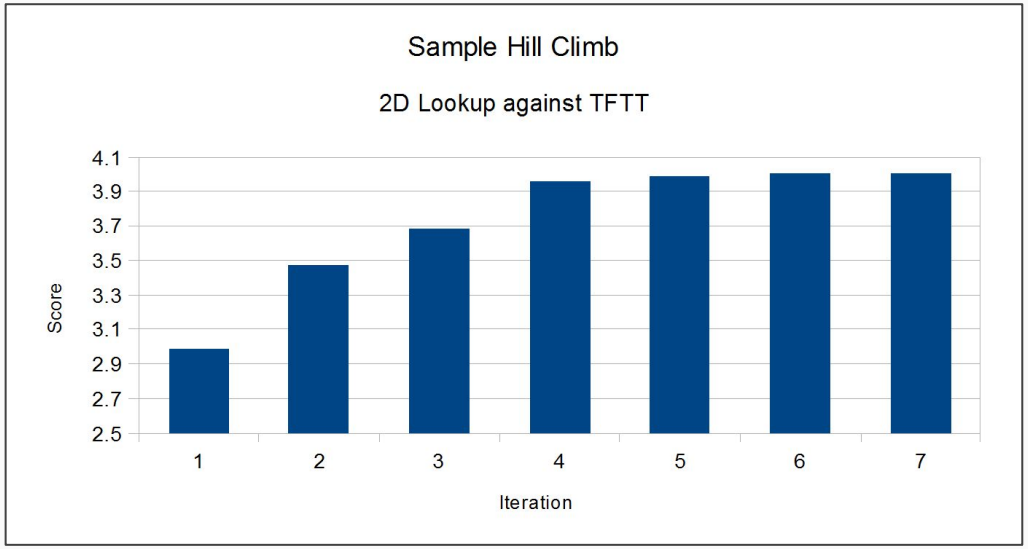
Original Table:	CCCC (Score : 2.51)
Neighbour 1:	CCC**D** (Score : 2.48)
Neighbour 2:	CC**D**C (Score : 2.33)
Neighbour 3:	C**D**CC (Score : 2.65) <= Best Neighbour
Neighbour 4:	**D**CCC (Score : 2.56)

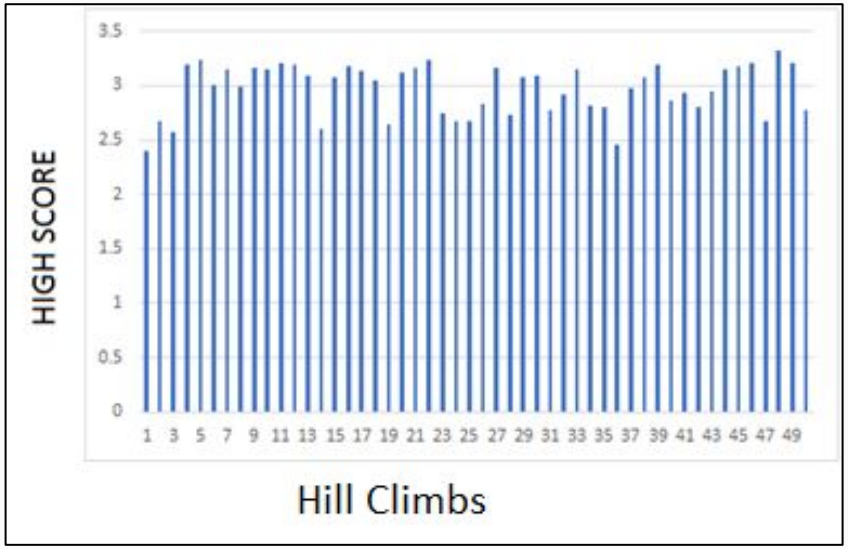
Hill Climbing - Findings

- 2D Lookup produces better results
  - Longer hill climbs, shorter histories & better solutions on average than 1D
- Solutions vary by length of history when trained against a variety of strategies

Hill Climbing - Findings

- Reaches a local max fairly quickly, which may not be optimal.
  - Solved by random restarts and sideways moves





Genetic Algorithm - Implementation

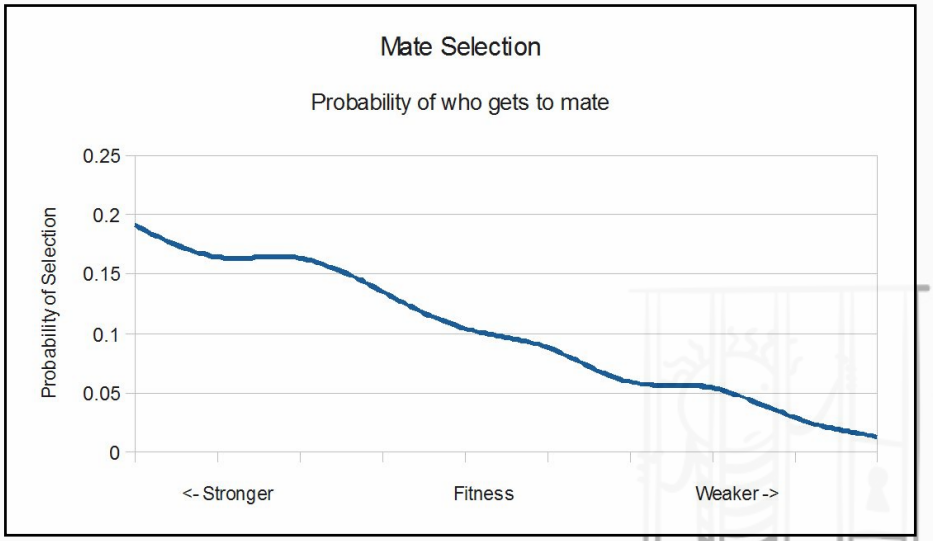
- Emulates evolution by selecting stronger individuals in a population for breeding.

Input: Population Size, Number of Generations, Number of Children per Couple
Output : Fittest Individual after Evolution

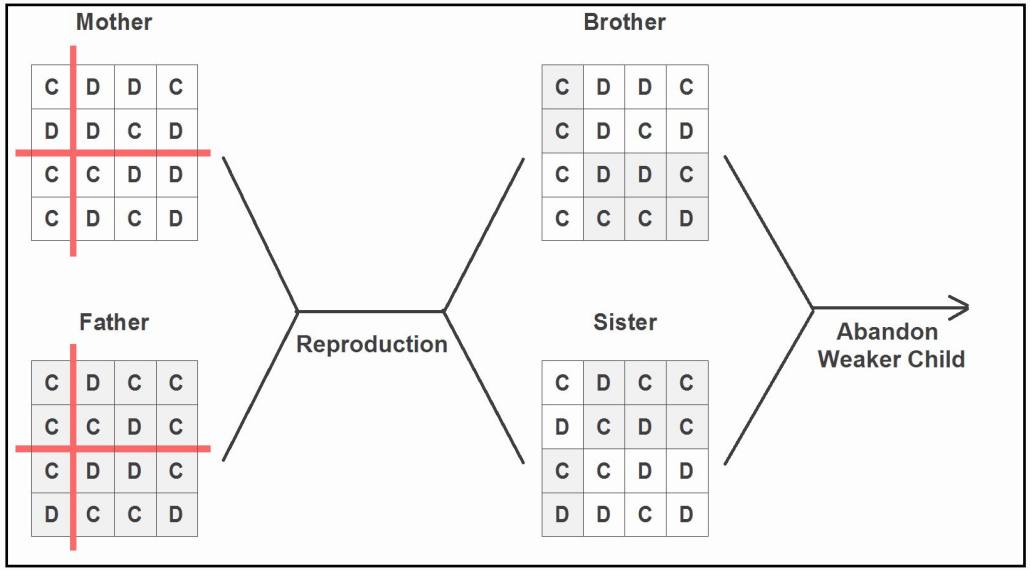
Population = *new* ***Population of Random Lookups***

For: (E*ach Generation*) {
    NextGeneration = Empty Population.
    For: (**Population size**) {
        Father = *selectRandomIndividual()*.
        Mother = *selectRandomIndividual()*.
        Child = *reproduce*(**Mother, Father**).
        Mutate **Child** with a small probability.
        Add the **Child** to **NextGeneration.**
    }
    Population = NextGeneration
}
Return the **Fittest Individual in Population**

- *selectRandomIndividual()*
  - Sorted by fitness, picks individual randomly according to a skewed distribution.

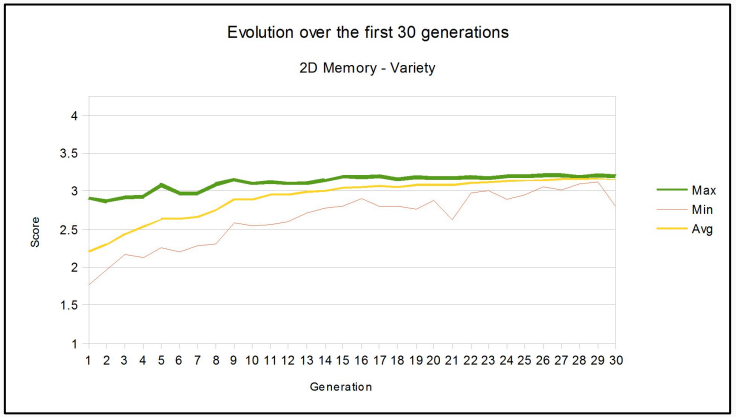


Genetic Algorithm - Implementation

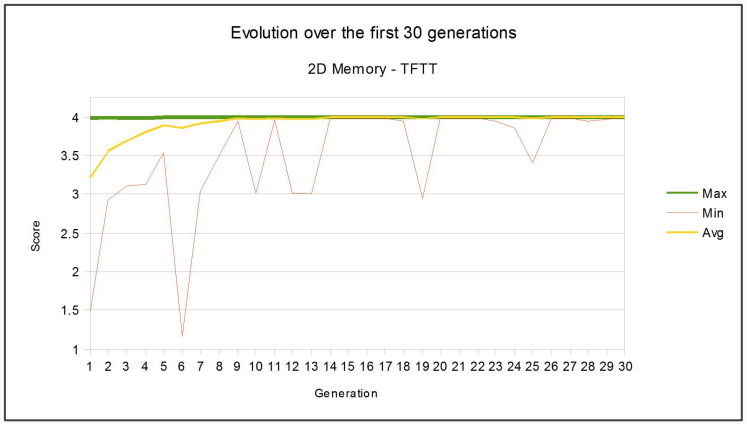


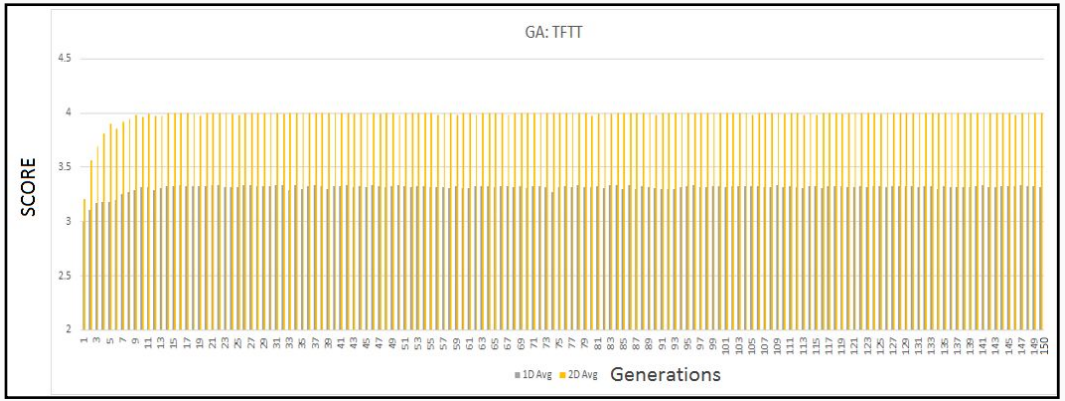
- *reproduce*(Mother,Father)
  - After Mother and Father selected,
  - Splits them at random indices,
  - Exchanges their parts to form a child,
  - And adds “fitter” child to Next Generation
    - *Our algorithm also has a second mode that keeps both children.*

Genetic Algorithm - Findings

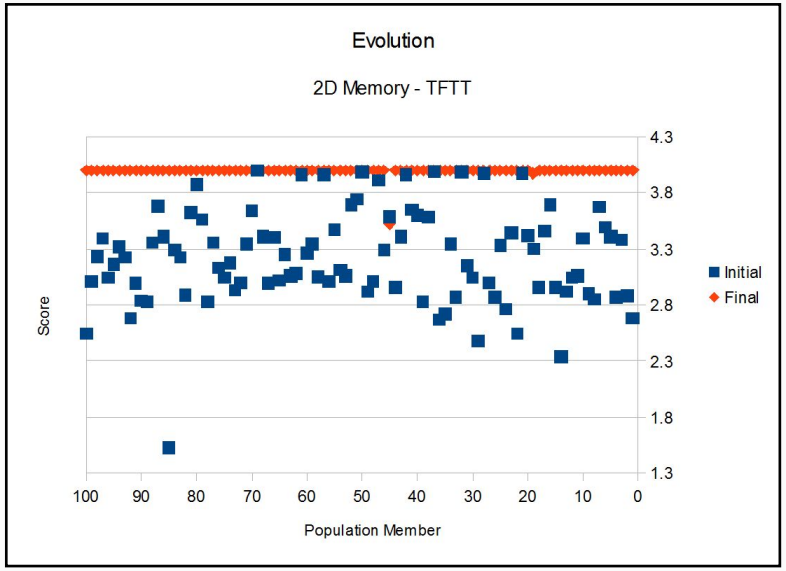


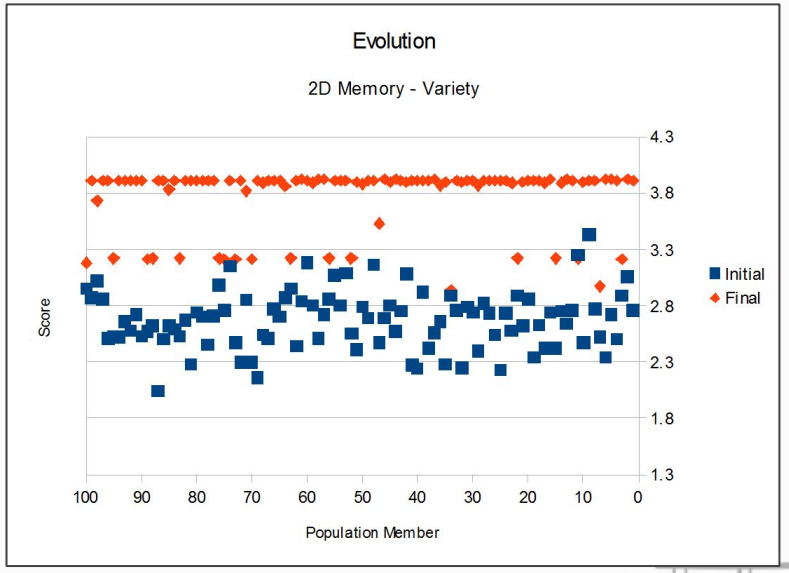
- The range of individual fitness declines over generations. Everyone gets fitter.
- Population's fitness peaks quickly
- 2D lookups perform better than 1D again.





Genetic Algorithm - Findings

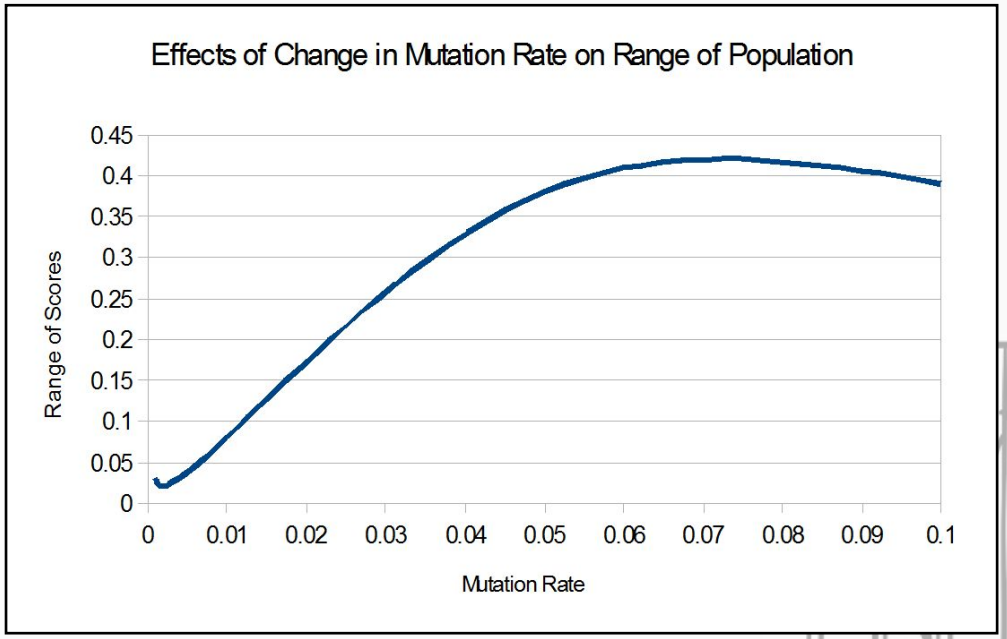




- When facing:
  - Single strategy: There is a peak score, the optimal lookup.
  - Multiple strategies: Often no optimal lookup, but close!
- Entire population tends toward the exact same lookup table, which is the optimal solution.

Genetic Algorithm - Findings

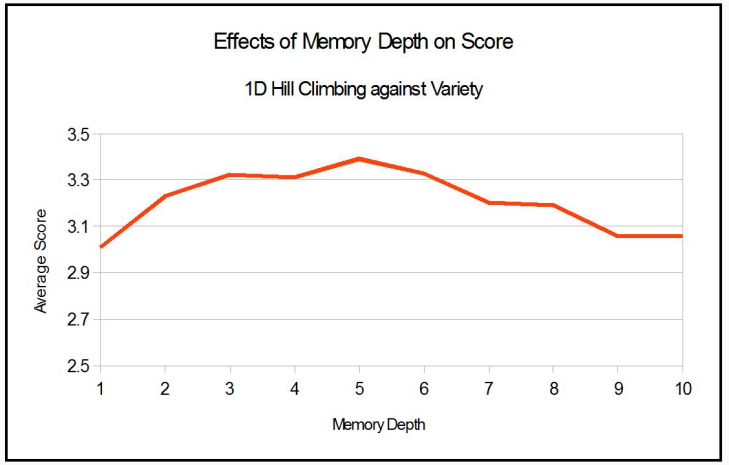
- Changing Population Size & Number of Generations
  - No significant effect on fitness past 10 generations or 10 individuals.
- Changing Mutation Rate
  - More diverse population

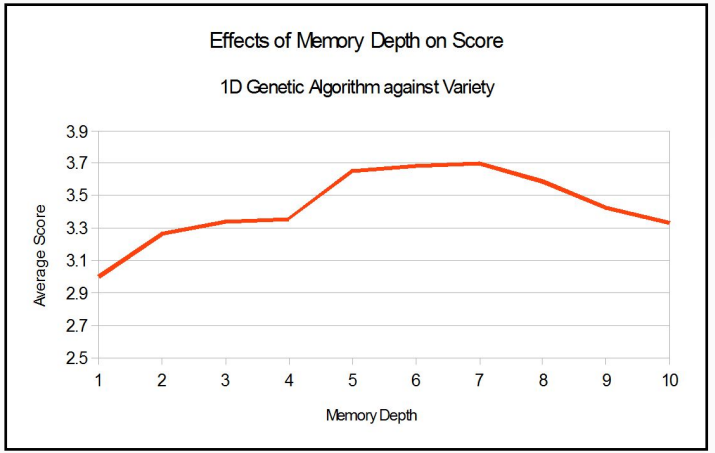


Conclusion

Compare & Contrast

- Both algorithms prefer similar history depths, according to their lookup dimension.

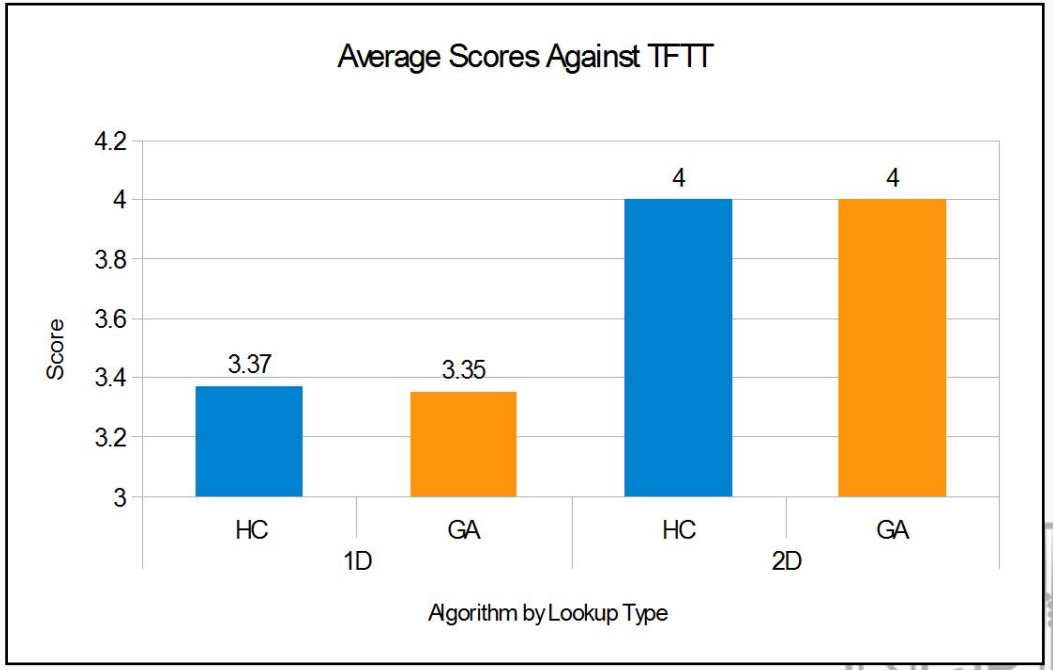


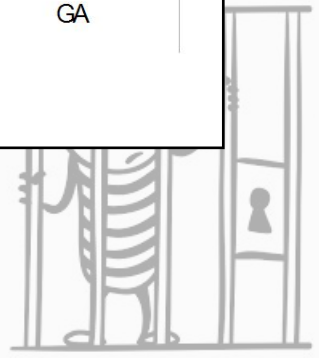


Conclusion

Compare & Contrast

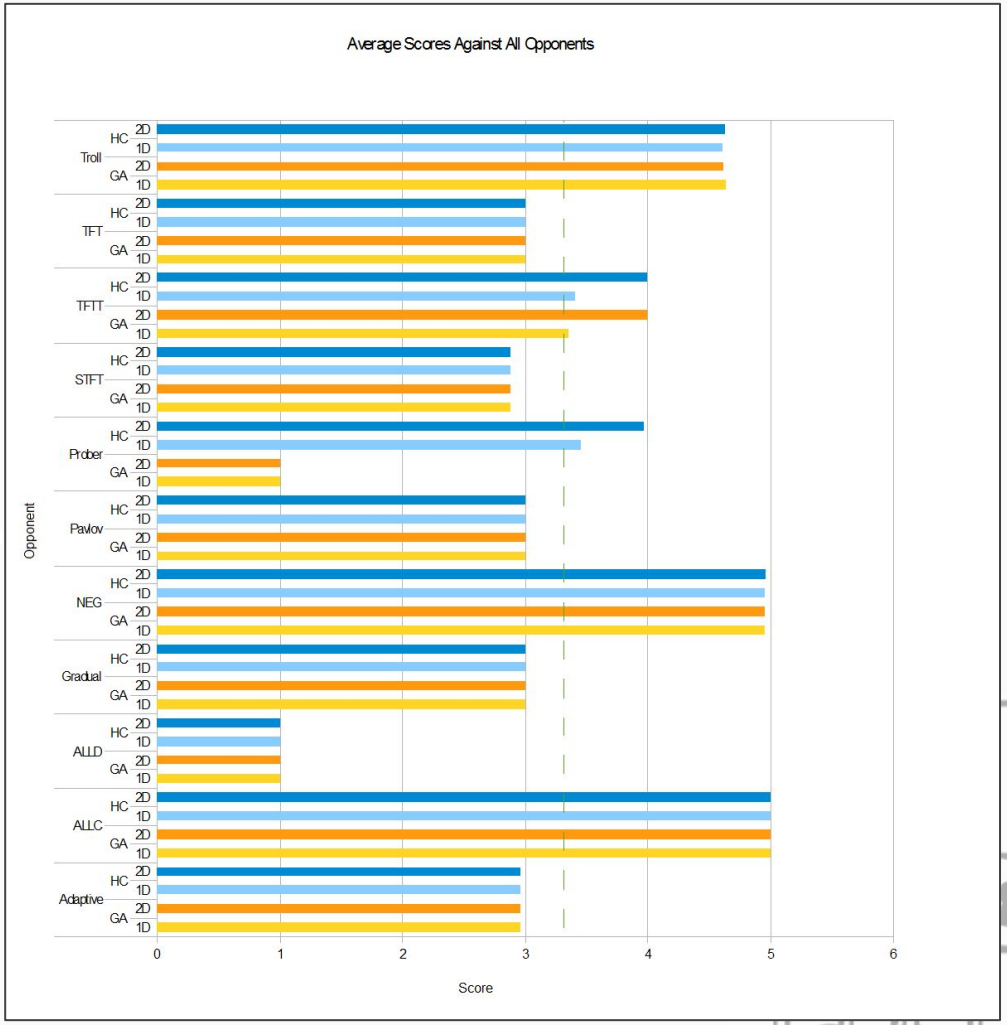
- 2D Lookups based on opponent & player history produce better results, for both algorithms.





Conclusion

- Both algorithms consistently produced optimal lookup tables when trained against particular strategies.
- Algorithms were also trained against human-design strategies successfully.



Conclusion

- Both algorithms perform very well on the Prisoner's Dilemma problem.
  - Hill Climb can produce optimal tables quickly, while genetic algorithms will produce them more consistently.
- 2D array lookups, which look at opponent & player history together, often perform better than 1D lookups.

The End… *Any Questions?*