

Creating Backgammon A.I. Players Using Expect-MiniMax Search Algorithm

Review Of Final Project In
Introduction To A.I.

Fall 2010

HUJI

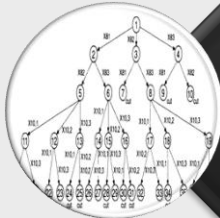
Submitted By:
Itai
Yosef



Overview



Goal



Theory & Background

- The Expect-Minimax Algorithm
- Design Of Heuristic Metrics & Heuristic Function



Implementation

- Empirical Results and Conclusions
- Java Implementation and Design



Goal

Create Artificial Intelligent Backgammon players using the **Expect-MiniMax** search algorithm, design useful **heuristic metrics** to evaluate nodes in the search tree and **empirically observe** different player “personalities” (i.e. different weights on heuristic metrics).

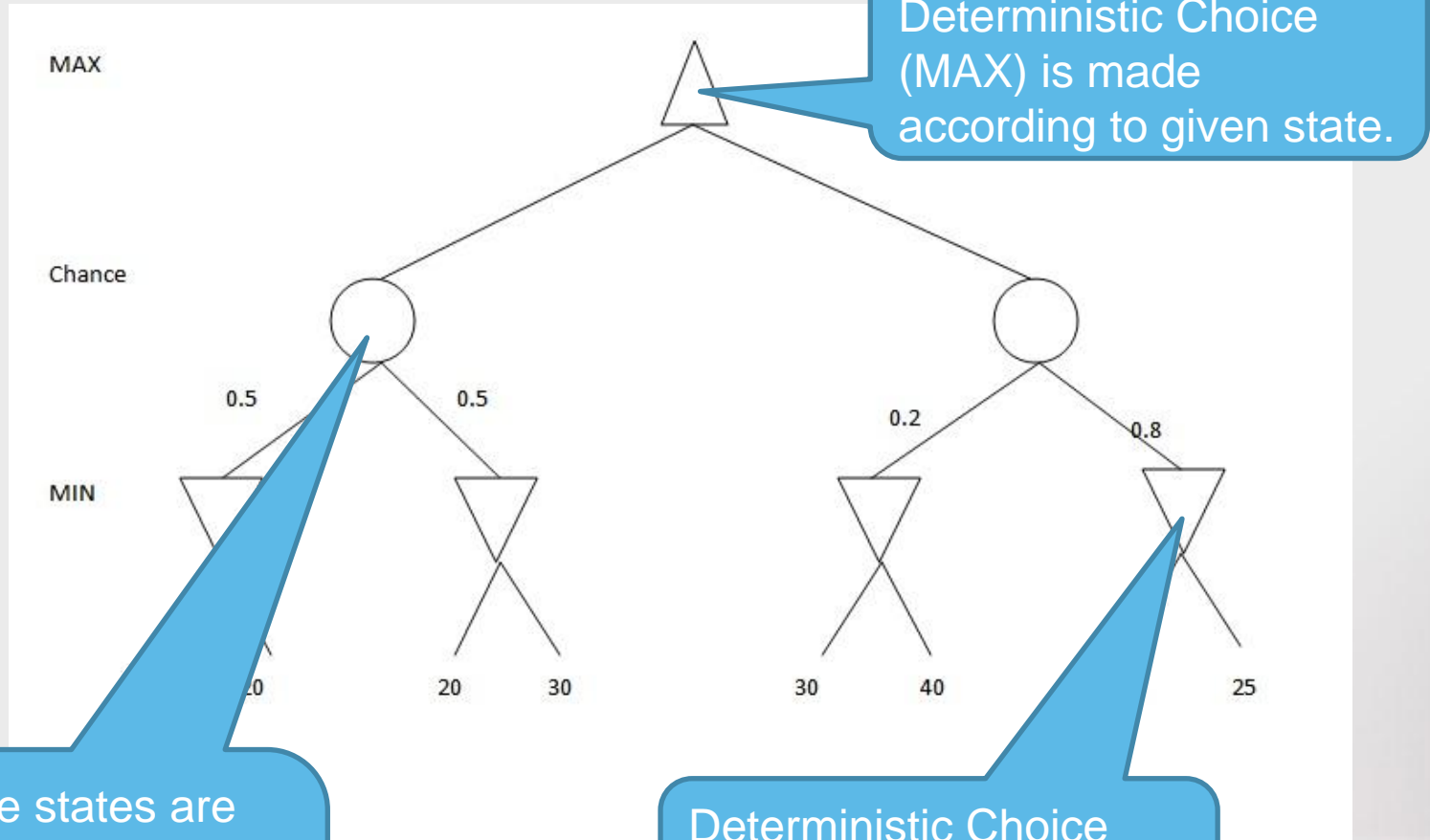


Expect-MiniMax

The Expect-MiniMax algorithm expands the deterministic MiniMax algorithm we learned in class to 2 player zero-sum finite **NONdeterministic** games of perfect information, by adding a “chance layer” between each player’s turn (layer) in the search tree.



Expect-MiniMax Tree



New possible states are generated with **weights** given to each branch according to the probability of that state to be reached.

Deterministic Choice (MIN) is made according to given state.



Expect-MiniMax Caveat Num. 1

Terminal states cannot be evaluated as $\pm\infty$, because then their value would render the probability weights useless!

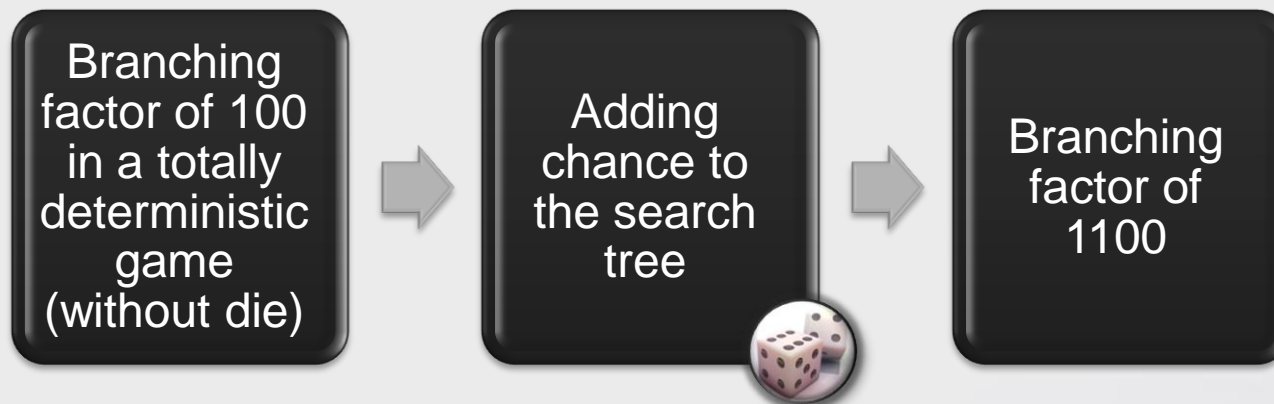
Solution: terminal states are given a **large** but **sensible** value, that changes according to each specific player's "personality" (heuristic metric values).



Expect-MiniMax Caveat Num. 2

Backgammon has a branching factor of 11 in each “chance” layer and a dice throw can present more than 100 different possible moves (in case of a “double”).

This means that **the search tree can have a branching factor of ~1100!**



Expect-MiniMax Caveat Num. 2

Solution: The search tree is limited to 2 turns and we implemented alpha-beta pruning.

In addition, for each turn, if the number of possible moves is larger than 100 then we search the tree's first level only, i.e. we choose the move that will create the best state in the current turn (no look ahead).



Heuristic Metrics

The Following metrics were used to evaluate the heuristic value of a given state (“state” = checkers’ position on a Backgammon board):



Heuristic Metrics For A Given Board

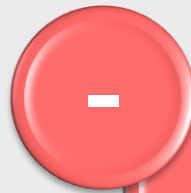


Houses (more than 1 checker in point)

Eat (opponent's checkers on bar)

Spread on board

Win



Towers (more than 3 checkers in point)

Exposed (only 1 checker in point)

Eaten (our checkers on bar)

Lose

Stuck (can't move)

Checkers left in base



Heuristic Functions – Player “Personalities”

Different player “personalities” were created by setting different values for each heuristic metric.

For example, a defensive player would set a very high value to having safe “houses” and a very low (negative) value to being eaten, while an aggressive “risk taker” player would set less extreme values to those metrics but would set very high values to eating the opponent’s checkers.



“Personalities” tested



Aggressive (The Punisher)

- Relatively low value for safety
- Very high value for eating opponent's checkers



Defensive (The Thing)

- Very High value for safety
- Very low value for getting eaten & being exposed



All-Around (The Megazord)

- Low value variance between different heuristic metrics
- A compromise of the other 2 personalities



Random (The Monkey)

- Doesn't use MiniMax alg. – randomly decides actions.



Empirical Observation - Process

Two classes were created for each “personality”, **based on two different search algorithms** (but the same heuristic evaluation):

One that uses the Pruned Expect-MiniMax algorithm and one that performs a complete search of the first level of the tree only (named “short range”).

Then, every different personality (short range & long range) was matched up against all the others in a “free-for-all” style tournament.



Empirical Observation - Process

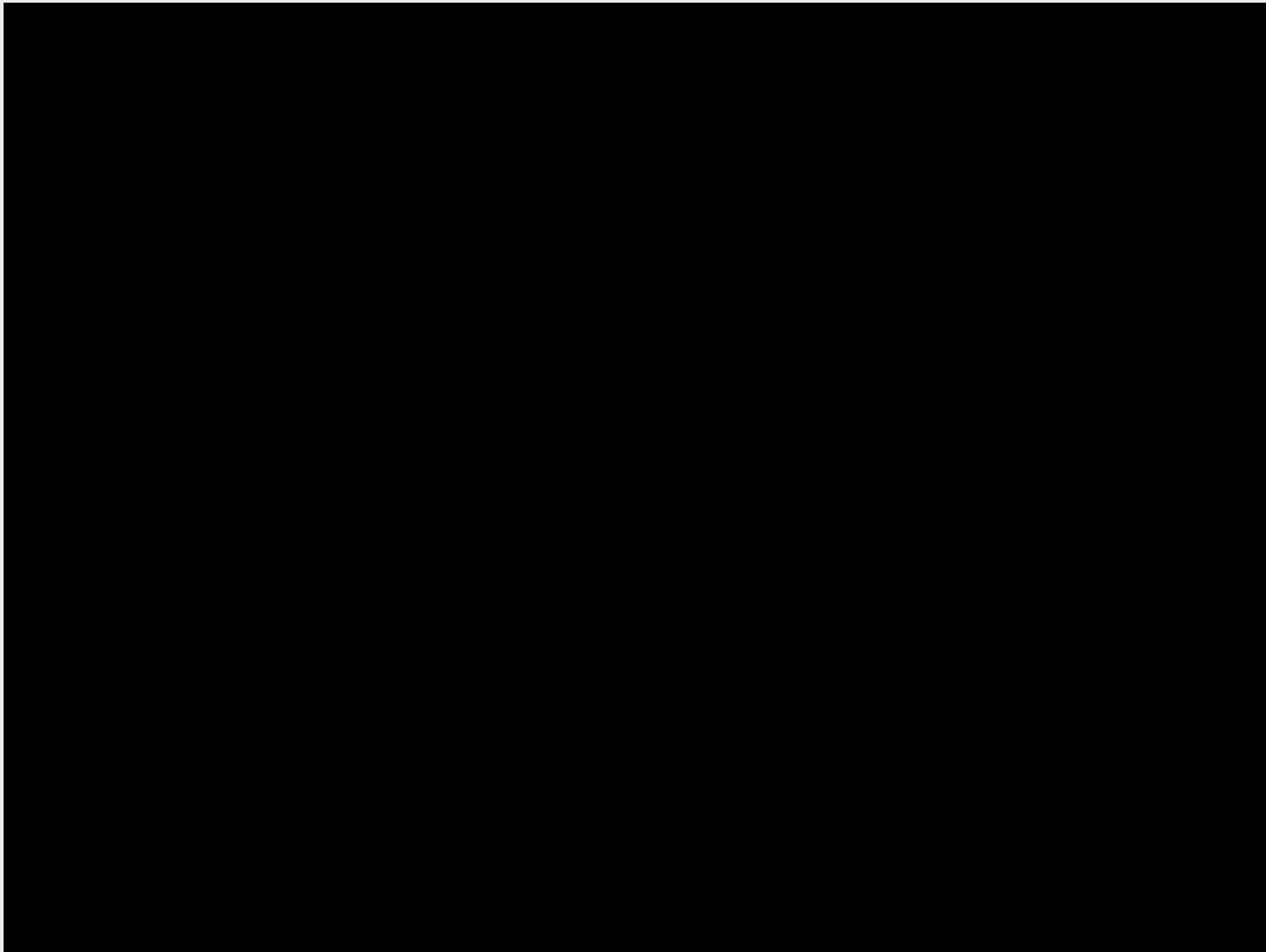
Each match between 2 personalities was in the form of best 2 out of 3, in order to try and clean up the noise generated by the random die. Each game with a personality that used the Expect-MiniMax search algorithm took about 10 minutes.

A random player (“The Monkey”) was introduced to the tournament as a control group.












































Empirical Observation - Exhibition

Click on the black box to view a video of one of the matches (turn sound ON!):



Empirical Observation - Results

Player/ Opponent	Short Range Aggressive	Short Range Defensive	Short Range All-Around	Aggressive	Defensive	All Around	Random (control)
SR Aggressive	X						
SR Defensive		X					
SR All-Around			X				
Aggressive				X			
Defensive					X		
All Around						X	
Random (control)							X
Num. Of Wins	3	3	3	6	3	3	0

Empirical Observation - Results

And The
Winner Is:
Aggressive!



Empirical Observation - Conclusions

Search

- Backgammon's die present a very large branching factor which limits the depth of the search tree.

Backgammon Strategies

- The chance factor in backgammon created a very close tournament.
- However, playing aggressively and taking chances seems to pay off.



Empirical Observation – Additional Backgammon Strategies Conclusions

- We observed that planning one step ahead doesn't give a personality a significant competitive edge (short range players usually did as well as long range players), probably due to the great effect of the die.
- Even though our results showed that aggressive achieved a flawless victory, some of the matches were close calls so it definitely is not a foolproof strategy.



Empirical Observation – Additional Backgammon Strategies Conclusions

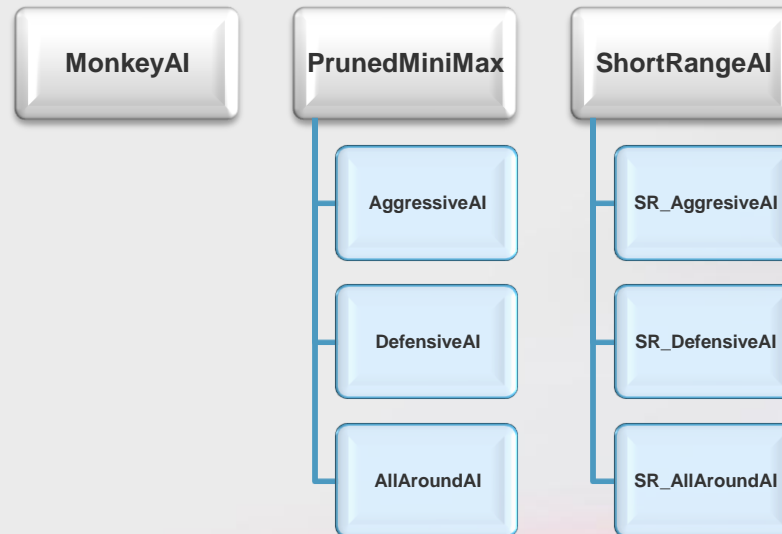
- Our results show that skill is still important in backgammon, since every single player lost to aggressive and yet **easily** beat the control group (the random player).
- The random player also played an important role in confirming the correctness of our heuristic design, since the aggressive player, which is a risk taker, beat the random player with a backgammon (מרס טורקי) many times. This confirms our design since taking risks (such as exposing checkers) is not very dangerous against such an opponent.



Java Implementation

We used the open source Java project “JGammon” as the infrastructure for our project.

We wrote 2 main abstract classes that interface with the open source code: PrunedMiniMax which uses the Expect-MiniMax algorithm and ShortRangeAI which uses a basic one level search. Each abstract class was implemented via inheritance by our “personality” classes. In addition, we wrote a 3rd class that interfaces with the open source code and implements a random player called MonkeyAI.



Java Implementation

In addition, we made some changes to some of the open source code to accommodate our needs and wrote a comparator class called “boardSetupComparator” to enable removing redundancies in the search trees.



Compilation Instructions

Running a simulation

In Linux (school computers):

Go to the “JGammon.src\src” directory in the extracted files and run “make” via the shell. The program will automatically run.

In Windows:

Several options:

- Open a new java project in Eclipse based on our code.
- Download some program to run the make file.
- Manually perform the commands inside the make file

Note: The makefile is located in the “JGammon.src\src” directory.

After JGammon loads, press “Game” and then “Show a demo game”.

Choosing the players

In order to change the players, open “Jgammon.java” in the package called “jgam” and change the names of the classes in lines 336 (white player) and 339 (red player).

The complete names of our AI classes that can be placed there (you should copy paste from here):

- jgam.ai.AggressiveAI
- jgam.ai.DefensiveAI
- jgam.ai.AllAroundAI
- jgam.ai.SR_AggressiveAI
- jgam.ai.SR_DefensiveAI
- jgam.ai.SR_AllAroundAI
- jgam.ai.MonkeyAI



The End!

