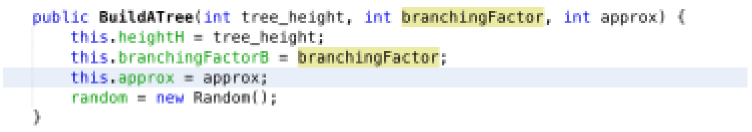
12252511
Zhesi Ning

COMP30260 "Artificial Intelligence for Games & Puzzles" Assignment 1

NegaScout/PVS with Principal Variation Reordering

Step One: Building a tree

The class BuildATree's constructor takes three positive integers as arguments:
1. branching factor - b
2. height - h
3. approximation - approx



```
public BuildATree(int tree_height, int branchingFactor, int approx) {
    this.heightH = tree_height;
    this.branchingFactorB = branchingFactor;
    this.approx = approx;
    random = new Random();
}
```

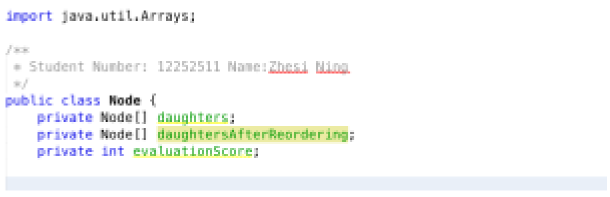
Each of the nodes have their own static evaluation(score) to indicate strength of performance.

The Node class contains:

1. an evaluation score: **private int evaluationScore;**

2. daughters, in some originally-generated order: **private Node[] daughters;**

3. daughters, in an order that might get changed by PV reordering during a search: **private Node[] daughtersAfterReordering;**



```
import java.util.Arrays;

/**
 * Student Number: 12252511 Name:Zhesi Ning
 */
public class Node {
    private Node[] daughters;
    private Node[] daughtersAfterReordering;
    private int evaluationScore;
```

We can generate a new tree with height h, branching factor b and approximation approx like this:

>new BuildATree(h, b, approx);

Step Two: Negamax-style alpha-beta algorithm with iterative deepening

Class AB is a simple implementation of negamax alpha-beta algorithm with iterative deepening.

(a) code to count the number of static evaluations performed:

The variable "private int noOfEvalutionPerformed " in class AB and class Pvs is used to count the number of static evaluations performed in both alpha beta search and PVS search.

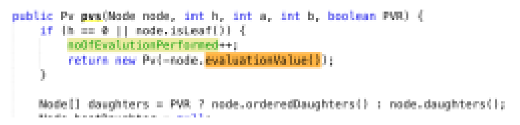
(b) code to return both a value and a principal variation:

As the screenshot shows below the method getScore gets the score value and the evaluationValue() returns the principal variation of the node.

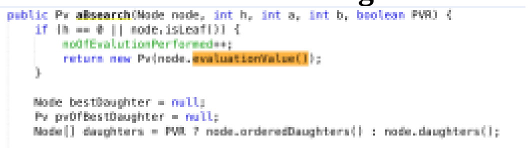
(c) code to unpick the returned values appropriately:

There is a toString method in class Pv where it can unpick the returned values.

(d) a parameter indicating whether or not to use the modified daughter:

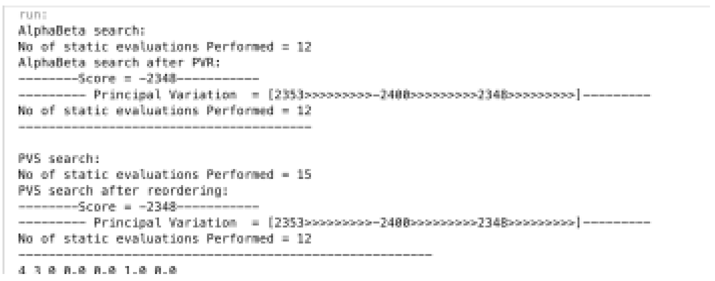


```
public Pv pvs(Node node, int h, int a, int b, boolean PVR) {
    if (h == 0 || node.isLeaf()) {
        noOfEvalutionPerformed++;
        return new Pv(-node.evaluationValue());
    }

    Node[] daughters = PVR ? node.orderedDaughters() : node.daughters();
```



```
public Pv aBsearch(Node node, int h, int a, int b, boolean PVR) {
    if (h == 0 || node.isLeaf()) {
        noOfEvalutionPerformed++;
        return new Pv(node.evaluationValue());
    }

    Node bestDaughter = null;
    Pv pvOfBestDaughter = null;
    Node[] daughters = PVR ? node.orderedDaughters() : node.daughters();
```

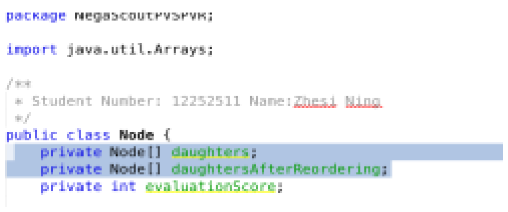
Both alpha beta search and PVS search take a Boolean value *PVR* in the search method. If *PVR* is true then this search will use the modified daughters after the reordering, so that the best-seen daughter is in the first position. Otherwise it is just a standard search with the original tree.



```
run:
AlphaBeta search:
No of static evaluations Performed = 12
AlphaBeta search after PVR:
--------Score = -2348-----------
--------- Principal Variation  = [2353>>>>>>>>>>-2400>>>>>>>>>>2348>>>>>>>>>>]---------
No of static evaluations Performed = 12
---------------------------------------

PVS search:
No of static evaluations Performed = 15
PVS search after reordering:
--------Score = -2348-----------
--------- Principal Variation  = [2353>>>>>>>>>>-2400>>>>>>>>>>2348>>>>>>>>>>]---------
No of static evaluations Performed = 12
-------------------------------------------------------
4 3 0 0.0 0.0 1.0 0.0
```

Step Three: Principal Variation Reordering

In class Node there are two arrays: one contains the daughters of the node and the other one is its daughters after performing Principal Variation Reordering.



```
package NegaScoutPVSPVR;

import java.util.Arrays;

/**
 * Student Number: 12252511 Name:Zhesi Ning
 */
public class Node {
    private Node[] daughters;
    private Node[] daughtersAfterReordering;
    private int evaluationScore;
```

Step Four: PVS with iterative deepening

Class Pvs is a simple implementation of the PVS search algorithm which calls the alpha beta search but tries with a narrower window every time. This increases the chance of a cut off. The algorithm uses iterative deepening as well.

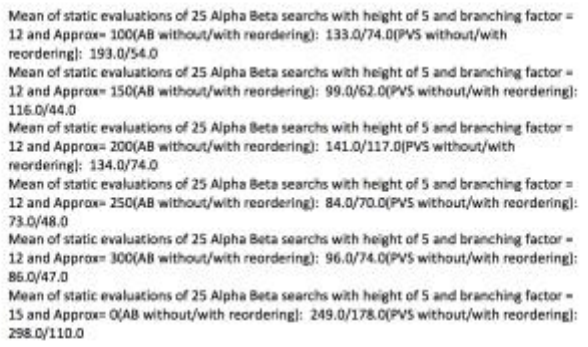
Step Five: Experiment and Report

In this step, I carried out a number of difference searches using both alpha beta search and PVS search with/without PV reordering. For each parameter combination below we generate 25 trees

12252511
Zhesi Ning

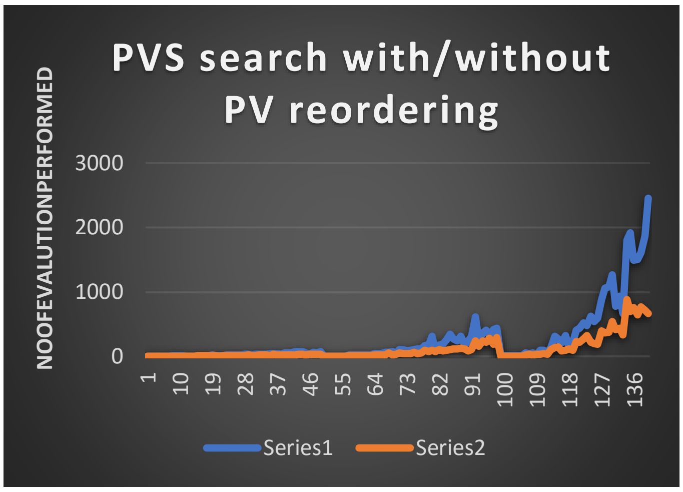
- tree height, from 4 to 6 in steps of 1
- branching factor, from 3 to 21 in steps of 3
- Approximation, from 0 to 300 in steps of 50

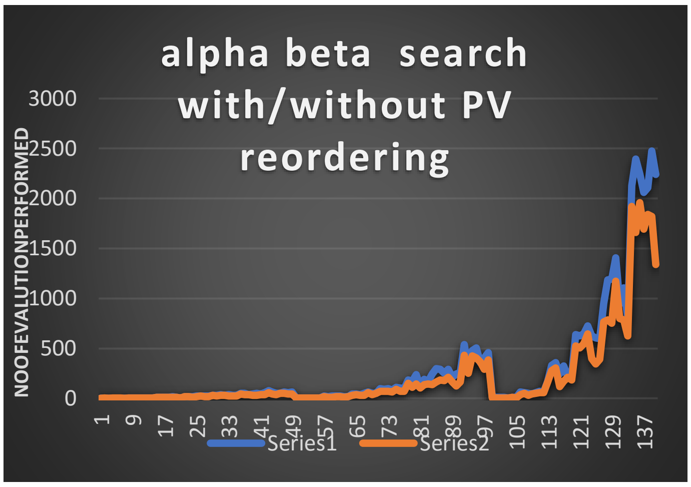
then we calculate the means of those 25 results for each combination, and compare the difference between them. Below image gives a rough idea of what the output data looks like (it took more than 20mins to run).



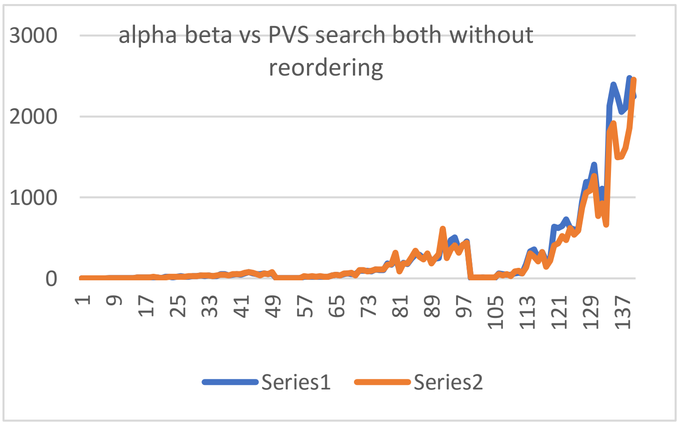
```
Mean of static evaluations of 25 Alpha Beta searchs with height of 5 and branching factor =
12 and Approx= 100(AB without/with reordering): 133.0/74.0(PVS without/with
reordering): 193.0/54.0
Mean of static evaluations of 25 Alpha Beta searchs with height of 5 and branching factor =
12 and Approx= 150(AB without/with reordering): 99.0/62.0(PVS without/with reordering):
116.0/44.0
Mean of static evaluations of 25 Alpha Beta searchs with height of 5 and branching factor =
12 and Approx= 200(AB without/with reordering): 141.0/117.0(PVS without/with
reordering): 134.0/74.0
Mean of static evaluations of 25 Alpha Beta searchs with height of 5 and branching factor =
12 and Approx= 250(AB without/with reordering): 84.0/70.0(PVS without/with reordering):
73.0/48.0
Mean of static evaluations of 25 Alpha Beta searchs with height of 5 and branching factor =
12 and Approx= 300(AB without/with reordering): 96.0/74.0(PVS without/with reordering):
86.0/47.0
Mean of static evaluations of 25 Alpha Beta searchs with height of 5 and branching factor =
15 and Approx= 0(AB without/with reordering): 249.0/178.0(PVS without/with reordering):
298.0/110.0
```

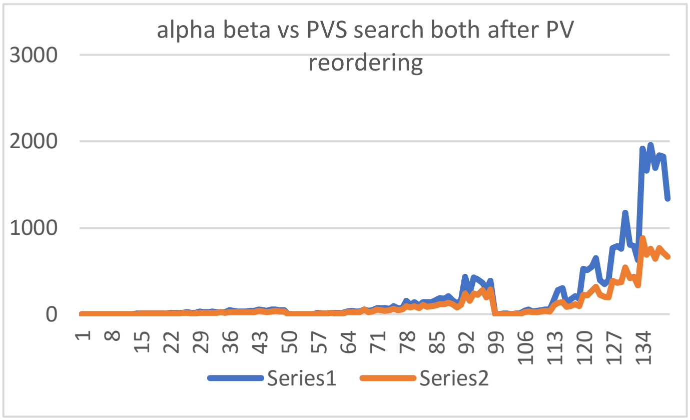
We can import the data into excel and plot some graphs





The graphs above in orange are searches after PV reordering and those in blue are searches without reordering. The graphs below in blue are alpha beta searches and those in orange are PVS searches.





As we obsereve from the graphes, generally speaking both alpha beta and pvs searches have better performance while using the tree after PV reordering. This is thanks to the principal variation. From an interior node is a sequence whose first element is the best child therefore we have a better chance to cut-off (prune) the tree. We also noticed that the bigger the branching factor and the deeper the tree, the performance difference between with and without pv reordering will be larger.

12252511
Zhesi Ning

There is a performance difference between alpha beta search and PVS search, since PVS search is calling the alpha-beta algorithm with varying values for its alpha and beta and it is “trying its luck” and hoping to get a chance to prune the rest of the tree in order to get a better performance. However, without PV reordering, the performance increase is pretty small since the scores of the children are distributed quite randomly. But after we reorder the children we can see a significant improvement for the performance because the best children will always at the first place and the search will go from the left to the right of the tree. Therefore, we can maximise the chance to prune the subtree.