/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Thinking Operation class.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Ramin Edjlal\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Drived Classess of Autamata Cellular Quantum Thinking Kernel\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 1394/12/19\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Crashed with Stack Overflow Exception\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Drives Caused Memory lack\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* New Version Cased Stack Overflow\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Scanning Four Dimension Homes of Thing Existences Taking A lot Of Time\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* All Data in This Scope From AllDraw Become Clear When Scope Changes\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Heuristic Work but the Movements And Attack Method Doesn’t work\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Probability Heuristic constant Table return\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Heuristic Working Not Constant Immunity\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Heuristic Constant Result Mechanism\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Things Order and Virtualization Error\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Misleading Things Order movement\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Multi Movements (3 ) In Chess Thinking\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Location of Horse 'Bob' (Gray) After Killer Un logically UnSelfSupported\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Check Thinking 'Alice' Malfunction\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* 'CheckMate' By 'Bob' Have Not Been Recognized.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* 'Check' By 'Bob' Not Recognized.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* 'Check' 'Alice' Detected. No ActionsString Was Done.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* 'Check' Mechanism Failure.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Strategy By 'Alice' Changed. 'Check' Not Recognized By 'Alice'.\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Heuristic Loop\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* 'Check' Mechanism For Penalty Regard Is Malfunction\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).

\* Things Location Failure. Row and Column of this Objects class Malfunction\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Malfunction Of Operating Lists in this class.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Some Movements of All Possible Movements is not Identified\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Malfunction Clone Data To be Copied. List Will be erased\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* King Cannot Hit UnSelfSupported Enemy Things at Check.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Thinking Time Taking al lot of time.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* There is No Reason For Mal Function of Thinking.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*[+]

\* Huristic SelfSupported at horse huristic cal at table content malfunction.\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*[+]

\* No Reason for malfunctioning of table content at huristic SelfSupported.\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*[+]

\* Thinking Finished Misleading.bool Variable of Think Finished Not Work on.\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* A non Identified King Table List Alice is in List and Unhabitly ignored.\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* The Location of Penalty Regard Mechansim is Misleading.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Penalty Reagrd List is Empty.No Misleading List of Penalty Regard Mec.\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* No Ilegal Non ObjectDanger and Check By 'Alice' at Current Game in PR Mech.\*\*\*\*\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* Mechansianm For Like Napeloonires KLish CheckMate is Incompletable.\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).

\* Ileegal Table Content Ignoring of Objects Kind.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).

\* Tree Construction of AStarGready is Uncompleted.Some Nodes Become Empty.\*\*\*\*RS\*\*0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<+>

\* All Penalty Leads to 16 Objects Unmovable or Make Penalty But in Reality Non Penalty Exist.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)

\* All Self and Enemy CheckMate Mechanisam is Logical else Mislaeading.\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).QC-Ok.

\* Proccess of Thinking Stop Misleading With Error.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}.QC-Ok.

\* All List of this class make differncy at several runable state of one table board state.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}..

\* Thinking Act Misleading.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}.

\* The Achmaz Removing and maybe SelfNotSupported in Attacker conflict and thus Ignore.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).

\* The Self Supporter in Attacker somthime goes to misleading act.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC-Ok.

\* Enemy Attacker Not Supported act Misleading.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC\_OK.

\* Heuristic proccesing dosne't haave any aim.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).

\* Rating of Alice as Computer Game is very weak as compatitor of users.\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_BAD.

\* Thinking gone to take some part of stones.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_BAD.

\* Thinking failed becuase of all possible movment penalties of first level\*\*.\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Object Dangour and Check is aditive of HeuristicCheckedand checked mated.\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Heuristics take some part of stone.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Branch Dept at Thinking Tree is low becuse of harware constraints and speed.\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_BAD.

\* Thinking falied becuase of All Possible of Penalties movments.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Tow Confliction Misleading in Self Attacked and King Dangoure Separatedly.\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Conflict in Restoring UsePenaltyRegardMechanisam value during User false.\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC\_OK.

\* Self Objects Movments Comes to Dangrous Location.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*).QC\_OK

\* Thinking Tree Construction was not Complition and have empty with no reason.\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*{\*}QC\_OK

\* Heuristic of 'Attack';'Movment';'Support';'CheckMate...' Undisiarable.\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*<\*>QC \_BAD

\* Huristic and Learning regime work in worth state.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC\_BAD

\* Mal Function in Boundray Conditions founding in Leaf Creation Tree.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC\_BAD

\* Index was out of range in same grope of thinking objects by hitting.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RC\*\*0.88\*\*1\*\*Risk Control\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(\*)QC\_BAD

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+:Sum(26)) (\*:Sum(1)) 5:(+:Sum(3)) 6.(+:Sum0.12\*\*4\*\*Managements and Cuation Programing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(+)) 7.(:Sum(1))

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

using System;

using System.Collections.Generic;

using System.Drawing;

using LearningMachine;

using System.IO;

using System.Threading.Tasks;

using System.Diagnostics;

namespace RefrigtzDLL

{

[Serializable]

public class ThinkingChess

{

StackFrame callStack = new StackFrame(1, true);

//Initiate Global and Static Variables.

public bool IsThereMateOfEnemy = false;

public static NetworkQuantumLearningKrinskyAtamata LearniningTable = null;

bool ThinkingAtRun = false;

public static String ActionsString = "";

String OutPutAction = "";

int ThinkingLevel = 0;

public List<bool[]> LearningVarsObject = new List<bool[]>();

public static bool LearningVarsCheckedMateOccured;

public static bool LearningVarsCheckedMateOccuredOneCheckedMate;

int DivisionPenaltyRegardHeuristicQueficient = 1;

public int SuppportCountStaticGray = 0;

public int SuppportCountStaticBrown = 0;

bool IsGardHighPriority = false;

const int ThresholdBlitz = 10000;

const int ThresholdFullGame = 20000;

public static double MaxHuristicx = Double.MinValue;

public bool MovementsAStarGreedyHuristicFoundT = false;

public bool IgnoreSelfObjectsT = false;

public bool UsePenaltyRegardMechnisamT = true;

public bool BestMovmentsT = false;

public bool PredictHuristicT = true;

public bool OnlySelfT = false;

public bool AStarGreedyHuristicT = false;

bool ArrangmentsChanged = false;

public int NumberOfPenalties = 0;

static int NumbersOfCurrentBranchesPenalties = 0;

public static int NumbersOfAllNode = 0;

/\*public int SodierMidle = 8;

public int SodierHigh = 16;

public int ElefantMidle = 2;

public int ElefantHigh = 4;

public int HourseMidle = 2;

public int HourseHight = 4;

public int CastleMidle = 2;

public int CastleHigh = 4;

public int MinisterMidle = 1;

public int MinisterHigh = 2;

public int KingMidle = 1;

public int KingHigh = 2;

\*/

public int SodierMidle = 0;

public int SodierHigh = 0;

public int ElefantMidle = 0;

public int ElefantHigh = 0;

public int HourseMidle = 0;

public int HourseHight = 0;

public int CastleMidle = 0;

public int CastleHigh = 0;

public int MinisterMidle = 0;

public int MinisterHigh = 0;

public int KingMidle = 0;

public int KingHigh = 0;

public static bool KingMaovableGray = false;

public static bool KingMaovableBrown = false;

public static int FoundFirstMating;

public int SodierValue = 1 \* 3;

public int ElefantValue = 2 \* 16;

public int HourseValue = 3 \* 8;

public int CastleValue = 5 \* 16;

public int MinisterValue = 8 \* 32;

public int KingValue = 10 \* 8;

public static int BeginThread = 0;

public static int EndThread = 0;

bool ExistingOfEnemyHiiting = false;

int IgnoreObjectDangour = -1;

public int CheckMateAStarGreedy = 0;

bool CheckMateOcuured = false;

int CurrentRow = -1, CurrentColumn = -1;

public bool IsCheck = false;

public int Kind = 0;

public List<int> HitNumber = new List<int>();

public static bool NotSolvedKingDanger = false;

public static bool ThinkingRun = false;

public int ThingsNumber = 0;

public int CurrentArray = 0;

public bool ThinkingBegin = false;

public bool ThinkingFinished = false;

public int IndexSoldier = 0;

public int IndexElefant = 0;

public int IndexHourse = 0;

public int IndexCastle = 0;

public int IndexMinister = 0;

public int IndexKing = 0;

static public int Index = 0;

static public int[,] RowColumn;

public List<int[]> RowColumnSoldier = new List<int[]>();

public List<int[]> RowColumnElefant = new List<int[]>();

public List<int[]> RowColumnHourse = new List<int[]>();

public List<int[]> RowColumnCastle = new List<int[]>();

public List<int[]> RowColumnMinister = new List<int[]>();

public List<int[]> RowColumnKing = new List<int[]>();

public int[,] TableT;

public List<int> HitNumberSoldier = new List<int>();

public List<int> HitNumberElefant = new List<int>();

public List<int> HitNumberHourse = new List<int>();

public List<int> HitNumberCastle = new List<int>();

public List<int> HitNumberMinister = new List<int>();

public List<int> HitNumberKing = new List<int>();

public int[,] TableConst;

public List<int[,]> TableListSolder = new List<int[,]>();

public List<int[,]> TableListElefant = new List<int[,]>();

public List<int[,]> TableListHourse = new List<int[,]>();

public List<int[,]> TableListCastle = new List<int[,]>();

public List<int[,]> TableListMinister = new List<int[,]>();

public List<int[,]> TableListKing = new List<int[,]>();

public List<double[]> HuristicListSolder = new List<double[]>();

public List<double[]> HuristicListElefant = new List<double[]>();

public List<double[]> HuristicListHourse = new List<double[]>();

public List<double[]> HuristicListCastle = new List<double[]>();

public List<double[]> HuristicListMinister = new List<double[]>();

public List<double[]> HuristicListKing = new List<double[]>();

public List<QuantumAtamata> PenaltyRegardListSolder = new List<QuantumAtamata>();

public List<QuantumAtamata> PenaltyRegardListElefant = new List<QuantumAtamata>();

public List<QuantumAtamata> PenaltyRegardListHourse = new List<QuantumAtamata>();

public List<QuantumAtamata> PenaltyRegardListCastle = new List<QuantumAtamata>();

public List<QuantumAtamata> PenaltyRegardListMinister = new List<QuantumAtamata>();

public List<QuantumAtamata> PenaltyRegardListKing = new List<QuantumAtamata>();

public int Max;

public int Row, Column;

public Color color;

public int Order;

[NonSerialized()] public Task t = null;

public List<AllDraw> AStarGreedy = new List<AllDraw>();

int[,] Value = new int[8, 8];

bool IgnoreFromCheckandMateHuristic = false;

int CurrentAStarGredyMax = -1;

List<int[,]> ObjectNumbers = new List<int[,]>();

///Log of Errors.

static void Log(Exception ex)

{

try

{

Object a = new Object();

lock (a)

{

string stackTrace = ex.ToString();

//Write to File.

File.AppendAllText(AllDraw.Root + "\\ErrorProgramRun.txt", stackTrace + ": On" + DateTime.Now.ToString()); /// path of file where stack trace will be stored.

}

}

catch (Exception t) { Log(t); }

}

void SetObjectNumbersInList(int[,] Tab)

{

SetObjectNumbers(Tab);

int[,] A = new int[2, 6];

A[0, 0] = SodierMidle;

A[1, 0] = SodierHigh;

A[0, 1] = ElefantMidle;

A[1, 1] = ElefantHigh;

A[0, 2] = HourseMidle;

A[1, 2] = HourseHight;

A[0, 3] = CastleMidle;

A[1, 3] = CastleHigh;

A[0, 4] = MinisterMidle;

A[1, 4] = MinisterHigh;

A[0, 5] = KingMidle;

A[1, 5] = KingHigh;

ObjectNumbers.Add(A);

}

public void SetObjectNumbers(int[,] TabS)

{

Object a = new Object();

lock (a)

{

SodierMidle = 0;

SodierHigh = 0;

ElefantMidle = 0;

ElefantHigh = 0;

HourseMidle = 0;

HourseHight = 0;

CastleMidle = 0;

CastleHigh = 0;

MinisterMidle = 0;

MinisterHigh = 0;

KingMidle = 0;

KingHigh = 0;

for (int h = 0; h < 8; h++)

for (int s = 0; s < 8; s++)

{

if (TabS[h, s] == 1)

{

SodierMidle++;

SodierHigh++;

}

else if (TabS[h, s] == 2)

{

ElefantMidle++;

ElefantHigh++;

}

else if (TabS[h, s] == 3)

{

HourseMidle++;

HourseHight++;

}

else if (TabS[h, s] == 4)

{

CastleMidle++;

CastleHigh++;

}

else if (TabS[h, s] == 5)

{

MinisterMidle++;

MinisterHigh++;

}

else if (TabS[h, s] == 6)

{

KingMidle++;

KingHigh++;

}

else

if (TabS[h, s] == -1)

{

SodierHigh++;

}

else if (TabS[h, s] == -2)

{

ElefantHigh++;

}

else if (TabS[h, s] == -3)

{

HourseHight++;

}

else if (TabS[h, s] == -4)

{

CastleHigh++;

}

else if (TabS[h, s] == -5)

{

MinisterHigh++;

}

else if (TabS[h, s] == -6)

{

KingHigh++;

}

}

}

}

//Constructor

public ThinkingChess(int CurrentAStarGredy, bool MovementsAStarGreedyHuristicTFou, bool IgnoreSelfObject, bool UsePenaltyRegardMechnisa, bool BestMovment, bool PredictHurist, bool OnlySel, bool AStarGreedyHuris, bool Arrangments, int i, int j)

{

Object O = new Object();

lock (O)

{

//Initiate Variables.

CurrentAStarGredyMax = CurrentAStarGredy;

MovementsAStarGreedyHuristicFoundT = MovementsAStarGreedyHuristicTFou;

IgnoreSelfObjectsT = IgnoreSelfObject;

UsePenaltyRegardMechnisamT = UsePenaltyRegardMechnisa;

BestMovmentsT = BestMovment;

PredictHuristicT = PredictHurist;

OnlySelfT = OnlySel;

AStarGreedyHuristicT = AStarGreedyHuris;

ArrangmentsChanged = Arrangments;

//SetObjectNumbers(TableConst);

Row = i;

Column = j;

//Clear Dearty Part.

/\*TableListSolder.Clear();

TableListElefant.Clear();

TableListHourse.Clear();

TableListCastle.Clear();

TableListMinister.Clear();

TableListKing.Clear();

RowColumnSoldier = new List<int[]>();

RowColumnElefant = new List<int[]>();

RowColumnHourse = new List<int[]>();

RowColumnCastle = new List<int[]>();

RowColumnMinister = new List<int[]>();

RowColumnKing = new List<int[]>();

HitNumberSoldier = new List<int>();

HitNumberElefant = new List<int>();

HitNumberHourse = new List<int>();

HitNumberCastle = new List<int>();

HitNumberMinister = new List<int>();

HitNumberKing = new List<int>();

PenaltyRegardListSolder = new List<QuantumAtamata>();

PenaltyRegardListElefant = new List<QuantumAtamata>();

PenaltyRegardListHourse = new List<QuantumAtamata>();

PenaltyRegardListCastle = new List<QuantumAtamata>();

PenaltyRegardListMinister = new List<QuantumAtamata>();

PenaltyRegardListKing = new List<QuantumAtamata>();

AStarGreedy = new List<AllDraw>();

\*/

//Network Quantum Atamata Book Initiate For Every Clone.

SetValueOfTabls(Row, Column);

}

}

void SetValueOfTabls(int Row, int Column)

{

Object o = new Object();

lock (o)

{

//for (int h = 0; h < 8; h++)

//for (int m = 0; m < 8; m++)

{

//if (h != Row || m != Column)

//return;

Value[Row, Column] = 0;

{

if (TableConst == null || TableConst[Row, Column] == 0)

return;

if (TableConst != null)

Value[Row, Column] += ObjectValueCalculator(TableConst, Order, Row, Column);

}

}

}

}

//determine When Arrangment of Table Objects is Validated at Begin.

bool BeginArragmentsOfOrderFinished(int[,] Table, int Order)

{

Object O = new Object();

lock (O)

{

int CH = 0;

if (ArrangmentsChanged)

{

if (Order == 1)

{

//Number of Gray Objects at Last Row Bottmm.

for (int i = 0; i < 2; i++)

for (int j = 6; j < 8; j++)

if (Table[i, j] > 0)

CH++;

}

else

{

//Number of Brown Objects at Last tow Row Upper.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 2; j++)

if (Table[i, j] < 0)

CH++;

}

}

else

{

if (Order == -1)

{

//Number of Brown Objects Table at Last tow row Uppper.

for (int i = 0; i < 8; i++)

for (int j = 6; j < 2; j++)

if (Table[i, j] > 0)

CH++;

}

else

{

//Number of Gray Objects Table at Last tow rown below.

for (int i = 0; i < 2; i++)

for (int j = 0; j < 8; j++)

if (Table[i, j] < 0)

CH++;

}

}

if (CH <= 8)

return true;

return false;

}

}

//Constructor

public ThinkingChess(int CurrentAStarGredy, bool MovementsAStarGreedyHuristicTFou, bool IgnoreSelfObject, bool UsePenaltyRegardMechnisa, bool BestMovment, bool PredictHurist, bool OnlySel, bool AStarGreedyHuris, bool Arrangments, int i, int j, Color a, int[,] Tab, int Ma, int Ord, bool ThinkingBeg, int CurA, int ThingN, int Kin)

{

Object O = new Object();

lock (O)

{

CurrentAStarGredyMax = CurrentAStarGredy;

MovementsAStarGreedyHuristicFoundT = MovementsAStarGreedyHuristicTFou;

IgnoreSelfObjectsT = IgnoreSelfObject;

UsePenaltyRegardMechnisamT = UsePenaltyRegardMechnisa;

BestMovmentsT = BestMovment;

PredictHuristicT = PredictHurist;

OnlySelfT = OnlySel;

AStarGreedyHuristicT = AStarGreedyHuris;

//Initiate Variables.

ArrangmentsChanged = Arrangments;

Kind = Kin;

SetObjectNumbers(Tab);

//THIS = TH;

AStarGreedy = new List<AllDraw>();

ThingsNumber = ThingN;

CurrentArray = CurA;

/\*TableListSolder.Clear();

TableListElefant.Clear();

TableListHourse.Clear();

TableListCastle.Clear();

TableListMinister.Clear();

TableListKing.Clear();

RowColumnSoldier = new List<int[]>();

RowColumnElefant = new List<int[]>();

RowColumnHourse = new List<int[]>();

RowColumnCastle = new List<int[]>();

RowColumnMinister = new List<int[]>();

RowColumnKing = new List<int[]>();

RowColumn = new int[1000000, 2];

HitNumberSoldier = new List<int>();

HitNumberElefant = new List<int>();

HitNumberHourse = new List<int>();

HitNumberCastle = new List<int>();

HitNumberMinister = new List<int>();

HitNumberKing = new List<int>();

PenaltyRegardListSolder = new List<QuantumAtamata>();

PenaltyRegardListElefant = new List<QuantumAtamata>();

PenaltyRegardListHourse = new List<QuantumAtamata>();

PenaltyRegardListCastle = new List<QuantumAtamata>();

PenaltyRegardListMinister = new List<QuantumAtamata>();

PenaltyRegardListKing = new List<QuantumAtamata>();

\*/

Row = i;

Column = j;

color = a;

Max = Ma;

TableT = Tab;

Index = 0;

IndexSoldier = 0;

IndexElefant = 0;

IndexHourse = 0;

IndexCastle = 0;

IndexMinister = 0;

IndexKing = 0;

TableConst = new int[8, 8];

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

TableConst[ii, jj] = Tab[ii, jj];

}

Order = Ord;

ThinkingBegin = ThinkingBeg;

//AStarGreedy = new List<AllDraw>();

/\*Object o = new Object();

lock (o)

{

for (int h = 0; h < 8; h++)

for (int m = 0; m < 8; m++)

{

if (TableConst != null)

{

if (TableConst[h, m] == 0)

continue;

Value[h, m] = ObjectValueCalculator(TableConst, Order, h, m);

}

}

}

\*/

//SetObjectNumbers(TableConst);

}

}

//Clone A Table

int[,] CloneATable(int[,] Tab)

{

Object O = new Object();

lock (O)

{

//Create and new an Object.

int[,] Table = new int[8, 8];

//Assigne Parameter To New Objects.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

Table[i, j] = Tab[i, j];

//Return New Object.

return Table;

}

}

//Clone A List.

int[] CloneAList(int[] Tab, int Count)

{

Object O = new Object();

lock (O)

{

//Initiate new Objects.

int[] Table = new int[Count];

//Asigne to new Objects.

for (int i = 0; i < Count; i++)

Table[i] = Tab[i];

//Retrun new Object.

return Table;

}

}

//Clone a copy of an array.

double[] CloneAList(double[] Tab, int Count)

{

Object O = new Object();

lock (O)

{

//Initiate New Object.

double[] Table = new double[Count];

//Assigne to new Object.,

for (int i = 0; i < Count; i++)

Table[i] = Tab[i];

//Return New Object.

return Table;

}

}

//Gwt Value of Book Netwrok Quantum Atamtat at Every Need time form parameters index.

int GetValue(int i, int j)

{

Object O = new Object();

lock (O)

{

return Value[i, j];

}

}

///Clone a Copy.

public void Clone(ref ThinkingChess AA)

{

Object O = new Object();

lock (O)

{

//Assignment Content to New Content Object.

//Initaite New Object.

if (AA == null)

AA = new ThinkingChess(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Row, Column);

AA.ArrangmentsChanged = ArrangmentsChanged;

//When Depth Object is not NULL.

if (AStarGreedy.Count != 0)

{

AA.AStarGreedy = new System.Collections.Generic.List<AllDraw>();

//For All Depth(s).

for (int i = 0; i < AStarGreedy.Count; i++)

{

try

{

//Clone a Copy From Depth Objects.

AStarGreedy[i].Clone(AA.AStarGreedy[i]);

}

catch (Exception tt) { Log(tt); }

}

}

//For All Moves Indexx Solders List Count.

for (int j = 0; j < RowColumnSoldier.Count; j++)

//Add a Clone To New Solder indexx Object.

AA.RowColumnSoldier.Add(CloneAList(RowColumnSoldier[j], 2));

//For All Castle List Count.

for (int j = 0; j < RowColumnCastle.Count; j++)

//Add a Clone to New Castle index Objects List.

AA.RowColumnCastle.Add(CloneAList(RowColumnCastle[j], 2));

//For All Elephant index List Count.

for (int j = 0; j < RowColumnElefant.Count; j++)

//Add a Clone to New Elephant Object List.

AA.RowColumnElefant.Add(CloneAList(RowColumnElefant[j], 2));

//For All Hourse index List Count.

for (int j = 0; j < RowColumnHourse.Count; j++)

//Add a Clone to New Hourse index List.

AA.RowColumnHourse.Add(CloneAList(RowColumnHourse[j], 2));

//For All King index List Count.

for (int j = 0; j < RowColumnKing.Count; j++)

//Add a Clone To New King Object List.

AA.RowColumnKing.Add(CloneAList(RowColumnKing[j], 2));

//For All Minister index Count.

for (int j = 0; j < RowColumnMinister.Count; j++)

//Add a Clone To Minister New index List.

AA.RowColumnMinister.Add(CloneAList(RowColumnMinister[j], 2));

//Assgine thread.

AA.t = t;

//Create and Initiate new Table Object.

AA.TableT = new int[8, 8];

//Create and Initaite New Table Object.

AA.TableConst = new int[8, 8];

//if Table is not NULL>

if (TableT != null)

//For All Items in Table Object.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

//Assgine Table items in New Table Object.

AA.TableT[i, j] = TableT[i, j];

//If Table is Not Null.

if (TableConst != null)

//For All Items in Table Object.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

//Assignm Items in New Table Object.

AA.TableConst[i, j] = TableConst[i, j];

//For All Table State Movements in Castles Objects.

for (int i = 0; i < TableListCastle.Count; i++)

//Add aclon of a Table in New Briges Table List.

AA.TableListCastle.Add(CloneATable(TableListCastle[i]));

//For All Table List Movements in Elephant Objects

for (int i = 0; i < TableListElefant.Count; i++)

//Add a Clone of Tables in Elephant Mevments Obejcts List To New One.

AA.TableListElefant.Add(CloneATable(TableListElefant[i]));

//For All Hourse Table Movemnts items.

for (int i = 0; i < TableListHourse.Count; i++)

//Add a Clone of Hourse Table Movement in New List.

AA.TableListHourse.Add(CloneATable(TableListHourse[i]));

//For All King Tables Movment Count.

for (int i = 0; i < TableListKing.Count; i++)

//Add a Clone To New King Table List.

AA.TableListKing.Add(CloneATable(TableListKing[i]));

//For All Minister Table Movment Items.

for (int i = 0; i < TableListMinister.Count; i++)

//Add a clone To New Minister Table Movment List.

AA.TableListMinister.Add(CloneATable(TableListMinister[i]));

//For All Solder Table Movment Count.

for (int i = 0; i < TableListSolder.Count; i++)

//Add a Clone of Table item to New Table List Movments.

AA.TableListSolder.Add(CloneATable(TableListSolder[i]));

//For All Solder Husrist List Count.

for (int i = 0; i < HuristicListSolder.Count; i++)

//Ad a Clone of Hueristic Solders To New List.

AA.HuristicListSolder.Add(CloneAList(HuristicListSolder[i], 4));

//For All Elephant Huristic List Count.

for (int i = 0; i < HuristicListElefant.Count; i++)

//Add A Clone of Copy to New Elephant Huristic List.

AA.HuristicListElefant.Add(CloneAList(HuristicListElefant[i], 4));

//For All Hours Huristic Hourse Count.

for (int i = 0; i < HuristicListHourse.Count; i++)

//Add a Clone of Copy To New Housre Huristic List.

AA.HuristicListHourse.Add(CloneAList(HuristicListHourse[i], 4));

//For All Castles Huristic List Count.

for (int i = 0; i < HuristicListCastle.Count; i++)

//Add a Clone of Copy to New Castles Huristic List.

AA.HuristicListCastle.Add(CloneAList(HuristicListCastle[i], 4));

//For All Minister Huristic List Count.

for (int i = 0; i < HuristicListMinister.Count; i++)

//Add a Clone of Copy to New Minister List.

AA.HuristicListMinister.Add(CloneAList(HuristicListMinister[i], 4));

//For All King Husrict List Items.

for (int i = 0; i < HuristicListKing.Count; i++)

//Add a Clone of Copy to New King Hursitic List.

AA.HuristicListKing.Add(CloneAList(HuristicListKing[i], 4));

//Initiate and create Penalty Solder List.

AA.PenaltyRegardListSolder = new List<QuantumAtamata>();

//For All Solder Penalty List Count.

if (Kind == 1)

{

AA.PenaltyRegardListSolder = new List<QuantumAtamata>();

for (int i = 0; i < PenaltyRegardListSolder.Count; i++)

{

//Initiate a new Quantum Atamata Object

//QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

//Add New Object Create to New Penalty Solder List.

AA.PenaltyRegardListSolder.Add(PenaltyRegardListSolder[i]);

}

}

else

if (Kind == 2)

{

//Initaite and Create Elephant Penalty List Object.

AA.PenaltyRegardListElefant = new List<QuantumAtamata>();

//For All Elepahtn Penalty List Count.

for (int i = 0; i < PenaltyRegardListElefant.Count; i++)

{

//Initiate a new Quantum Atamata Object

//QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

//Clone a Copy Of Penalty Elephant.

AA.PenaltyRegardListElefant.Add(PenaltyRegardListElefant[i]);

//Add New Object Create to New Penalty Elephant List.

//AA.PenaltyRegardListElefant.Add(Current);

}

}

else

if (Kind == 3)

{

//Initaite and Create Hourse Penalty List Object.

AA.PenaltyRegardListHourse = new List<QuantumAtamata>();

//For All Solder Hourse List Count.

for (int i = 0; i < PenaltyRegardListHourse.Count; i++)

{

//Initiate a new Quantum Atamata Object

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

//Clone a Copy Of Penalty Hourse.

//PenaltyRegardListHourse[i].Clone(ref Current);

//Add New Object Create to New Penalty Hourse List.

AA.PenaltyRegardListHourse.Add(PenaltyRegardListHourse[i]);

}

}

else

if (Kind == 4)

{

//Initaite and Create Castles Penalty List Object.

AA.PenaltyRegardListCastle = new List<QuantumAtamata>();

//For All Solder Castle List Count.

for (int i = 0; i < PenaltyRegardListCastle.Count; i++)

{

//Initiate a new Quantum Atamata Object

//QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

//Clone a Copy Of Penalty Castles.

//PenaltyRegardListCastle[i].Clone(ref Current);

//Add New Object Create to New Penalty Castles List.

AA.PenaltyRegardListCastle.Add(PenaltyRegardListCastle[i]);

}

}

else

if (Kind == 5)

{

//Initaite and Create Minister Penalty List Object.

AA.PenaltyRegardListMinister = new List<QuantumAtamata>();

//For All Solder Minster List Count.

for (int i = 0; i < PenaltyRegardListMinister.Count; i++)

{

//Initiate a new Quantum Atamata Object

//QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

//Clone a Copy Of Penalty Minsiter.

//PenaltyRegardListMinister[i].Clone(ref Current);

//Add New Object Create to New Penalty Minsietr List.

AA.PenaltyRegardListMinister.Add(PenaltyRegardListMinister[i]);

}

}

else

if (Kind == 6)

{

//Initaite and Create King Penalty List Object.

AA.PenaltyRegardListKing = new List<QuantumAtamata>();

//For All Solder King List Count.

for (int i = 0; i < PenaltyRegardListKing.Count; i++)

{

//Initiate a new Quantum Atamata Object

//QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

//Clone a Copy Of Penalty King.

//PenaltyRegardListKing[i].Clone(ref Current);

//Add New Object Create to New Penalty King List.

AA.PenaltyRegardListKing.Add(PenaltyRegardListKing[i]);

}

}

//Iniktiate Same Obejcts to New Same Obejcts.

AA.AStarGreedy = AStarGreedy;

AA.CastleValue = CastleValue;

AA.color = color;

AA.Column = Column;

AA.CurrentArray = CurrentArray;

AA.CurrentColumn = CurrentColumn;

AA.CurrentRow = CurrentRow;

AA.ElefantValue = ElefantValue;

AA.ExistingOfEnemyHiiting = ExistingOfEnemyHiiting;

AA.HourseValue = HourseValue;

AA.IgnoreObjectDangour = IgnoreObjectDangour;

AA.IndexCastle = IndexCastle;

AA.IndexElefant = IndexElefant;

AA.IndexHourse = IndexHourse;

AA.IndexKing = IndexKing;

AA.IndexMinister = IndexMinister;

AA.IndexSoldier = IndexSoldier;

AA.IsCheck = IsCheck;

AA.Kind = Kind;

AA.KingValue = KingValue;

AA.CheckMateAStarGreedy = CheckMateAStarGreedy;

AA.CheckMateOcuured = CheckMateOcuured;

AA.Max = Max;

AA.MinisterValue = MinisterValue;

AA.Order = Order;

AA.Row = Row;

AA.SodierValue = SodierValue;

AA.ThingsNumber = ThingsNumber;

AA.ThinkingBegin = ThinkingBegin;

AA.ThinkingFinished = ThinkingFinished;

}

}

///Huristic of Attacker.

double HuristicAttack(bool Before, int[,] Table, int Ord, Color aa, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

double HuristicAttackValue = 0;

double HA = 0;

int DumOrder = Order;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

///When AStarGreedy Huristic is Not Assigned.

try

{

//When Huristic is not Greedy.

if (!AStarGreedyHuristicT)

{

int Order = new int();

Color a = new Color();

a = aa;

if (i == ii && j == jj)

return HuristicAttackValue;

double Sign = new double();

Order = DummyOrder;

///When Attack is true. means [ii,jj] is in Attacked [i,j].

///What is Attack!

///Ans:When [ii,jj] is Attacked [i,j] return true when enemy is located in [ii,jj].

if (Table[ii, jj] > 0 && DummyOrder == -1 && Table[i, j] < 0)

{

Order = -1;

Sign = 1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

a = Color.Brown;

}

else if (Table[ii, jj] < 0 && DummyOrder == 1 && Table[i, j] > 0)

{

Order = 1;

Sign = 1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

a = Color.Gray;

}

else

return HuristicAttackValue;

//For Attack Movments.- GetObjectValueHuristic

Object O1 = new Object();

lock (O1)

{

if (Before)

{

if (Attack(Table, i, j, ii, jj, a, Order))

{

//Find Huristic Value Of Current and Add to Sumation.

HA += (Sign \* (System.Math.Abs(GetValue(ii, jj) + GetValue(i, j))));

//When there is supporter of attacked Objects take huristic negative else take muliply sign and muliply huristic.

bool Supported = new bool();

Supported = false;

//For All Enemy Obejcts.

Parallel.For(0, 8, g =>

{

if (Supported)

return;

Parallel.For(0, 8, h =>

{

if (Supported)

return;

//Ignore Of Self Objects.

if (Order == 1 && Table[g, h] >= 0)

return;

if (Order == -1 && Table[g, h] <= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

aaa = Color.Gray;

if (Order \* -1 == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

//When Enemy is Supported.

bool A = new bool();

Object O2 = new Object();

lock (O2)

{

A = Support(Table, g, h, ii, jj, aaa, Order \* -1);

}

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Supported = true;

return;

}

});

if (Supported)

return;

});

if (!Supported)

//When is Not Supported multyply 20.

HA \*= 20;

else

//When is Supported Multyply -20.

HA \*= -20;

}

}

}

}

//For All Table Homes find Attack Huristic.

else

{

int Order = new int();

Color a = new Color();

a = aa;

//Ignore of Current.

if (i == ii && j == jj)

return HuristicAttackValue;

Order = DummyOrder;

double Sign = 1;

///When Attack is true. means [ii,jj] is in Attacked [i,j].

///What is Attack!

///Ans:When [ii,jj] is Attacked [i,j] return true when enemy is located in [ii,jj].

if (Table[ii, jj] > 0 && DummyOrder == -1 && Table[i, j] < 0)

{

Order = -1;

Sign = 1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

a = Color.Brown;

}

else if (Table[ii, jj] < 0 && DummyOrder == 1 && Table[i, j] > 0)

{

Order = 1;

Sign = 1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

a = Color.Gray;

}

else

return HuristicAttackValue;

bool Supported = new bool();

Supported = false;

//For Attack Movments.

Object O2 = new Object();

lock (O2)

{

if (Before)

{

if (Attack(Table, i, j, ii, jj, a, Order))

{

HA += (Sign \* (System.Math.Abs(GetValue(ii, jj) + GetValue(i, j)

)));

//When there is supporter of attacked Objects take huristic negative else take muliply sign and muliply huristic.

//For All Enemy Obejcts.

Parallel.For(0, 8, g =>

{

if (Supported)

return;

Parallel.For(0, 8, h =>

{

if (Supported)

return;

//Ignore Of Self Objects.

if (Order == 1 && Table[g, h] >= 0)

return;

if (Order == -1 && Table[g, h] <= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

aaa = Color.Gray;

if (Order \* -1 == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

//When Enemy is Supported.

bool A = new bool();

Object O1 = new Object();

lock (O1)

{

A = Support(Table, g, h, ii, jj, aaa, Order \* -1);

}

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Supported = true;

return;

}

});

if (Supported)

return;

});

//When there is supported take positive multiply else take negative multiply.

if (!Supported)

//When is Not Supported multyply 20.

HA \*= 20;

else

//When is Supported Multyply -20.

HA \*= -20;

}

}

}

}

}

catch (Exception t)

{

Log(t);

}

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

Order = DumOrder;

//Initiate to Begin Call Orders.

return HA;

}

}

double HuristicReducsedAttack(bool Before, int[,] Table, int Ord, Color aa, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

double HuristicReducedAttackValue = 0;

//Initiate Objects.

double HA = 0;

int DumOrder = Order;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

///When AStarGreedy Huristic is Not Assigned.

try

{

if (!AStarGreedyHuristicT)

{

///For Every Objects.

int Order = new int();

Color a = new Color();

a = aa;

if (i == ii && j == jj)

return HuristicReducedAttackValue;

double Sign = 1;

Order = DummyOrder;

///When Attack is true. means [ii,jj] is in Attacked [i,j].

///What is Attack!

///Ans:When [ii,jj] is Attacked [i,j] return true when enemy is located in [ii,jj].

if (Table[ii, jj] > 0 && DummyOrder == -1 && Table[i, j] < 0)

{

Order = 1;

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = 1;

a = Color.Gray;

}

else if (Table[ii, jj] < 0 && DummyOrder == 1 && Table[i, j] > 0)

{

Order = -1;

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

a = Color.Brown;

}

else

return HuristicReducedAttackValue;

//For Attack Movments.

Object O1 = new Object();

lock (O1)

{

if (Before)

{

if (Attack(Table, ii, jj, i, j, a, Order))

{

HA += (Sign \* (System.Math.Abs(GetValue(i, j) + GetValue(ii, jj))));

bool Reduced = new bool();

Reduced = false;

Parallel.For(0, 8, g =>

{

if (Reduced)

return;

Parallel.For(0, 8, h =>

{

if (Reduced)

return;

//Ignore Of Enemy Objects.

if (Order == 1 && Table[g, h] <= 0)

return;

if (Order == -1 && Table[g, h] >= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

if (Order \* -1 == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

bool A = new bool();

Object O2 = new Object();

lock (O2)

{

A = Support(Table, g, h, ii, jj, aaa, Order \* 1);

}

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Reduced = true;

return;

}

});

if (Reduced)

return;

});

if (Reduced)

//When is Not Supported multyply 100.

HA \*= 20;

else

//When is Supported Multyply -100.

HA \*= -20;

}

}

}

}

//For All Table Homes find Attack Huristic.

else

{

int Order = new int();

Color a = new Color();

a = aa;

if (i == ii && j == jj)

return HuristicReducedAttackValue;

Order = DummyOrder;

double Sign = 1;

///When Attack is true. means [ii,jj] is in Attacked [i,j].

///What is Attack!

///Ans:When [ii,jj] is Attacked [i,j] return true when enemy is located in [ii,jj].

if (Table[ii, jj] > 0 && DummyOrder == -1 && Table[i, j] < 0)

{

Order = 1;

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = 1;

a = Color.Gray;

}

else if (Table[ii, jj] < 0 && DummyOrder == 1 && Table[i, j] > 0)

{

Order = -1;

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

a = Color.Brown;

}

else

return HuristicReducedAttackValue;

//For Attack Movments.

Object O1 = new Object();

lock (O1)

{

if (Before)

{

if (Attack(Table, ii, jj, i, j, a, Order))

{

HA += (Sign \* (System.Math.Abs(GetValue(i, j) + GetValue(ii, jj))));

bool Reduced = new bool();

Reduced = false;

//For All Self Obejcts.

Parallel.For(0, 8, g =>

{

if (Reduced)

return;

Parallel.For(0, 8, h =>

{

if (Reduced)

return;

//Ignore Of Enemy Objects.

if (Order == 1 && Table[g, h] <= 0)

return;

if (Order == -1 && Table[g, h] >= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

if (Order \* -1 == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

bool A = new bool();

Object O2 = new Object();

lock (O2)

{

A = Support(Table, g, h, ii, jj, aaa, Order \* 1);

}

//When Self is Supported.

if (A)

{

//Assgine variable.

Reduced = true;

return;

}

});

if (Reduced)

return;

});

if (Reduced)

//When is Not Supported multyply 100.

HA \*= 20;

else

//When is Supported Multyply -100.

HA \*= -20;

}

}

}

}

}

catch (Exception t)

{

Log(t);

}

//Initiate to Begin Call Orders.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

Order = DumOrder;

//Add Local Huristic to Global One.

return HA;

}

}

///Value of Object method.

int GetObjectValue(int[,] Tabl, int ii, int jj, int Order)

{

Object O = new Object();

lock (O)

{

return System.Math.Abs(Tabl[ii, jj]);

}

}

///Huristic of ObjectDanger.

double HuristicObjectDangour(int[,] Table, int Order, Color a, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

double HuristicObjectDangourCheckMateValue = 0;

double HA = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

///When There is no AStarGreedyHuristicT

try

{

if (!AStarGreedyHuristicT)

{

///For All Object in Current Table.

if (i == ii && j == jj)

return HuristicObjectDangourCheckMateValue;

Order = DummyOrder;

double Sign = 1;

///When ObjectDanger is true. means [ii,jj] is in ObjectDanger by [i,j].

///What is ObjectDanger!

///Ans:When [i,j] is Attacked [ii,jj] return true when enemy is located in [ii,jj].

if (Table[ii, jj] > 0 && DummyOrder == -1 && Table[i, j] < 0)

{

Order = 1;

Object O1 = new Object();

lock (O1)

{

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = 1;

}

a = Color.Gray;

}

else if (Table[ii, jj] < 0 && DummyOrder == 1 && Table[i, j] > 0)

{

Order = -1;

Object O1 = new Object();

lock (O1)

{

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

}

a = Color.Brown;

}

else

return HuristicObjectDangourCheckMateValue;

//For ObjectDanger Movments.

if (ObjectDanger(Table, ii, jj, i, j, a, Order))

{

//Find Local Sumation of ObjectDanger Huristic.

HA += Sign \* (GetObjectValue(Table, ii, jj, Order) + GetObjectValue(Table, i, j, Order));

}

}

//For All Table Home Find ObjectDanger Huristic

else

{

if (i == ii && j == jj)

return HuristicObjectDangourCheckMateValue;

double Sign = 1;

///When ObjectDanger is true. means [ii,jj] is in ObjectDanger by [i,j].

///What is ObjectDanger!

///Ans:When [i,j] is Attacked [ii,jj] return true when enemy is located in [ii,jj].

if (Table[ii, jj] > 0 && DummyOrder == -1 && Table[i, j] < 0)

{

Order = 1;

Object O2 = new Object();

lock (O2)

{

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = 1;

}

a = Color.Gray;

}

else if (Table[ii, jj] < 0 && DummyOrder == 1 && Table[i, j] > 0)

{

Order = -1;

Object O3 = new Object();

lock (O3)

{

Sign = -1 \* AllDraw.SignAttack;

ChessRules.CurrentOrder = -1;

}

a = Color.Brown;

}

else

return HuristicObjectDangourCheckMateValue;

//For ObjectDanger Movments.

Object O1 = new Object();

lock (O1)

{

if (ObjectDanger(Table, ii, jj, i, j, a, Order))

{

//Find Local Sumation of ObjectDanger Huristic.

HA += Sign \* (GetObjectValue(Table, ii, jj, Order) + GetObjectValue(Table, i, j, Order));

}

}

}

}

catch (Exception t)

{

Log(t);

}

//Initiate Orders to Call Begining.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//Assignments of Global Huristic with Local One.

//return Local Huristic.

return HA;

}

}

double HuristicKiller(int Killed, int[,] Tabl, int i, int j, int ii, int jj, int Ord, Color aa, bool Hit)

{

Object O = new Object();

lock (O)

{

int[,] Tab = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Tabl[ik, jk];

double HuristicKillerValue = 0;

//Defualt is Gray Order.

double HA = 0.0;

double Sign = AllDraw.SignKiller;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

//Make live when there is killed.

if (Killed != 0)

{

Tab[ii, jj] = Tab[i, j];

Tab[i, j] = Killed;

}

try

{

int Order = new int();

Order = DummyOrder;

Color a = new Color();

a = aa;

Color colorAS = a;

//Ignore of Self.

if (Order == 1 && Tab[ii, jj] >= 0)

return HuristicKillerValue;

if (Order == -1 && Tab[ii, jj] <= 0)

return HuristicKillerValue;

bool EnemyNotSupported = false;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//Wehn Curfrent Movemnet is on attack.

Object O1 = new Object();

lock (O1)

{

EnemyNotSupported = InAttackEnemyThatIsNotSupported(Killed, Tab, Order, aa, i, j, ii, jj);

//When there is Attacks to Current Objects and is killable..

if (Attack(Tab, i, j, ii, jj, a, Order) && EnemyNotSupported)

{

//Huristic positive.

HA += AllDraw.SignKiller \* (double)((GetValue(i, j) + GetValue(ii, jj)

) \* Sign);

}

a = colorAS;

}

}

catch (Exception t)

{

Log(t);

}

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

return HA;

}

}

//Attacks Of Enemy that is not Supported.QC\_OK

bool InAttackEnemyThatIsNotSupported(int Kilded, int[,] Table, int Order, Color a, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

//Initiate Global Variables.

int Ord = Order;

bool S = true;

//int i = iij, j = jji;

bool EnemyNotSupported = true;

if (Kilded != 0)

{

EnemyNotSupported = true;

//Enemy

Parallel.For(0, 8, iii =>

{

Parallel.For(0, 8, jjj =>

{

if (!EnemyNotSupported)

return;

int Order1 = new int();

Order1 = Ord;

int[,] Tab = new int[8, 8];

Parallel.For(0, 8, ik =>

{

if (!EnemyNotSupported)

return;

Parallel.For(0, 8, jk =>

{

Object O3 = new Object();

lock (O3)

{

Tab[ik, jk] = Table[ik, jk];

}

});

});

Object O2 = new Object();

lock (O2)

{

Tab[i, j] = Tab[ii, jj];

Tab[ii, jj] = Kilded;

}

//Ignore of Current

if (Order1 == 1 && Tab[iii, jjj] >= 0)

return;

else

if (Order1 == -1 && Tab[iii, jjj] <= 0)

return;

a = Color.Gray;

if (Order1 \* -1 == -1)

a = Color.Brown;

//When Enemy is Supported.

Object O1 = new Object();

lock (O1)

{

if (Support(Tab, iii, jjj, ii, jj, a, Order1 \* -1)

&& GetValue(i, j) >= GetValue(ii, jj)

)

//Wehn [i,j] (Current) is less or equal than [ii,jj] (Enemy)

//EnemyNotSupported method Should return [valid]

//By this situation return not valid

{

EnemyNotSupported = false;

return;

}

}

});

if (!EnemyNotSupported)

return;

});

if (EnemyNotSupported)

S = false;

}

//When S is not valid there is one node in [EnemyNotSupported]

if (!S)

{

Order = Ord;

return true;

}

Order = Ord;

return false;

}

}

//When at least one Attacked Self Object return true.

bool InAttackEnemyThatIsNotSupportedAll(bool EnemyIsValuable, int[,] Table, int Order, Color a, int ij, int ji, int iij, int jji, ref List<int[]> ValuableEnemyNotSupported)

{

Object O = new Object();

lock (O)

{

//Initiate Global Variables.

int Ord = Order;

Object O4 = new Object();

lock (O4)

{

int[,] Tab = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Table[ik, jk];

bool S = true;

bool EnemyNotSupported = true;

bool InAttackedNotEnemySupported = false;

//int i = iij, j = jji;

//For Current

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

//Ignore of Enemy

if (Order == 1 && Tab[i, j] <= 0)

continue;

else

if (Order == -1 && Tab[i, j] >= 0)

continue;

//For Enemies.

for (int ii = 0; ii < 8; ii++)

{

for (int jj = 0; jj < 8; jj++)

{

//Ignore of Curent

if (Order == 1 && Tab[ii, jj] >= 0)

continue;

else

if (Order == -1 && Tab[ii, jj] <= 0)

continue;

Object O1 = new Object();

lock (O1)

{

if (EnemyIsValuable && (!IsObjectValaubleObjectEnemy(ii, jj, Tab[ii, jj], ref ValuableEnemyNotSupported)))

continue;

EnemyNotSupported = true;

InAttackedNotEnemySupported = false;

if (Attack(Tab, i, j, ii, jj, a, Order))

{

InAttackedNotEnemySupported = true;

//Enemy

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

//Ignore of Current

if (Order == 1 && Tab[iii, jjj] >= 0)

continue;

else

if (Order == -1 && Tab[iii, jjj] <= 0)

continue;

a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

//

if (Support(Tab, iii, jjj, ii, jj, a, Order \* -1)

//&& (GetObjectValue(Tab, i, j, Order) >= GetObjectValue(Tab, ii, jj, Order \* -1)

//Wehn [i,j] (Current) is less or equal than [ii,jj] (Enemy)

//EnemyNotSupported method Should return [valid]

//By this situation return not valid

)

{

EnemyNotSupported = false;

}

}

if (!EnemyNotSupported)

break;

}

}

if (EnemyNotSupported && InAttackedNotEnemySupported)

{

S = false;

break;

}

}

}

if (!S)

{

break;

}

}

if (!S)

{

break;

}

}

if (!S)

{

break;

}

}

//When there is at leat tow enmy of attackment.

if (!S)

{

Order = Ord;

return true;

}

Order = Ord;

}

return false;

}

}

//When there is more than tow self object not supported on atacked by movement return true.

int IsNotSafeToMoveAenemeyToAttackMoreThanTowObject(int AttackCount, int[,] Table, int Order, int i, int j, int ii, int jj)

{

//For All Enemie

Object O1 = new Object();

lock (O1)

{

//Ignore of Self

if (Order == 1 && Table[i, j] >= 0)

return 0;

if (Order == -1 && Table[i, j] <= 0)

return 0;

//For All Self and Empty.

//Ignore of Enemy.

if (Order == 1 && Table[ii, jj] < 0)

return 0;

if (Order == -1 && Table[ii, jj] > 0)

return 0;

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[i, j], Table, Order \* -1, i, j);

Color a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

int[,] Tab = new int[8, 8];

Object O = new Object();

lock (O)

{

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Table[ik, jk];

}

//When there is attack to some self node.

Object OO = new Object();

lock (OO)

{

if (A.Rules(i, j, ii, jj, a, Tab[i, j]))

{

//take move

Tab[ii, jj] = Tab[i, j];

Tab[i, j] = 0;

AttackCount = 0;

//For All Self

Parallel.For(0, 8, iii =>

{

//if (AttackCount > 1)

//return;

Parallel.For(0, 8, jjj =>

{

if (AttackCount > 1)

return;

//Ignore of Enemy.

if (Order == 1 && Tab[iii, jjj] <= 0)

return;

if (Order == -1 && Tab[iii, jjj] >= 0)

return;

a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

//when there is attack to some self node.

if (Attack(Tab, ii, jj, iii, jjj, a, Order \* -1))

{

bool Supporte = false;

//For All Self

Parallel.For(0, 8, iiii =>

{

if (AttackCount > 1)

return;

Parallel.For(0, 8, jjjj =>

{

if (AttackCount > 1)

return;

//Ignore of Enemy.

if (Order == 1 && Tab[iiii, jjjj] <= 0)

return;

if (Order == -1 && Tab[iiii, jjjj] >= 0)

return;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//when there is attack of self node to that enemy node.

if (Support(Tab, iiii, jjjj, iii, jjj, a, Order) || Attack(Tab, iiii, jjjj, ii, jj, a, Order))

{

Supporte = true;

return;

}

});

});

if (!Supporte)

AttackCount++;

}

else

return;

if (AttackCount > 1)

return;

});

if (AttackCount > 1)

return;

});

}

else

return 0;

}

return AttackCount;

}

}

//Supported of Self that is Not Attacks.QC\_BAD

bool InAttackSelfThatNotSupported(int[,] TableS, int Order, Color a, int ij, int ji, int ii, int jj)

{

Object O = new Object();

lock (O)

{

//Initiate Variables.

int[,] Tab = new int[8, 8];

Object O1 = new Object();

lock (O1)

{

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

int Ord = Order;

bool SelfSupported = false;

bool InAttackedNotSelfSupported = false;

bool IsObjDangerest = false;

bool S = true;

int i = ii, j = jj;

//Ignore of Current

//For Enemy.

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

//Ignore of Current

if (Order == 1 && Tab[iii, jjj] >= 0)

continue;

else

if (Order == -1 && Tab[iii, jjj] <= 0)

continue;

//Enemy

a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

InAttackedNotSelfSupported = false;

SelfSupported = false;

Object OO = new Object();

lock (OO)

{

if (Attack(Tab, iii, jjj, i, j, a, Order \* -1))

{

InAttackedNotSelfSupported = true;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//For Self.

for (int iiii = 0; iiii < 8; iiii++)

{

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of Enemies

if (Order == 1 && Tab[iiii, jjjj] <= 0)

continue;

else

if (Order == -1 && Tab[iiii, jjjj] >= 0)

continue;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

//When there is support and cuurent is less than enemy.

//method return true when is not supporte and the enemy is less than cuurent in to be hitten.

if (Support(Tab, iiii, jjjj, i, j, a, Order))

{

SelfSupported = true;

S = S && true;

break;

}

}

if (SelfSupported)

break;

}

//When a source enemy object attack a destination source object

//a source object is greater than another source object. Is = -1 Is another object valuable.

//a source object is less than or equal than another source object.Is = 1 Is not another object valuable.

//IsObjDangerest = IsAnotherObjectMakeDangoure(TableS, Order, color, i, j, iii, jjj);

}

}

if ((!SelfSupported && InAttackedNotSelfSupported) //|| IsObjDangerest

)

{

S = false;

break;

}

}

if ((!SelfSupported && InAttackedNotSelfSupported) || IsObjDangerest

)

{

S = false;

break;

}

}

if (!SelfSupported

&& InAttackedNotSelfSupported

)

{

S = false;

}

if (!SelfSupported && InAttackedNotSelfSupported)

{

S = false;

}

Order = Ord;

//When S is valid the any is not in [SelfNotSupported];Self is Supporeted.

if (S)

return false;

return true;

}

}

}

//When there is at least on self object that is not safty.

bool InAttackSelfThatNotSupportedAll(int[,] TableS, int Order, Color a, int i, int j, int iii, int jjj, int ikk, int jkk, int iik, int jjk)

{

Object O = new Object();

lock (O)

{

bool S = true;

int Ord = Order;

List<int[]> ValuableSelfSupported = new List<int[]>();

bool IsTowValuableObject = false;

Object O1 = new Object();

lock (O1)

{

IsTowValuableObject = InAttackSelfThatNotSupportedCalculateValuableAll(TableS, Order, color, ikk, jkk, iik, jjk, ref ValuableSelfSupported);

//Initiate Variables.

int[,] Tab = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

bool SelfSupported = false;

bool InAttackedNotSelfSupported = false;

S = true;

Order = Ord;

//Ignore of Enemies

if (Order == 1 && Tab[i, j] <= 0)

return false;

else

if (Order == -1 && Tab[i, j] >= 0)

return false;

//when there is another object valuable in List continue.

if (IsTowValuableObject && (!IsObjectValaubleObjectSelf(i, j, Tab[i, j], ref ValuableSelfSupported)))

return false;

Order = Ord;

//Ignore of Current

if (Order == 1 && Tab[iii, jjj] >= 0)

return false;

else

if (Order == -1 && Tab[iii, jjj] <= 0)

return false;

if (i == iii && j == jjj)

return false;

//Enemy

a = Color.Gray;

Order = Ord;

if (Order \* -1 == -1)

a = Color.Brown;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

InAttackedNotSelfSupported = false;

SelfSupported = false;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

Object O2 = new Object();

lock (O2)

{

if (Attack(Tab, iii, jjj, i, j, a, Order \* -1))

{

InAttackedNotSelfSupported = true;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//For Self.

for (int iiii = 0; iiii < 8; iiii++)

{

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of Enemies

if (Order == 1 && Tab[iiii, jjjj] <= 0)

continue;

else

if (Order == -1 && Tab[iiii, jjjj] >= 0)

continue;

if (i == iiii && j == jjjj)

continue;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

//When there is supporte and cuurent is less than enemy.

//method return true when is not supporte and the enemy is less than cuurent in to be hitten.

if (Support(Tab, iiii, jjjj, i, j, a, Order) && (GetObjectValue(Tab, i, j, Order) <= GetObjectValue(Tab, iii, jjj, Order \* -1)))

{

SelfSupported = true;

S = S && true;

break;

}

}

//When a source enemy object attack a destination source object

//a source object is greater than another source object. Is = -1 Is another object valuable.

//a source object is less than or equal than another source object.Is = 1 Is not another object valuable.

if (SelfSupported)

break;

}

}

}

if ((!SelfSupported && InAttackedNotSelfSupported))

{

S = false;

}

}

Order = Ord;

//When S is valid the any is not in [SelfNotSupported];Self is Supporeted.

if (S)

return false;

return true;

}

}

//Creation A Complete List of Attacked Self Object(s).

bool InAttackSelfThatNotSupportedCalculateValuableAll(int[,] TableS, int Order, Color a, int ij, int ji, int ii, int jj, ref List<int[]> ValuableSelfSupported)

{

Object O = new Object();

lock (O)

{

//Initiate Variables.

int[,] Tab = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

int Ord = Order;

bool SelfSupported = false;

bool InAttackedNotSelfSupported = false;

bool S = true;

//int i = ii, j = jj;

//For Self

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

S = true;

//Ignore of Enemy

if (Order == 1 && Tab[i, j] <= 0)

continue;

else

if (Order == -1 && Tab[i, j] >= 0)

continue;

//For Enemy.

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

//Ignore of Current

if (Order == 1 && Tab[iii, jjj] >= 0)

continue;

else

if (Order == -1 && Tab[iii, jjj] <= 0)

continue;

//Enemy

a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

InAttackedNotSelfSupported = false;

SelfSupported = false;

S = true;

//Wehn an Object of Enemy Attack Self Object

Object O1 = new Object();

lock (O1)

{

if (Attack(Tab, iii, jjj, i, j, a, Order \* -1))

{

InAttackedNotSelfSupported = true;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//For Self.

for (int iiii = 0; iiii < 8; iiii++)

{

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of Enemies

if (Order == 1 && Tab[iiii, jjjj] <= 0)

continue;

else

if (Order == -1 && Tab[iiii, jjjj] >= 0)

continue;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = TableS[ik, jk];

//When There is Supporter For Attacked Self Object and Is Greater than Attacking Object.

if (Support(Tab, iiii, jjjj, i, j, a, Order) && (GetObjectValue(Tab, i, j, Order) <= GetObjectValue(Tab, iii, jjj, Order \* -1)))

{

SelfSupported = true;

S = S && true;

break;

}

}

if (SelfSupported)

break;

}

//When a source enemy object attack a destination source object

//a source object is greater than another source object. Is = -1 Is another object valuable.

//a source object is less than or equal than another source object.Is = 1 Is not another object valuable.

}

}

//When Attacked Current Object is not supported and there is another object valuable

Object O2 = new Object();

lock (O2)

{

if ((!SelfSupported && InAttackedNotSelfSupported))

{

S = false;

if (!S)

{

int[] Valuable = new int[3];

//First is Value;Second and Third is Row and Column.

Valuable[0] = TableS[i, j];

Valuable[1] = i;

Valuable[2] = j;

if (!ExistValuble(Valuable, ref ValuableSelfSupported))

ValuableSelfSupported.Add(Valuable);

S = true;

}

}

}

}

}

}

}

Order = Ord;

//When There is at last tow SelfNotSupporeted Object.

if (ValuableSelfSupported.Count > 1)

return true;

return false;

}

}

bool ExistValuble(int[] Table, ref List<int[]> ValuableSelfSupported)

{

Object O = new Object();

lock (O)

{

bool Is = false;

for (int i = 0; i < ValuableSelfSupported.Count; i++)

{

if (ValuableSelfSupported[i][0] == Table[0] && ValuableSelfSupported[i][1] == Table[1] && ValuableSelfSupported[i][2] == Table[2])

return true;

}

return Is;

}

}

bool MaxObjecvts(List<int> Obj, int Max)

{

Object O = new Object();

lock (O)

{

bool MaxO = true;

if (Obj.Count > 0)

{

if (Max == 0)

return !MaxO;

if (Max > 0)

if (Obj[0] < 0)

return !MaxO;

if (Max < 0)

if (Obj[0] > 0)

return !MaxO;

for (int i = 0; i < Obj.Count; i++)

{

if (System.Math.Abs(Obj[i]) > System.Math.Abs(Max))

{

MaxO = true;

return MaxO;

}

else

MaxO = false;

}

}

return MaxO;

}

}

//When Current Movment Take Supporte.QC\_OK

bool IsCurrentMoveTakeSupporte(int[,] Table, int Order, Color a, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

//Initiate Variables.

int[,] Tab = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tab[ik, jk] = Table[ik, jk];

bool SelfSupported = false;

int Dum = ChessRules.CurrentOrder;

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

//Ignore of Enemy Objects.

if (Tab[iii, jjj] <= 0 && Order == 1)

continue;

if (Tab[iii, jjj] >= 0 && Order == -1)

continue;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//When there is Attacks.

if (Support(Tab, iii, jjj, ii, jj, a, Order))

SelfSupported = true;

}

}

return SelfSupported;

}

}

///Huristic of King safty.

double HeuristicKingSafety(int[,] Tab, int Order, Color a, int i, int j, int iii, int jjj, int CurrentAStarGredy)

{

Object O = new Object();

lock (O)

{

double HeuristicKingSafe = 0;

double HA = 0;

//For Enemies.

Parallel.For(0, 8, ii =>

{

Parallel.For(0, 8, jj =>

{

if (Order == 1 && Tab[ii, jj] >= 0)

return;

if (Order == -1 && Tab[ii, jj] <= 0)

return;

ChessRules A = new ChessRules(CurrentAStarGredy, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab[ii, jj], Tab, Order \* -1, ii, jj);

//For Current and Empty

Parallel.For(0, 8, iiii =>

{

Parallel.For(0, 8, jjjj =>

{

//Ignore of Enemy.

if (Order == 1 && Tab[iiii, jjjj] < 0)

return;

if (Order == -1 && Tab[iiii, jjjj] > 0)

return;

int[,] Table = new int[8, 8];

//Clone a Copy.

Parallel.For(0, 8, ij =>

{

Parallel.For(0, 8, ji =>

{

Table[ij, ji] = Tab[ij, ji];

});

});

Color AA = Color.Gray;

if (Order \* -1 == -1)

AA = Color.Brown;

//When Enemy can Move

Object O1 = new Object();

lock (O1)

{

if (A.Rules(ii, jj, iiii, jjjj, AA, Table[ii, jj]))

{

//Take Movment.

Table[iiii, jjjj] = Table[ii, jj];

Table[ii, jj] = 0;

//Is Dangrous for King.

A = new ChessRules(CurrentAStarGredy, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab[iiii, jjjj], Table, Order, iiii, jjjj);

if (A.ObjectDangourKingMove(Order, Table, false))

{

//Clone a Copy.

Parallel.For(0, 8, ij =>

{

Parallel.For(0, 8, ji =>

{

Table[ij, ji] = Tab[ij, ji];

});

});

//When Before Move such situation is observed calculate huristic count.

/\*if (Order == 1 && A.CheckGrayObjectDangour)

HA += AllDraw.SignKingSafe \* (GetObjectValue(Table, ii, jj, Order \* -1) + GetObjectValue(Table, iiii, jjjj, Order));

else

if (Order == -1 && A.CheckBrownObjectDangour)

HA += AllDraw.SignKingSafe \* (GetObjectValue(Table, ii, jj, Order \* -1) + GetObjectValue(Table, iiii, jjjj, Order));

\*/

if (Order == 1 && A.CheckMateGray)

HA += AllDraw.SignKingSafe \* (GetObjectValue(Table, ii, jj, Order \* -1) + GetObjectValue(Table, iiii, jjjj, Order));

else

if (Order == -1 && A.CheckMateBrown)

HA += AllDraw.SignKingSafe \* (GetObjectValue(Table, ii, jj, Order \* -1) + GetObjectValue(Table, iiii, jjjj, Order));

}

}

}

});

});

});

});

//For Enemy and Self Sign.

HeuristicKingSafe += HA;

return HeuristicKingSafe;

}

}

double HeuristicKingDangourous(int[,] Tab, int Order, Color a, int i, int j, int iii, int jjj, int CurrentAStarGredy)

{

Object O = new Object();

lock (O)

{

double HeuristicKingDangour = 0;

double HA = 0;

//For Self.

Parallel.For(0, 8, ii =>

{

Parallel.For(0, 8, jj =>

{

//Ignore of Enemy and Empty.

if (Order == 1 && Tab[ii, jj] <= 0)

return;

if (Order == -1 && Tab[ii, jj] >= 0)

return;

ChessRules A = new ChessRules(CurrentAStarGredy, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab[ii, jj], Tab, Order, ii, jj);

//For Enemy and Empty.

Parallel.For(0, 8, iiii =>

{

Parallel.For(0, 8, jjjj =>

{

//Ignore of Self.

if (Order == 1 && Tab[iiii, jjjj] > 0)

return;

if (Order == -1 && Tab[iiii, jjjj] < 0)

return;

int[,] Table = new int[8, 8];

//Clone a Copy.

Parallel.For(0, 8, ij =>

{

Parallel.For(0, 8, ji =>

{

Object O2 = new Object();

lock (O2)

{

Table[ij, ji] = Tab[ij, ji];

}

});

});

Color AA = Color.Gray;

if (Order == -1)

AA = Color.Brown;

//When Self Move

if (A.Rules(ii, jj, iiii, jjjj, AA, Table[ii, jj]))

{

//Take Mo0vment

Object O2 = new Object();

lock (O2)

{

Table[iiii, jjjj] = Table[ii, jj];

Table[ii, jj] = 0;

}

//The Move is Dqangrous.

Object O3 = new Object();

lock (O3)

{

if (A.ObjectDangourKingMove(Order, Table, false))

{

int[,] Table1 = new int[8, 8];

//Clone a Copy.

Parallel.For(0, 8, ij =>

{

Parallel.For(0, 8, ji =>

{

Object O1 = new Object();

lock (O1)

{

Table1[ij, ji] = Tab[ij, ji];

}

});

});

//When Situation Observed Take Situation calcualte Huristic.

Object O4 = new Object();

lock (O4)

{

/\*if (Order == -1 && A.CheckGrayObjectDangour)

HA += AllDraw.SignKingDangour \* (GetObjectValue(Table1, ii, jj, Order \* -1) + GetObjectValue(Table1, iiii, jjjj, Order));

else

if (Order == 1 && A.CheckBrownObjectDangour)

HA += AllDraw.SignKingDangour \* (GetObjectValue(Table1, ii, jj, Order \* -1) + GetObjectValue(Table1, iiii, jjjj, Order));

\*/

if (Order == -1 && A.CheckMateGray)

HA += AllDraw.SignKingDangour \* (GetObjectValue(Table1, ii, jj, Order \* -1) + GetObjectValue(Table1, iiii, jjjj, Order));

else

if (Order == 1 && A.CheckMateBrown)

HA += AllDraw.SignKingDangour \* (GetObjectValue(Table1, ii, jj, Order \* -1) + GetObjectValue(Table1, iiii, jjjj, Order));

}

}

}

}

});

});

});

});

//For Order Sign.

HeuristicKingDangour += HA \* SignOrderToPlate(Order);

return HeuristicKingDangour;

}

}

//Huristic of Supportation.

double HuristicSelfSupported(int[,] Tab, int Ord, Color aa, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

double HuristicSelfSupportedValue = 0;

//Initiate Local Vrariables.

double HA = 0;

int DumOrder = Order;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

//If There is Not AStarGreedy Huristic Boolean Value.

try

{

if (!AStarGreedyHuristicT)

{

//For Current Object Lcation.

int Order = new int();

Order = DumOrder;

Color a = new Color();

a = aa;

//Ignore Current Unnessery Home.

if (i == ii && j == jj)

return HuristicSelfSupportedValue;

//Default Is Gray One.

double Sign = 1;

Order = DummyOrder;

///When Supporte is true. means [ii,jj] Supportes [i,j].

///What is Supporte!

///Ans:When [i,j] is Supporte [ii,jj] return true when Self is located in [ii,jj].

if (Tab[ii, jj] < 0 && DummyOrder == -1 && Tab[i, j] <= 0)

{

Order = -1;

Object O1 = new Object();

lock (O1)

{

Sign = 1 \* AllDraw.SignSupport;

ChessRules.CurrentOrder = -1;

}

a = Color.Brown;

}

else if (Tab[ii, jj] > 0 && DummyOrder == 1 && Tab[i, j] >= 0)

{

Order = 1;

Object O1 = new Object();

lock (O1)

{

Sign = 1 \* AllDraw.SignSupport;

ChessRules.CurrentOrder = 1;

}

a = Color.Gray;

}

else

return HuristicSelfSupportedValue;

//For Support Movments.

if (Support(Tab, i, j, ii, jj, a, Order))

{

//Calculate Local Support Huristic.

HA += (Sign \* (System.Math.Abs((GetValue(ii, jj) + GetValue(i, j)

))));

bool Supported = new bool();

Supported = false;

//For All Self Obejcts.

Parallel.For(0, 8, g =>

{

if (Supported)

return;

Parallel.For(0, 8, h =>

{

Object O2 = new Object();

lock (O2)

{

if (Supported)

return;

//Ignore Of Enemy Objects.

if (Order == 1 && Tab[g, h] <= 0)

return;

if (Order == -1 && Tab[g, h] >= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

aaa = Color.Gray;

if (Order == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

//When Enemy is Supported.

bool A = new bool();

A = Support(Tab, g, h, ii, jj, aaa, Order);

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Supported = true;

return;

}

}

});

if (Supported)

return;

});

Object O1 = new Object();

lock (O1)

{

if (Supported)

//When is Not Supported multyply 100.

HA \*= 20;

else

//When is Supported Multyply -100.

HA \*= -20;

}

}

}

//For All Homes Table.

else

{

int Order = new int();

Color a = new Color();

a = aa;

{

//Ignore Current Home.

if (i == ii && j == jj)

return HuristicSelfSupportedValue;

//Initiate Local Variables.

double Sign = 1;

Order = DummyOrder;

///When Supporte is true. means [ii,jj] is in SelfSupported.by [i,j].

///What is Supporte!

///Ans:When [i,j] is Supporte [ii,jj] return true when Self is located in [ii,jj].

if (Tab[ii, jj] < 0 && DummyOrder == -1 && Tab[i, j] <= 0)

{

Order = -1;

Object O2 = new Object();

lock (O2)

{

Sign = 1 \* AllDraw.SignSupport;

ChessRules.CurrentOrder = -1;

a = Color.Brown;

}

}

else if (Tab[ii, jj] > 0 && DummyOrder == 1 && Tab[i, j] >= 0)

{

Order = 1;

Object O2 = new Object();

lock (O2)

{

Sign = 1 \* AllDraw.SignSupport;

ChessRules.CurrentOrder = 1;

a = Color.Gray;

}

}

else

return HuristicSelfSupportedValue;

//For Support Movments.

if (Support(Tab, i, j, ii, jj, a, Order))

{

//Calculate Local Support Huristic.

HA += (Sign \* (System.Math.Abs((GetValue(ii, jj) + GetValue(i, j)

))));

bool Supported = new bool();

Supported = false;

//For All Self Obejcts.

Parallel.For(0, 8, g =>

{

if (Supported)

return;

Parallel.For(0, 8, h =>

{

if (Supported)

return;

//Ignore Of Enemy Objects.

if (Order == 1 && Tab[g, h] <= 0)

return;

if (Order == -1 && Tab[g, h] >= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

Object O2 = new Object();

lock (O2)

{

aaa = Color.Gray;

if (Order == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

}

//When Enemy is Supported.

bool A = new bool();

Object O1 = new Object();

lock (O1)

{

A = Support(Tab, g, h, ii, jj, aaa, Order);

}

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Supported = true;

return;

}

});

if (Supported)

return;

});

if (!Supported)

//When is Not Supported multyply 100.

HA \*= 20;

else

//When is Supported Multyply -100.

HA \*= -20;

}

}

}

}

catch (Exception t)

{

Log(t);

}

//Reassignments of Global Orders with Local Begining One.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

Order = DumOrder;

return HA;

}

}

///Identification of Equality

public static bool TableEqual(int[,] Tab1, int[,] Tab2)

{

Object O = new Object();

lock (O)

{

try

{

//For All Home

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

//When there is different values in same location of tow Table return non equality.

if (Tab1[i, j] != Tab2[i, j])

return false;

}

//Else return equlity.

return true;

}

catch (Exception t)

{

Log(t);

return false;

}

}

}

//If tow int Objects is equal.

public static bool TableEqual(int Tab1, int Tab2)

{

Object O = new Object();

lock (O)

{

try

{

//When there is different values in same location of tow Table return non equality.

if (Tab1 != Tab2)

return false;

//Else return equlity.

return true;

}

catch (Exception t)

{

Log(t);

return false;

}

}

}

//Deterimination of Existance of Table in List..

static public bool ExistTableInList(int[,] Tab, List<int[,]> List, int Index)

{

Object O = new Object();

lock (O)

{

//Initiate Local Variables.

bool Exist = false;

//For All Tables of Table List.

for (int i = Index; i < List.Count; i++)

{

//Strore Equality Value.

bool Eq = TableEqual(Tab, List[i]);

//When is Equality is Occurred.

if (Eq)

{

//Store Equality Local Value in a Global static value.

AllDraw.LoopHuristicIndex = i;

return Eq;

}

Exist |= Eq;

}

//return Equality Local value of all lists.

return Exist;

}

}

///Move Determination.

public bool Movable(int[,] Tab, int i, int j, int ii, int jj, Color a, int Order)

{

Object O = new Object();

lock (O)

{

int[,] Table = new int[8, 8];

for (int p = 0; p < 8; p++)

for (int k = 0; k < 8; k++)

Table[p, k] = Tab[p, k];

//Initiate Local Variables.

int Store = Table[ii, jj];

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[i, j], Table, Order, i, j);

///Table[ii, jj] = 0;

//Menen Parameter is Moveble to Second Parameters Location returm Movable.

if (A.Rules(i, j, ii, jj, a, Order))

{

//Initiate Movments.

Table[ii, jj] = Table[i, j];

Table[i, j] = 0;

//Default Order Assignments.

int Ord = 1;

//Brown Order Consideration.

if (Table[ii, jj] < 0)

Ord = -1;

//Store of Local Order Assignments in Global Assignments.

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

//Consider Global Check Variables.

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[ii, jj], Table, Ord, ii, jj);

AA.Check(Table, Ord);

//Reaasignment of Premitive Variables.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//Reassignments of Table Content and Consider CheckMate Specific Order.

Table[i, j] = Table[ii, jj];

//When Gray.

if (Table[i, j] > 0)

{

//And CheckedMated is Occured for gray. return false.

Table[ii, jj] = Store;

if (AA.CheckMateGray)

return false;

return true;

}

//When Brown.

if (Table[i, j] < 0)

{

Table[ii, jj] = Store;

//When CheckedMated occured for Brown return false.

if (AA.CheckMateBrown)

return false;

return true;

}

}

Table[ii, jj] = Store;

return false;

}

}

//

//When Oredrs of OrderPalte and Calculation Order is not equal return negative one and else return one.

double SignOrderToPlate(int Order)

{

Object O = new Object();

lock (O)

{

double Sign = 1.0;

//When Current Order Sign Positive.

if (Order == AllDraw.OrderPlate)

Sign = 1.0;

else

//When Order is Opposite Sign Negative.

if (Order != AllDraw.OrderPlate)

Sign = -1.0;

return Sign;

}

}

//Remove Penalties of Unnesserily Nodes.

public bool RemovePenalty(int[,] Tab, int Order, int i, int j)

{

Object O = new Object();

lock (O)

{

bool Remove = false;

//Create Objects.

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab[i, j], Tab, Order, i, j);

//When is Check.

if (AA.Check(Tab, Order))

{

//When there is Current Checked or Objects Danger return false.

if (Order == 1 && (AA.CheckGray || AA.CheckGrayObjectDangour))

return Remove;

if (Order == -1 && (AA.CheckBrown || AA.CheckBrownObjectDangour))

return Remove;

}

//For Enemy.

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

if (Order == 1 && Tab[ii, jj] >= 0)

continue;

if (Order == -1 && Tab[ii, jj] <= 0)

continue;

//Clone a Copy.

int[,] Table = new int[8, 8];

//Clone a Table.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

Table[iii, jjj] = Tab[iii, jjj];

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[ii, jj], Table, Order \* -1, ii, jj);

Color a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

//When there is movment to current OPbject.

if (A.Rules(ii, jj, i, j, a, Table[ii, jj]))

{

//Number of Attacks and take move.

int Count = AttackerCount(Table, Order \* -1, a, ii, jj);

//When there is Object Danger.

//Clone a Copy.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

Table[iii, jjj] = Tab[iii, jjj];

//Create New Chess Rule Object.

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[ii, jj], Table, Order, ii, jj);

//Detect int.

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//When Current Movments Attacks Enemy.

if (Attack(Table, i, j, ii, jj, a, Order))

{

//For Current Home.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

//Ignore of Enemy.

if (Order == 1 && Tab[iii, jjj] <= 0)

continue;

if (Order == -1 && Tab[iii, jjj] >= 0)

continue;

//Whn Value Of Current is Less That Enemy.

if (GetObjectValue(Table, i, j, Order) < GetObjectValue(Table, ii, jj, Order))

{

//Take Move.

Table[ii, jj] = Table[i, j];

Table[i, j] = 0;

a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

//When Enemy Attacks Current Moved.

if (!Attack(Table, iii, jjj, ii, jj, a, Order \* -1))

{

//For Current Order.

for (int iiii = 0; iiii < 8; iiii++)

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of Enemy.

if (Order == 1 && Tab[iiii, jjjj] <= 0)

continue;

if (Order == -1 && Tab[iiii, jjjj] >= 0)

continue;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

//When Self Supported Current

if (Support(Table, iiii, jjjj, i, j, a, Order))

{

//If V alue of Enemy is Greater Than Current and Value of Enemy is Greater than Supporter.

if (GetObjectValue(Table, iii, jjj, Order) < GetObjectValue(Table, ii, jj, Order) && GetObjectValue(Table, iii, jjj, Order) > GetObjectValue(Table, iiii, jjjj, Order))

{

Remove = true;

return Remove;

}

else

return Remove;

}

else

return Remove;

}

}

else

return

Remove;

}

else

return Remove;

}

}

}

}

return Remove;

}

}

//Dangouring of current movment fo current Order.

bool IsCurrentStateIsDangreousForCurrentOrder(int[,] Tabl, int Order, Color a, int ii, int jj)

{

Object O = new Object();

lock (O)

{

//Initiate Object.

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, 1, Tabl, 1, Row, Column);

//Gray Order.

if (Order == 1)

{

//Find location of Gray King.

int RowG = -1, ColumnG = -1;

A.FindGrayKing(Tabl, ref RowG, ref ColumnG);

//When found.

if (RowG != -1 && ColumnG != -1)

{

//For Brown

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

//Ignore of Gray and Empty

if (Tabl[i, j] >= 0)

continue;

if (i != ii && j != jj)

{

//Create new Objects of Table

int[,] TablCon = new int[8, 8];

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

TablCon[iii, jjj] = Tabl[iii, jjj];

//For Enemy Order.

if (TablCon[i, j] < 0)

{

//For Gray and Empty Objects.

if (TablCon[ii, jj] >= 0)

{

//Setting Enemy Order.

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TablCon[i, j], TablCon, -1, i, j);

//When Enemy is Attacked Gray Objects.

if (A.Rules(i, j, ii, jj, Color.Brown, TablCon[i, j]))

{

//Take Movments.

TablCon[ii, jj] = TablCon[i, j];

TablCon[i, j] = 0;

//Settting Current Order.

ChessRules.CurrentOrder = 1;

//Settting Object.

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TablCon[ii, jj], TablCon, 1, ii, jj);

//When Occured Check.

if (A.Check(TablCon, 1))

{

//When Gray is Check.

if (A.CheckGray)

{

//For Enemy Order Objects.

for (int iiii = 0; iiii < 8; iiii++)

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//When is not Conflict.

if (iiii != i && jjjj != j && iiii != ii && jjjj != jj)

{

//Setting Enemy.

ChessRules.CurrentOrder = -1;

//When Enemy is Supported

if (Support(TablCon, iiii, jjjj, i, j, Color.Brown, -1))

{

//restore and return true.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

return true;

}

}

}

}

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

}

}

}

}

}

}

}

}

//For Brown Order.

else if (Order == -1)

{

//Found of Brown King.

int RowB = -1, ColumnB = -1;

A.FindBrownKing(Tabl, ref RowB, ref ColumnB);

//When found.

if (RowB != -1 && ColumnB != -1)

{

//For Gray.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

if (Tabl[i, j] <= 0)

continue;

if (i != ii && j != jj)

{

//Create new Objects of Table

int[,] TablCon = new int[8, 8];

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

TablCon[iii, jjj] = Tabl[iii, jjj];

//For Enemy Objects.

if (TablCon[i, j] > 0)

{

//For Self Objects and Empty.

if (TablCon[ii, jj] <= 0)

{

//Store and Enemy Order.

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TablCon[i, j], TablCon, 1, i, j);

ChessRules.CurrentOrder = 1;

//When Enemy Attacked Self Objects.

if (A.Rules(i, j, ii, jj, Color.Gray, TablCon[i, j]))

{

//Take movemnts.

TablCon[ii, jj] = TablCon[i, j];

TablCon[i, j] = 0;

//Setting current Order.

ChessRules.CurrentOrder = -1;

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TablCon[ii, jj], TablCon, -1, ii, jj);

//When Check Occured.

if (A.Check(TablCon, -1))

{

//When Current is Check.

if (A.CheckBrown)

{

//For Enemy Objecvts.

for (int iiii = 0; iiii < 8; iiii++)

for (int jjjj = 0; jjjj < 8; jjjj++)

{

//Ignore of Conflit.

if (iiii != i && jjjj != j && iiii != ii && jjjj != jj)

{

//Setting Enemy Order

ChessRules.CurrentOrder = 1;

//When Enemy is Supported.

if (Support(TablCon, iiii, jjjj, i, j, Color.Gray, 1))

{

//restore and return true.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

return true;

}

}

}

}

//restore.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

}

}

}

}

}

}

}

}

//return false.

return false;

}

}

//When Next Movements is Checked.QC\_OK.

int[] IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(int Order, int[,] Tabl, int ik, int jk, int iki, int jki, int OrderPalte, int OrderPalteMulMinuse, int Depth, bool KindCheckedSelf)

{

Object O = new Object();

lock (O)

{

int[] Is = new int[4];

Object O3 = new Object();

lock (O3)

{

Is[0] = 0;

Is[1] = 0;

Is[2] = 0;

Is[3] = 0;

int[,] Tab2 = CloneATable(Tabl);

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab2[ik, jk], Tab2, Order \* -1, ik, jk);

if (Order \* -1 == 1)

color = Color.Gray;

else

color = Color.Brown;

//When Enemy Attack Currnet.

if (A.Rules(ik, jk, iki, jki, color, Tab2[ik, jk]))

{

Tab2[iki, jki] = Tab2[ik, jk];

Tab2[ik, jk] = 0;

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab2[iki, jki], Tab2, Order \* -1, iki, jki);

//When Current Always is in CheckedMate.

if (A.CheckMate(Tab2, Order \* -1))

{

//When Order is Gray.

if (OrderPalte == 1)

{

if (A.CheckMateGray)

{

Is[0] = 1;

if (KindCheckedSelf)

Is[1] = Depth;

}

else

{

//if (A.CheckMateBrown)

//return Is;

}

}

//When Order is Brown.

else

if (OrderPalte == -1)

{

if (A.CheckMateBrown)

{

Is[0] = 1;

Is[1] = Depth;

}

else

{

//if (A.CheckMateGray)

//return Is;

}

}

//When Order \* -1 is Gray

if (OrderPalteMulMinuse == 1)

{

if (A.CheckMateGray)

{

Is[2] = 1;

Is[3] = Depth;

}

else

{

//if (A.CheckMateBrown)

//return Is;

}

}

//When Order \* -1 is Brown

else

if (OrderPalteMulMinuse == -1)

{

if (A.CheckMateBrown)

{

Is[2] = 1;

Is[3] = Depth;

}

else

{

//if (A.CheckMateGray)

//return Is;

}

}

}

if (Order \* -1 == 1)

color = Color.Gray;

else

color = Color.Brown;

//if (Tab2[iki, jki] == 0)

//return Is;

//For Movements.

int Ord = Order \* -1;

int[,] Tab = CloneATable(Tab2);

Color a = color;

if (Ord == 1)

a = Color.Gray;

else

a = Color.Brown;

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMulMinuse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

Object O1 = new Object();

int[] IS = null;

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovment(Tab, Ord, a, Depth1, OrderP, OrderM, KindCheckedSelf1);

}

if (IS[0] == 1) Is[0] = 1;

if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1];

Is[3] = IS[3];

}

}

return Is;

}

}

//When Next Movements is Checked.QC\_OK.

bool IsNextMovmentIsCheckOrCheckMateForCurrentMovmentOnCurrentMovemnet(int Order, int[,] Tabl, int ik, int jk, int iki, int jki, int OrderPalte)

{

Object O = new Object();

lock (O)

{

bool Is = false;

int[,] Tab2 = new int[8, 8];

for (int ki = 0; ki < 8; ki++)

for (int kj = 0; kj < 8; kj++)

Tab2[ki, kj] = Tabl[ki, kj];

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab2[ik, jk], Tab2, Order - 1, ik, jk);

//When Enemy Attack Currnet.

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab2[iki, jki], Tab2, OrderPalte, iki, jki);

//When Current Always is in CheckedMate.

if (A.CheckMate(Tab2, OrderPalte))

{

//When for penalty.

if (OrderPalte == AllDraw.OrderPlate)

{

//When Order is Gray.

if (OrderPalte == 1)

{

if (A.CheckMateGray)

Is = true;

else

{

if (A.CheckMateBrown)

return Is;

}

}

//When Order is Brown.

else

if (OrderPalte == -1)

{

if (A.CheckMateBrown)

Is = true;

else

{

if (A.CheckMateGray)

return Is;

}

}

}

//When for regard.

else

{

//When Order \* -1 is Gray

if (OrderPalte == 1)

{

if (A.CheckMateGray)

Is = true;

else

{

if (A.CheckMateBrown)

return Is;

}

}

//When Order \* -1 is Brown

else

if (OrderPalte == -1)

{

if (A.CheckMateBrown)

Is = true;

else

{

if (A.CheckMateGray)

return Is;

}

}

}

}

return Is;

}

}

int[] IsNextMovmentIsCheckOrCheckMateForCurrentMovment(int[,] Tabl, int Order, Color a, int Depth, int OrderPalte, int OrderPalteMinusPluse, bool KindCheckedSelf)

{

Object O = new Object();

lock (O)

{

int[] Is = new int[4];

Object O3 = new Object();

lock (O3)

{

Is[0] = 0;

Is[1] = 0;

Is[2] = 0;

Is[3] = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

if (Depth >= AllDraw.MaxAStarGreedy)

return Is;

//For All Enemies.

Parallel.For(0, 8, ik =>

Parallel.For(0, 8, jk =>

{

//Ignore of Current

if (Order == 1 && Tabl[ik, jk] >= 0)

return;

if (Order == -1 && Tabl[ik, jk] <= 0)

return;

if (System.Math.Abs(Tabl[ik, jk]) == 1)

{

//For Current Home

Parallel.For(ik - 2, ik + 3, iki =>

Parallel.For(jk - 2, jk + 3, jki =>

// init subtotal

{

if (!Scop(ik, jk, iki, jki, 1))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

}

));

}

else

if (System.Math.Abs(Tabl[ik, jk]) == 2)

{

//For Current Home

Parallel.For(0, 8, iki =>

{

int jki = iki + jk - ik;

if (!Scop(ik, jk, iki, jki, 2))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

});

//For Current Home

Parallel.For(0, 8, iki =>

{

int jki = iki \* -1 + jk + ik;

if (!Scop(ik, jk, iki, jki, 2))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

});

}

else

if (System.Math.Abs(Tabl[ik, jk]) == 3)

{

//For Current Home

Parallel.For(ik - 2, ik + 3, iki =>

Parallel.For(jk - 2, jk + 3, jki =>

{

if (!Scop(ik, jk, iki, jki, 3))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

}));

}

else

if (System.Math.Abs(Tabl[ik, jk]) == 4)

{

//For Current Home

Parallel.For(0, 8, iki =>

{

int jki = jk;

if (!Scop(ik, jk, iki, jki, 4))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

});

//For Current Home

Parallel.For(0, 8, jki =>

{

int iki = ik;

if (!Scop(ik, jk, iki, jki, 4))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

});

}

else

if (System.Math.Abs(Tabl[ik, jk]) == 5)

{

//For Current Home

Parallel.For(0, 8, iki =>

Parallel.For(0, 8, jki =>

{

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (!Scop(ik, jk, iki, jki, 5))

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

}));

}

else

if (System.Math.Abs(Tabl[ik, jk]) == 6)

{

//For Current Home

Parallel.For(ik - 1, ik + 2, iki =>

Parallel.For(jk - 1, jk + 2, jki =>

{

if (!Scop(ik, jk, iki, jki, 6))

return;

//Ignore of Enemy

if (Order == 1 && Tabl[iki, jki] < 0)

return;

if (Order == -1 && Tabl[iki, jki] > 0)

return;

if (Is[0] == 1)

return;

int Ord = Order;

int[,] Tab = CloneATable(Tabl);

int ik1 = ik, jk1 = jk, iki1 = iki, jki1 = jki, OrderP = OrderPalte, OrderM = OrderPalteMinusPluse, Depth1 = Depth + 1;

bool KindCheckedSelf1 = KindCheckedSelf;

int[] IS = null;

Object O1 = new Object();

lock (O1)

{

IS = IsNextMovmentIsCheckOrCheckMateForCurrentMovmentBaseKernel(Ord, Tab, ik1, jk1, iki1, jki1, OrderP, OrderM, Depth1, KindCheckedSelf1);

if (IS[0] == 1) Is[0] = 1; if (IS[2] == 1) Is[2] = 1;

Is[1] = IS[1]; Is[3] = IS[3];

}

}));

}

}));

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

}

return Is;

}

}

//When Current Movements is in dangrous and is not movable.

bool IsGardForCurrentMovmentsAndIsNotMovable(int[,] Tab, int Order, Color a, int ii, int jj, int iii, int jjj)

{

Object O = new Object();

lock (O)

{

//Setting false.

bool Attacked = true;

int NumberOfCurrentEnemyAttackSuchObject = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

//For Enemy Order.

Object O1 = new Object();

lock (O1)

{

//Ignore of Self Objects.

if (Order == 1 && Tab[ii, jj] >= 0)

return false;

else

if (Order == -1 && Tab[ii, jj] <= 0)

return false;

//Restore

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

NumberOfCurrentEnemyAttackSuchObject = 0;

//For Self Objects and Empty.

//Ignore of Enemy Objects.

if (Order == 1 && Tab[iii, jjj] < 0)

return false;

else

if (Order == -1 && Tab[iii, jjj] > 0)

return false;

//For Enemy Order.

ChessRules.CurrentOrder = Order \* -1;

//Initiate for not exiting from abnormal loop.

Attacked = false;

Color aa = Color.Gray;

if (Order \* -1 == -1)

aa = Color.Brown;

//When Enemy Attacked Current Movements.

if (Attack(Tab, ii, jj, iii, jjj, aa, Order \* -1) && (GetObjectValue(Tab, ii, jj, Order \* -1) < GetObjectValue(Tab, iii, jjj, Order)))

{

NumberOfCurrentEnemyAttackSuchObject++;

//Clone a Table.

int[,] TabS = new int[8, 8];

for (int p = 0; p < 8; p++)

for (int m = 0; m < 8; m++)

TabS[p, m] = Tab[p, m];

TabS[iii, jjj] = TabS[ii, jj];

TabS[ii, jj] = 0;

//For Self Objects.

Parallel.For(0, 8, iiii =>

{

if (!Attacked || NumberOfCurrentEnemyAttackSuchObject > 1)

return;

Parallel.For(0, 8, jjjj =>

{

if (!Attacked || NumberOfCurrentEnemyAttackSuchObject > 1)

//return;//Ignore Enmy Objects.

if (Order == 1 && Tab[iiii, jjjj] <= 0)

return;

else

if (Order == -1 && Tab[iiii, jjjj] >= 0)

return;

//Show the Attacked.

Attacked = true;

//For Self Objects and Empty.

Parallel.For(0, 8, iiiii =>

{

Parallel.For(0, 8, jjjjj =>

{

//Ignore of Enemy Objects.

if (Order == 1 && Tab[iiiii, jjjjj] < 0)

return;

else

if (Order == -1 && Tab[iiiii, jjjjj] > 0)

return;

//When Current Objects Movable not need to consideration mor going to next Current object.

Object O2 = new Object();

lock (O2)

{

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TabS[iiii, jjjj], TabS, Order, iiii, jjjj)).Rules(iiii, jjjj, iiiii, jjjjj, a, TabS[iiii, jjjj]))

{

Attacked = Attacked && false;

return;

}

}

});

if (!Attacked || NumberOfCurrentEnemyAttackSuchObject > 1)

return;

});

if (Attacked || NumberOfCurrentEnemyAttackSuchObject > 1)

return;

});

if (Attacked || NumberOfCurrentEnemyAttackSuchObject > 1)

return;

});

}

else

return false;

}

//Restore.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//Return Variable when true show an object is not movable or one enemy object attack more than one current Object.

return Attacked || NumberOfCurrentEnemyAttackSuchObject > 1;

}

}

///when current movments gards enemy with higer priority at movment.QC\_OK

bool IsCurrentCanGardHighPriorityEnemy(int Depth, int[,] Table, int Order, Color a, int ij, int ji, int iij, int jji, int OrderPlate)

{

Object O = new Object();

lock (O)

{

if (Depth >= CurrentAStarGredyMax)

return false;

Object O4 = new Object();

lock (O4)

{

Depth++;

IsGardHighPriority = false;

int[,] Tabl1 = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tabl1[ik, jk] = Table[ik, jk];

//For Current.

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

//Ignore of Enemy.QC\_OK.

if (Order == 1 && Tabl1[i, j] <= 0)

continue;

else

if (Order == -1 && Tabl1[i, j] >= 0)

continue;

//For Enemy.

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

//Ignore of Current.QC\_OK.

if (Order == 1 && Tabl1[ii, jj] >= 0)

continue;

else

if (Order == -1 && Tabl1[ii, jj] >= 0)

continue;

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Tabl1[ik, jk] = Table[ik, jk];

//Take Movement.

if (Attack(Tabl1, i, j, ii, jj, a, Order \* -1))

{

//When Current Movments is

if (GetObjectValue(Tabl1, i, j, Order \* -1) <= GetObjectValue(Tabl1, ii, jj, Order \* -1))

{

if (Order == OrderPlate)

IsGardHighPriority = true;

}

else

{

Tabl1[ii, jj] = Tabl1[i, j];

Tabl1[i, j] = 0;

if (Order \* -1 == 1)

a = Color.Gray;

else

a = Color.Brown;

IsGardHighPriority = IsGardHighPriority || IsCurrentCanGardHighPriorityEnemy(Depth, Table, Order \* -1, a, ii, jj, i, j, OrderPlate);

}

}

}

}

}

return IsGardHighPriority;

}

}

///Huristic of Check and CheckMate.

public double HuristicCheckAndCheckMate(int[,] Table, Color a)

{

Object O = new Object();

lock (O)

{

double HA = 0;

int DummyOrder = AllDraw.OrderPlate;

int DummyCurrentOrder = ChessRules.CurrentOrder;

//double ObjectDangour = 1;

//double Check = 1000;

double ObjectDangour = 0;

double Check = 0;

double CheckMate = 100000;

//When is self objects order divide valuse by 100

//Becuse reduce from danger is most favareable of caused to enemy attack

if (Order == AllDraw.OrderPlate)

{

//ObjectDangour = 0.01;

//Check = 10;

CheckMate = 1000;

}

try

{

Object O1 = new Object();

lock (O1)

{

//Consider Global Check CheckMate ObjectDanger Variables Orderly.

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[Row, Column], Table, Order, Row, Column);

A.CheckMate(Table, Order);

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[Row, Column], Table, Order, Row, Column);

AA.ObjectDangourKingMove(Order, Table, false);

{

//Consider Value to More Valuable Positive and Negative Check CheckMate ObjectDanger

if (A.CheckMateGray || A.CheckMateBrown)

{//When is Brown CheckedMate.

if (DummyOrder == 1 && A.CheckMateBrown)

{

HA += CheckMate;

MovementsAStarGreedyHuristicFoundT = true;

}

//When is Gray CheckedMate.

if (DummyOrder == -1 && A.CheckMateGray)

{

HA += CheckMate;

MovementsAStarGreedyHuristicFoundT = true;

}

}

//When is Checked.

if (A.CheckGray || A.CheckBrown)

{

//When is Gray Checked

if (DummyOrder == 1 && A.CheckBrown)

{

HA += Check;

MovementsAStarGreedyHuristicFoundT = true;

}

//When is Brown Check.

if (DummyOrder == -1 && A.CheckGray)

{

HA += Check;

MovementsAStarGreedyHuristicFoundT = true;

}

}

//When is Objects Dangoure.

if (AA.CheckGrayObjectDangour || AA.CheckBrownObjectDangour)

{

//when is Gray Objects Dangoure.

if (DummyOrder == 1 && AA.CheckBrownObjectDangour)

{

HA += ObjectDangour;

MovementsAStarGreedyHuristicFoundT = true;

}

//when is Gray Objects Dangoure.

if (DummyOrder == -1 && AA.CheckGrayObjectDangour)

{

HA += ObjectDangour;

MovementsAStarGreedyHuristicFoundT = true;

}

}

//When is CheckMate

if (A.CheckMateGray || A.CheckMateBrown)

{

//When is Gray Check Mate.

if (DummyOrder == 1 && A.CheckMateGray)

{

HA -= CheckMate;

}

//when is Brown CheckMate.

if (DummyOrder == -1 && A.CheckMateBrown)

{

HA -= CheckMate;

}

}

//when is Check

if (A.CheckGray || A.CheckBrown)

{

//when is Gray Check

if (DummyOrder == 1 && A.CheckGray)

{

HA -= Check;

}

//When is Brown Check.

if (DummyOrder == -1 && A.CheckBrown)

{

HA -= Check;

}

}

//When is Object Dangoure.

if (AA.CheckBrownObjectDangour || AA.CheckGrayObjectDangour)

{

//When is Gray Object.

if (DummyOrder == 1 && AA.CheckGrayObjectDangour)

{

HA -= ObjectDangour;

}

//When is Brown Object Dangoure.

if (DummyOrder == -1 && AA.CheckBrownObjectDangour)

{

HA -= ObjectDangour;

}

}

}

}

}

catch (Exception t)

{

Log(t);

}

if (HA < 0)

IgnoreFromCheckandMateHuristic = true;

ChessRules.CurrentOrder = DummyCurrentOrder;

return HA;

}

}

//Veryfy and detect Object Value.

int VeryFye(int[,] Table, int Order, Color a)

{

Object O = new Object();

lock (O)

{

int HA = 0;

int Object = Table[Row, Column];

//Wehn Solider.

if (System.Math.Abs(Object) == 1)

HA = 1;

//When Elephant.

else if (System.Math.Abs(Object) == 2)

HA = 2;

//When Hourse.

else if (System.Math.Abs(Object) == 3)

HA = 3;

//When Castles.

else if (System.Math.Abs(Object) == 4)

HA = 5;

//When Minster.

else if (System.Math.Abs(Object) == 5)

HA = 8;

//When King.

else if (System.Math.Abs(Object) == 6)

HA = 10;

return HA;

}

}

//QC\_OK

//Numbers of Supporting Current Objects method.

int SupporterCount(int[,] Table, int Order, Color a, int ii, int jj)

{

Object O = new Object();

lock (O)

{

int Count = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

if (Order == 1)

ChessRules.CurrentOrder = 1;

else

ChessRules.CurrentOrder = -1;

bool[,] Tab = new bool[8, 8];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

if (Order == 1 && Table[i, j] <= 0)

continue;

else

if (Order == -1 && Table[i, j] >= 0)

continue;

if (Support(Table, i, j, ii, jj, a, Order))

{

Count++;

}

}

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

return Count;

}

}

//Attacks on Enemies.

int AttackerCount(int[,] Table, int Order, Color a, int i, int j)

{

Object O = new Object();

lock (O)

{

int Count = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

int[,] Tab = new int[8, 8];

for (int h = 0; h < 8; h++)

for (int k = 0; k < 8; k++)

Tab[h, k] = Table[h, k];

//For Slef Objects..

for (int ii = 0; ii < 8; ii++)

for (int jj = 0; jj < 8; jj++)

{

//Ignore Of Self Objects

if (Order == 1 && Tab[ii, jj] >= 0)

continue;

else

if (Order == -1 && Tab[ii, jj] <= 0)

continue;

//If Current Attacks Enemy.

if (Attack(Tab, i, j, ii, jj, a, Order))

{

Count++;

}

}

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

return Count;

}

}

//Attackers of Enemies.QC\_OK.

int EnemyAttackerCount(int[,] Table, int Order, Color a, int ii, int jj)

{

Object O = new Object();

lock (O)

{

int Count = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

if (Order == 1)

ChessRules.CurrentOrder = 1;

else

ChessRules.CurrentOrder = -1;

int[,] Tab = new int[8, 8];

for (int h = 0; h < 8; h++)

for (int k = 0; k < 8; k++)

Tab[h, k] = Table[h, k];

for (int i = 0; i < 8; i++)

for (int j = 0; j < 8; j++)

{

if (Order == 1 && Table[i, j] >= 0)

continue;

else

if (Order == -1 && Table[i, j] <= 0)

continue;

if (Attack(Table, i, j, ii, jj, a, Order \* -1))

{

Count++;

}

}

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

return Count;

}

}

//Distance of Enemy Kings from Current Object.

public double HeuristicDistabceOfCurrentMoveFromEnemyKing(int[,] Tab, int Order, int ii, int jj)

{

Object O = new Object();

lock (O)

{

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = 0;

//Initiate.

int RowG = -1, ColumnG = -1, RowB = -1, ColumnB = -1;

//Create ChessRules Objects.

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Tab[ii, jj], Tab, Order, ii, jj);

double Dis = 0;

//Order is Gray.

if (Order == -1)

{

//Found of Gray King Location.

A.FindGrayKing(Tab, ref RowG, ref ColumnG);

//When Soldier.

if (System.Math.Abs(Tab[ii, jj]) == 1)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowG, 2) + System.Math.Pow(jj - ColumnG, 2));

else

//When Elephant.

if (System.Math.Abs(Tab[ii, jj]) == 2)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowG, 2) + System.Math.Pow(jj - ColumnG, 2));

else

//When Hourse.

if (System.Math.Abs(Tab[ii, jj]) == 3)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowG, 2) + System.Math.Pow(jj - ColumnG, 2));

else

//When Castles.

if (System.Math.Abs(Tab[ii, jj]) == 4)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowG, 2) + System.Math.Pow(jj - ColumnG, 2));

else

//When minister.

if (System.Math.Abs(Tab[ii, jj]) == 5)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowG, 2) + System.Math.Pow(jj - ColumnG, 2));

else

//When King.

if (System.Math.Abs(Tab[ii, jj]) == 6)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowG, 2) + System.Math.Pow(jj - ColumnG, 2));

HeuristicDistabceOfCurrentMoveFromEnemyKingValue += Dis \* SignOrderToPlate(Order);

}

//Brown Order.

else

{

//Found of Brown King Location.

A.FindBrownKing(Tab, ref RowB, ref ColumnB);

//When Soldier.

if (System.Math.Abs(Tab[ii, jj]) == 1)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowB, 2) + System.Math.Pow(jj - ColumnB, 2));

else

//When Elephant.

if (System.Math.Abs(Tab[ii, jj]) == 2)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowB, 2) + System.Math.Pow(jj - ColumnB, 2));

else

//When Hourse.

if (System.Math.Abs(Tab[ii, jj]) == 3)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowB, 2) + System.Math.Pow(jj - ColumnB, 2));

else

//When Castles.

if (System.Math.Abs(Tab[ii, jj]) == 4)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowB, 2) + System.Math.Pow(jj - ColumnB, 2));

else

//When Minister.

if (System.Math.Abs(Tab[ii, jj]) == 5)

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowB, 2) + System.Math.Pow(jj - ColumnB, 2));

else

//When King.

Dis = AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - RowB, 2) + System.Math.Pow(jj - ColumnB, 2));

//Dis = -1000.0;

HeuristicDistabceOfCurrentMoveFromEnemyKingValue += Dis \* SignOrderToPlate(Order);

}

return HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

}

}

public double HuristicSoldierFromCenter(int[,] Table, Color aa, int Ord, int ii, int jj, int i, int j)

{

Object O = new Object();

lock (O)

{

double HA = 0;

Object O1 = new Object();

lock (O1)

{

if (Table[ii, jj] == 0)

{

if (!ArrangmentsChanged)

{

if (Order == 1)

{

if (i < 4 && j < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 3, 2) + System.Math.Pow(j - 3, 2));

}

if (i < 4 && j >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 3, 2) + System.Math.Pow(j - 4, 2));

}

}

else

{

if (i >= 4 && j < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 4, 2) + System.Math.Pow(j - 3, 2));

}

if (i >= 4 && j >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 4, 2) + System.Math.Pow(j - 4, 2));

}

}

}

else

{

if (Order == -1)

{

if (i < 4 && j < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 3, 2) + System.Math.Pow(j - 3, 2));

}

if (i < 4 && j >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 3, 2) + System.Math.Pow(j - 4, 2));

}

}

else

{

if (i >= 4 && j < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 4, 2) + System.Math.Pow(j - 3, 2));

}

if (i >= 4 && j >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(i - 4, 2) + System.Math.Pow(j - 4, 2));

}

}

}

}

else if (Table[i, j] == 0)

{

if (!ArrangmentsChanged)

{

if (Order == 1)

{

if (ii < 4 && jj < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 3, 2) + System.Math.Pow(jj - 3, 2));

}

if (ii < 4 && jj >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 3, 2) + System.Math.Pow(jj - 4, 2));

}

}

else

{

if (ii >= 4 && jj < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 4, 2) + System.Math.Pow(jj - 3, 2));

}

if (ii >= 4 && jj >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 4, 2) + System.Math.Pow(jj - 4, 2));

}

}

}

else

{

if (Order == -1)

{

if (ii < 4 && jj < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 3, 2) + System.Math.Pow(jj - 3, 2));

}

if (ii < 4 && jj >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 3, 2) + System.Math.Pow(jj - 4, 2));

}

}

else

{

if (ii >= 4 && jj < 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 4, 2) + System.Math.Pow(jj - 3, 2));

}

if (ii >= 4 && jj >= 4)

{

HA += AllDraw.SignDistance \* System.Math.Sqrt(System.Math.Pow(ii - 4, 2) + System.Math.Pow(jj - 4, 2));

}

}

}

}

}

return HA \* SignOrderToPlate(Order);

}

}

public double[] HuristicAll(bool Before, int Killed, int[,] Table, Color aa, int Ord, int iii, int jjj, int iiii, int jjjj)

{

Object O = new Object();

lock (O)

{

double[] Huristic = new double[6];

//Initiate Local Variable.

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

///When AStarGreedy Huristic is Not Assigned.

try

{

if (!AStarGreedyHuristicT)

{

//For Current Objects.

Parallel.For(0, 8, i =>

{

Parallel.For(0, 8, j =>

{

Object O1 = new Object();

lock (O1)

{

int i1 = i, j1 = j, iiii1 = iii, jjjj1 = jjj;

int[,] Table1 = CloneATable(Table);

int Ord1 = Ord;

Color aa1 = aa;

double HAA1 = HuristicAttack(Before, Table1, Ord1, aa1, i1, j1, iiii1, jjjj1);

if (HAA1 != 0)

Huristic[0] += HAA1;

int i2 = i, j2 = j, iiii2 = iii, jjjj2 = jjj;

int[,] Table2 = CloneATable(Table);

int Ord2 = Ord;

Color aa2 = aa;

int Killed1 = Killed;

double HAA2 = HuristicKiller(Killed1, Table2, i2, j2, iiii2, jjjj2, Ord2, aa2, false);

if (HAA2 != 0)

Huristic[1] += HAA2;

int i3 = i, j3 = j, iiii3 = iii, jjjj3 = jjj;

int[,] Table3 = CloneATable(Table);

int Ord3 = Ord;

Color aa3 = aa;

double HAA3 = HuristicMovment(Before, Table3, aa3, Ord3, i3, j3, iiii3, jjjj3);

if (HAA3 != 0)

Huristic[2] += HAA3;

int i4 = i, j4 = j, iiii4 = iii, jjjj4 = jjj;

int[,] Table4 = CloneATable(Table);

int Ord4 = Ord;

Color aa4 = aa;

double HAA4 = HuristicObjectDangour(Table4, Ord4, aa4, i4, j4, iiii4, jjjj4);

if (HAA4 != 0)

Huristic[3] += HAA4;

int i5 = i, j5 = j, iiii5 = iii, jjjj5 = jjj;

int[,] Table5 = CloneATable(Table);

int Ord5 = Ord;

Color aa5 = aa;

double HAA5 = HuristicReducsedAttack(Before, Table5, Ord5, aa5, i5, j5, iiii5, jjjj5);

if (HAA5 != 0)

Huristic[4] += HAA5;

int i6 = i, j6 = j, iiii6 = iii, jjjj6 = jjj;

int[,] Table6 = CloneATable(Table);

int Ord6 = Ord;

Color aa6 = aa;

double HAA6 = HuristicSelfSupported(Table6, Ord6, aa6, i6, j6, iiii6, jjjj6);

if (HAA6 != 0)

Huristic[5] += HAA6;

}

});

});

}

//For All Homes Table.

else

{

Parallel.For(0, 8, i =>

{

Parallel.For(0, 8, j =>

{

Parallel.For(0, 8, ii =>

{

Parallel.For(0, 8, jj =>

{

Object O1 = new Object();

lock (O1)

{

int i1 = i, j1 = j, iiii1 = iii, jjjj1 = jjj;

int[,] Table1 = CloneATable(Table);

int Ord1 = Ord;

Color aa1 = aa;

double HAA1 = HuristicAttack(Before, Table1, Ord1, aa1, i1, j1, iiii1, jjjj1);

Huristic[0] += HAA1;

int i2 = i, j2 = j, iiii2 = iii, jjjj2 = jjj;

int[,] Table2 = CloneATable(Table);

int Ord2 = Ord;

Color aa2 = aa;

int Killed1 = Killed;

double HAA2 = HuristicKiller(Killed1, Table2, i2, j2, iiii2, jjjj2, Ord2, aa2, false);

Huristic[1] += HAA2;

int i3 = i, j3 = j, iiii3 = iii, jjjj3 = jjj;

int[,] Table3 = CloneATable(Table);

int Ord3 = Ord;

Color aa3 = aa;

double HAA3 = HuristicMovment(Before, Table3, aa3, Ord3, i3, j3, iiii3, jjjj3);

Huristic[2] += HAA3;

int i4 = i, j4 = j, iiii4 = iii, jjjj4 = jjj;

int[,] Table4 = CloneATable(Table);

int Ord4 = Ord;

Color aa4 = aa;

double HAA4 = HuristicObjectDangour(Table4, Ord4, aa4, i4, j4, iiii4, jjjj4);

Huristic[3] += HAA4;

int i5 = i, j5 = j, iiii5 = iii, jjjj5 = jjj;

int[,] Table5 = CloneATable(Table);

int Ord5 = Ord;

Color aa5 = aa;

double HAA5 = HuristicReducsedAttack(Before, Table5, Ord5, aa5, i5, j5, iiii5, jjjj5);

Huristic[4] += HAA5;

int i6 = i, j6 = j, iiii6 = iii, jjjj6 = jjj;

int[,] Table6 = CloneATable(Table);

int Ord6 = Ord;

Color aa6 = aa;

double HAA6 = HuristicSelfSupported(Table6, Ord6, aa6, i6, j6, iiii6, jjjj6);

Huristic[5] += HAA6;

}

});

});

});

});

}

}

catch (Exception t)

{

Log(t);

}

//Reassignments of Begin Call Global Orders.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//Store Local Huristic in Global One.

//Huristic[0] = (Huristic[0]\* SignOrderToPlate(Order));

//Huristic[1] = (Huristic[1]\* SignOrderToPlate(Order));

//Huristic[2] = (Huristic[2]\* SignOrderToPlate(Order));

//Huristic[3] = (Huristic[3]\* SignOrderToPlate(Order));

//Huristic[4] = (Huristic[4]\* SignOrderToPlate(Order));

//Huristic[5] = (Huristic[5]\* SignOrderToPlate(Order));

//Return Local Huristic.

return Huristic;

}

}

///Huristic of Movments.

public double HuristicMovment(bool Before, int[,] Table, Color aa, int Ord, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

double HuristicMovementValue = 0;

//Initiate Local Variable.

double HA = 0;

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

///When AStarGreedy Huristic is Not Assigned.

try

{

if (!AStarGreedyHuristicT)

{

int Order = new int();

Color a = new Color();

a = aa;

Order = DummyOrder;

double Sign = new double();

///When Moveble is true. means [ii,jj] is in Movmebale to [i,j].

///What is Moveable!

///Ans:When [ii,jj] is Movebale to [i,j] return true when Empty or Enemy is located in [ii,jj].

if (Table[i, j] == 0 && DummyOrder == -1 && Table[ii, jj] < 0)

{

Order = -1;

Object O1 = new Object();

lock (O1)

{

Sign = 1 \* AllDraw.SignMovments;

ChessRules.CurrentOrder = -1;

}

a = Color.Brown;

}

else if (Table[i, j] == 0 && DummyOrder == 1 && Table[ii, jj] > 0)

{

Order = 1;

Object O1 = new Object();

lock (O1)

{

Sign = 1 \* AllDraw.SignMovments;

ChessRules.CurrentOrder = 1;

}

a = Color.Gray;

}

else

return HuristicMovementValue;

if (Before)

{

//When is Movable Movement inCurrent.

if (Movable(Table, ii, jj, i, j, a, Order))

{

HA += (Sign \* (System.Math.Abs(GetValue(ii, jj) + GetValue(i, j))));

bool Supported = false;

//For All Enemy Obejcts.

Parallel.For(0, 8, g =>

{

if (Supported)

return;

Parallel.For(0, 8, h =>

{

Object O2 = new Object();

lock (O2)

{

if (Supported)

return;

//Ignore Of Self Objects.

if (Order == 1 && Table[g, h] >= 0)

return;

if (Order == -1 && Table[g, h] <= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

aaa = Color.Gray;

if (Order \* -1 == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

//When Enemy is Supported.

bool A = new bool();

A = Support(Table, g, h, i, j, aaa, Order \* -1);

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Supported = true;

return;

}

}

});

if (Supported)

return;

});

Object O1 = new Object();

lock (O1)

{

if (!Supported)

//When is Not Supported multyply 100.

HA \*= 20;

else

//When is Supported Multyply -100.

HA \*= -20;

}

}

}

}

//For All Homes Table.

else

{

int Order = new int();

Color a = new Color();

a = aa;

if (i == ii && j == jj)

return HuristicMovementValue;

double Sign = new double();

Order = DummyOrder;

///When Moveble is true. means [ii,jj] is in Movmebale to [i,j].

///What is Moveable!

///Ans:When [ii,jj] is Movebale to [i,j] return true when Empty or Enemy is located in [ii,jj].

if (Table[i, j] == 0 && DummyOrder == -1 && Table[ii, jj] < 0)

{

Order = -1;

Object O1 = new Object();

lock (O1)

{

Sign = 1 \* AllDraw.SignMovments;

ChessRules.CurrentOrder = -1;

a = Color.Brown;

}

}

else if (Table[i, j] == 0 && DummyOrder == 1 && Table[ii, jj] > 0)

{

Order = 1;

Object O1 = new Object();

lock (O1)

{

Sign = 1 \* AllDraw.SignMovments;

ChessRules.CurrentOrder = 1;

a = Color.Gray;

}

}

else

return HuristicMovementValue;

if (Before)

{

//When is Movable Movement inCurrent.

if (Movable(Table, ii, jj, i, j, a, Order))

{

HA += (Sign \* (System.Math.Abs(GetValue(ii, jj) + GetValue(i, j))));

bool Supported = false;

//For All Enemy Obejcts.

Parallel.For(0, 8, g =>

{

if (Supported)

return;

//for (int h = 0; h < 8; h++)

Parallel.For(0, 8, h =>

{

if (Supported)

return;

//Ignore Of Self Objects.

if (Order == 1 && Table[g, h] >= 0)

return;

if (Order == -1 && Table[g, h] <= 0)

return;

Color aaa = new Color();

//Assgin Enemy ints.

aaa = Color.Gray;

if (Order \* -1 == -1)

aaa = Color.Brown;

else

aaa = Color.Gray;

//When Enemy is Supported.

bool A = new bool();

Object O2 = new Object();

lock (O2)

{

A = Support(Table, g, h, i, j, aaa, Order \* -1);

}

//When Enemy is Supported.

if (A)

{

//Assgine variable.

Supported = true;

return;

}

});

if (Supported)

return;

});

if (!Supported)

//When is Not Supported multyply 100.

HA \*= 20;

else

//When is Supported Multyply -100.

HA \*= -20;

}

}

}

}

catch (Exception t)

{

Log(t);

}

//Reassignments of Begin Call Global Orders.

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//Store Local Huristic in Global One.

return HA;

}

}

///Attack Determination.QC\_Ok

public bool Attack(int[,] Tab, int i, int j, int ii, int jj, Color a, int Order)

{

Object O = new Object();

lock (O)

{

int CCurentOrder = ChessRules.CurrentOrder;

//Initiate Global static Variable.

ChessRules.CurrentOrder = Order;

int[,] Table = new int[8, 8];

for (int ik = 0; ik < 8; ik++)

for (int jk = 0; jk < 8; jk++)

Table[ik, jk] = Tab[ik, jk];

//when there is a Movment from Parameter One to Second Parameter return Attacke..

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[i, j], Table, Order, i, j)).Rules(i, j, ii, jj, a, Order) //&& Table[ii, jj] != 0

)

{

ChessRules.CurrentOrder = CCurentOrder;

return true;

}

ChessRules.CurrentOrder = CCurentOrder;

return false;

}

}

///ObjectDanger Determination.

public bool ObjectDanger(int[,] Tab, int i, int j, int ii, int jj, Color a, int Order)

{

Object O = new Object();

lock (O)

{

int CCurrentOrder = ChessRules.CurrentOrder;

//Initiate Local Varibales.

int[,] Table = new int[8, 8];

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

Table[iii, jjj] = Tab[iii, jjj];

}

ChessRules.CurrentOrder = Order;

///When [i,j] is Attacked [ii,jj] retrun true when enemy is located in [ii,jj].

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[i, j], Table, Order, i, j)).Rules(i, j, ii, jj, a, Order))

{

//Initiate Local Variables.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

Table[iii, jjj] = Tab[iii, jjj];

}

//Take Movments.

Table[ii, jj] = Table[i, j];

Table[i, j] = 0;

//Consider Check.

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[ii, jj], Table, Order, ii, jj);

if (AA.ObjectDangourKingMove(Order, Table, false))

{

ChessRules.CurrentOrder = CCurrentOrder;

//Return ObjectDanger.

if ((AA.CheckGrayObjectDangour) && Order == 1)

return true;

else

if ((AA.CheckBrownObjectDangour) && Order == -1)

return true;

}

if (AA.CheckMate(Table, Order))

{

ChessRules.CurrentOrder = CCurrentOrder;

//Return ObjectDanger.

if ((AA.CheckGray || AA.CheckMateGray) && Order == 1)

return true;

else

if ((AA.CheckBrown || AA.CheckMateBrown) && Order == -1)

return true;

}

}

ChessRules.CurrentOrder = CCurrentOrder;

//return Non ObjectDanger.

return false;

}

}

///Supportation Determination.QC\_OK

public bool Support(int[,] Tab, int i, int j, int ii, int jj, Color a, int Order)

{

Object O = new Object();

lock (O)

{

//Initiate Local Variables.

int[,] Table = new int[8, 8];

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

Table[iii, jjj] = Tab[iii, jjj];

///When All Tables is Gray.

if (Table[i, j] > 0 && Table[ii, jj] >= 0)

{

///When [i,j] Supporte [ii,jj].

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[i, j], Table, Order, i, j)).Rules(i, j, ii, jj, a, Table[i, j], false))

{

return true;

}

}

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

Table[iii, jjj] = Tab[iii, jjj];

///When All is Brown.

if (Table[i, j] < 0 && Table[ii, jj] <= 0)

{

///When [i,j] Supporetd [ii,jj].

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[i, j], Table, Order, i, j)).Rules(i, j, ii, jj, a, Table[i, j], false))

{

return true;

}

}

return false;

}

}

//Return Msx Huiristic of Child Level.

public bool MaxHuristic(ref int j, int Kin, ref double Less, int Order)

{

Object O = new Object();

lock (O)

{

bool Found = false;

//When Solders.

if (Kin == 1)

{

for (int i = 0; i < this.PenaltyRegardListSolder.Count; i++)

{

if (PenaltyRegardListSolder[i].IsPenaltyAction() != 0)

{

if (Order == AllDraw.OrderPlate)

{

if (Less > HuristicListSolder[i][0] +

HuristicListSolder[i][1] +

HuristicListSolder[i][2] +

HuristicListSolder[i][3] +

HuristicListSolder[i][4] +

HuristicListSolder[i][5] +

HuristicListSolder[i][6] +

HuristicListSolder[i][7] +

HuristicListSolder[i][8] +

HuristicListSolder[i][9])

{

Less = HuristicListSolder[i][0] +

HuristicListSolder[i][1] +

HuristicListSolder[i][2] +

HuristicListSolder[i][3] +

HuristicListSolder[i][4] +

HuristicListSolder[i][5] +

HuristicListSolder[i][6] +

HuristicListSolder[i][7] +

HuristicListSolder[i][8] +

HuristicListSolder[i][9];

j = i;

Found = true;

}

}

else

{

if (Less < HuristicListSolder[i][0] +

HuristicListSolder[i][1] +

HuristicListSolder[i][2] +

HuristicListSolder[i][3] +

HuristicListSolder[i][4] +

HuristicListSolder[i][5] +

HuristicListSolder[i][6] +

HuristicListSolder[i][7] +

HuristicListSolder[i][8] +

HuristicListSolder[i][9])

{

Less = HuristicListSolder[i][0] +

HuristicListSolder[i][1] +

HuristicListSolder[i][2] +

HuristicListSolder[i][3] +

HuristicListSolder[i][4] +

HuristicListSolder[i][5] +

HuristicListSolder[i][6] +

HuristicListSolder[i][7] +

HuristicListSolder[i][8] +

HuristicListSolder[i][9];

j = i;

Found = true;

}

}

}

}

}

else//When Elephant.

if (Kin == 2)

{

for (int i = 0; i < this.PenaltyRegardListElefant.Count; i++)

{

if (PenaltyRegardListElefant[i].IsPenaltyAction() != 0)

{

if (Order == AllDraw.OrderPlate)

{

if (Less > HuristicListElefant[i][0] +

HuristicListElefant[i][1] +

HuristicListElefant[i][2] +

HuristicListElefant[i][3] +

HuristicListElefant[i][4] +

HuristicListElefant[i][5] +

HuristicListElefant[i][6] +

HuristicListElefant[i][7] +

HuristicListElefant[i][8] +

HuristicListElefant[i][9])

{

Less = HuristicListElefant[i][0] +

HuristicListElefant[i][1] +

HuristicListElefant[i][2] +

HuristicListElefant[i][3] +

HuristicListElefant[i][4] +

HuristicListElefant[i][5] +

HuristicListElefant[i][6] +

HuristicListElefant[i][7] +

HuristicListElefant[i][8] +

HuristicListElefant[i][9];

j = i;

Found = true;

}

}

else

{

if (Less < HuristicListElefant[i][0] +

HuristicListElefant[i][1] +

HuristicListElefant[i][2] +

HuristicListElefant[i][3] +

HuristicListElefant[i][4] +

HuristicListElefant[i][5] +

HuristicListElefant[i][6] +

HuristicListElefant[i][7] +

HuristicListElefant[i][8] +

HuristicListElefant[i][9])

{

Less = HuristicListElefant[i][0] +

HuristicListElefant[i][1] +

HuristicListElefant[i][2] +

HuristicListElefant[i][3] +

HuristicListElefant[i][4] +

HuristicListElefant[i][5] +

HuristicListElefant[i][6] +

HuristicListElefant[i][7] +

HuristicListElefant[i][8] +

HuristicListElefant[i][9];

j = i;

Found = true;

}

}

}

}

}

else//When Hourse.

if (Kin == 3)

{

for (int i = 0; i < this.PenaltyRegardListHourse.Count; i++)

{

if (PenaltyRegardListHourse[i].IsPenaltyAction() != 0)

{

if (Order == AllDraw.OrderPlate)

{

if (Less > HuristicListHourse[i][0] +

HuristicListHourse[i][1] +

HuristicListHourse[i][2] +

HuristicListHourse[i][3] +

HuristicListHourse[i][4] +

HuristicListHourse[i][5] +

HuristicListHourse[i][6] +

HuristicListHourse[i][7] +

HuristicListHourse[i][8] +

HuristicListHourse[i][9])

{

Less = HuristicListHourse[i][0] +

HuristicListHourse[i][1] +

HuristicListHourse[i][2] +

HuristicListHourse[i][3] +

HuristicListHourse[i][4] +

HuristicListHourse[i][5] +

HuristicListHourse[i][6] +

HuristicListHourse[i][7] +

HuristicListHourse[i][8] +

HuristicListHourse[i][9];

j = i;

Found = true;

}

}

else

{

if (Less < HuristicListHourse[i][0] +

HuristicListHourse[i][1] +

HuristicListHourse[i][2] +

HuristicListHourse[i][3] +

HuristicListHourse[i][4] +

HuristicListHourse[i][5] +

HuristicListHourse[i][6] +

HuristicListHourse[i][7] +

HuristicListHourse[i][8] +

HuristicListHourse[i][9])

{

Less = HuristicListHourse[i][0] +

HuristicListHourse[i][1] +

HuristicListHourse[i][2] +

HuristicListHourse[i][3] +

HuristicListHourse[i][4] +

HuristicListHourse[i][5] +

HuristicListHourse[i][6] +

HuristicListHourse[i][7] +

HuristicListHourse[i][8] +

HuristicListHourse[i][9];

j = i;

Found = true;

}

}

}

}

}

else//When Castles.

if (Kin == 4)

{

for (int i = 0; i < this.PenaltyRegardListCastle.Count; i++)

{

if (PenaltyRegardListCastle[i].IsPenaltyAction() != 0)

{

if (Order == AllDraw.OrderPlate)

{

if (Less > HuristicListCastle[i][0] +

HuristicListCastle[i][1] +

HuristicListCastle[i][2] +

HuristicListCastle[i][3] +

HuristicListCastle[i][4] +

HuristicListCastle[i][5] +

HuristicListCastle[i][6] +

HuristicListCastle[i][7] +

HuristicListCastle[i][8] +

HuristicListCastle[i][9])

{

Less = HuristicListCastle[i][0] +

HuristicListCastle[i][1] +

HuristicListCastle[i][2] +

HuristicListCastle[i][3] +

HuristicListCastle[i][4] +

HuristicListCastle[i][5] +

HuristicListCastle[i][6] +

HuristicListCastle[i][7] +

HuristicListCastle[i][8] +

HuristicListCastle[i][9];

j = i;

Found = true;

}

}

else

{

if (Less < HuristicListCastle[i][0] +

HuristicListCastle[i][1] +

HuristicListCastle[i][2] +

HuristicListCastle[i][3] +

HuristicListCastle[i][4] +

HuristicListCastle[i][5] +

HuristicListCastle[i][6] +

HuristicListCastle[i][7] +

HuristicListCastle[i][8] +

HuristicListCastle[i][9])

{

Less = HuristicListCastle[i][0] +

HuristicListCastle[i][1] +

HuristicListCastle[i][2] +

HuristicListCastle[i][3] +

HuristicListCastle[i][4] +

HuristicListCastle[i][5] +

HuristicListCastle[i][6] +

HuristicListCastle[i][7] +

HuristicListCastle[i][8] +

HuristicListCastle[i][9];

j = i;

Found = true;

}

}

}

}

}

else//When Minister.

if (Kin == 5)

{

for (int i = 0; i < this.PenaltyRegardListMinister.Count; i++)

{

if (PenaltyRegardListMinister[i].IsPenaltyAction() != 0)

{

if (Order == AllDraw.OrderPlate)

{

if (Less > HuristicListMinister[i][0] +

HuristicListMinister[i][1] +

HuristicListMinister[i][2] +

HuristicListMinister[i][3] +

HuristicListMinister[i][4] +

HuristicListMinister[i][5] +

HuristicListMinister[i][6] +

HuristicListMinister[i][7] +

HuristicListMinister[i][8] +

HuristicListMinister[i][9]

)

{

Less = HuristicListMinister[i][0] +

HuristicListMinister[i][1] +

HuristicListMinister[i][2] +

HuristicListMinister[i][3] +

HuristicListMinister[i][4] +

HuristicListMinister[i][5] +

HuristicListMinister[i][6] +

HuristicListMinister[i][7] +

HuristicListMinister[i][8] +

HuristicListMinister[i][9];

j = i;

Found = true;

}

}

else

{

if (Less < HuristicListMinister[i][0] +

HuristicListMinister[i][1] +

HuristicListMinister[i][2] +

HuristicListMinister[i][3] +

HuristicListMinister[i][4] +

HuristicListMinister[i][5] +

HuristicListMinister[i][6] +

HuristicListMinister[i][7] +

HuristicListMinister[i][8] +

HuristicListMinister[i][9]

)

{

Less = HuristicListMinister[i][0] +

HuristicListMinister[i][1] +

HuristicListMinister[i][2] +

HuristicListMinister[i][3] +

HuristicListMinister[i][4] +

HuristicListMinister[i][5] +

HuristicListMinister[i][6] +

HuristicListMinister[i][7] +

HuristicListMinister[i][8] +

HuristicListMinister[i][9];

j = i;

Found = true;

}

}

}

}

}

else//When King.

if (Kin == 6)

{

for (int i = 0; i < this.PenaltyRegardListKing.Count; i++)

{

if (PenaltyRegardListKing[i].IsPenaltyAction() != 0)

{

if (Order == AllDraw.OrderPlate)

{

if (Less > HuristicListKing[i][0] +

HuristicListKing[i][1] +

HuristicListKing[i][2] +

HuristicListKing[i][3] +

HuristicListKing[i][4] +

HuristicListKing[i][5] +

HuristicListKing[i][6] +

HuristicListKing[i][7] +

HuristicListKing[i][8] +

HuristicListKing[i][9])

{

Less = HuristicListKing[i][0] +

HuristicListKing[i][1] +

HuristicListKing[i][2] +

HuristicListKing[i][3] +

HuristicListKing[i][4] +

HuristicListKing[i][5] +

HuristicListKing[i][6] +

HuristicListKing[i][7] +

HuristicListKing[i][8] +

HuristicListKing[i][9];

j = i;

Found = true;

}

}

else

{

if (Less < HuristicListKing[i][0] +

HuristicListKing[i][1] +

HuristicListKing[i][2] +

HuristicListKing[i][3] +

HuristicListKing[i][4] +

HuristicListKing[i][5] +

HuristicListKing[i][6] +

HuristicListKing[i][7] +

HuristicListKing[i][8] +

HuristicListKing[i][9])

{

Less = HuristicListKing[i][0] +

HuristicListKing[i][1] +

HuristicListKing[i][2] +

HuristicListKing[i][3] +

HuristicListKing[i][4] +

HuristicListKing[i][5] +

HuristicListKing[i][6] +

HuristicListKing[i][7] +

HuristicListKing[i][8] +

HuristicListKing[i][9];

j = i;

Found = true;

}

}

}

}

}

return Found;

}

}

//Setting Numbers of Objects in Current Table boards.

//Count of Solders on Table.

int SolderOnTableCount(ref DrawSoldier[] So, bool Mi, int MaxCount)

{

Object O = new Object();

lock (O)

{

int Count = 0, i = 0;

//For Alll Solders on int Calculate Solkder Count.

while (i < MaxCount)

{

//The Index out of range exeption is not fixable.

try

{

if (So != null) if (So[i] != null)

{

//When int is Gray or Brown.

if (So[i].color == Color.Gray || So[i].color == Color.Brown)

{

if (Mi)

{

if (So[i].color == Color.Gray)

Count++;

}

else

Count++;

}

else

So[i] = null;

}

}

catch (Exception t) { Log(t); }

i++;

};

return Count;

}

}

//Elepahnt On Table Count.

int ElefantOnTableCount(ref DrawElefant[] So, bool Mi, int MaxCount)

{

Object O = new Object();

lock (O)

{

int Count = 0, i = 0;

//For All Elephant items in Table.

while (i < MaxCount)

{

try

{

//The Index out of range exeption is not fixable.

if (So != null) if (So[i] != null)

{

//when Elaphant int is Gray or Brown.

if (So[i].color == Color.Gray || So[i].color == Color.Brown)

{

if (Mi)

{

if (So[i].color == Color.Gray)

Count++;

}

else

Count++;

}

else

So[i] = null;

}

}

catch (Exception t) { Log(t); }

i++;

};

return Count;

}

}

//Calculate Hourse on table.

int HourseOnTableCount(ref DrawHourse[] So, bool Mi, int MaxCount)

{

Object O = new Object();

lock (O)

{

int Count = 0, i = 0;

while (i < MaxCount)

{

//For All Hourse on Table .

//The Index out of range exeption is not fixable.

try

{

if (So != null) if (So[i] != null)

{

//When int is Gray or Brown.

if (So[i].color == Color.Gray || So[i].color == Color.Brown)

{

if (Mi)

{

if (So[i].color == Color.Gray)

Count++;

}

else

Count++;

}

else

So[i] = null;

}

}

catch (Exception t) { Log(t); }

i++;

};

return Count;

}

}

//Calculate Castles Count.

int CastleOnTableCount(ref DrawCastle[] So, bool Mi, int MaxCount)

{

Object O = new Object();

lock (O)

{

int Count = 0, i = 0;

while (i < MaxCount)

{

try

{

//The Index out of range exeption is not fixable.

if (So != null) if (So[i] != null)

{

//When Castles int is Gray or Brown.

if (So[i].color == Color.Gray || So[i].color == Color.Brown)

{

if (Mi)

{

if (So[i].color == Color.Gray)

Count++;

}

else

Count++;

}

else

So[i] = null;

}

}

catch (Exception t) { Log(t); }

i++;

};

return Count;

}

}

//Calculate Minsiter Count.

int MinisterOnTableCount(ref DrawMinister[] So, bool Mi, int MaxCount)

{

Object O = new Object();

lock (O)

{

int Count = 0, i = 0;

while (i < MaxCount)

{

try

{

//The Index out of range exeption is not fixable.

if (So != null) if (So[i] != null)

{

//When int of items is gray or Brown.

if (So[i].color == Color.Gray || So[i].color == Color.Brown)

{

if (Mi)

{

if (So[i].color == Color.Gray)

Count++;

}

else

Count++;

}

else

So[i] = null;

}

}

catch (Exception t) { Log(t); }

i++;

};

return Count;

}

}

//Calculate King on Table.

int KingOnTableCount(ref DrawKing[] So, bool Mi, int MaxCount)

{

Object O = new Object();

lock (O)

{

int Count = 0, i = 0;

while (i < MaxCount)

{

try

{

//The Index out of range exeption is not fixable.

if (So != null) if (So[i] != null)

{

//when int is Gray or Brown.

if (So[i].color == Color.Gray || So[i].color == Color.Brown)

{

if (Mi)

{

if (So[i].color == Color.Gray)

Count++;

}

else

Count++;

}

else

So[i] = null;

}

}

catch (Exception t) { Log(t); }

i++;

};

return Count;

}

}

//Return Huristic.

public double ReturnHuristic(int ii, int j, int Order, bool AA)

{

Object O = new Object();

lock (O)

{

ActionsString = "";

//AllDraw.ActionStringReady = false;

//NumbersOfCurrentBranchesPenalties = 0;

//calculation of huristic methos and storing value retured.

double Hur = new double();

Object O1 = new Object();

lock (O1)

{

if (!AA)

{

if (ii >= 0 && UsePenaltyRegardMechnisamT)

Hur = ReturnHuristicCalculartor(0, ii, j, Order) \* LearniningTable.LearingValue(Row, Column);

else

Hur = ReturnHuristicCalculartor(0, ii, j, Order);

}

else

Hur = ReturnHuristicCalculartor(0, ii, j, Order) + 1000;

}

//Optimization depend of numbers of unpealties nodes quefficient.

return Hur \* ((double)(NumbersOfAllNode - NumbersOfCurrentBranchesPenalties) / (double)(NumbersOfAllNode));

}

}

String Alphabet(int RowRealesed)

{

Object O = new Object();

lock (O)

{

String A = "";

if (RowRealesed == 0)

A = "a";

else

if (RowRealesed == 1)

A = "b";

else

if (RowRealesed == 2)

A = "c";

else

if (RowRealesed == 3)

A = "d";

else

if (RowRealesed == 4)

A = "e";

else

if (RowRealesed == 5)

A = "f";

else

if (RowRealesed == 6)

A = "g";

else

if (RowRealesed == 7)

A = "h";

return A;

}

}

String Number(int ColumnRealeased)

{

Object O = new Object();

lock (O)

{

String A = "";

if (ColumnRealeased == 7)

A = "0";

else

if (ColumnRealeased == 6)

A = "1";

else

if (ColumnRealeased == 5)

A = "2";

else

if (ColumnRealeased == 4)

A = "3";

else

if (ColumnRealeased == 3)

A = "4";

else

if (ColumnRealeased == 2)

A = "5";

else

if (ColumnRealeased == 1)

A = "6";

else

if (ColumnRealeased == 0)

A = "7";

return A;

}

}

public double ReturnHuristicCalculartor(int iAstarGready, int ii, int j, int Order)

{

Object O = new Object();

lock (O)

{

//if (iAstarGready > AllDraw.MaxAStarGreedy)

//return 0;

SetObjectNumbers(TableConst);

//NumbersOfCurrentBranchesPenalties = 0;

double Huristic = 0; ;

int[] iIndex = { -1, -1, -1, -1, -1, -1 }, mIndex = { -1, -1, -1, -1, -1, -1 }, jIndex = { -1, -1, -1, -1, -1, -1 }, Kin = { -1, -1, -1, -1, -1, -1 };

double[] Less = new double[6];

if (Order == AllDraw.OrderPlate)

{

for (int i = 0; i < 6; i++)

{

Less[i] = Double.MinValue;

}

}

else

{

for (int i = 0; i < 6; i++)

{

Less[i] = Double.MaxValue;

}

}

iAstarGready++;

//Calculate numbers of current branches penalties.

NumbersOfCurrentBranchesPenalties += NumberOfPenalties;

int DummyOrder = Order;

if (ii != -1)

{

//When is Gray.

if (Order == 1)

{

//For All Depth Count.

for (int i = 0; i < AStarGreedy.Count; i++)

{

//For All solder DrawOn Table Count.

for (int m = 0; m < SolderOnTableCount(ref AStarGreedy[i].SolderesOnTable, true, AStarGreedy[i].SodierHigh); m++)

{

//When Depth of Solders On Table is Not NULL.

if (AStarGreedy[i].SolderesOnTable[m] != null)

{

//Calculate Maximum Huristic in Branch.

if (AStarGreedy[i].SolderesOnTable[m].SoldierThinking[0].MaxHuristic(ref jIndex[0], 1, ref Less[0], Order \* -1))

{

iIndex[0] = i;

mIndex[0] = m;

Kin[0] = 1;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Elephant On Table Count.

for (int m = 0; m < ElefantOnTableCount(ref AStarGreedy[i].ElephantOnTable, true, AStarGreedy[i].ElefantHigh); m++)

{

//For All Elephant in Depth Count.

if (AStarGreedy[i].ElephantOnTable[m] != null)

{

//Found of Maxmimum in Branch.

if (AStarGreedy[i].ElephantOnTable[m].ElefantThinking[0].MaxHuristic(ref jIndex[1], 2, ref Less[1], Order \* -1))

{

iIndex[1] = i;

mIndex[1] = m;

Kin[1] = 2;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Hourse on Table Count.

for (int m = 0; m < HourseOnTableCount(ref AStarGreedy[i].HoursesOnTable, true, AStarGreedy[i].HourseHight); m++)

{

//When is HourseOn Table Depth Object is Not NULL.

if (AStarGreedy[i].HoursesOnTable[m] != null)

{

//Forund of Maximum on on Branch.

if (AStarGreedy[i].ElephantOnTable[m].ElefantThinking[0].MaxHuristic(ref jIndex[1], 2, ref Less[1], Order \* -1))

{

iIndex[2] = i;

mIndex[2] = m;

Kin[2] = 3;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Castles on table Count.

for (int m = 0; m < CastleOnTableCount(ref AStarGreedy[i].CastlesOnTable, true, AStarGreedy[i].CastleHigh); m++)

{

//When Depth Objects of Hourse Table is Not NULL.

if (AStarGreedy[i].CastlesOnTable[m] != null)

{

//Found of Maximum Castles Branch.

if (AStarGreedy[i].CastlesOnTable[m].CastleThinking[0].MaxHuristic(ref jIndex[3], 4, ref Less[3], Order \* -1))

{

iIndex[3] = i;

mIndex[3] = m;

Kin[3] = 4;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Minsiter on table count.

for (int m = 0; m < MinisterOnTableCount(ref AStarGreedy[i].MinisterOnTable, true, AStarGreedy[i].MinisterHigh); m++)

{

//When Minster of Depth is Not Null.

if (AStarGreedy[i].MinisterOnTable[m] != null)

{

//Found of Maximum Minster on table Branches.

if (AStarGreedy[i].MinisterOnTable[m].MinisterThinking[0].MaxHuristic(ref jIndex[4], 5, ref Less[4], Order \* -1))

{

iIndex[4] = i;

mIndex[4] = m;

Kin[4] = 5;

//Huristic = Less;

}

}

}

//For All King on table Count.

for (int m = 0; m < KingOnTableCount(ref AStarGreedy[i].KingOnTable, true, AStarGreedy[i].KingHigh); m++)

{

//When Depth Object of King Table is Not NULL.

if (AStarGreedy[i].KingOnTable[m] != null)

{

//Found of Maximum on table Branches.

if (AStarGreedy[i].KingOnTable[m].KingThinking[0].MaxHuristic(ref jIndex[5], 1, ref Less[5], Order \* -1))

{

iIndex[5] = i;

mIndex[5] = m;

Kin[5] = 6;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName()); ;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

}

}

else

{

//For All Depth Variables.

for (int i = 0; i < AStarGreedy.Count; i++)

{

//For All Brown Solders on table count.

for (int m = SolderOnTableCount(ref AStarGreedy[i].SolderesOnTable, true, AStarGreedy[i].SodierHigh); m < SolderOnTableCount(ref AStarGreedy[i].SolderesOnTable, false, AStarGreedy[i].SodierHigh); m++)

{

//When solderis on table depth obejcts is nopt null.

if (AStarGreedy[i].SolderesOnTable[m] != null)

{

//Found of Maximum on Depth solders on table items.

if (AStarGreedy[i].SolderesOnTable[m].SoldierThinking[0].MaxHuristic(ref jIndex[0], 1, ref Less[0], Order \* -1))

{

iIndex[0] = i;

mIndex[0] = m;

Kin[0] = 1;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Elephant On Table Count.

for (int m = ElefantOnTableCount(ref AStarGreedy[i].ElephantOnTable, true, AStarGreedy[i].ElefantHigh); m < ElefantOnTableCount(ref AStarGreedy[i].ElephantOnTable, false, AStarGreedy[i].ElefantHigh); m++)

{

//For All Elephant in Depth Count.

if (AStarGreedy[i].ElephantOnTable[m] != null)

{

//Found of Maxmimum in Branch.

if (AStarGreedy[i].ElephantOnTable[m].ElefantThinking[0].MaxHuristic(ref jIndex[1], 2, ref Less[1], Order \* -1))

{

iIndex[1] = i;

mIndex[1] = m;

Kin[1] = 2;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Hourse on Table Count.

for (int m = HourseOnTableCount(ref AStarGreedy[i].HoursesOnTable, true, AStarGreedy[i].HourseHight); m < HourseOnTableCount(ref AStarGreedy[i].HoursesOnTable, false, AStarGreedy[i].HourseHight); m++)

{

//When is HourseOn Table Depth Object is Not NULL.

if (AStarGreedy[i].HoursesOnTable[m] != null)

{

//Forund of Maximum on on Branch.

if (AStarGreedy[i].HoursesOnTable[m].HourseThinking[0].MaxHuristic(ref jIndex[2], 3, ref Less[2], Order \* -1))

{

iIndex[2] = i;

mIndex[2] = m;

Kin[2] = 3;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Castles on table Count.

for (int m = CastleOnTableCount(ref AStarGreedy[i].CastlesOnTable, true, AStarGreedy[i].CastleHigh); m < CastleOnTableCount(ref AStarGreedy[i].CastlesOnTable, false, AStarGreedy[i].CastleHigh); m++)

{

//When Depth Objects of Hourse Table is Not NULL.

if (AStarGreedy[i].CastlesOnTable[m] != null)

{

//Found of Maximum Castles Branch.

if (AStarGreedy[i].CastlesOnTable[m].CastleThinking[0].MaxHuristic(ref jIndex[3], 4, ref Less[3], Order \* -1))

{

iIndex[3] = i;

mIndex[3] = m;

Kin[3] = 4;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All Minsiter on table count.

for (int m = MinisterOnTableCount(ref AStarGreedy[i].MinisterOnTable, true, AStarGreedy[i].MinisterHigh); m < MinisterOnTableCount(ref AStarGreedy[i].MinisterOnTable, false, AStarGreedy[i].MinisterHigh); m++)

{

//When Minster of Depth is Not Null.

if (AStarGreedy[i].MinisterOnTable[m] != null)

{

//Found of Maximum Minster on table Branches.

if (AStarGreedy[i].MinisterOnTable[m].MinisterThinking[0].MaxHuristic(ref jIndex[4], 5, ref Less[4], Order \* -1))

{

iIndex[4] = i;

mIndex[4] = m;

Kin[4] = 5;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

//For All King on table Count.

for (int m = KingOnTableCount(ref AStarGreedy[i].KingOnTable, true, AStarGreedy[i].KingHigh); m < KingOnTableCount(ref AStarGreedy[i].KingOnTable, false, AStarGreedy[i].KingHigh); m++)

{

//When Minster of Depth is Not Null.

if (AStarGreedy[i].KingOnTable[m] != null)

{

//When Depth Object of King Table is Not NULL.

if (AStarGreedy[i].KingOnTable[m].KingThinking[0].MaxHuristic(ref jIndex[5], 1, ref Less[5], Order \* -1))

{

iIndex[5] = i;

mIndex[5] = m;

Kin[5] = 6;

//Huristic = Less;

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

else

CodeClass.SaveByCode(2, callStack.GetFileLineNumber(), callStack.GetFileName());

}

}

}

int IJ = -1;

//if (Order == AllDraw.OrderPlate)

//IJ = MaxOfSixHuristic(Less) + 1;

//else

//IJ = MinOfSixHuristic(Less) + 1;

//Calculate Huristic of Current Node.

//When Sodleris Kind.

if (System.Math.Abs(Kind) == 1 && HuristicListSolder.Count > 0)

{

Huristic += HuristicListSolder[j][0] +

HuristicListSolder[j][1] +

HuristicListSolder[j][2] +

HuristicListSolder[j][3] +

HuristicListSolder[j][4] +

HuristicListSolder[j][5] +

HuristicListSolder[j][6] +

HuristicListSolder[j][7] +

HuristicListSolder[j][8] +

HuristicListSolder[j][9];

Object O1 = new Object();

lock (O1)

{

ActionsString += " " + Alphabet(Row) + Number(Column) + Alphabet(RowColumnSoldier[j][0]) + Number(RowColumnSoldier[j][1]);

if (Order == 1)

ActionsString = "\r\nHuristic Soldier AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

else

ActionsString = "\r\nHuristic Soldier AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

}

}

else

//When Elephant Kind.

if (System.Math.Abs(Kind) == 2 && HuristicListElefant.Count > 0)

{

Huristic += HuristicListElefant[j][0] +

HuristicListElefant[j][1] +

HuristicListElefant[j][2] +

HuristicListElefant[j][3] +

HuristicListElefant[j][4] +

HuristicListElefant[j][5] +

HuristicListElefant[j][6] +

HuristicListElefant[j][7] +

HuristicListElefant[j][8] +

HuristicListElefant[j][9];

Object O1 = new Object();

lock (O1)

{

ActionsString += " " + Alphabet(Row) + Number(Column) + Alphabet(RowColumnElefant[j][0]) + Number(RowColumnElefant[j][1]);

if (Order == 1)

ActionsString = "\r\nHuristic Elephant AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

else

ActionsString = "\r\nHuristic Elephant AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

}

}

else

//When Hourse Kind.

if (System.Math.Abs(Kind) == 3 && HuristicListHourse.Count > 0)

{

Huristic += HuristicListHourse[j][0] +

HuristicListHourse[j][1] +

HuristicListHourse[j][2] +

HuristicListHourse[j][3] +

HuristicListHourse[j][4] +

HuristicListHourse[j][5] +

HuristicListHourse[j][6] +

HuristicListHourse[j][7] +

HuristicListHourse[j][8] +

HuristicListHourse[j][9];

Object O1 = new Object();

lock (O1)

{

ActionsString += " " + Alphabet(Row) + Number(Column) + Alphabet(RowColumnHourse[j][0]) + Number(RowColumnHourse[j][1]);

if (Order == 1)

ActionsString = "\r\nHuristic Hourse AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

else

ActionsString = "\r\nHuristic Hourse AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

}

}

else

//When Castles Kind.

if (System.Math.Abs(Kind) == 4 && HuristicListCastle.Count > 0)

{

Huristic += HuristicListCastle[j][0] +

HuristicListCastle[j][1] +

HuristicListCastle[j][2] +

HuristicListCastle[j][3] +

HuristicListCastle[j][4] +

HuristicListCastle[j][5] +

HuristicListCastle[j][6] +

HuristicListCastle[j][7] +

HuristicListCastle[j][8] +

HuristicListCastle[j][9];

Object O1 = new Object();

lock (O1)

{

ActionsString += " " + Alphabet(Row) + Number(Column) + Alphabet(RowColumnCastle[j][0]) + Number(RowColumnCastle[j][1]);

if (Order == 1)

ActionsString = "\r\nHuristic Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

else

ActionsString = "\r\nHuristic Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

}

}

else

//When Minister Kind.

if (System.Math.Abs(Kind) == 5 && HuristicListMinister.Count > 0)

{

Huristic += HuristicListMinister[j][0] +

HuristicListMinister[j][1] +

HuristicListMinister[j][2] +

HuristicListMinister[j][3] +

HuristicListMinister[j][4] +

HuristicListMinister[j][5] +

HuristicListMinister[j][6] +

HuristicListMinister[j][7] +

HuristicListMinister[j][8] +

HuristicListMinister[j][9];

Object O1 = new Object();

lock (O1)

{

ActionsString += " " + Alphabet(Row) + Number(Column) + Alphabet(RowColumnMinister[j][0]) + Number(RowColumnMinister[j][1]);

if (Order == 1)

ActionsString = "\r\nHuristic Minister AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

else

ActionsString = "\r\nHuristic Minister AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

}

}

else

//When King Kind.

if (System.Math.Abs(Kind) == 6 && HuristicListKing.Count > 0)

{

Huristic += HuristicListKing[j][0] +

HuristicListKing[j][1] +

HuristicListKing[j][2] +

HuristicListKing[j][3] +

HuristicListKing[j][4] +

HuristicListKing[j][5] +

HuristicListKing[j][6] +

HuristicListKing[j][7] +

HuristicListKing[j][8] +

HuristicListKing[j][9];

Object O1 = new Object();

lock (O1)

{

ActionsString += " " + Alphabet(Row) + Number(Column) + Alphabet(RowColumnKing[j][0]) + Number(RowColumnKing[j][1]);

if (Order == 1)

ActionsString = "\r\nHuristic King AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

else

ActionsString = "\r\nHuristic King AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at Level " + iAstarGready.ToString() + " By Action String " + ActionsString;

}

}

//When Kind Found.

//if (IJ != -1)

{

//Reapeate for Solders.

if (//IJ == 1 &&

AStarGreedy.Count > 0 && iIndex[0] != -1)

Huristic += AStarGreedy[iIndex[0]].SolderesOnTable[mIndex[0]].SoldierThinking[0].ReturnHuristicCalculartor(iAstarGready, ii, jIndex[0], Order \* -1);

//Repeate for Elephant.

if (//IJ == 2 &&

AStarGreedy.Count > 0 && iIndex[1] != -1)

Huristic += AStarGreedy[iIndex[1]].ElephantOnTable[mIndex[1]].ElefantThinking[0].ReturnHuristicCalculartor(iAstarGready, ii, jIndex[1], Order \* -1);

//Repeate for Hourse.

if (//IJ == 3 &&

AStarGreedy.Count > 0 && iIndex[2] != -1)

Huristic += AStarGreedy[iIndex[2]].HoursesOnTable[mIndex[2]].HourseThinking[0].ReturnHuristicCalculartor(iAstarGready, ii, jIndex[2], Order \* -1);

//Repeate for Castles.

if (//IJ == 4 &&

AStarGreedy.Count > 0 && iIndex[3] != -1)

Huristic += AStarGreedy[iIndex[3]].CastlesOnTable[mIndex[3]].CastleThinking[0].ReturnHuristicCalculartor(iAstarGready, ii, jIndex[3], Order \* -1);

//Repeate for Minstre.

if (//IJ == 5 &&

AStarGreedy.Count > 0 && iIndex[4] != -1)

Huristic += AStarGreedy[iIndex[4]].MinisterOnTable[mIndex[4]].MinisterThinking[0].ReturnHuristicCalculartor(iAstarGready, ii, jIndex[4], Order \* -1);

//Repeate for King.

if (//IJ == 6 &&

AStarGreedy.Count > 0 && iIndex[5] != -1)

Huristic += AStarGreedy[iIndex[5]].KingOnTable[mIndex[5]].KingThinking[0].ReturnHuristicCalculartor(iAstarGready, ii, jIndex[5], Order \* -1);

}

}

else

{

//When Solder Kind.

if (System.Math.Abs(Kind) == 1 && HuristicListSolder.Count > 0)

{

Huristic += HuristicListSolder[j][0] +

HuristicListSolder[j][1] +

HuristicListSolder[j][2] +

HuristicListSolder[j][3] +

HuristicListSolder[j][4] +

HuristicListSolder[j][5] +

HuristicListSolder[j][6] +

HuristicListSolder[j][7];

}

else

//When Elephant Kind.

if (System.Math.Abs(Kind) == 2 && HuristicListElefant.Count > 0)

{

Huristic += HuristicListElefant[j][0] +

HuristicListElefant[j][1] +

HuristicListElefant[j][2] +

HuristicListElefant[j][3] +

HuristicListElefant[j][4] +

HuristicListElefant[j][5] +

HuristicListElefant[j][6] +

HuristicListElefant[j][7];

}

else

//When Hourse Kind.

if (System.Math.Abs(Kind) == 3 && HuristicListHourse.Count > 0)

{

Huristic += HuristicListHourse[j][0] +

HuristicListHourse[j][1] +

HuristicListHourse[j][2] +

HuristicListHourse[j][3] +

HuristicListHourse[j][4] +

HuristicListHourse[j][5] +

HuristicListHourse[j][6] +

HuristicListHourse[j][7];

}

else

//When Castles Kind.

if (System.Math.Abs(Kind) == 4 && HuristicListCastle.Count > 0)

{

Huristic += HuristicListCastle[j][0] +

HuristicListCastle[j][1] +

HuristicListCastle[j][2] +

HuristicListCastle[j][3] +

HuristicListCastle[j][4] +

HuristicListCastle[j][5] +

HuristicListCastle[j][6] +

HuristicListCastle[j][7];

}

else

//When Minister Kind.

if (System.Math.Abs(Kind) == 5 && HuristicListMinister.Count > 0)

{

Huristic += HuristicListMinister[j][0] +

HuristicListMinister[j][1] +

HuristicListMinister[j][2] +

HuristicListMinister[j][3] +

HuristicListMinister[j][4] +

HuristicListMinister[j][5] +

HuristicListMinister[j][6] +

HuristicListMinister[j][7];

}

else

//When King Kind.

if (System.Math.Abs(Kind) == 6 && HuristicListKing.Count > 0)

{

Huristic += HuristicListKing[j][0] +

HuristicListKing[j][1] +

HuristicListKing[j][2] +

HuristicListKing[j][3] +

HuristicListKing[j][4] +

HuristicListKing[j][5] +

HuristicListKing[j][6] +

HuristicListKing[j][7];

}

}

Order = DummyOrder;

return Huristic;

}

}

//Returrn of Hurestic Tree.QC\_Ok.

//Scope of Every Objects Movments.

bool Scop(int i, int j, int ii, int jj, int Kind)

{

Object O = new Object();

lock (O)

{

if (i == ii && j == jj)

return false;

//Scope of index out of range.

if (i < 0)

return false;

if (j < 0)

return false;

if (ii < 0)

return false;

if (jj < 0)

return false;

if (i > 7)

return false;

if (j > 7)

return false;

if (ii > 7)

return false;

if (jj > 7)

return false;

bool Validity = false;

//Scope on estimation on rule movment.

if (Kind == 1)//Sodier

{

if (ArrangmentsChanged)

{

if (Order == 1)

{

if (j <= jj)

return false;

}

else

{

if (j >= jj)

return false;

}

}

else if (!ArrangmentsChanged)

{

if (Order == -1)

{

if (j <= jj)

return false;

}

else

{

if (j >= jj)

return false;

}

}

if (System.Math.Abs(i - ii) <= 2 && System.Math.Abs(j - jj) <= 2)

Validity = true;

}

else

if (Kind == 2)//Elephant

{

if (System.Math.Abs(i - ii) == System.Math.Abs(j - jj))

{

Validity = true;

}

}

else

if (Kind == 3)//Hourse

{

if (System.Math.Abs(i - ii) == 1 && System.Math.Abs(j - jj) == 2)

Validity = true;

if (System.Math.Abs(i - ii) == 2 && System.Math.Abs(j - jj) == 1)

Validity = true;

}

else

if (Kind == 4)//Castle

{

if ((i == ii && j != jj) || (i != ii && j == jj))

Validity = true;

}

else

if (Kind == 5)//Minister

{

if (((i == ii && j != jj) || (i != ii && j == jj)) || System.Math.Abs(i - ii) == System.Math.Abs(j - jj))

Validity = true;

}

else

if (Kind == 6)//King

{

if (System.Math.Abs(i - ii) <= 1 && System.Math.Abs(j - jj) <= 1)

Validity = true;

}

return Validity;

}

}

//Calculate Maximum of Six Max Huristic of Six Kind Objects.

int MaxOfSixHuristic(double[] Less)

{

Object O = new Object();

lock (O)

{

int Value = -1;

double Les = Double.MinValue;

for (int i = 0; i < 6; i++)

{

if (Less[i] > Les)

{

Les = Less[i];

Value = i;

}

}

return Value;

}

}

//Calculate Minimum of Six Min Huristic of Six Kind Objects.note the enemy Huristic are negative.

int MinOfSixHuristic(double[] Less)

{

Object O = new Object();

lock (O)

{

int Value = -1;

double Les = Double.MaxValue;

for (int i = 0; i < 6; i++)

{

if (Less[i] < Les)

{

Les = Less[i];

Value = i;

}

}

return Value;

}

}

void KingThinkingChess(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object O = new Object();

lock (O)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///When There is Movments.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj)).Rules(ii, jj, i, j, color, TableS[ii, jj]))

{

try

{

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

//KingValue = ObjectValueCalculator(TableS, Order, ii, jj);

///Add Table to List of Private.

HitNumberKing.Add(TableS[i, j]);

Object O1 = new Object();

lock (O1)

{

ThinkingRun = true;

}

///Predict Huristic.

Object A = new object();

lock (A)

{

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A1 = new object();

lock (A1)

{

NumbersOfAllNode++;

}

///Calculate Huristic and Add to List Speciifically and Cal Syntax.

///ActionsString of Movements.

int Killed = 0;

Object A2 = new object();

lock (A2)

{

Killed = TableS[i, j];

TableS[i, j] = TableS[ii, jj];

TableS[ii, jj] = 0;

}

//RemoveAtList(Kind);

//Learning Autamata

Object A3 = new object();

lock (A3)

{

PenaltyMechanisam(Killed, false, 6, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

}

Object A4 = new object();

lock (A4)

{

int RowB = 0, ColumnB = 0;

int RowG = 0, ColumnG = 0;

ChessRules AAA = new ChessRules(0, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, 6, TableS, Order, ii, jj);

AAA.FindBrownKing(TableS, ref RowB, ref ColumnB);

AAA.FindGrayKing(TableS, ref RowG, ref ColumnG);

//Gray Order.

//Illegal King Foundation.

if (System.Math.Abs(RowB - RowG) <= 1 && System.Math.Abs(ColumnB - ColumnG) <= 1)

{

NumberOfPenalties++;

Current.LearningAlgorithmPenalty();

PenaltyRegardListKing.Add(Current);

}

}

//Consideration of Prevention from going Check State by self order.

///Store of Indexes Changes and Table in specific List.

Object A5 = new object();

lock (A5)

{

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnKing.Add(AS);

//Index++;

TableListKing.Add(CloneATable(TableS)); ;

IndexKing++;

}

Object A6 = new object();

lock (A6)

{

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, 0, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A7 = new object();

lock (A7)

{

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListKing.Add(Hu);

}

Object O2 = new Object();

lock (O2)

{

OutPutAction = " " + Alphabet(ii) + Number(jj) + Alphabet(i) + Number(j);

if (Order == 1)

AllDraw.OutPut = "\r\nThinking King AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

else

AllDraw.OutPut = "\r\nThinking King AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

ThinkingLevel++;

ThinkingAtRun = false;

}

}

catch (Exception t)

{

Log(t);

ThinkingAtRun = false;

}

}

}

}

void MinisterThinkingChess(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object O11 = new Object();

lock (O11)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///When There is Movments.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj)).Rules(ii, jj, i, j, color, TableS[ii, jj]))

{

try

{

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

///Add Table to List of Private.

HitNumberMinister.Add(TableS[i, j]);

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

///Predict Huristic.

Object A = new object();

lock (A)

{

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A1 = new object();

lock (A1)

{

NumbersOfAllNode++;

}

int Killed = 0;

Object A2 = new object();

lock (A2)

{

Killed = TableS[i, j];

TableS[i, j] = TableS[ii, jj];

TableS[ii, jj] = 0;

}

Object A3 = new object();

lock (A3)

{

PenaltyMechanisam(Killed, false, 5, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

}

///Store of Indexes Changes and Table in specific List.

Object A4 = new object();

lock (A4)

{

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnMinister.Add(AS);

//RowColumn[Index, 0] = i;

//RowColumn[Index, 1] = j;

//Index++;

TableListMinister.Add(CloneATable(TableS)); ;

IndexMinister++;

}

Object A5 = new object();

lock (A5)

{

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, 0, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

///Calculate Huristic and Add to List Speciifically and Cal Syntax.

Object A6 = new object();

lock (A6)

{

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListMinister.Add(Hu);

}

Object O1 = new Object();

lock (O1)

{

OutPutAction = " " + Alphabet(ii) + Number(jj) + Alphabet(i) + Number(j);

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Minister AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

else

AllDraw.OutPut = "\r\nThinking Minister AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

ThinkingLevel++;

ThinkingAtRun = false;

}

}

catch (Exception t)

{

Log(t);

ThinkingAtRun = false;

}

}

}

}

bool IsPrviousMovemntIsDangrousForCurrent(int[,] TableS, int Order)

{

Object O = new Object();

lock (O)

{

bool Dang = false;

int BREAK = 0;

Object O1 = new Object();

lock (O1)

{

//.Current

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

BREAK = 0;

if (Order == 1 && TableS[i, j] <= 0)

continue;

else

if (Order == -1 && TableS[i, j] >= 0)

continue;

//Enemy

for (int ii = 0; ii < 8; ii++)

{

for (int jj = 0; jj < 8; jj++)

{

BREAK = 0;

if (Order == 1 && TableS[ii, jj] >= 0)

continue;

else

if (Order == -1 && TableS[ii, jj] <= 0)

continue;

Color a = Color.Gray;

if (Order \* -1 == -1)

a = Color.Brown;

if (Attack(TableS, ii, jj, i, j, a, Order \* -1))

{

BREAK = 1;

//Current

for (int iii = 0; iii < 8; iii++)

{

for (int jjj = 0; jjj < 8; jjj++)

{

BREAK = 0;

if (Order == 1 && TableS[iii, jjj] <= 0)

continue;

else

if (Order == -1 && TableS[iii, jjj] >= 0)

continue;

a = Color.Gray;

if (Order == -1)

a = Color.Brown;

if (Support(TableS, iii, jjj, i, j, a, Order))

{

BREAK = 2;

break;

}

}

if (BREAK == 2)

break;

}

}

if (BREAK == 1)

break;

}

if (BREAK == 1)

break;

}

if (BREAK == 1)

break;

}

if (BREAK == 1)

break;

}

if (BREAK == 1)

Dang = true;

}

return Dang;

}

}

//When There is not valuable Object in List Greater than Target Self Object return true.

bool IsObjectValaubleObjectSelf(int i, int j, int Object, ref List<int[]> ValuableSelfSupported)

{

Object O = new Object();

lock (O)

{

bool Is = true;

for (int k = 0; k < ValuableSelfSupported.Count; k++)

{

if (ValuableSelfSupported[k][0] > 0 && Object > 0)

{

if (System.Math.Abs(ValuableSelfSupported[k][0]) > System.Math.Abs(Object))

Is = false;

}

else

if (ValuableSelfSupported[k][0] < 0 && Object < 0)

{

if (System.Math.Abs(ValuableSelfSupported[k][0]) > System.Math.Abs(Object))

Is = false;

}

if (Is == false)

break;

}

return Is;

}

}

bool IsObjectValaubleObjectEnemy(int i, int j, int Object, ref List<int[]> ValuableEnemyNotSupported)

{

Object O = new Object();

lock (O)

{

bool Is = true;

for (int k = 0; k < ValuableEnemyNotSupported.Count; k++)

if (System.Math.Abs(ValuableEnemyNotSupported[k][0]) < System.Math.Abs(Object))

{

Is = false;

break;

}

return Is;

}

}

bool[] SomeLearningVarsCalculator(int[,] TableS, int ik, int jk, int iik, int jjk)

{

Object O22 = new Object();

lock (O22)

{

int AttackCount = 0;

bool[] LearningV = new bool[3];

Object O = new Object();

lock (O)

{

Parallel.For(0, 8, i =>

{

if ((LearningV[0] || LearningV[1] || LearningV[2]))

return;

Parallel.For(0, 8, j =>

{

if ((LearningV[0] || LearningV[1] || LearningV[2]))

return;

Parallel.For(0, 8, iii =>

{

if ((LearningV[0] || LearningV[1] || LearningV[2]))

return;

Parallel.For(0, 8, jjj =>

{

if ((LearningV[0] || LearningV[1] || LearningV[2]))

return;

Parallel.Invoke(() =>

{

Object O1 = new Object();

lock (O1)

{

if (!(LearningV[0] || LearningV[1] || LearningV[2]))

LearningV[0] = LearningV[0] || InAttackSelfThatNotSupportedAll(TableS, Order, color, i, j, iii, jjj, ik, jk, iik, jjk);

}

}, () =>

{

Object O1 = new Object();

lock (O1)

{

if ((LearningV[0] || LearningV[1] || LearningV[2]))

return;

if (AttackCount <= 1 && (!(LearningV[0] || LearningV[1] || LearningV[2])))

AttackCount = AttackCount + IsNotSafeToMoveAenemeyToAttackMoreThanTowObject(AttackCount, TableS, Order, i, j, iii, jjj//, ii, jj, iiii, jjjj

);

else

if (!(LearningV[0] || LearningV[1] || LearningV[2]))

LearningV[1] = true;

}

}, () =>

{

Object O1 = new Object();

lock (O1)

{

if (!(LearningV[0] || LearningV[1] || LearningV[2]))

LearningV[2] = LearningV[2] || IsGardForCurrentMovmentsAndIsNotMovable(TableS, Order, color, i, j, iii, jjj//, ii, jj, iiii, jjjj

);

}

});

});

});

});

});

}

return LearningV;

}

}

bool[] CalculateLearningVars(int Killed, int[,] TableS, int i, int j, int ii, int jj)

{

Object O = new Object();

lock (O)

{

bool[] LearningV = new bool[14];

bool IsCurrentCanGardHighPriorityEne = new bool();

bool IsNextMovemntIsCheckOrCheckMateForCurrent = new bool();

bool IsDangerous = new bool();

bool CanKillerAnUnSupportedEnemy = new bool();

bool InDangrousUnSupported = new bool();

bool Support = new bool();

bool IsNextMovemntIsCheckOrCheckMateForEnemy = new bool();

bool IsPrviousMovemntIsDangrousForCurr = new bool();

bool PDo = new bool();

bool RDo = new bool();

bool SelfNotSupported = new bool();

bool EnemyNotSupported = new bool();

bool IsGardForCurrentMovmentsAndIsNotMova = new bool();

bool IsNotSafeToMoveAenemeyToAttackMoreThanTowObj = new bool();

bool P = new bool();

bool R = new bool();

bool IsTowValuableObjectEnemy = false;

List<int[]> ValuableEnemyNotSupported = new List<int[]>();

List<int[]> ValuableSelfSupported = new List<int[]>();

//When true must penalty

Object O11 = new Object();

lock (O11)

{

IsPrviousMovemntIsDangrousForCurr = IsPrviousMovemntIsDangrousForCurrent(TableS, Order);

//when true must penalty

if (!IsPrviousMovemntIsDangrousForCurr)

SelfNotSupported = InAttackSelfThatNotSupported(TableS, Order, color, i, j, ii, jj);

//when true must regard

Support = false;

int SelfChackedMateDepth = 0;

int EnemyCheckedMateDepth = 0;

IsDangerous = false;//No Needed.

//For All Current

bool[] LearningVars = SomeLearningVarsCalculator(TableS, ii, jj, i, j);

Object O4 = new Object();

lock (O4)

{

SelfNotSupported = LearningVars[0];

IsNotSafeToMoveAenemeyToAttackMoreThanTowObj = LearningVars[1];

IsGardForCurrentMovmentsAndIsNotMova = LearningVars[2];

}

if ((!IsNextMovemntIsCheckOrCheckMateForCurrent) && (!SelfNotSupported) && (!IsPrviousMovemntIsDangrousForCurr) && (!IsGardForCurrentMovmentsAndIsNotMova) && (!IsNotSafeToMoveAenemeyToAttackMoreThanTowObj) && (!IsDangerous))

{

int[] Is = new int[4];

Is[0] = 0;

Is[1] = 0;

Is[2] = 0;

Is[3] = 0;

if (CurrentAStarGredyMax == 0)

{

int Depth = new int();

Depth = 0;

int[,] Tab = CloneATable(TableS);

int Ord = Order;

Color a = color;

int Ord1 = AllDraw.OrderPlate;

int Ord2 = AllDraw.OrderPlate \* -1;

//when is true must penalty(Superposition)

Is = IsNextMovmentIsCheckOrCheckMateForCurrentMovment(Tab, Ord, a, Depth, Ord1, Ord2, true);

//A

}

Object OO1 = new Object();

lock (OO1)

{

if (Is[0] >= 1)

IsNextMovemntIsCheckOrCheckMateForCurrent = true;

else

IsNextMovemntIsCheckOrCheckMateForCurrent = false;

if (Is[2] >= 1)

IsNextMovemntIsCheckOrCheckMateForEnemy = true;

else

IsNextMovemntIsCheckOrCheckMateForEnemy = false;

SelfChackedMateDepth = Is[1];

EnemyCheckedMateDepth = Is[3];

}

}

//Order Depth Consideration Constraint.

if (IsNextMovemntIsCheckOrCheckMateForCurrent && IsNextMovemntIsCheckOrCheckMateForEnemy)

{

Object OO2 = new Object();

lock (OO2)

{

if (SelfChackedMateDepth < EnemyCheckedMateDepth)

IsNextMovemntIsCheckOrCheckMateForEnemy = false;

else

if (SelfChackedMateDepth > EnemyCheckedMateDepth)

IsNextMovemntIsCheckOrCheckMateForCurrent = false;

}

}

if ((!IsNextMovemntIsCheckOrCheckMateForCurrent) && (!SelfNotSupported) && (!IsPrviousMovemntIsDangrousForCurr) && (!IsGardForCurrentMovmentsAndIsNotMova) && (!IsNotSafeToMoveAenemeyToAttackMoreThanTowObj) && (!IsDangerous))

{

EnemyNotSupported = InAttackEnemyThatIsNotSupportedAll(IsTowValuableObjectEnemy, TableS, Order, color, i, j, ii, jj, ref ValuableEnemyNotSupported);

}

if ((!IsNextMovemntIsCheckOrCheckMateForCurrent) && (!SelfNotSupported) && (!IsPrviousMovemntIsDangrousForCurr) && (!IsGardForCurrentMovmentsAndIsNotMova) && (!IsNotSafeToMoveAenemeyToAttackMoreThanTowObj) && (!EnemyNotSupported) && (!IsDangerous))

EnemyNotSupported = InAttackEnemyThatIsNotSupported(Killed, TableS, Order, color, i, j, ii, jj);

if ((!IsNextMovemntIsCheckOrCheckMateForCurrent) && (!SelfNotSupported) && (!IsPrviousMovemntIsDangrousForCurr) && (!IsGardForCurrentMovmentsAndIsNotMova) && (!IsNotSafeToMoveAenemeyToAttackMoreThanTowObj) && (!EnemyNotSupported) && (!IsDangerous))

{

EnemyNotSupported = InAttackEnemyThatIsNotSupportedAll(IsTowValuableObjectEnemy, TableS, Order, color, i, j, ii, jj, ref ValuableEnemyNotSupported);

}

if (CurrentAStarGredyMax == 0 && (!IsNextMovemntIsCheckOrCheckMateForCurrent) && (!SelfNotSupported) && (!IsPrviousMovemntIsDangrousForCurr) && (!IsGardForCurrentMovmentsAndIsNotMova) && (!IsNotSafeToMoveAenemeyToAttackMoreThanTowObj) && (!EnemyNotSupported) && (!IsDangerous))

{

//when is true must regard.

IsCurrentCanGardHighPriorityEne = IsCurrentCanGardHighPriorityEnemy(0, TableS, Order, color, i, j, ii, jj, Order);

}

if (SelfNotSupported || IsNextMovemntIsCheckOrCheckMateForCurrent || IsPrviousMovemntIsDangrousForCurr || IsGardForCurrentMovmentsAndIsNotMova && IsDangerous)

{

IsCurrentCanGardHighPriorityEne = false;

EnemyNotSupported = false;

IsNextMovemntIsCheckOrCheckMateForEnemy = false;

}

Object OO = new Object();

lock (OO)

{

LearningV[0] = IsCurrentCanGardHighPriorityEne;

LearningV[1] = IsNextMovemntIsCheckOrCheckMateForCurrent;

LearningV[2] = IsDangerous;

LearningV[3] = CanKillerAnUnSupportedEnemy;

LearningV[4] = InDangrousUnSupported;

LearningV[5] = Support;

LearningV[6] = IsNextMovemntIsCheckOrCheckMateForEnemy;

LearningV[7] = IsPrviousMovemntIsDangrousForCurr;

LearningV[8] = PDo;

LearningV[9] = RDo;

LearningV[10] = SelfNotSupported;

LearningV[11] = EnemyNotSupported;

LearningV[12] = IsGardForCurrentMovmentsAndIsNotMova;

LearningV[13] = IsNotSafeToMoveAenemeyToAttackMoreThanTowObj;

if (IsNextMovemntIsCheckOrCheckMateForCurrent)

IgnoreFromCheckandMateHuristic = true;

CanKillerAnUnSupportedEnemy = Support || EnemyNotSupported || IsCurrentCanGardHighPriorityEne || IsNextMovemntIsCheckOrCheckMateForEnemy || IsNextMovemntIsCheckOrCheckMateForCurrent;//B

P = IsNotSafeToMoveAenemeyToAttackMoreThanTowObj || IsGardForCurrentMovmentsAndIsNotMova || IsPrviousMovemntIsDangrousForCurr || SelfNotSupported || IsDangerous || IsCurrentCanGardHighPriorityEne || IsNextMovemntIsCheckOrCheckMateForEnemy || IsNextMovemntIsCheckOrCheckMateForCurrent;//C

R = CanKillerAnUnSupportedEnemy;//D

InDangrousUnSupported = P && (!R);

PDo = P & (!R);

//B+C

RDo = R && (!P);

}

}

return LearningV;

}

}

void CastlesThinkingChess(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle

)

{

Object O22 = new Object();

lock (O22)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///When There is Movments.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj)).Rules(ii, jj, i, j, color, TableS[ii, jj]))

{

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

///Add Table to List of Private.

HitNumberCastle.Add(TableS[i, j]);

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

///Predict Huristic.

Object A = new object();

lock (A)

{

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A1 = new object();

lock (A1)

{

NumbersOfAllNode++;

}

int Killed = 0;

Object A2 = new object();

lock (A2)

{

Killed = TableS[i, j];

TableS[i, j] = TableS[ii, jj];

TableS[ii, jj] = 0;

}

Object A3 = new object();

lock (A3)

{

PenaltyMechanisam(Killed, false, 4, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

}

///Store of Indexes Changes and Table in specific List.

Object A4 = new object();

lock (A4)

{

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnCastle.Add(AS);

TableListCastle.Add(CloneATable(TableS)); ;

IndexCastle++;

}

Object A5 = new object();

lock (A5)

{

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, Killed, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

///Calculate Huristic and Add to List Speciifically and Cal Syntax.

Object A6 = new object();

lock (A6)

{

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListCastle.Add(Hu);

}

Object O3 = new Object();

lock (O3)

{

OutPutAction = " " + Alphabet(ii) + Number(jj) + Alphabet(i) + Number(j);

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

else

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

ThinkingLevel++;

ThinkingAtRun = false;

}

}

}

}

void HourseThinkingChess(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object OO = new Object();

lock (OO)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///When There is Movments.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj)).Rules(ii, jj, i, j, color, TableS[ii, jj]))

{

try

{

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

///Add Table to List of Private.

HitNumberHourse.Add(TableS[i, j]);

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

///Predict Huristic.

Object A = new object();

lock (A)

{

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A1 = new object();

lock (A1)

{

NumbersOfAllNode++;

}

int Killed = 0;

Object A2 = new object();

lock (A2)

{

Killed = TableS[i, j];

TableS[i, j] = TableS[ii, jj];

TableS[ii, jj] = 0;

}

Object A3 = new object();

lock (A3)

{

PenaltyMechanisam(Killed, false, 3, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

}

///Store of Indexes Changes and Table in specific List.

Object A4 = new object();

lock (A4)

{

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnHourse.Add(AS);

//RowColumn[Index, 0] = i;

//RowColumn[Index, 1] = j;

//Index++;

TableListHourse.Add(CloneATable(TableS)); ;

IndexHourse++;

}

///Wehn Predict of Operation Do operate a Predict of this movments.

Object A5 = new object();

lock (A5)

{

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, Killed, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

//Calculate Huristic and Add to List and Cal Syntax.

Object A6 = new object();

lock (A6)

{

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListHourse.Add(Hu);

}

Object O4 = new Object();

lock (O4)

{

OutPutAction = " " + Alphabet(ii) + Number(jj) + Alphabet(i) + Number(j);

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Hourse AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

else

AllDraw.OutPut = "\r\nThinking Hourse AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

ThinkingLevel++;

ThinkingAtRun = false;

}

}

catch (Exception t)

{

ThinkingAtRun = false;

Log(t);

}

}

}

}

void ElephantThinkingChess(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object OO = new Object();

lock (OO)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///When There is Movments.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj)).Rules(ii, jj, i, j, color, TableS[ii, jj]))

{

try

{

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

///Add Table to List of Private.

HitNumberElefant.Add(TableS[i, j]);

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

Object A = new object();

lock (A)

{

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A1 = new object();

lock (A1)

{

NumbersOfAllNode++;

}

Object A2 = new object();

int Killed = 0;

lock (A2)

{

Killed = TableS[i, j];

TableS[i, j] = TableS[ii, jj];

TableS[ii, jj] = 0;

}

Object A3 = new object();

lock (A3)

{

PenaltyMechanisam(Killed, false, 2, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

}

///Store of Indexes Changes and Table in specific List.

Object A4 = new object();

lock (A4)

{

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnElefant.Add(AS);

TableListElefant.Add(CloneATable(TableS)); ;

IndexElefant++;

}

Object A5 = new object();

lock (A5)

{

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, Killed, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A6 = new object();

lock (A6)

{

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListElefant.Add(Hu);

}

Object O5 = new Object();

lock (O5)

{

OutPutAction = " " + Alphabet(ii) + Number(jj) + Alphabet(i) + Number(j);

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Elephant AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

else

AllDraw.OutPut = "\r\nThinking Elephant AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

ThinkingLevel++;

ThinkingAtRun = false;

}

}

catch (Exception t)

{

ThinkingAtRun = false;

Log(t);

}

}

}

}

bool EqualitTow(bool PenRegStrore, int kind)

{

Object O = new Object();

lock (O)

{

bool Equality = false;

if (kind == 1 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListSolder.Count == TableListSolder.Count)

Equality = true;

else

if (kind == 2 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListElefant.Count == TableListElefant.Count)

Equality = true;

else

if (kind == 3 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListHourse.Count == TableListHourse.Count)

Equality = true;

else

if (kind == 4 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListCastle.Count == TableListCastle.Count)

Equality = true;

else

if (kind == 5 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListMinister.Count == TableListMinister.Count)

Equality = true;

else

if (kind == 6 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListKing.Count == TableListKing.Count)

Equality = true;

return Equality;

}

}

bool EqualitOne(QuantumAtamata Current, int kind)

{

Object O = new Object();

lock (O)

{

bool Equality = false;

if (kind == 1 && Current.IsPenaltyAction() != 0 && UsePenaltyRegardMechnisamT && PenaltyRegardListSolder.Count == TableListSolder.Count)

Equality = true;

else

if (kind == 2 && Current.IsPenaltyAction() != 0 && UsePenaltyRegardMechnisamT && PenaltyRegardListElefant.Count == TableListElefant.Count)

Equality = true;

else

if (kind == 3 && Current.IsPenaltyAction() != 0 && UsePenaltyRegardMechnisamT && PenaltyRegardListHourse.Count == TableListHourse.Count)

Equality = true;

else

if (kind == 4 && Current.IsPenaltyAction() != 0 && UsePenaltyRegardMechnisamT && PenaltyRegardListMinister.Count == TableListMinister.Count)

Equality = true;

else

if (kind == 5 && Current.IsPenaltyAction() != 0 && UsePenaltyRegardMechnisamT && PenaltyRegardListKing.Count == TableListKing.Count)

Equality = true;

else

if (kind == 6 && Current.IsPenaltyAction() != 0 && UsePenaltyRegardMechnisamT && PenaltyRegardListSolder.Count == TableListSolder.Count)

Equality = true;

return Equality;

}

}

void AddAtList(int kind, QuantumAtamata Current)

{

Object O = new Object();

lock (O)

{

//Adding QuantumAutamata Object to Specified List.

if (kind == 1)

//Soldier

PenaltyRegardListSolder.Add(Current);

else

if (kind == 2)

//Elefant

PenaltyRegardListElefant.Add(Current);

else

if (kind == 3)

//Hourse

PenaltyRegardListHourse.Add(Current);

else

if (kind == 4)

//Castles.

PenaltyRegardListCastle.Add(Current);

else

if (kind == 5)

//Minister.

PenaltyRegardListMinister.Add(Current);

else

if (kind == 6)

//King.

PenaltyRegardListKing.Add(Current);

}

}

void RemoveAtList(int kind)

{

Object O = new Object();

lock (O)

{

//Remove Last QuantumAtutamata Object.

if (kind == 1)

//Soldier

PenaltyRegardListSolder.RemoveAt(PenaltyRegardListSolder.Count - 1);

else

if (kind == 2)

//Elefant

PenaltyRegardListElefant.RemoveAt(PenaltyRegardListElefant.Count - 1);

else

if (kind == 3)

//Hourse

PenaltyRegardListHourse.RemoveAt(PenaltyRegardListHourse.Count - 1);

else

if (kind == 4)

//Castles

PenaltyRegardListCastle.RemoveAt(PenaltyRegardListCastle.Count - 1);

else

if (kind == 5)

//Minister

PenaltyRegardListMinister.RemoveAt(PenaltyRegardListMinister.Count - 1);

else

if (kind == 6)

//King.

PenaltyRegardListKing.RemoveAt(PenaltyRegardListKing.Count - 1);

}

}

bool PenaltyMechanisam(int Killed, bool Before, int kind, int[,] TableS, int ii, int jj, ref QuantumAtamata Current, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object OO = new Object();

lock (OO)

{

bool RETURN = false;

Object O3 = new Object();

ChessRules AA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj);

Object O = new Object();

lock (O)

{

if (!UsePenaltyRegardMechnisamT)

{

RETURN = true;

AddAtList(kind, Current);

}

//Consideration to go to Check.

//if (!UsePenaltyRegardMechnisamT)

AA.CheckMate(TableS, Order);

{

if (AllDraw.OrderPlate == 1 && AA.CheckMateBrown)

{

Object A = new Object();

lock (A)

{

IsThereMateOfEnemy = true;

FoundFirstMating++;

Current.LearningAlgorithmRegard();

RemoveAtList(kind);

AddAtList(kind, Current);

return true;

}

}

if (AllDraw.OrderPlate == -1 && AA.CheckMateGray)

{

DoEnemySelf = false;

Object A = new Object();

lock (A)

{

IsThereMateOfEnemy = true;

FoundFirstMating++;

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

return true;

}

}

if (Order == 1 && AA.CheckMateBrown)

{

DoEnemySelf = false;

EnemyCheckMateActionsString = true;

}

if (Order == -1 && AA.CheckMateGray)

{

DoEnemySelf = false;

EnemyCheckMateActionsString = true;

}

if (Order == 1 && AA.CheckMateGray)

{

EnemyCheckMateActionsString = false;

}

if (Order == -1 && AA.CheckMateBrown)

{

EnemyCheckMateActionsString = false;

}

if (Order == 1 && AA.CheckGray)

{

//KishBefore = true;

Object A = new object();

lock (A)

{

NumberOfPenalties++;

}

}

else

if (Order == -1 && AA.CheckBrown)

{

//KishBefore = true;

Object A = new object();

lock (A)

{

NumberOfPenalties++;

}

}

}

if (RETURN)

return false;

}

//Initiate Local Variables.

bool IsCurrentCanGardHighPriorityEne = new bool();

bool IsNextMovemntIsCheckOrCheckMateForCurrent = new bool();

bool IsNextMovemntIsCheckOrCheckMateForEnemy = new bool();

bool IsDangerous = new bool();

bool CanKillerAnUnSupportedEnemy = new bool();

bool InDangrousUnSupported = new bool();

bool Support = new bool();

bool IsPrviousMovemntIsDangrousForCurr = new bool();

bool PDo = new bool(), RDo = new bool();

bool SelfNotSupported = new bool();

bool EnemyNotSupported = new bool();

bool IsGardForCurrentMovmentsAndIsNotMova = new bool();

bool IsNotSafeToMoveAenemeyToAttackMoreThanTowObj = new bool();

bool[] LearningV = null;

//Mechanisam of Regrad.

Object O1 = new Object();

lock (O1)

{

if (kind == 1 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListSolder.Count == TableListSolder.Count)

LearningV = CalculateLearningVars(Killed, TableS, ii, jj, i, j);

else

if (kind == 2 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListElefant.Count == TableListElefant.Count)

LearningV = CalculateLearningVars(Killed, TableS, ii, jj, i, j);

else

if (kind == 3 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListHourse.Count == TableListHourse.Count)

LearningV = CalculateLearningVars(Killed, TableS, ii, jj, i, j);

else

if (kind == 4 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListMinister.Count == TableListMinister.Count)

LearningV = CalculateLearningVars(Killed, TableS, ii, jj, i, j);

else

if (kind == 5 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListKing.Count == TableListKing.Count)

LearningV = CalculateLearningVars(Killed, TableS, ii, jj, i, j);

else

if (kind == 6 && PenRegStrore && UsePenaltyRegardMechnisamT && PenaltyRegardListSolder.Count == TableListSolder.Count)

LearningV = CalculateLearningVars(Killed, TableS, ii, jj, i, j);

}

Object O2 = new Object();

lock (O2)

{

IsCurrentCanGardHighPriorityEne = LearningV[0];

IsNextMovemntIsCheckOrCheckMateForCurrent = LearningV[1];

IsDangerous = LearningV[2];

CanKillerAnUnSupportedEnemy = LearningV[3];

InDangrousUnSupported = LearningV[4];

Support = LearningV[5];

IsNextMovemntIsCheckOrCheckMateForEnemy = LearningV[6];

IsPrviousMovemntIsDangrousForCurr = LearningV[7];

PDo = LearningV[8];

RDo = LearningV[9];

SelfNotSupported = LearningV[10];

EnemyNotSupported = LearningV[11];

IsGardForCurrentMovmentsAndIsNotMova = LearningV[12];

IsNotSafeToMoveAenemeyToAttackMoreThanTowObj = LearningV[13];

}

//Consideration of Itterative Movments to ignore.

//Operation of Penalty Regard Mechanisam on Check and mate speciffically.

bool Equality = EqualitOne(Current, kind);

Object O4 = new Object();

lock (O4)

{

if (Equality)

{

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, Row, Column);

if (A.Check(TableS, Order))

{

if (Order == 1 && (A.CheckGray))

{

NumberOfPenalties++;

Current.LearningAlgorithmPenalty();

}

else

if (Order == -1 && (A.CheckBrown))

{

NumberOfPenalties++;

Current.LearningAlgorithmPenalty();

}

AddAtList(kind, Current);

}

else

{

if (IsCurrentStateIsDangreousForCurrentOrder(TableS, Order, color, i, j) && DoEnemySelf)

{

NumberOfPenalties++;

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

else

AddAtList(kind, Current);

}

//When There is Penalty or Regard.To Side can not be equal.

if (PDo || RDo)

{

//Penalty.

if (PDo)

{

Object OO1 = new Object();

lock (OO1)

{

for (int ik = 0; ik < System.Math.Abs(TableS[i, j]); ik++)

LearniningTable.LearningAlgorithmPenaltyNet(ii, jj);

}

DivisionPenaltyRegardHeuristicQueficient = 3;

//When previous Move of Enemy goes to Dangoure Current Object.

if (IsPrviousMovemntIsDangrousForCurr && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

//For Not Suppored In Attacked.

if (SelfNotSupported && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

//When Current Move Dos,'t Supporte.

//For Ocuuring in Enemy CheckMate.

if (SelfNotSupported && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsGardForCurrentMovmentsAndIsNotMova && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsNotSafeToMoveAenemeyToAttackMoreThanTowObj && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsDangerous && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (EnemyNotSupported && Current.IsPenaltyAction() != 0 && Current.IsRewardAction() != 1)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

}

else if (RDo)

{

Object OOO = new Object();

lock (OOO)

{

for (int ik = 0; ik < System.Math.Abs(TableS[i, j]); ik++)

LearniningTable.LearningAlgorithmRegardNet(ii, jj);

}

DivisionPenaltyRegardHeuristicQueficient = 3;

if (SelfNotSupported && Current.IsPenaltyAction() != 0)

{

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsGardForCurrentMovmentsAndIsNotMova && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsNotSafeToMoveAenemeyToAttackMoreThanTowObj && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsDangerous && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (EnemyNotSupported && Current.IsPenaltyAction() != 0 && Current.IsRewardAction() != 1)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

if (IsCurrentCanGardHighPriorityEne && Current.IsPenaltyAction() != 0 && Current.IsRewardAction() != 1)

{

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

//For Ocuuring Enemy Garding Objects.

if (Support && Current.IsPenaltyAction() != 0 && Current.IsRewardAction() != 1)

{

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

}

}

else

{

Object OO1 = new Object();

lock (OO1)

{

for (int ik = 0; ik < System.Math.Abs(TableS[i, j]); ik++)

{

LearniningTable.LearningAlgorithmRegardNet(ii, jj);

LearniningTable.LearningAlgorithmPenaltyNet(ii, jj);

}

}

DivisionPenaltyRegardHeuristicQueficient = 2;

if (IsNextMovemntIsCheckOrCheckMateForCurrent && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (SelfNotSupported && Current.IsPenaltyAction() != 0)

{

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsGardForCurrentMovmentsAndIsNotMova && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsNotSafeToMoveAenemeyToAttackMoreThanTowObj && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsDangerous && Current.IsPenaltyAction() != 0)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmPenalty();

AddAtList(kind, Current);

}

if (IsNextMovemntIsCheckOrCheckMateForEnemy && Current.IsPenaltyAction() != 0)

{

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

if (IsCurrentCanGardHighPriorityEne && Current.IsPenaltyAction() != 0)

{

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

if (EnemyNotSupported && Current.IsPenaltyAction() != 0 && Current.IsRewardAction() != 1)

{

NumberOfPenalties++;

RemoveAtList(kind);

Current.LearningAlgorithmRegard();

AddAtList(kind, Current);

}

}

}

}

return false;

}

}

void SolderThinkingChess(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///When There is Movments.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, ii, jj)).Rules(ii, jj, i, j, color, TableS[ii, jj]))

{

try

{

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

///Add Table to List of Private.

HitNumberSoldier.Add(TableS[i, j]);

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

Object A = new object();

lock (A)

{

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

Object A1 = new object();

lock (A1)

{

NumbersOfAllNode++;

}

///ActionsString of Movements.

int Killed = 0;

Object A2 = new object();

lock (A2)

{

Killed = TableS[i, j];

TableS[i, j] = TableS[ii, jj];

TableS[ii, jj] = 0;

}

Object A3 = new object();

lock (A3)

{

PenaltyMechanisam(Killed, false, 1, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

}

//}

///Store of Indexes Changes and Table in specific List.

Object A4 = new object();

lock (A4)

{

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnSoldier.Add(AS);

TableListSolder.Add(CloneATable(TableS)); ;

IndexSoldier++;

}

Object A5 = new object();

lock (A5)

{

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, Killed, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

}

///Calculate Huristic and Add to List Speciifically and Cal Syntax.

Object A6 = new object();

lock (A6)

{

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListSolder.Add(Hu);

}

Object O8 = new Object();

lock (O8)

{

OutPutAction = " " + Alphabet(ii) + Number(jj) + Alphabet(i) + Number(j);

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Soldeir AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

else

AllDraw.OutPut = "\r\nThinking Soldeir AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + OutPutAction;

ThinkingLevel++;

ThinkingAtRun = false;

}

}

catch (Exception t)

{

Log(t);

ThinkingAtRun = false;

}

}

}

}

void CastleThinkingBrown(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//When is Brown Castles King.

//Calcuilate Huristic Before Movment.

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

Object A = new object();

lock (A)

{

NumbersOfAllNode++;

}

int Killed = 0;

if (i < ii)

{

TableS[ii - 1, j] = -4;

TableS[ii - 2, j] = -6;

TableS[ii, jj] = 0;

TableS[0, jj] = 0;

}

else

{

TableS[ii + 1, j] = -4;

TableS[ii + 2, j] = -6;

TableS[ii, jj] = 0;

TableS[7, jj] = 0;

}

PenaltyMechanisam(Killed, false, 7, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

//Store Movments Items.

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnKing.Add(AS);

TableListKing.Add(CloneATable(TableS)); ;

IndexKing++;

//Calculate Huristic Sumation and Store in Specific List.

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

HuristicListKing.Add(Hu);

Castle = true;

Object O7 = new Object(); SetObjectNumbersInList(TableS);

lock (O7)

{

if (i < ii)

{

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O-O";

else

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O-O";

ThinkingLevel++;

}

else

{

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O";

else

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O";

ThinkingLevel++;

}

ThinkingAtRun = false;

}

}

}

public void CalculateHuristics(bool Before, int Killed, int[,] TableSS, int i, int j, int ii, int jj, Color color

, ref double HuristicAttackValue

, ref double HuristicMovementValue

, ref double HuristicSelfSupportedValue

, ref double HuristicObjectDangourCheckMateValue

, ref double HuristicKillerValue

, ref double HuristicReducedAttackValue

, ref double HeuristicDistabceOfCurrentMoveFromEnemyKingValue

, ref double HeuristicKingSafe

, ref double HeuristicFromCenter

, ref double HeuristicKingDangour)

{

Object OO = new Object();

lock (OO)

{

double[] Huriistic = null;

double HCheck = new double();

double HDistance = new double();

double HKingSafe = new double();

double HKingDangour = new double();

double HFromCenter = 0;

Parallel.Invoke(() =>

{

Object O = new Object();

lock (O)

{

Huriistic = HuristicAll(Before, Killed, TableSS, color, Order, i, j, ii, jj);

}

}

, () =>

{

Object O = new Object();

lock (O)

{

HCheck = HuristicCheckAndCheckMate(TableSS, color//, ref HuristicObjectDangourCheckMateValue

);

}

}

, () =>

{

Object O = new Object();

lock (O)

{

HDistance = HeuristicDistabceOfCurrentMoveFromEnemyKing(TableSS, Order, i, j//, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue

);

}

}

, () =>

{

Object O = new Object();

lock (O)

{

HKingSafe = HeuristicKingSafety(TableSS, Order, color, i, j, ii, jj//, ref HeuristicKingSafe

, CurrentAStarGredyMax);

}

}

, () =>

{

Object O = new Object();

lock (O)

{

HKingDangour = HeuristicKingDangourous(TableSS, Order, color, i, j, ii, jj//, ref HeuristicKingSafe

, CurrentAStarGredyMax);

}

}

, () =>

{

Object O = new Object();

lock (O)

{

HFromCenter = HuristicSoldierFromCenter(TableSS, color, Order, i, j, ii, jj);

}

}

);

Object O1 = new Object();

lock (O1)

{

HuristicAttackValue = Huriistic[0] \* SignOrderToPlate(Order);

HuristicKillerValue = Huriistic[1] \* SignOrderToPlate(Order);

HuristicMovementValue = Huriistic[2] \* SignOrderToPlate(Order);

HuristicObjectDangourCheckMateValue = (Huriistic[3] + HCheck) \* SignOrderToPlate(Order);

HuristicReducedAttackValue = Huriistic[4] \* SignOrderToPlate(Order);

HuristicSelfSupportedValue = Huriistic[5] \* SignOrderToPlate(Order);

HeuristicDistabceOfCurrentMoveFromEnemyKingValue = HDistance \* SignOrderToPlate(Order);

HeuristicKingSafe = HKingSafe \* SignOrderToPlate(Order);

HeuristicFromCenter = HFromCenter \* SignOrderToPlate(Order);

HeuristicKingDangour = HKingDangour \* SignOrderToPlate(Order);

}

}

}

void CastleThinkingGray(int DummyOrder, int DummyCurrentOrder, int[,] TableS, int ii, int jj, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, int i, int j, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

double HuristicAttackValue = new double();

double HuristicMovementValue = new double();

double HuristicSelfSupportedValue = new double();

double HuristicObjectDangourCheckMateValue = new double();

double HuristicKillerValue = new double();

double HuristicReducedAttackValue = new double();

double HeuristicDistabceOfCurrentMoveFromEnemyKingValue = new double();

double HeuristicKingSafe = new double();

double HeuristicFromCenter = new double();

double HeuristicKingDangour = new double();

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

ThinkingAtRun = true;

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

//When is Castles Gray King.

//Predict Huristic Caluculatio Before Movments.

Object O = new Object();

lock (O)

{

ThinkingRun = true;

}

CalculateHuristics(true, 0, TableS, ii, jj, i, j, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

Object A = new object();

lock (A)

{

NumbersOfAllNode++;

}

int Killed = 0;

if (i < ii)

{

TableS[ii - 1, j] = 4;

TableS[ii - 2, j] = 6;

TableS[ii, jj] = 0;

TableS[0, jj] = 0;

}

else

{

TableS[ii + 1, j] = 4;

TableS[ii + 2, j] = 6;

TableS[ii, jj] = 0;

TableS[7, jj] = 0;

}

PenaltyMechanisam(Killed, false, 7, TableS, ii, jj, ref Current, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

//{ ThinkingAtRun = false; return; }

//Store Movments Items.

int[] AS = new int[2];

AS[0] = i;

AS[1] = j;

RowColumnKing.Add(AS);

TableListKing.Add(CloneATable(TableS));

IndexKing++;

//Calculate Movment Huristic After Movments.

//Caused this for Stachostic results.

SetValueOfTabls(ii, jj); CalculateHuristics(false, Killed, TableS, i, j, ii, jj, color, ref HuristicAttackValue, ref HuristicMovementValue, ref HuristicSelfSupportedValue, ref HuristicObjectDangourCheckMateValue, ref HuristicKillerValue, ref HuristicReducedAttackValue, ref HeuristicDistabceOfCurrentMoveFromEnemyKingValue, ref HeuristicKingSafe, ref HeuristicFromCenter, ref HeuristicKingDangour);

double[] Hu = new double[10]; HuristicPenaltyValuePerform(Current, Order, ref HuristicAttackValue);

if (IgnoreFromCheckandMateHuristic)

HuristicObjectDangourCheckMateValue = 0;

Hu[0] += HuristicAttackValue;

Hu[1] += HuristicMovementValue;

Hu[2] += HuristicSelfSupportedValue;

Hu[3] += HuristicObjectDangourCheckMateValue;

Hu[4] += HuristicKillerValue;

Hu[5] += HuristicReducedAttackValue;

Hu[6] += HeuristicDistabceOfCurrentMoveFromEnemyKingValue;

Hu[7] += HeuristicKingSafe;

Hu[8] = HeuristicFromCenter;

Hu[9] = HeuristicKingDangour;

Object O9 = new Object();

lock (O9)

{

if (i < ii)

{

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O-O";

else

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O-O";

ThinkingLevel++;

}

else

{

if (Order == 1)

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Bob at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O";

else

AllDraw.OutPut = "\r\nThinking Castle AstarGreedy By Level " + CurrentAStarGredyMax.ToString() + " Alice at " + ThinkingLevel.ToString() + "th Thinking String " + "O-O";

ThinkingLevel++;

}

HuristicListKing.Add(Hu);

//ChessRules.CastleActGray = true;

Castle = true;

ThinkingAtRun = false;

}

}

}

public void HuristicPenaltyValuePerform(QuantumAtamata Current, int Order, ref double HuristicAttackValue, bool AllDrawClass = false)

{

Object O1 = new Object();

lock (O1)

{

if (LearningVarsObject.Count == 0 || AllDrawClass)

{

if (AllDraw.OrderPlate == Order)

{

if (Current.IsPenaltyAction() == 0)

HuristicAttackValue += (-300 / DivisionPenaltyRegardHeuristicQueficient);

}

else

if (AllDraw.OrderPlate != Order)

{

if (Current.IsPenaltyAction() == 0)

HuristicAttackValue += (300 / DivisionPenaltyRegardHeuristicQueficient);

}

if (AllDraw.OrderPlate == Order)

{

if (Current.IsRewardAction() == 1)

HuristicAttackValue += (300 / DivisionPenaltyRegardHeuristicQueficient);

}

else

if (AllDraw.OrderPlate != Order)

{

if (Current.IsRewardAction() == 1)

HuristicAttackValue += (-300 / DivisionPenaltyRegardHeuristicQueficient);

}

}

else

{

if ((LearningVarsObject[LearningVarsObject.Count - 1][1] && !LearningVarsObject[LearningVarsObject.Count - 1][4]))

{

if (AllDraw.OrderPlate == Order)

{

if (Current.IsPenaltyAction() == 0)

HuristicAttackValue += (-1000000 / DivisionPenaltyRegardHeuristicQueficient);

}

else

if (AllDraw.OrderPlate != Order)

{

if (Current.IsPenaltyAction() == 0)

HuristicAttackValue += (1000000 / DivisionPenaltyRegardHeuristicQueficient);

}

if (AllDraw.OrderPlate == Order)

{

if (Current.IsRewardAction() == 1)

HuristicAttackValue += (1000000 / DivisionPenaltyRegardHeuristicQueficient);

}

else

if (AllDraw.OrderPlate != Order)

{

if (Current.IsRewardAction() == 1)

HuristicAttackValue += (-1000000 / DivisionPenaltyRegardHeuristicQueficient);

}

}

}

}

}

public void ThinkingSoldierBase(int ord, int ii, int jj, int i, int j, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O = new Object();

lock (O)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

if (Scop(ii, jj, i, j, 1) && System.Math.Abs(TableS[ii, jj]) == 1 && System.Math.Abs(Kind) == 1)

{

Order = ord;

SolderThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

}

}

}

public void ThinkingSoldier(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

Parallel.For(ii - 2, ii + 3, i =>

{

Parallel.For(jj - 2, jj + 3, j =>

{

Object O = new Object();

lock (O)

{

if (Scop(ii, jj, i, j, 1))

{

ThinkingSoldierBase(ord, ii, jj, i, j, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

}

});

});

}

}

public void ThinkingElephantBase(int ord, int ii, int jj, int i, int j, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

///Else for Elephant Thinking.

if (Scop(ii, jj, i, j, 2) && System.Math.Abs(TableS[ii, jj]) == 2 && System.Math.Abs(Kind) == 2)

{

Order = ord;

ElephantThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

}

}

}

}

public void ThinkingElephant(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O2 = new Object();

lock (O2)

{

Object O1 = new Object();

lock (O1)

{

Parallel.For(0, 8, i =>

{

Object O = new Object();

lock (O)

{

int j = i + jj - ii;

if (Scop(ii, jj, i, j, 2))

{

ThinkingElephantBase(ord, ii, jj, i, j, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

}

});

//==================

Parallel.For(0, 8, i =>

{

Object O = new Object();

lock (O)

{

while (ThinkingAtRun) { }

int j = i \* -1 + ii + jj;

if (Scop(ii, jj, i, j, 2))

ThinkingElephantBase(ord, ii, jj, i, j, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

ThinkingAtRun = false;

}

});

}

}

}

public void ThinkingHourseOne(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii + 2, jj + 1, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii + 2, jj + 1, Castle);

}

}

}

public void ThinkingHourseTwo(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii - 2, jj - 1, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii - 2, jj - 1, Castle

);

}

}

public void ThinkingHourseThree(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii + 2, jj - 1, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii + 2, jj - 1, Castle);

}

}

}

public void ThinkingHourseFour(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii - 2, jj + 1, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii - 2, jj + 1, Castle

);

}

}

public void ThinkingHourseFive(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii + 1, jj + 2, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii + 1, jj + 2, Castle

);

}

}

}

public void ThinkingHourseSix(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii - 1, jj - 2, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii - 1, jj - 2, Castle);

}

}

}

public void ThinkingHourseSeven(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O = new Object();

lock (O)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O111 = new Object();

lock (O111)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii + 1, jj - 2, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii + 1, jj - 2, Castle);

}

}

}

public void ThinkingHourseEight(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O111 = new Object();

lock (O111)

{

int[,] TableS = new int[8, 8];

///Initiate a Local Variables.

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

Order = ord;

if (Scop(ii, jj, ii - 1, jj + 2, 3))

HourseThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, ii - 1, jj + 2, Castle);

}

}

}

public void ThinkingHourse(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O = new Object();

lock (O)

{

ThinkingHourseOne(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O1 = new Object();

lock (O1)

{

ThinkingHourseTwo(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O2 = new Object();

lock (O2)

{

ThinkingHourseThree(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O3 = new Object();

lock (O3)

{

ThinkingHourseFour(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O4 = new Object();

lock (O4)

{

ThinkingHourseFive(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O5 = new Object();

lock (O5)

{

ThinkingHourseSix(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O6 = new Object();

lock (O6)

{

ThinkingHourseSeven(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

Object O7 = new Object();

lock (O7)

{

ThinkingHourseEight(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

while (ThinkingAtRun) { }

}

}

public void ThinkingCastleOne(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

Parallel.For(0, 8, i =>

{

Object O = new Object();

lock (O)

{

int j = jj;

///Initiate a Local Variables.

int[,] TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

if (Scop(ii, jj, i, j, 4) && System.Math.Abs(TableS[ii, jj]) == 4 && System.Math.Abs(Kind) == 4)

{

while (ThinkingAtRun) { }

Order = ord;

CastlesThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

}

}

});

}

}

public void ThinkingCastleTow(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{ //==================

Object O1 = new Object();

lock (O1)

{

Parallel.For(0, 8, j =>

{

Object O = new Object();

lock (O)

{

int i = ii;

///Initiate a Local Variables.

int[,] TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

if (Scop(ii, jj, i, j, 4) && System.Math.Abs(TableS[ii, jj]) == 4 && System.Math.Abs(Kind) == 4)

{

while (ThinkingAtRun) { }

Order = ord;

CastlesThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle

);

}

}

});

}

}

public void ThinkingCastle(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O = new Object();

lock (O)

{

ThinkingCastleOne(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

ThinkingCastleTow(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

}

public void ThinkingMinisterBase(int ord, int ii, int jj, int i, int j, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

///Initiate a Local Variables.

int[,] TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

Object O = new Object();

lock (O)

{

while (ThinkingAtRun) { }

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

if (Scop(ii, jj, i, j, 5) && System.Math.Abs(TableS[ii, jj]) == 5 && System.Math.Abs(Kind) == 5)

{

while (ThinkingAtRun) { }

Order = ord;

MinisterThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle

);

}

}

}

}

public void ThinkingMinister(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

Parallel.For(0, 8, i =>

{

Parallel.For(0, 8, j =>

{

Object O = new Object();

lock (O)

{

ThinkingMinisterBase(ord, ii, jj, i, j, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

});

});

}

}

public void ThinkingCastleGray(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O = new Object();

lock (O)

{

for (int i = ii - 2; i < ii + 2; i++)

{

while (ThinkingAtRun) { }

///Initiate a Local Variables.

int[,] TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

///Calculate of Castles of Brown.

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, -7, TableS, Order, ii, jj)).Rules(ii, jj, i, jj, color, -7) && (ChessRules.CastleKingAllowedBrown))

{

CastleThinkingBrown(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, jj, Castle);

}

ThinkingAtRun = false;

}

}

}

public void ThinkingCastleBrown(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O = new Object();

lock (O)

{

for (int i = ii - 2; i < ii + 2; i++)

{

while (ThinkingAtRun) { }

///Initiate a Local Variables.

int[,] TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

if ((new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, 7, TableS, Order, ii, jj)).Rules(ii, jj, i, jj, color, 7) && (ChessRules.CastleKingAllowedGray))

{

CastleThinkingGray(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, jj, Castle);

}

ThinkingAtRun = false;

}

}

}

public void ThinkingKing(int ord, int ii, int jj, int DummyOrder, int DummyCurrentOrder, bool DoEnemySelf, bool PenRegStrore, bool EnemyCheckMateActionsString, bool Castle)

{

Object O1 = new Object();

lock (O1)

{

int[,] TableS = new int[8, 8];

Object O = new Object();

lock (O)

{

Parallel.For(ii - 1, ii + 2, i =>

{

Parallel.For(jj - 1, jj + 2, j =>

{

if (i == ii && j == jj)

return;

///Initiate a Local Variables.

TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

if (Scop(ii, jj, i, j, 6) && System.Math.Abs(TableS[ii, jj]) == 6 && System.Math.Abs(Kind) == 6)

{

while (ThinkingAtRun) { }

Order = ord;

KingThinkingChess(DummyOrder, DummyCurrentOrder, TableS, ii, jj, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, i, j, Castle);

}

});

});

}

}

}

///Kernel of Thinking

public void Thinking()

{

int ord = Order;

Object O = new Object();

lock (O)

{

while (!ThinkingBegin) { System.Threading.Thread.Sleep(1); }// S += 2; if (AllDraw.Blitz) { if (S > ThresholdBlitz)break; } else { if (S > ThresholdFullGame)break; } }

NumberOfPenalties = 0;

SetObjectNumbers(CloneATable(TableConst));

bool PenRegStrore = true;

// if (Order != AllDraw.OrderPlate)

// PenRegStrore = false;

//Thread.Sleep(500);

Object O1 = new Object();

lock (O1)

{

BeginThread++;

}

//if (!AllDraw.Blitz)

{

if (//CheckMateOcuured ||

FoundFirstMating > AllDraw.MaxAStarGreedy

)

{

Object O2 = new Object();

lock (O2)

{

AllDraw.OutPut = "\r\nBoundry Condition at Thinking at " + ThinkingChess.FoundFirstMating.ToString() + " Checkmate";

ThinkingBegin = false;

ThinkingFinished = true;

EndThread++;

}

return;

}

}

int DummyOrder = Order;

int DummyCurrentOrder = ChessRules.CurrentOrder;

//Initiate Locallly Global Variables.

IndexSoldier = 0;

IndexElefant = 0;

IndexHourse = 0;

IndexCastle = 0;

IndexMinister = 0;

IndexKing = 0;

int[,] TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

///Most Dot Net FrameWork Hot Path

///Create A Clone of Current Table Constant in ThinkingChess Object Tasble.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

///For Stored Location of Objects.

int ii = Row;

int jj = Column;

if (CheckMateOcuured

|| FoundFirstMating > AllDraw.MaxAStarGreedy

)

{

Object O2 = new Object();

lock (O2)

{

AllDraw.OutPut = "\r\nBoundry Condition at Thinking at " + ThinkingChess.FoundFirstMating.ToString() + " Checkmate";

ThinkingFinished = true;

ThinkingBegin = false;

EndThread++;

}

return;

}

IgnoreObjectDangour = -1;

///Initiate a Local Variables.

TableS = new int[8, 8];

///"Inizialization of This New Class (Current is Dynamic class Object) is MalFunction (Constant Variable Count).

QuantumAtamata Current = new QuantumAtamata(3, 3, 3);

///Most Dot Net FrameWork Hot Path

///Create A Clone of Current Table Constant in ThinkingChess Object Tasble.

for (int iii = 0; iii < 8; iii++)

for (int jjj = 0; jjj < 8; jjj++)

{

TableS[iii, jjj] = TableConst[iii, jjj];

}

///Deterimine for Castle King Wrongly Desision.

bool Castle = false;

//ExistInDestinationEnemy = false;

bool DoEnemySelf = true;

ChessRules AAA = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, TableS[ii, jj], TableS, Order, Row, Column);

if (AAA.CheckMate(TableS, Order))

{

if (AAA.CheckMateGray || AAA.CheckMateBrown)

{

Object O2 = new Object();

lock (O2)

{

AllDraw.OutPut = "\r\nBoundry Condition at Thinking at " + ThinkingChess.FoundFirstMating.ToString() + " Checkmate";

ThinkingFinished = true;

CheckMateOcuured = true;

if ((AAA.CheckMateGray && Order == 1) || (AAA.CheckMateBrown && Order == -1))

FoundFirstMating++;

EndThread++;

}

return;

}

}

if (Order == 1 && AAA.CheckGray)

{

IgnoreObjectDangour = 0;

IsCheck = true;

DoEnemySelf = false;

}

if (Order == -1 && AAA.CheckBrown)

{

IgnoreObjectDangour = 0;

IsCheck = true;

DoEnemySelf = false;

}

//When Root is CheckMate Benefit of Current Order No Consideration.

int CDumnmy = ChessRules.CurrentOrder;

bool EnemyCheckMateActionsString = false;

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

///Calculate Castles of Gray King.

///

try

{

if (Kind == 7)

{

ThinkingCastleBrown(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

else

if (Kind == -7)

{

ThinkingCastleGray(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

else

if (System.Math.Abs(Kind) == 1)///For Soldier Thinking

{

ThinkingSoldier(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

else

if (System.Math.Abs(Kind) == 2)///For Elephant Thinking

{

ThinkingElephant(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

///Else for Hourse Thinking.

else

if (System.Math.Abs(Kind) == 3)///For Hourse Thinking

{

ThinkingHourse(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

///Else For Castles Thinking.

else

if (System.Math.Abs(Kind) == 4)///For Castle Thinking

{

ThinkingCastle(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

///Else for Minister Thinkings.

else

if (System.Math.Abs(Kind) == 5)///For Minister Thinking

{

ThinkingMinister(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

///Else For Kings Thinkings.

else

if (System.Math.Abs(Kind) == 6)///For King Thinking

{

ThinkingKing(ord, ii, jj, DummyOrder, DummyCurrentOrder, DoEnemySelf, PenRegStrore, EnemyCheckMateActionsString, Castle);

}

}

catch (Exception t)

{

Log(t);

}

Object O3 = new Object();

lock (O3)

{

///Initiate Global Varibales at END.

ThinkingBegin = false;

///This Variable Not Work!

ThinkingFinished = true;

Order = DummyOrder;

ChessRules.CurrentOrder = DummyCurrentOrder;

EndThread++;

}

//UsePenaltyRegardMechnisamT = PenRegStrore;

//

///Return at End.

}

return;

}

int ObjectValueCalculator(int[,] Table, int Order, int ii, int jj)

{

Object O1 = new Object();

lock (O1)

{

int Val = 0;

//if (BeginArragmentsOfOrderFinished(Tabl, Order))

{

ChessRules A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[ii, jj], Table, Order, ii, jj);

Color a = Color.Gray;

Color aa = Color.Gray;

if (Order == -1)

a = Color.Brown;

if (Order \* -1 == -1)

aa = Color.Brown;

Parallel.For(0, 8, Row =>

Parallel.For(0, 8, Column =>

{

Parallel.For(0, 8, iii =>

{

Parallel.For(0, 8, jjj =>

{

Object O = new Object();

lock (O)

{

if (Scop(Row, Column, iii, jjj, System.Math.Abs(Table[Row, Column])) && (Table[Row, Column] != 0) && (Table[iii, jjj] == 0))

{

if (SignEqualOrEmptySelf(Table[Row, Column], Table[iii, jjj], Order))

{

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[Row, Column], Table, Order, Row, Column);

if (A.Rules(Row, Column, iii, jjj, a, Order))

Val++;

}

else

if (SignEqualOrEmptyEnemy(Table[Row, Column], Table[iii, jjj], Order))

{

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[Row, Column], Table, Order, Row, Column);

if (A.Rules(Row, Column, iii, jjj, a, Order))

Val--;

}

else

return;

}

else

if (Scop(Row, Column, iii, jjj, System.Math.Abs(Table[Row, Column])) && (Table[Row, Column] != 0) && (Table[iii, jjj] != 0))

{

//When is self level

if (Order == AllDraw.OrderPlate)

{

if (SignNotEqual(Table[Row, Column], Table[iii, jjj]))

{

//When there is enemy attack dec.

if (Attack(Table, Row, Column, iii, jjj, aa, Order \* -1))

Val -= 999999999;

}

else

if (SignEqual(Table[Row, Column], Table[iii, jjj]))

{

//when there is self support inc.

if (Support(Table, Row, Column, iii, jjj, a, Order))

Val++;

}

else

if (SignNotEqual(Table[Row, Column], Table[iii, jjj]))

{

//When there is self attack inc.

if (Attack(Table, Row, Column, iii, jjj, a, Order))

Val++;

}

else

//when there is enemy support dec..

if (SignEqual(Table[Row, Column], Table[iii, jjj]))

{

if (Support(Table, Row, Column, iii, jjj, aa, Order \* -1))

Val--;

}

else

return;

}

//When is enemy level.

else

{

if (SignNotEqual(Table[Row, Column], Table[iii, jjj]))

{

//When there is enemy attack inc.

if (Attack(Table, Row, Column, iii, jjj, aa, Order \* -1))

Val += 9999;

}

else

if (SignEqual(Table[Row, Column], Table[iii, jjj]))

{

//when there is self support dec.

if (Support(Table, Row, Column, iii, jjj, a, Order))

Val--;

}

else

//When there is self attack dec.

if (SignNotEqual(Table[Row, Column], Table[iii, jjj]))

{

if (Attack(Table, Row, Column, iii, jjj, a, Order))

Val--;

}

else

if (SignEqual(Table[Row, Column], Table[iii, jjj]))

{

//when there is enemy support inc.

if (Support(Table, Row, Column, iii, jjj, aa, Order \* -1))

Val++;

}

else

return;

}

}

}

});

});

}));

A = new ChessRules(CurrentAStarGredyMax, MovementsAStarGreedyHuristicFoundT, IgnoreSelfObjectsT, UsePenaltyRegardMechnisamT, BestMovmentsT, PredictHuristicT, OnlySelfT, AStarGreedyHuristicT, ArrangmentsChanged, Table[ii, jj], Table, Order, ii, jj);

if (A.ObjectDangourKingMove(Order, Table, false))

{

if (Order == 1 && A.CheckGrayObjectDangour)

Val -= 5;

if (Order == -1 && A.CheckBrownObjectDangour)

Val -= 5;

if (Order == -1 && A.CheckGrayObjectDangour)

Val += 5;

if (Order == 1 && A.CheckBrownObjectDangour)

Val += 5;

}

if (System.Math.Abs(Table[ii, jj]) == 2)

{

Val = Val \* 3;

}

else

if (System.Math.Abs(Table[ii, jj]) == 3)

{

Val = Val \* 3;

}

else

if (System.Math.Abs(Table[ii, jj]) == 4)

{

Val = Val \* 5;

}

else

if (System.Math.Abs(Table[ii, jj]) == 5)

{

Val = Val \* 9;

}

}

if (Val < 0)

Val = 0;

return Val;

}

}

bool SignEqualOrEmptySelf(int Obj1, int Obj2, int Order)

{

Object O = new Object();

lock (O)

{

bool Is = false;

if (Order == AllDraw.OrderPlate)

{

if (Order == 1)

{

if (Obj1 > 0 && Obj2 == 0)

Is = true;

}

else

{

if (Obj1 < 0 && Obj2 == 0)

Is = true;

}

}

return Is;

}

}

bool SignEqualOrEmptyEnemy(int Obj1, int Obj2, int Order)

{

Object O = new Object();

lock (O)

{

bool Is = false;

if (Order != AllDraw.OrderPlate)

{

if (Order == 1)

{

if (Obj1 > 0 && Obj2 == 0)

Is = true;

}

else

{

if (Obj1 < 0 && Obj2 == 0)

Is = true;

}

}

return Is;

}

}

bool SignEqual(int Obj1, int Obj2)

{

Object O = new Object();

lock (O)

{

bool Is = false;

if (Obj1 > 0 && Obj2 > 0)

Is = true;

if (Obj1 < 0 && Obj2 < 0)

Is = true;

return Is;

}

}

bool SignNotEqual(int Obj1, int Obj2)

{

Object O = new Object();

lock (O)

{

bool Is = false;

if (Obj1 < 0 && Obj2 > 0)

Is = true;

if (Obj1 > 0 && Obj2 < 0)

Is = true;

return Is;

}

}

}

}

//End of Documentation.