

Politechnika Świętokrzyska	
Programowanie systemów rozproszonych - Projekt	
Gra warcaby	Simlat Arkadiusz

1. Założenia projektowe

Napisać program do gry w warcaby. Program powinien umożliwić przeprowadzenie gry w warcaby. Zastosować rozproszony algorytm sztucznej inteligencji do przewidywania ruchów i kalkulowania zysków i strat. Gracz powinien móc określić poziom trudności. Przeprowadzić testy wydajnościowe i jakościowe.

Minimalne wymagania programowe:

- System operacyjny: Windows XP
- Platforma programistyczna: .NET Framework 4.0

2. Propozycja realizacji

2.1 Wstęp teoretyczny

Zasady gry warcaby:

- gra warcaby rozgrywana jest na planszy składającej się na przemian z jasnych i ciemnych pól,
- gra rozgrywana jest na ciemnych polach planszy o rozmiarze 8×8 lub 10×10 pól,
- każdy gracz rozpoczyna grę z dwunastoma (lub dwudziestoma) pionami (jeden koloru białego, drugi czarnego) ustawionymi na ciemnych polach planszy,
- jako pierwszy ruch wykonuje grający pionami białymi, po czym gracze wykonują na zmianę kolejne ruchy,
- celem gry jest zabicie wszystkich pionów przeciwnika albo zablokowanie wszystkich, które pozostają na planszy, pozbawiając przeciwnika możliwości wykonania ruchu,
- piony mogą poruszać się o jedno pole do przodu po przekątnej (na ukos) na wolne pola,
- bicie pionem następuje przez przeskoczenie sąsiedniego pionu (lub damki) przeciwnika na pole znajdujące się tuż za nim po przekątnej (pole to musi być wolne). Zbite piony są usuwane z planszy po zakończeniu ruchu,
- piony mogą bić zarówno do przodu, jak i do tyłu,
- w jednym ruchu wolno wykonać więcej niż jedno bicie tym samym pionem, przeskakując przez kolejne piony przeciwnika,
- bicia są obowiązkowe,
- pion, który dojdzie do ostatniego rzędu planszy, staje się damką, przy czym jeśli w jednym ruchu w wyniku wielokrotnego bicia przejdzie przez ostatni rząd (linię przemianą), ale nie zakończy na niej ruchu, to nie staje się damką i kończy ruch jako pionek,
- kiedy pion staje się damką, kolej ruchu przypada dla przeciwnika,
- damki mogą poruszać się w jednym ruchu o dowolną liczbę pól do przodu lub do tyłu po przekątnej, zatrzymując się na wolnych polach,
- bicie damką jest możliwe z dowolnej odległości po linii przekątnej i następuje przez przeskoczenie pionu (lub damki) przeciwnika, za którym musi znajdować się co

najmniej jedno wolne pole - damka przeskakuje na dowolne z tych pól i może kontynuować bicie (na tej samej lub prostopadłej linii),

- podczas bicia nie można przeskakiwać więcej niż jeden raz przez ten sam pion.

Algorytm mini-max:

- algorytm jest standardowo stosowany we wszelkich programach grających w gry planszowe i nie tylko,
- celem algorytmu jest wskazanie w danej sytuacji ruchu dającego graczowi strategię wygrywającą, lub przynajmniej dającego możliwie duże prawdopodobieństwo zwycięstwa.
- nazwa algorytmu bierze źródło od przyjętej w nim strategii postępowania: symulujemy na zmianę ruchy własne i przeciwnika, gdy znajdujemy się na poziomie przeciwnika wybieramy ruch najgorszy dla gracza, natomiast na poziomie gracza wybieramy ruch dla niego najlepszy,
- analizując drzewo gry w ten sposób, nie da się dojść do liści drzewa, tj. do sytuacji w której gra jest rozegrana,
- konieczne jest zastosowanie heurystycznych funkcji oceniających stan gry,
- drzewo gry przeszukiwane jest do pewnego ustalonego poziomu na którym stan gry jest ustalany przez zadaną funkcję.

Pseudokod algorytmu mini-max:

```
MINI-MAX(sytuacja, gracz, głębokość)
  if głębokość > limit_głębokości then
    return OCENĄ_SYTUACJĘ(sytuacja)
  r = WYGENERUJ_MOŻLIWE_RUCHY(sytuacja, gracz)
  if gracz = 0
    minimax := 0
  else
    minimax := inf
  foreach x in r
    sytuacja' := WYKONAJ_RUCH(sytuacja, x)
    t := MINI-MAX(sytuacja, PRZECIWNIK(gracz), głębokość + 1)
    if gracz = 0 and t > minimax then
      minimax:=t
    else if gracz = 1 and t < minimax then
      minimax:=t
  return minimax
```

Algorytm alfa-beta:

Algorytm alfa-beta jest algorytmem przeszukującym, redukującym liczbę węzłów, które muszą być rozwiązywane w drzewach przeszukujących przez algorytm mini-max.. Warunkiem stopu jest znalezienie przynajmniej jednego rozwiązania czyniącego obecnie badaną opcję ruchu gorszą od poprzednio zbadanych opcji. Wybranie takiej opcji ruchu nie przyniosłoby korzyści graczowi ruszającemu się, dlatego też nie ma potrzeby przeszukiwać dalej gałęzi drzewa tej opcji. Ta technika pozwala zaoszczędzić czas poszukiwania bez zmiany wyniku działania algorytmu. Korzyść płynąca z algorytmu alfa-beta leży w fakcie, że niektóre gałęzie drzewa przeszukiwania mogą zostać odcięte. Czas przeszukiwania ograniczony zostaje do przeszukania najbardziej obiecujących poddrzew, w związku z czym możemy zejść głębiej w tym samym czasie. Tak samo jak klasyczny min-max, algorytm należy do algorytmów

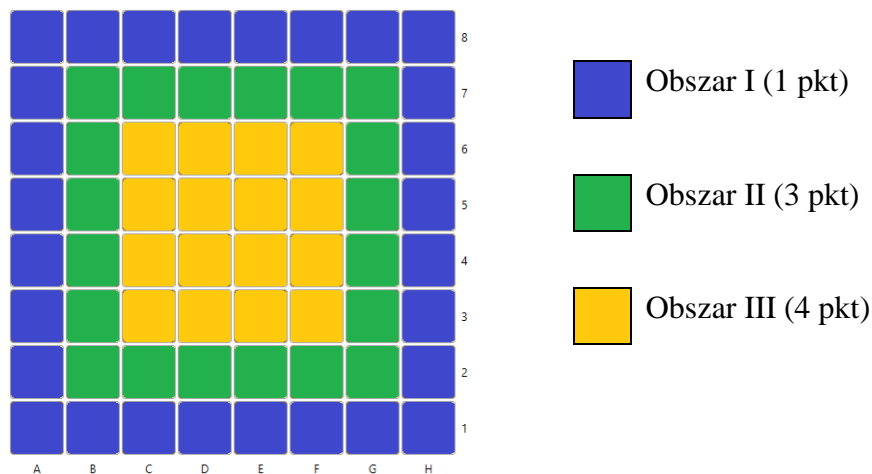
wykorzystujących metody podziału i ograniczeń (branch and bound). Współczynnik rozgałęzienia jest dwukrotnie mniejszy niż w metodzie min-max.

Pseudokod algorytmu alfa-beta:

```
funkcja alfabeta(węzeł, głębokość,  $\alpha$ ,  $\beta$ )
  jeżeli węzeł jest końcowy lub głębokość = 0
    zwróć wartość heurystyczną węzła
  jeżeli przeciwnik ma zagrać w węźle
    dla każdego potomka węzła
       $\beta := \min(\beta, \text{alfabeta}(\text{potomek}, \text{głębokość}-1, \alpha, \beta))$ 
      jeżeli  $\alpha \geq \beta$ 
        przerwij przeszukiwanie {odcinamy gałąź Alfa}
    zwróć  $\beta$ 
  w przeciwnym przypadku {my mamy zagrać w węźle}
    dla każdego potomka węzła
       $\alpha := \max(\alpha, \text{alfabeta}(\text{potomek}, \text{głębokość}-1, \alpha, \beta))$ 
      jeżeli  $\alpha \geq \beta$ 
        przerwij przeszukiwanie {odcinamy gałąź Beta}
    zwróć  $\alpha$ 
```

Krawędziowa funkcja oceniająca stan gry:

Funkcja krawędziowa polega na zapobieganiu poruszania się pionków w skrajnych polach szachownicy, gdyż zmniejsza to znacznie ich możliwości ruchowe. Punktujemy każdego pionka znajdującego się w obszarze niebieskim, zielonym, żółtym. Pionki znajdujące się w obszarze III są najlepiej punktowane a w obszarze I najgorzej.



2.2 Opis rozwiązania

Zastosowany algorytm:

W naszej aplikacji zastosowaliśmy modyfikację algorytmu alfa-beta. Modyfikacja polega na tym, że rekurencyjną wersję algorytmu zamieniliśmy na przygotowaną przez nas, która wykorzystuje klasę Stack. Użyta funkcja oceniająca jest połączeniem funkcji krawędziowej oraz prostej funkcji, która na podstawie liczby pionków oraz damek pozostałych na planszy dodaje do otrzymanego wyniku odpowiednią liczbę punktów.

Interfejs aplikacji:

Interfejsy aplikacji zostały stworzone za pomocą technologii Windows Presentation Foundation (WPF).

Komunikacja:

Komunikacja została stworzona za pomocą technologii Windows Communication Foundation (WCF).

Aplikacja została podzielona na trzy programy:

- Klient – program komunikujący się z użytkownikiem, który wizualizuje przebieg rozgrywki. Wysyła żądania do serwera.
- Serwer – Otrzymuje żądania od klientów. Dzieli główne zadanie na odpowiednią liczbę podzadań (liczba podzadań zależy od liczby węzłów), przydziela zadania do węzłów, a następnie zwraca wynik do klienta.
- Worker – Otrzymuje żądania od serwera. Wykonuje potrzebne obliczenia i zwraca wynik do serwera.

3. Realizacja

Logika gry została umieszczona w przygotowanej przez nas bibliotece GameCore.

Klasa *Game* – klasa kontrolująca przebieg rozgrywki.

Klasa *Board* – klasa używana do opisu stanu planszy. Zawiera metody do wykonywania podstawowych operacji na planszy. Przechowuje podstawowe informacje takie jak:

- liczba pionków oraz damek graczy,
- kolor gracza i komputera,
- kto teraz powinien wykonać ruch,
- rozmiar planszy,
- pozycje pionków na planszy.

Pozycje pionków przechowywane są w tablicy typu `int`. Jedna wartość typu `int` opisuje jeden wiersz na planszy. Każde pole opisane jest za pomocą 3 bitów:

- 0 – puste
- 1 – dama
- 2 – biały
- 4 - czarny

Klasa *MoveController* – klasa używana do kontroli wykonywanych ruchów. Klasa pozwala na sprawdzenie czy ruch, który gracz chce wykonać jest dozwolony. Przechowuje ona stan aktualnie wykonywanego ruchu przez gracza. Gdy ruch wykonany przez gracza jest ostatnim, który może wykonać w obecnej turze, wykonuje go.

Klasa *MoveFinder* – klasa odpowiadająca za odnajdywanie wszystkich dostępnych scenariuszy ruchów dla podanego koloru pionków. Główną metodą tej klasy jest:

```
public List<MoveScenarios> getAllowMoveScenarios(Board board, int color)
```

Metoda ta zwraca wszystkie dostępne bicia lub w przypadku ich braku dostępne ruchy.

Metoda służąca do odnajdywania bić w określonym kierunku:

```
private List<Position> findCapture(Board board, Direction direction,
    Position fromPosition, int piece)
{
    List<Position> foundCapture = new List<Position>();

    Position currPos = new Position(fromPosition);
    bool pieceIsDame = Board.isDame(piece);
    int maxMoveOffset = pieceIsDame ? board.Size : 1;
    int currMoveOffset = 0;

    while (currMoveOffset < maxMoveOffset)
    {
        currPos.move(direction);

        if (!currPos.isInRange(0, board.Size))
            break;

        if (!Board.isEmpty(board.getPiece(currPos)))
        {
            if (!Board.matchPieceColor(board.getPiece(currPos), piece))
            {
                currPos.move(direction);

                if (currPos.isInRange(0, board.Size) &&
                    Board.isEmpty(board.getPiece(currPos)))
                {
                    foundCapture.Add(new Position(currPos));

                    if (pieceIsDame)
                    {
                        foundCapture.AddRange(getEmptyPlaces(board, direction,
                            currPos, board.Size));
                    }

                    break;
                }
            }
            else
            {
                break;
            }
        }

        currMoveOffset++;
    }

    return foundCapture;
}
```


Działanie metody:

Metoda zaczyna poszukiwania od pozycji *fromPosition* i przesuwa się w kierunku *direction* na odległość nie większą niż *maxMoveOffset* (dla pionka wynosi 1, natomiast dla damki jest równy rozmiarowi planszy). Funkcja sprawdza czy na polu o aktualnych współrzędnych znajduje się pionek. Jeżeli tak sprawdza czy pionek jest innego koloru niż ten którym się porusza (*piece*). Pozostaje jedynie sprawdzić czy następne pole jest polem dozwolonym oraz pustym. Spełnienie tego warunku oznacza znalezienie możliwości bicia.

Przedstawiona metoda służy do odnajdywania pojedynczych bić. Do wyszukiwania scenariuszy wszystkich możliwych bić służy metoda:

private MoveScenarios getAllCaptureScenarios(Board board, Position basePosition).

```
private MoveScenarios getAllCaptureScenarios(Board board, Position basePosition)
{
    int piece = board.getPiece(basePosition);
    Stack<MoveOptionsContext> stack = new Stack<MoveOptionsContext>();
    List<List<Position>> allScenarios = new List<List<Position>>();
    List<Position> currScenario = new List<Position>();
    MoveOptionsContext currContext;
    MoveOptions currMoveOpt;

    MoveOptions rootMoveOpt = new MoveOptions(new Board(board), basePosition);

    stack.Push(new MoveOptionsContext(rootMoveOpt));

    while (stack.Count > 0)
    {
        currContext = stack.Pop();
        currMoveOpt = currContext.CurrMoveOpt;

        if (!currContext.IsVisited)
        {
            if (!currMoveOpt.Equals(rootMoveOpt))
                currScenario.Add(currMoveOpt.CurrentPos);

            currContext.IsVisited = true;
            stack.Push(currContext);

            List<Position> foundCapture = findCaptures(currMoveOpt.CurrentBoard,
                currMoveOpt.CurrentPos, piece);
        }
    }
}
```

```

        foreach (Position currCapturePos in foundCapture)
        {
            Board newBoardState = new Board(currMoveOpt.CurrentBoard);
            MoveController.execMove(newBoardState, currMoveOpt.CurrentPos,
                currCapturePos);

            MoveOptions captureMoveOpt = new MoveOptions(newBoardState,
                currCapturePos);
            currMoveOpt.Add(captureMoveOpt);

            stack.Push(new MoveOptionsContext(captureMoveOpt));
        }
    }
    else
    {
        if (currMoveOpt.AllMoveOptPos.Count == 0 && currScenario.Count > 0)
        {
            List<Position> leafPath = new List<Position>();
            leafPath.AddRange(currScenario);

            allScenarios.Add(leafPath);
        }

        if (currScenario.Count > 0)
        {
            currScenario.RemoveAt(currScenario.Count - 1);
        }
    }
}

if (allScenarios.Count == 0)
    return null;

return new MoveScenarios(basePosition, allScenarios, true);
}

```

Przedstawiona metoda wykorzystuje do działania metodę wcześniejszą. Metoda działa w oparciu o stos. Dla każdego znalezionej bicia szuka kolejnych do momentu aż wszystkie możliwości zostaną odnalezione. Podczas szukania jednocześnie tworzona jest ścieżka składająca się z współrzędnych dla aktualnie wykonywanego ruchu.

Działanie metod do wyszukiwania ruchów jest bardzo proste gdyż sprowadza się jedynie do odnajdywania pustych pól i dlatego zostaną one pominięte.

Klasa *MoveScenarios* – przechowuje wszystkie scenariusze ruchów dla pojedynczego pionka.

```
public class MoveScenarios
{
    private bool isCaptureScenario;
    private Position fromPosition;
    private List<List<Position>> allMoveSconarios;

    . . .
}
```

Pole:

- *isCaptureScenario* informuje czy scenariusze dotyczą bicia,
- *fromPosition* – współrzędne pola na którym znajduje się pionek,
- *allMoveSconarios* – lista zawierająca możliwe scenariusze ruchów. Każdy scenariusz składa się z co najmniej jednego ruchu.

Klasa *MiniMax* – implementacja algorytmu do znajdowania najlepszego ruchu. Aby wykonywanie algorytmu można było rozproszyć określone działania rozstały wyodrębnione do osobnych metod.

Poniżej przedstawiony został kod i opis działania tego algorytmu w wersji nierozproszonej na przykładzie metody *getBestMove()* .

```
public MoveScenarios getBestMove(Board currBoard, int secTimeout)
{
    int maxColor = currBoard.CurrentColor;
    MoveContext rootContext = new MoveContext(currBoard, null, null, 0, null, true);

    findBestMoveValue(rootContext, maxColor, secTimeout);

    if (rootContext.BestValue != int.MinValue)
        return new MoveScenarios(rootContext.CurrPosition,
                                rootContext.CurrMoveScenario);

    return null;
}
```

Działanie metody:

Na początku działania algorytmu tworzony jest tzw. korzeń (*rootContext*), który jest stanem zerowym. Po zakończeniu działania algorytmu obiekt będzie zawierał informację o najlepszym znalezionym posunięciu.

Metoda *findBestMoveValue(rootContext, maxColor, secTimeout)*:

```
public int findBestMoveValue(MoveContext context, int maxColor, int secTimeout)
{
    context.addChildBoardStates(moveFinder);

    Stack<MoveContext> stack = new Stack<MoveContext>();

    while (context.ChildsContext.Count > 0)
    {
        stack.Push(context.ChildsContext.Pop());
    }

    return resolve(stack, maxColor, maxDepth, secTimeout);
}
```

Metoda dla podanego stanu gry tworzy stany potomne, a następnie odkłada je na stos. Stos ze stanami potomnymi przekazywany jest do głównej metody rozwiązującej, po której zakończeniu obiekt rodzic (*context*) będzie zawierał informacje o najlepszym posunięciu. Metoda zwraca czas jaki pozostał po wykonaniu obliczeń. Maksymalny czas obliczeń przekazywany jest za pomocą zmiennej *secTimeout*.

Metoda *public int resolve(Stack<MoveContext> stack, int maxColor, int currMaxDepth, int secTimeout)*:

```
public int resolve(Stack<MoveContext> stack, int maxColor, int currMaxDepth,
    int secTimeout)
{
    CountdownTimer countdownTimer = new CountdownTimer();

    if (secTimeout > 0)
    {
        countdownTimer.Start(secTimeout);
    }

    MoveContext currContext;
    int moveWeight;
```

```

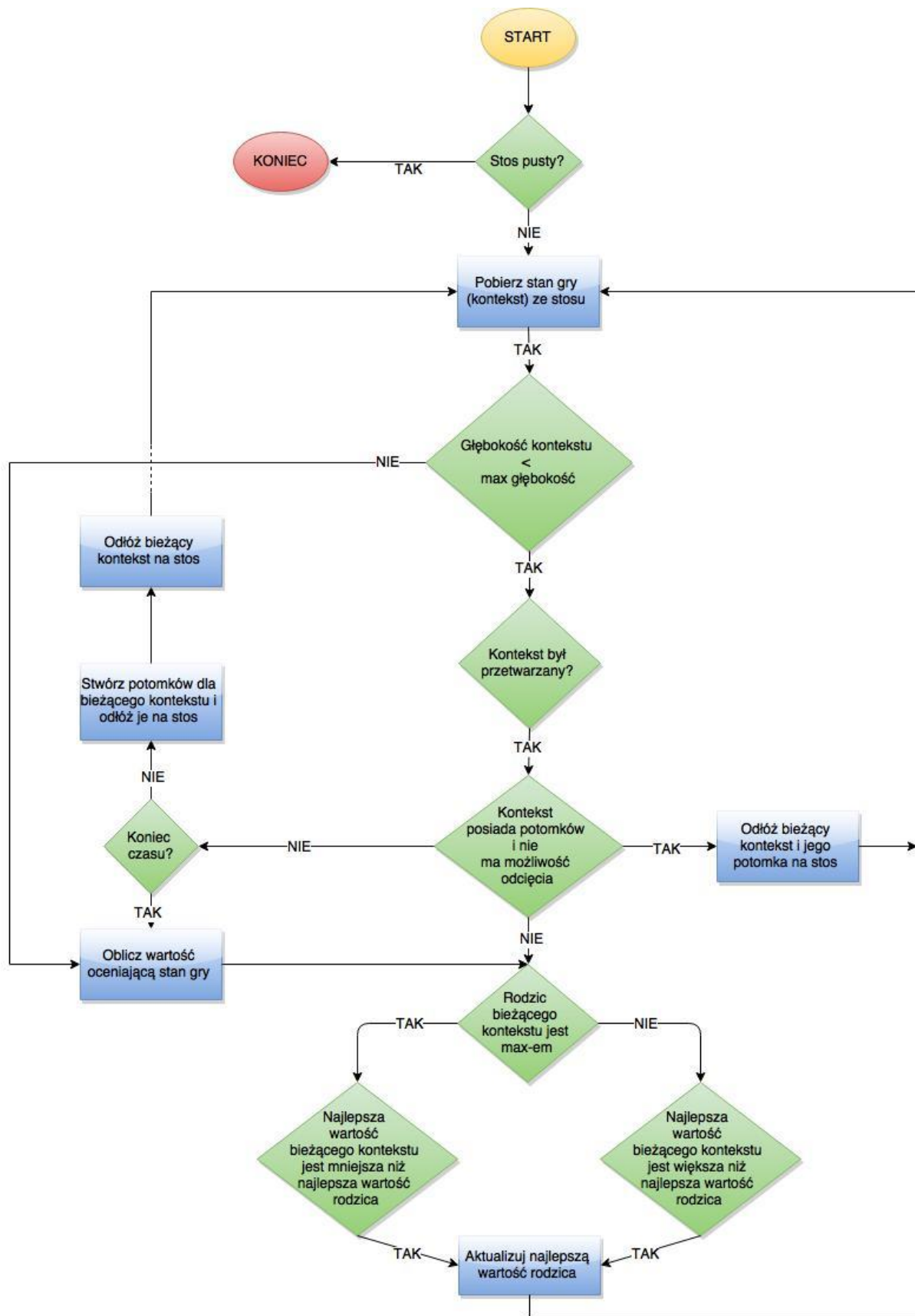
while (stack.Count > 0)
{
    currContext = stack.Pop();

    if (currContext.CurrDepth < currMaxDepth)
    {
        if (currContext.IsVisited)
        {
            if (currContext.ChildsContext.Count > 0 &&
                !currContext.cutOffPossible())
            {
                stack.Push(currContext);
                stack.Push(currContext.ChildsContext.Pop());
            }
            else
            {
                if (currContext.ParentContext.IsMax)
                {
                    currContext.setParentMax();
                }
                else
                {
                    currContext.setParentMin();
                }
            }
        }
        else if (!countdownTimer.isTimeout())
        {
            currContext.addChildBoardStates(moveFinder);
            stack.Push(currContext);
        }
        else
        {
            moveWeight = getBoardStateWeight(currContext.BoardState, maxColor);
            currContext.setParentValue(moveWeight);
        }
    }
    else
    {
        moveWeight = getBoardStateWeight(currContext.BoardState, maxColor);
        currContext.setParentValue(moveWeight);
    }
}

countdownTimer.Stop();

return countdownTimer.getRemainTime();
}

```



Dodatkowo klasa posiada metodę *public int generateMoveContext(MoveContext context, int minCount, List<MoveContext> list)*, która wykorzystywana jest przez serwer do utworzenia *minCount* potomków. Metoda generuje kolejne poziomy drzewa do momentu, aż liczba potoków znajdujących się na danym poziomie będzie wystarczająca

Klasa *MoveContext* – jest wykorzystywana do przechowywania informacji o bieżącym kontekście wykonania algorytmu mini-max.

```
public class MoveContext
{
    public Board BoardState { get; set; }
    public Position CurrPosition { get; set; }
    public List<Position> CurrMoveScenario { get; set; }
    public int CurrDepth { get; set; }
    public bool IsVisited { get; set; }
    public int BestValue { get; set; }
    public MoveContext ParentContext { get; set; }
    public Stack<MoveContext> ChildsContext { get; set; }
    public bool IsMax { get; private set; }
```

- *BoardState* – bieżący stan planszy,
- *CurrPosition* – bieżące położenie pionka na planszy,
- *CurrMoveScenario* – bieżący scenariusz ruchu,
- *CurrDepth* – głębokość wywołania kontekstu,
- *IsVisited* – czy kontekst był przetwarzany/odwiedzony,
- *BestValue* – bieżąca najlepsza wartość stanu gry,
- *ParentContext* – referencja do kontekstu rodzica,
- *ChildsContext* – stany potomne.

Metoda do dodawania stanów potomnych:

```
public bool addChildBoardStates(MoveFinder moveFinder)
{
    List<MoveScenarios> allScenarios = moveFinder.getAllowMoveScenarios(BoardState,
        BoardState.CurrentColor);

    foreach (MoveScenarios currScenario in allScenarios)
    {
        for (int i = 0; i < currScenario.Count(); i++)
        {
            Board newBoardState = new Board(BoardState);
            MoveController.execScenario(newBoardState, currScenario, i);

            ChildsContext.Push(
                new MoveContext(
                    newBoardState,
                    currScenario.getFromPosition(),
                    currScenario.getScenario(i),
                    CurrDepth + 1,
                    this,
                    !IsMax
                )
            );
        }
    }

    IsVisited = true;

    if (allScenarios.Count > 0)
        return allScenarios[0].isCapture();

    return false;
}
```

- - - SERVER - - -

Główną metodą serwera, która wywoływana jest po stronie klienta jest metoda:
MoveScenarios GetBestMove(Board board, int maxDepth, int secTimeout);


```

public MoveScenarios GetBestMove(Board board, int maxDepth, int secTimeout)
{
    MiniMax miniMax = new MiniMax(maxDepth);
    WorkerManager workerManager = WorkerManager.GetInstance();

    if (workerManager.getWorkersCount() == 0)
    {
        return miniMax.GetBestMove(board, secTimeout);
    }

    int maxColor = board.CurrentColor;
    MoveContext rootContext = new MoveContext(board, null, null, 0, null, true);

    Stack<MoveContext> mainStack = new Stack<MoveContext>();
    Stack<MoveContext> childsStack = new Stack<MoveContext>();
    List<MoveContext> contextList = new List<MoveContext>();

    int generatedChilds = miniMax.GenerateMoveContext(rootContext,
        workerManager.getWorkersCount(), contextList);

    if (generatedChilds == 0)
    {
        return null;
    }
    else if (generatedChilds == 1)
    {
        return miniMax.GetBestMove(board, secTimeout);
    }
    else if (generatedChilds <= -1)
    {
        foreach (MoveContext mc in contextList)
            mainStack.Push(mc);

        miniMax.Resolve(mainStack, maxColor, 2, -1);
    }
    else
    {
        int i = 0;

        for (; i < contextList.Count - generatedChilds; i++)
        {
            mainStack.Push(contextList[i]);
        }

        for (; i < contextList.Count; i++)
        {
            childsStack.Push(contextList[i]);
        }

        workerManager.SetParameters(childsStack, mainStack, maxDepth, maxColor,
            secTimeout - 15);
        workerManager.RunWorkers();
        workerManager.WaitForWorkers();
        childsStack.Clear();
        miniMax.Resolve(mainStack, maxColor, maxDepth, -1);
    }

    if (rootContext.BestValue != int.MinValue)
    {
        return new MoveScenarios(rootContext.CurrPosition,
            rootContext.CurrMoveScenario);
    }
    else
    {
        return new MiniMax(2).GetBestMove(board, -1);
    }
}

```

Na początku metody sprawdzana jest liczba podłączonych węzłów. Jeżeli brak węzłów serwer samodzielnie szuka najlepszego posunięcia i zwraca wynik do klienta. W przeciwnym wypadku generuje co najmniej tyle stanów potomnych ile węzłów. Wygenerowane stany zostają rozdzielone na dwa stosy. Pierwszy (*mainStack*) jest głównym stosem który zawiera stany z poziomów wyższych niż ten który został wygenerowany jako ostatni. Do tego stosu będą również trafiać stany przeliczone przez węzły. Na końcu stos ten będzie przekazany do metody *resolve()* w celu dokończenia obliczeń. Drugi stos (*childsStack*) zawiera stany dla których należy wykonać dalsze obliczenia. Stos ten jest stosem wejściowym dla wątków zarządzających węzłami.

Klasa *WorkerManager* – służy do utworzenia i sterowania wątkami, które z pomocą węzłów wykonują zadania.

```
public class WorkerManager
{
    private static WorkerManager instance = null;

    private List<WorkerInfo> workersInfo = null;
    private WorkerThread[] workersThread = null;
    private ManualResetEvent[] workersStopEvent = null;
    private Semaphore semaphore = null;
```

Klasa *WorkerThread* – obiekty tej klasy posiadają referencję do pojedynczego węzła. Dla każdego węzła tworzony jest jeden obiekt tej klasy. Metoda wykonująca obliczenia przedstawiona została poniżej.

```

private void doWork()
{
    MoveContext context = null;

    while (true)
    {
        semaphore.WaitOne();

        countdownTimer.Start(secTimeout);

        while (true)
        {
            lock (inStack)
            {
                if (inStack.Count == 0)
                    break;

                context = inStack.Pop();
            }

            context.BestValue = iworker.FindBestValue(context, maxDepth, maxColor,
                countdownTimer.getRemainTime());
            context.IsVisited = true;

            lock (outStack)
            {
                outStack.Push(context);
            }

            if (countdownTimer.isTimeout())
            {
                break;
            }
        }

        countdownTimer.Stop();
        stopEvent.Set();
    }
}

```

- - - WORKER - - -

Metoda wykonywana przez węzeł przedstawiona została poniżej.

```
public int FindBestValue(MoveContext moveContext, int maxDepth, int maxColor,
    int secTimeout)
{
    new MiniMax(maxDepth).findBestMoveValue(moveContext, maxColor, secTimeout);

    return moveContext.BestValue;
}
```

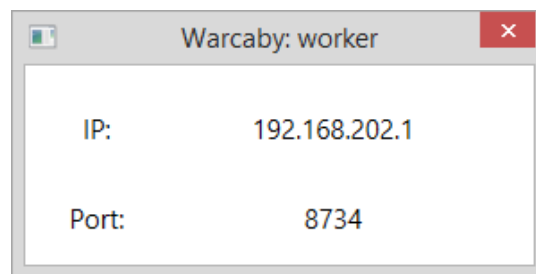
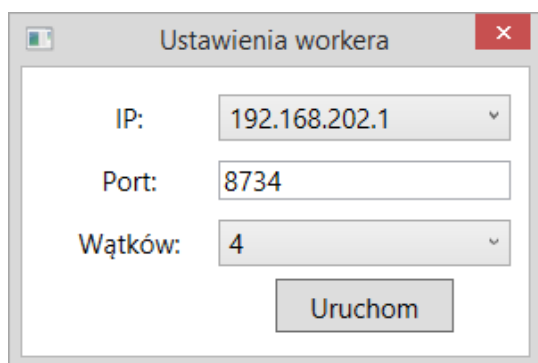
4. Obsługa aplikacji

Aplikacja została przygotowana w taki sposób, aby jej obsługa była jak najprostsza. Aplikacja nie wymaga ręcznego szukania i wpisywania adresów do serwera czy węzła co znacznie ułatwia i przyspiesza pracę z aplikacją. Program automatycznie wyszukuje adresy, które pochodzą z przygotowanej aplikacji. Użytkownik jednak dalej ma możliwość wyboru z którymi adresami chce się łączyć. Interfejs aplikacji klienckiej również został przygotowany w taki sposób, aby potrzebne informacje były widoczne, opcje łatwo dostępne, a gra przyjemna.

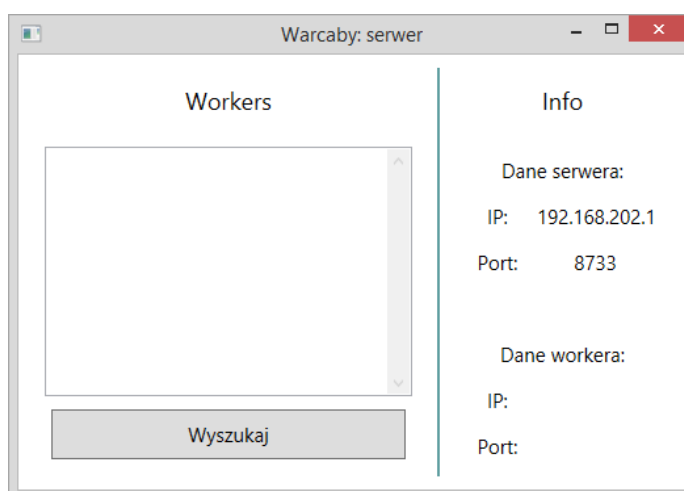
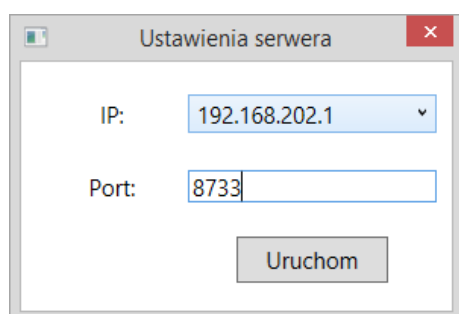
Aby prawidłowo uruchomić aplikację należy uruchomić:

- jedną instancję klienta,
- jedną instancję serwera,
- co najmniej jedną instancję workera.

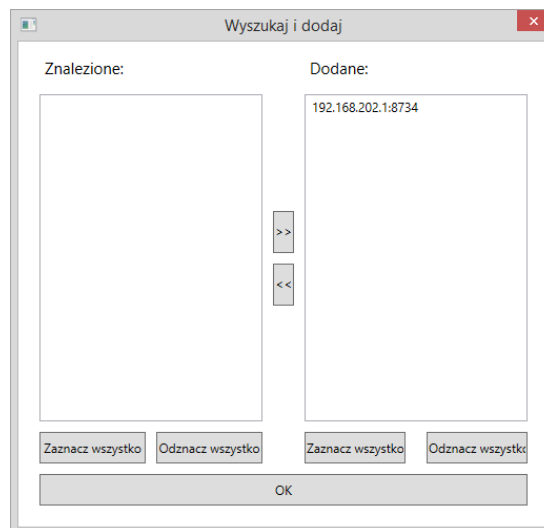
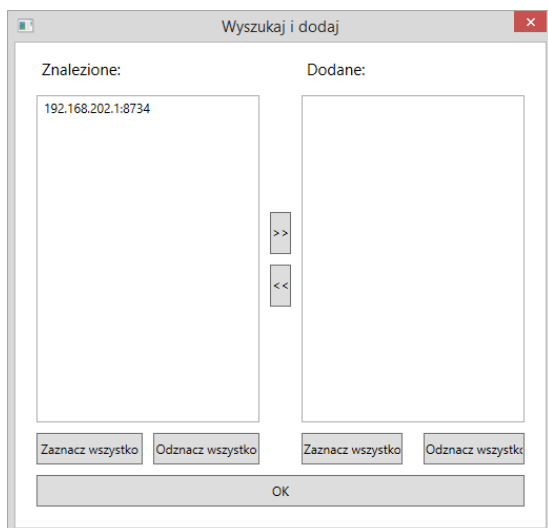
Po uruchomieniu programu workera należy wybrać adres IP z listy dostępnych adresów oraz wpisać numer portu. Jeżeli na jednym komputerze zostanie uruchomiony serwer i worker lub kilka instancji workera o jednakowym IP należy podane numery portów muszą być różne. Na koniec należy nacisnąć przycisk *Uruchom*. Usługa workera zostanie uruchomiona.



Teraz należy uruchomić serwer. Po podaniu wymaganych danych należy kliknąć przycisk Uruchom. Usługa serwera zostanie uruchomiona.

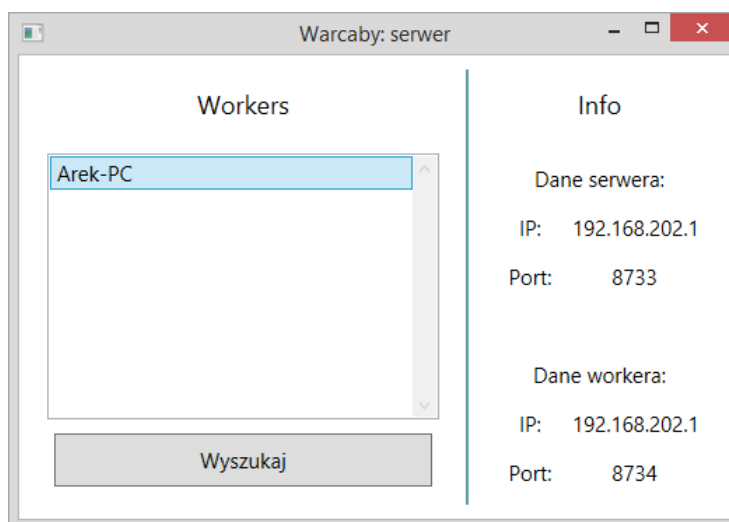


Teraz należy dodać workerów. W tym celu należy wybrać przycisk Wyszukaj. Workerów można również dodawać podczas gry, gdy jest nasza kolej ruchu.

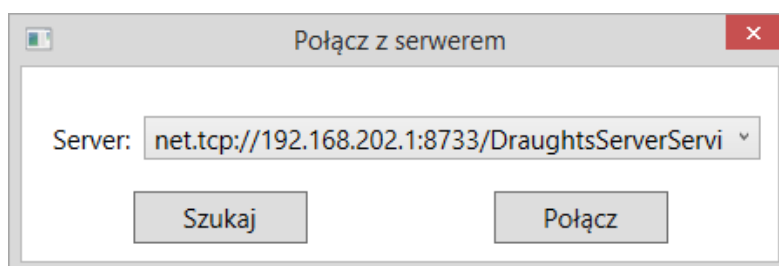


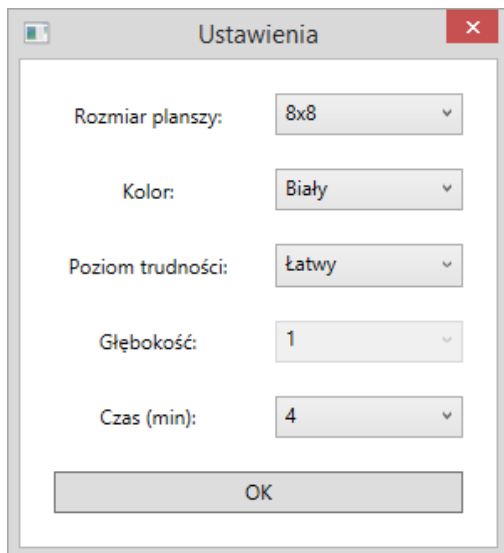
Z listy *Znalezione* należy wybrać te, które mają wykonywać obliczenia. Następnie należy kliknąć przycisk „>>” i zatwierdzić klikając OK.

Dodane węzły widoczne będą na liście *Workers*. Po kliknięciu na wybranego workera, jego szczegółowe informacje będą widoczne w części *Dane workera*.



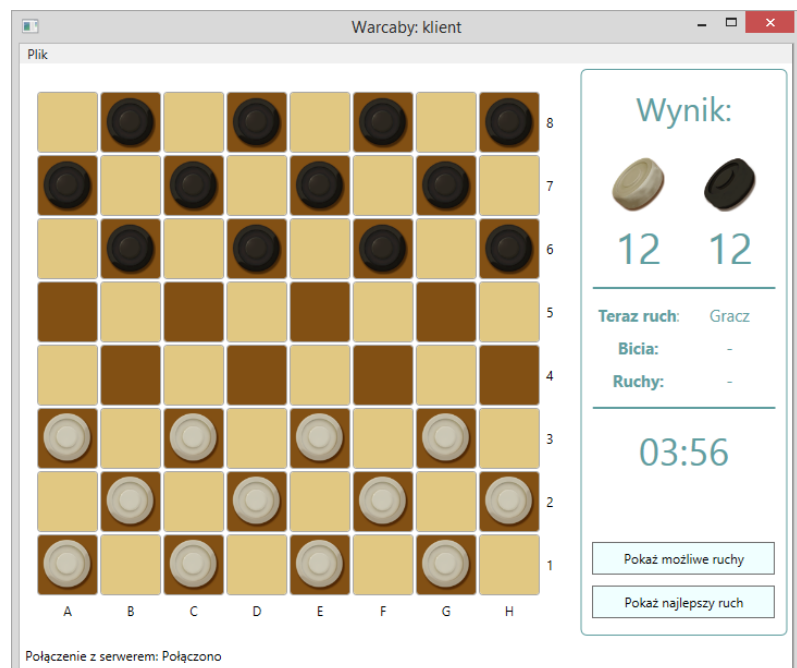
Pozostało jedynie uruchomić aplikację kliencką. Należy kliknąć przycisk *Szukaj*, a następnie wybrać jeden z odnalezionych serwerów i zatwierdzić przyciskiem *Połącz*.





Kolejnym krokiem jest dostosowanie ustawień gry. Zatwierdzamy przyciskiem OK.

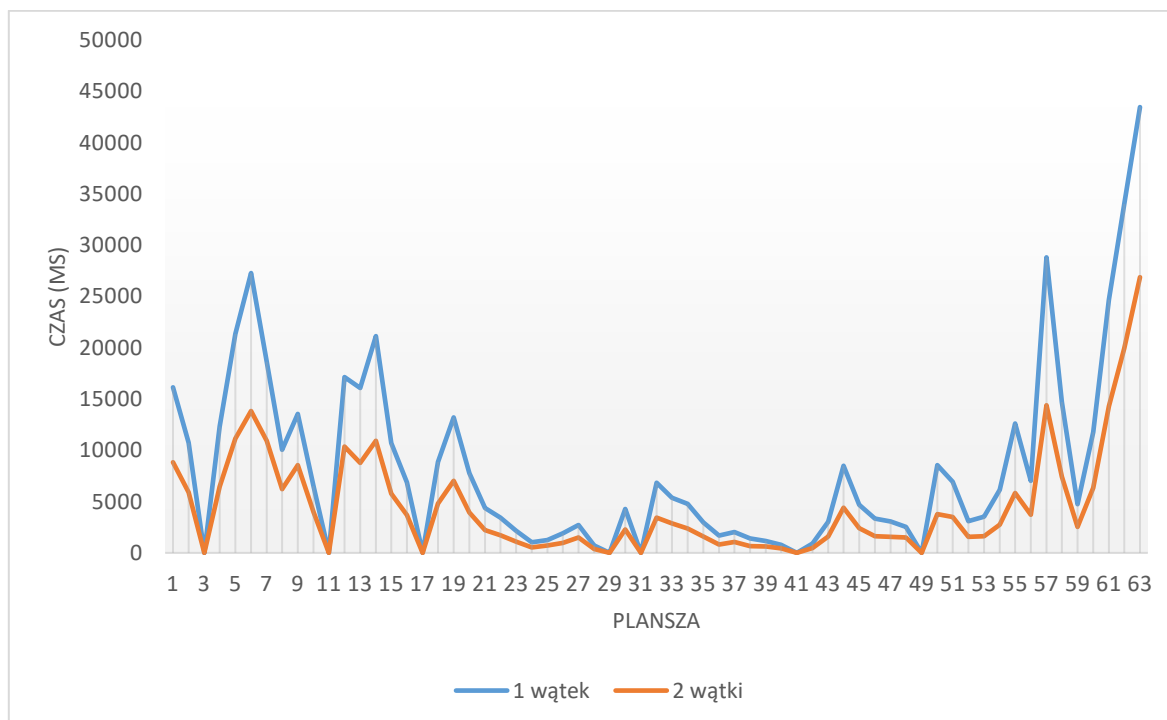
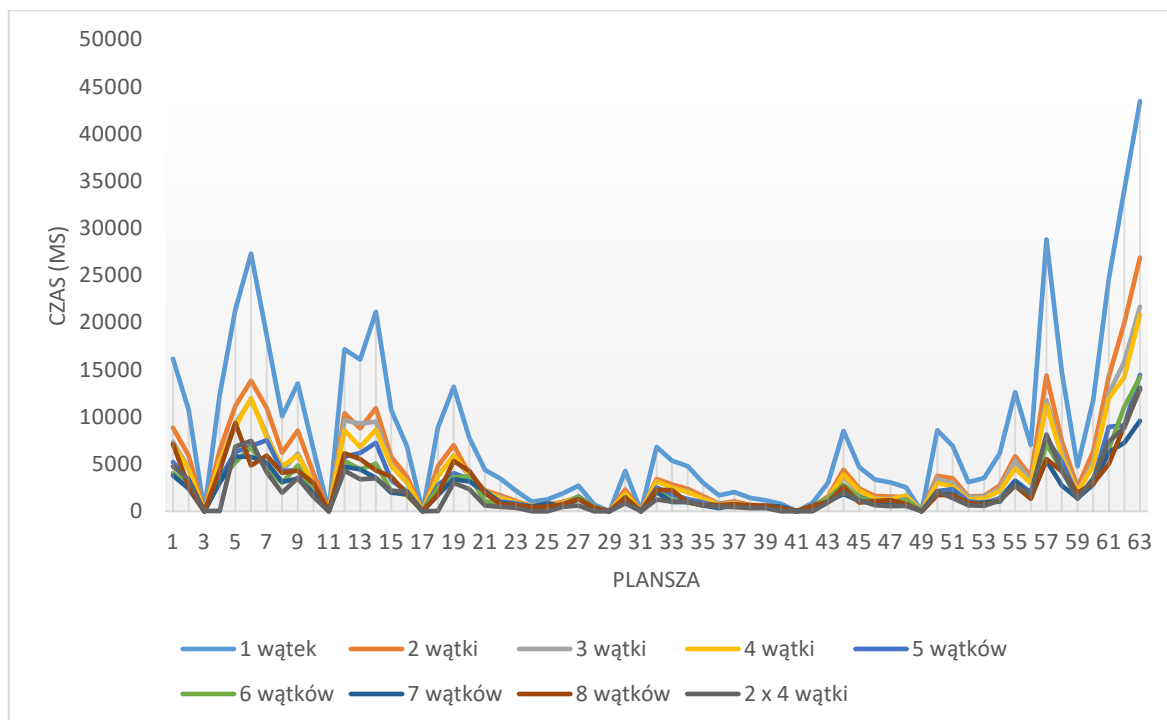
Wszystko gotowe. Gra została uruchomiona.

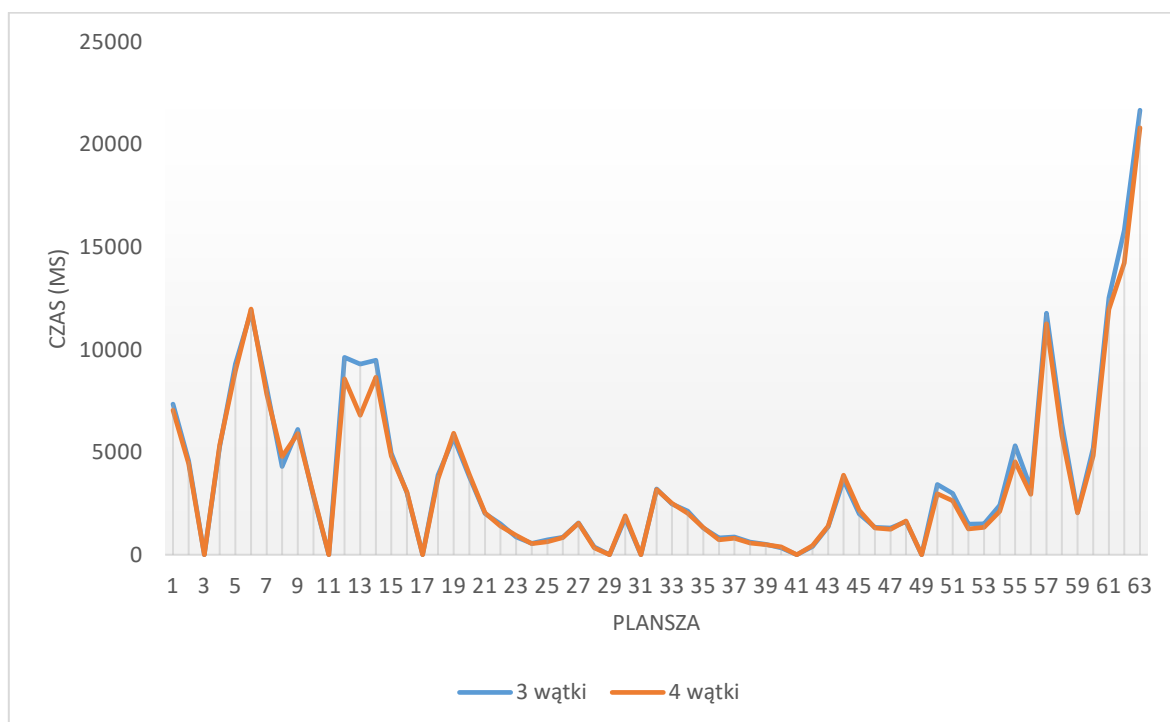
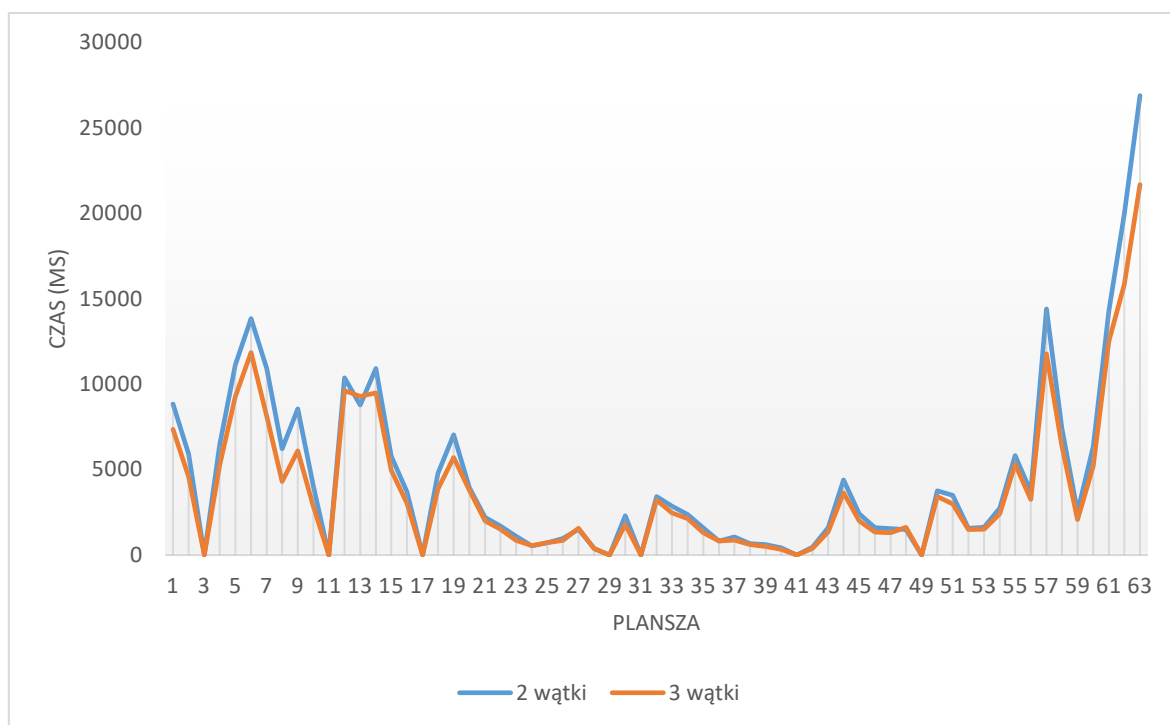


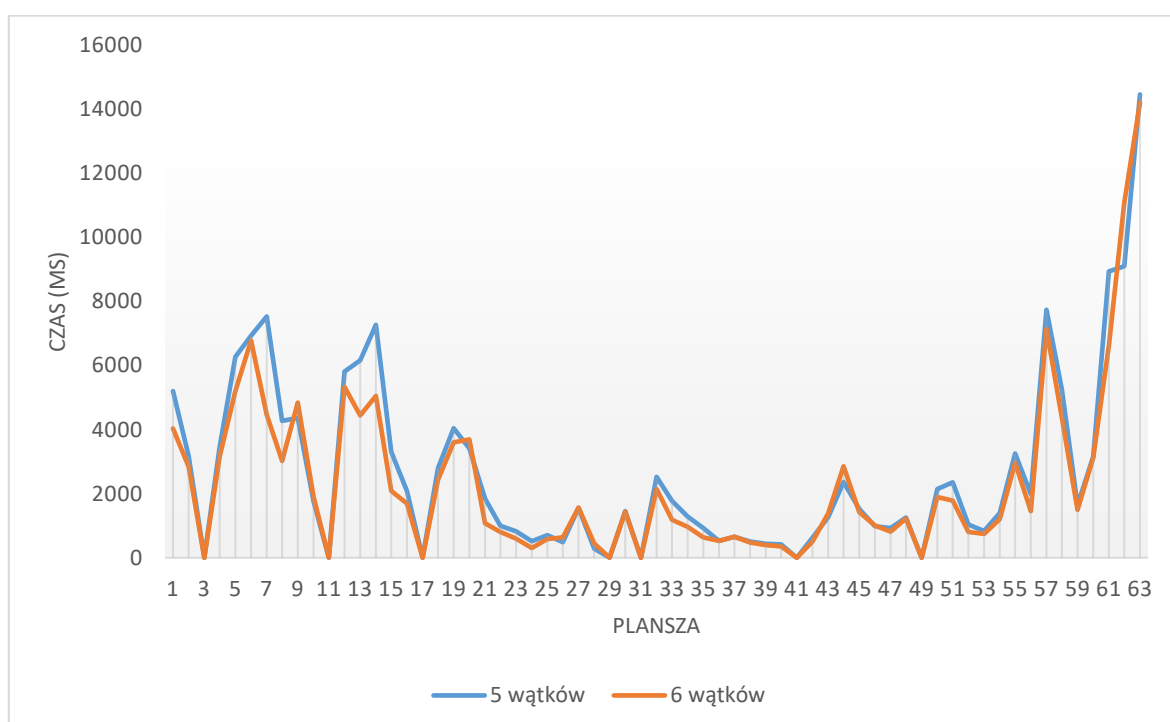
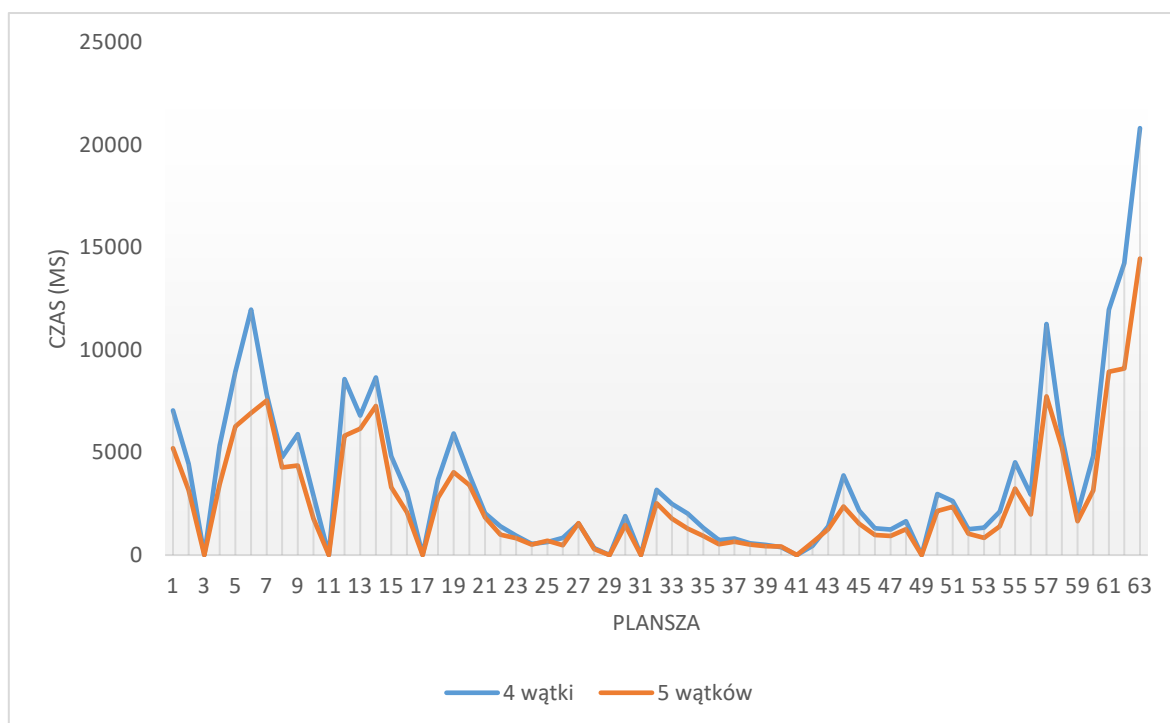
5. Testy aplikacji

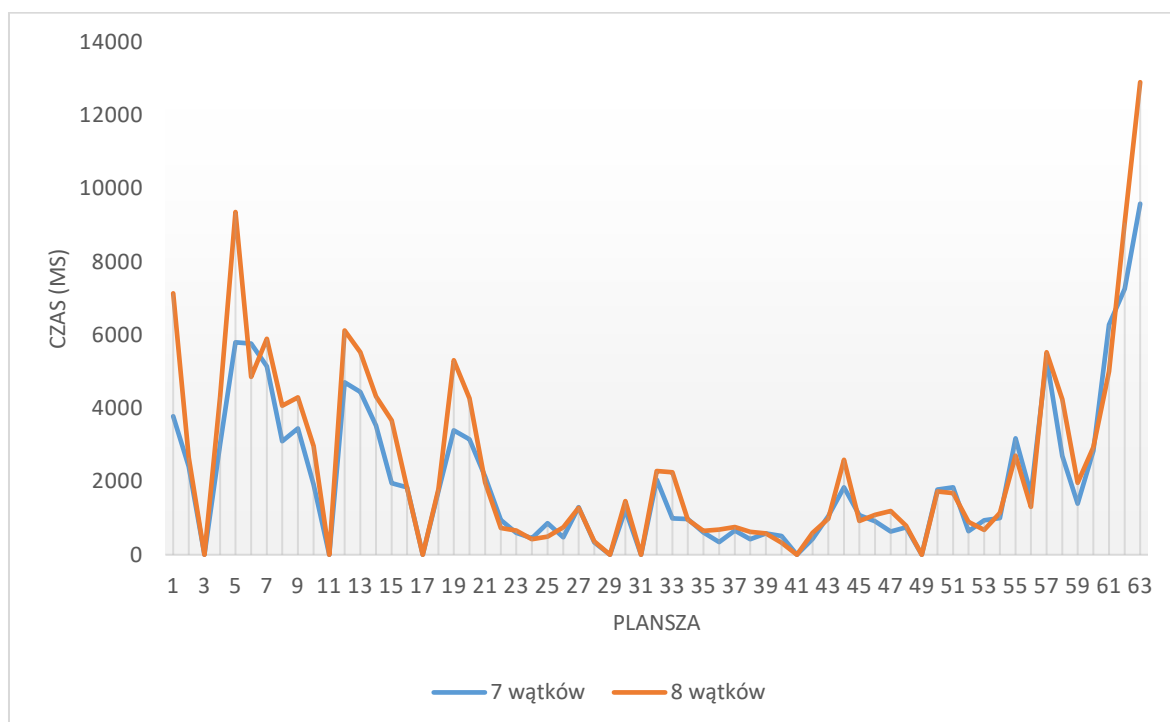
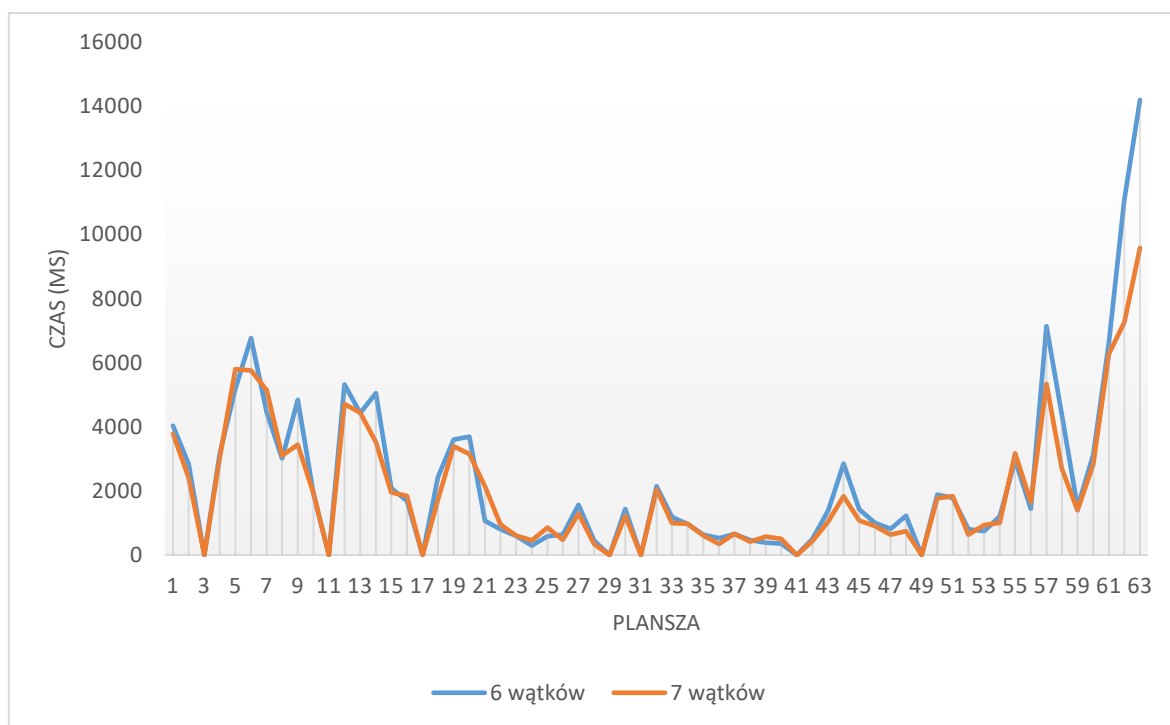
Testy zostały przeprowadzone dla przetwarzania na jednym oraz dwóch komputerach, dla różnej liczby workerów. Worker uruchamiany był jako jeden wątek. Przetwarzane stany planszy były takie same dla każdego testu. Wyniki testów zostały przedstawione poniżej.

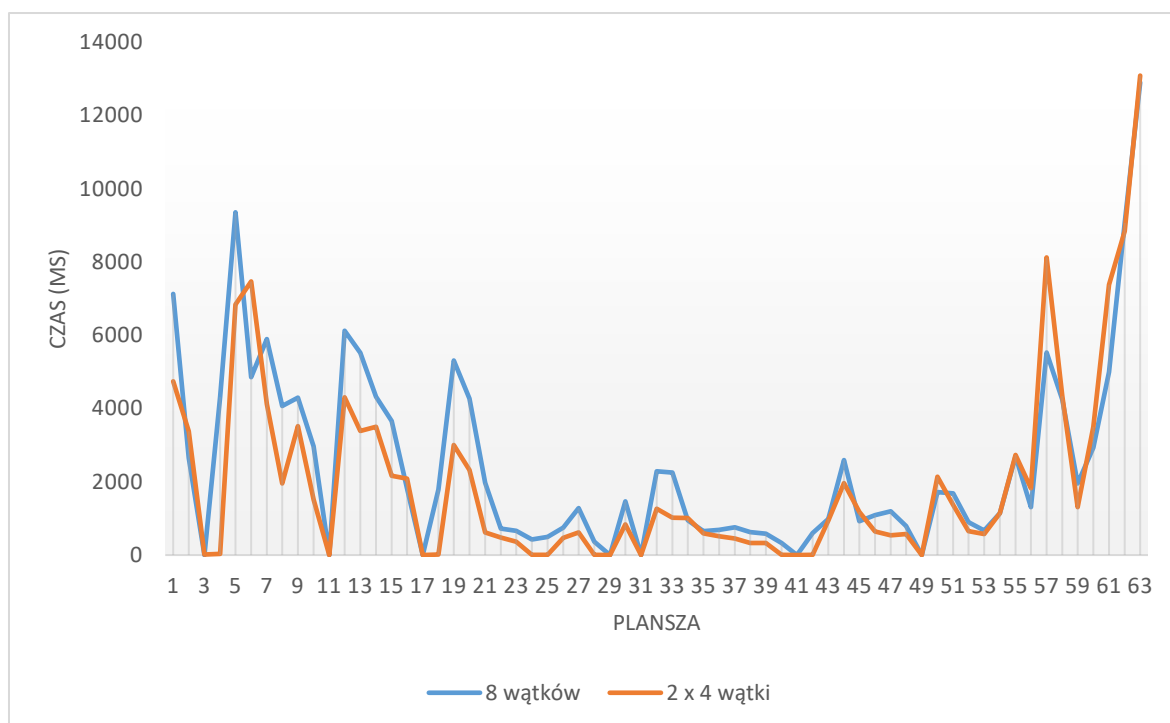
Wyniki testów dla średniego poziomu trudności (głębokość minimax – 9)
na planszy 8x8 pól:



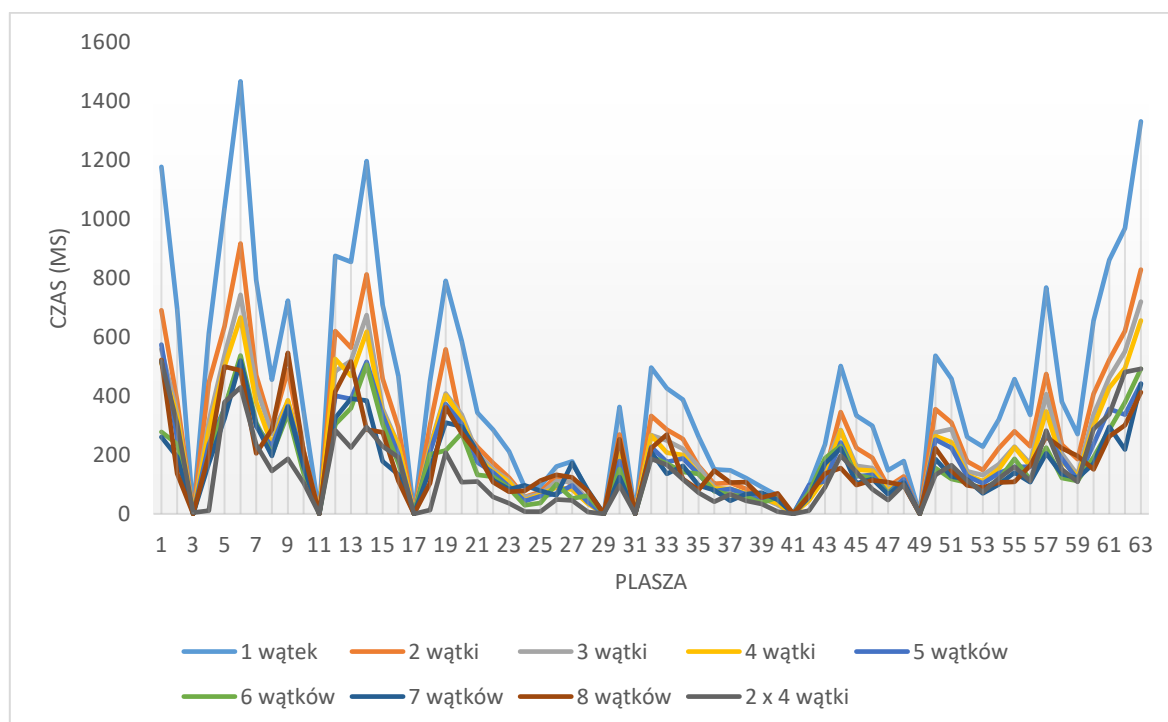


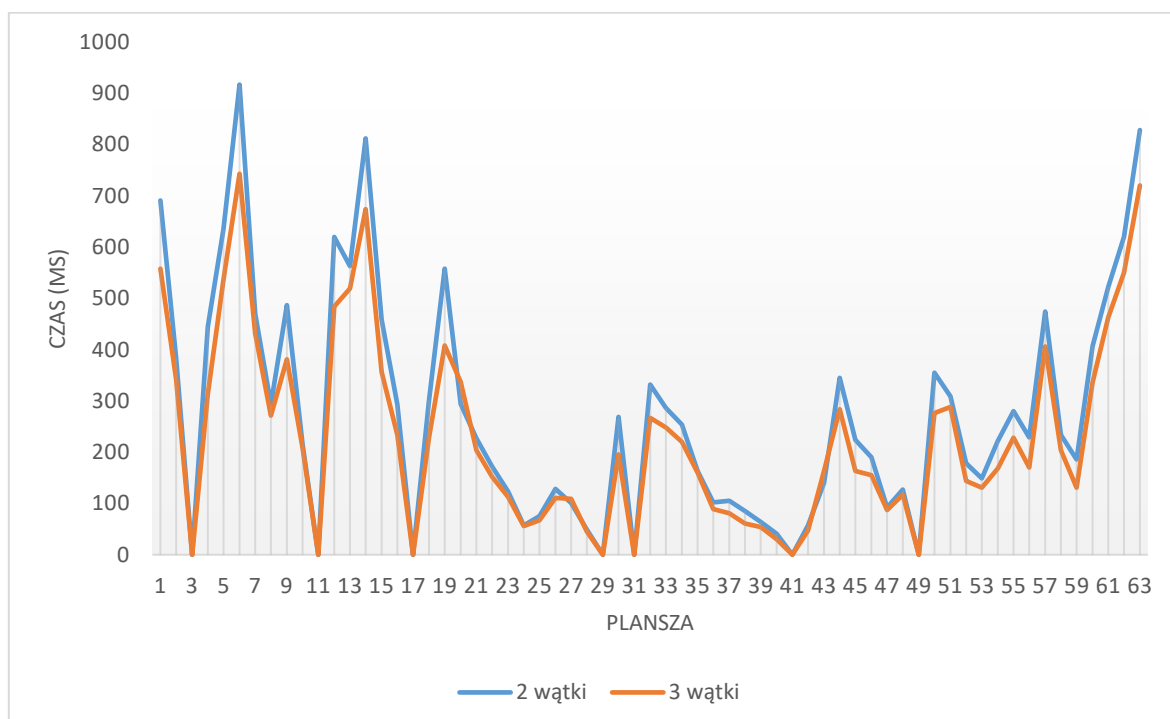
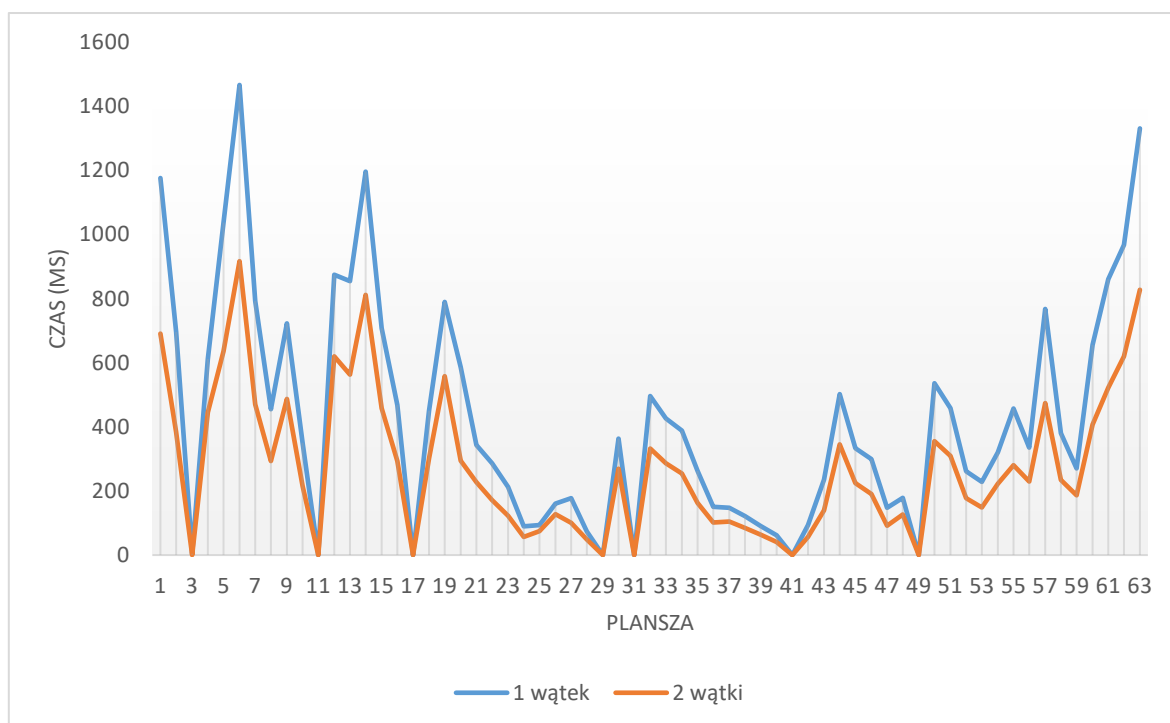


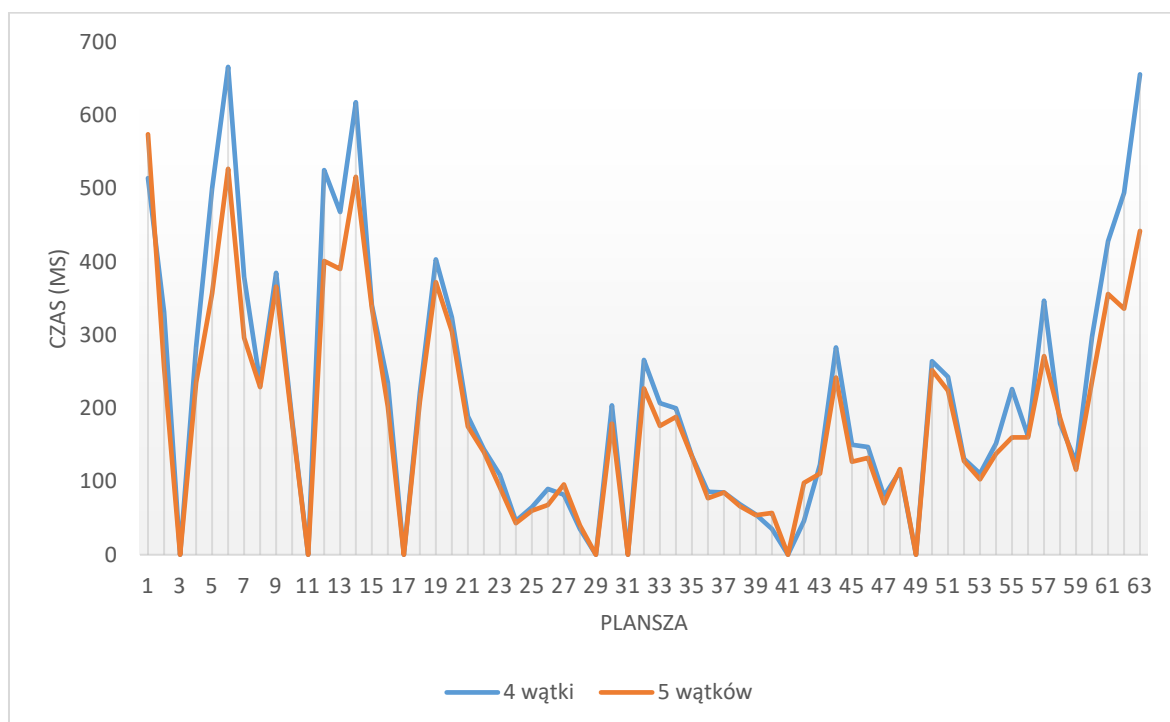
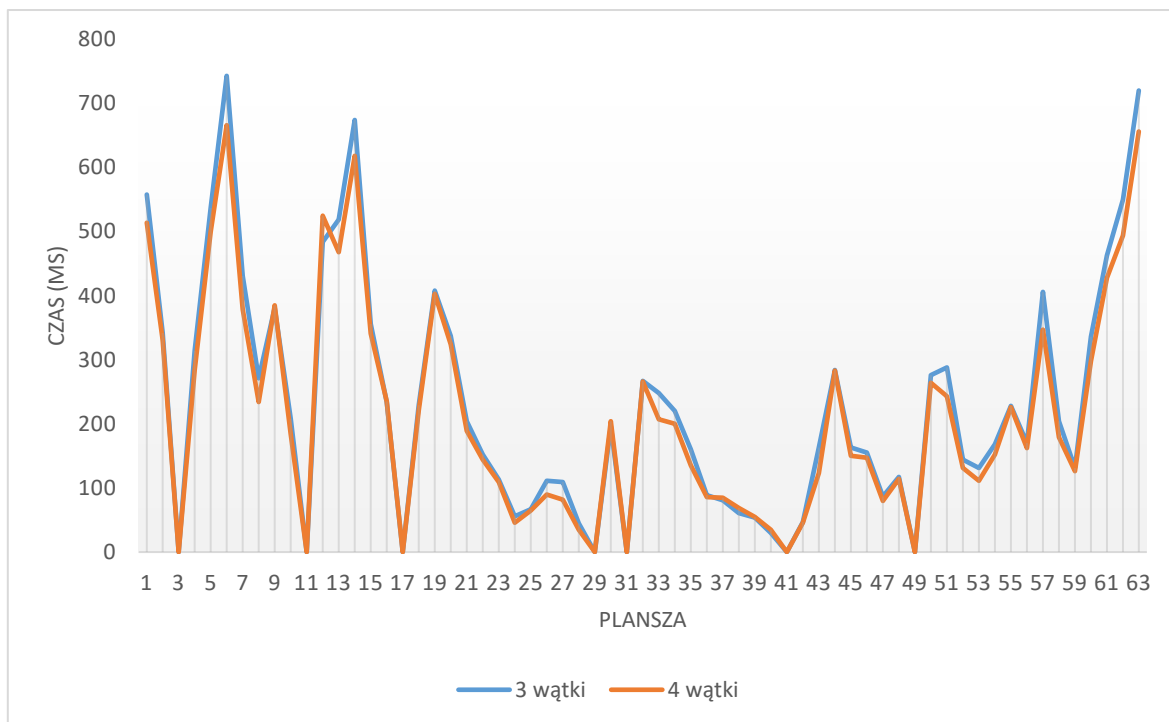


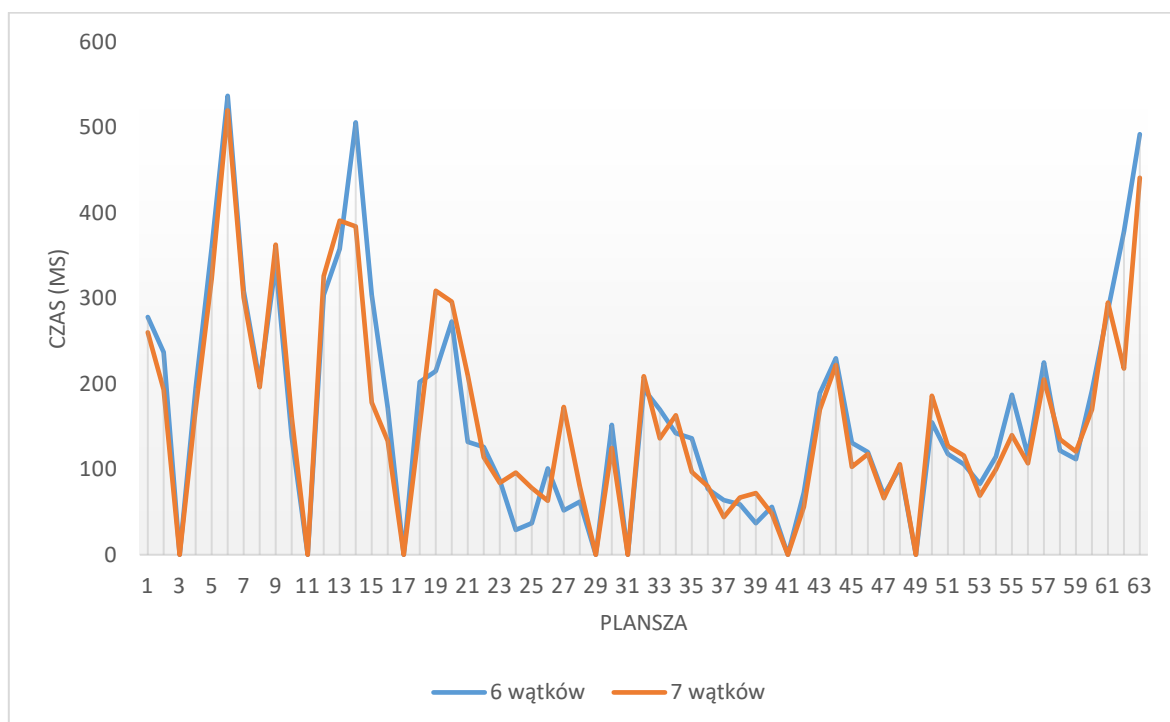
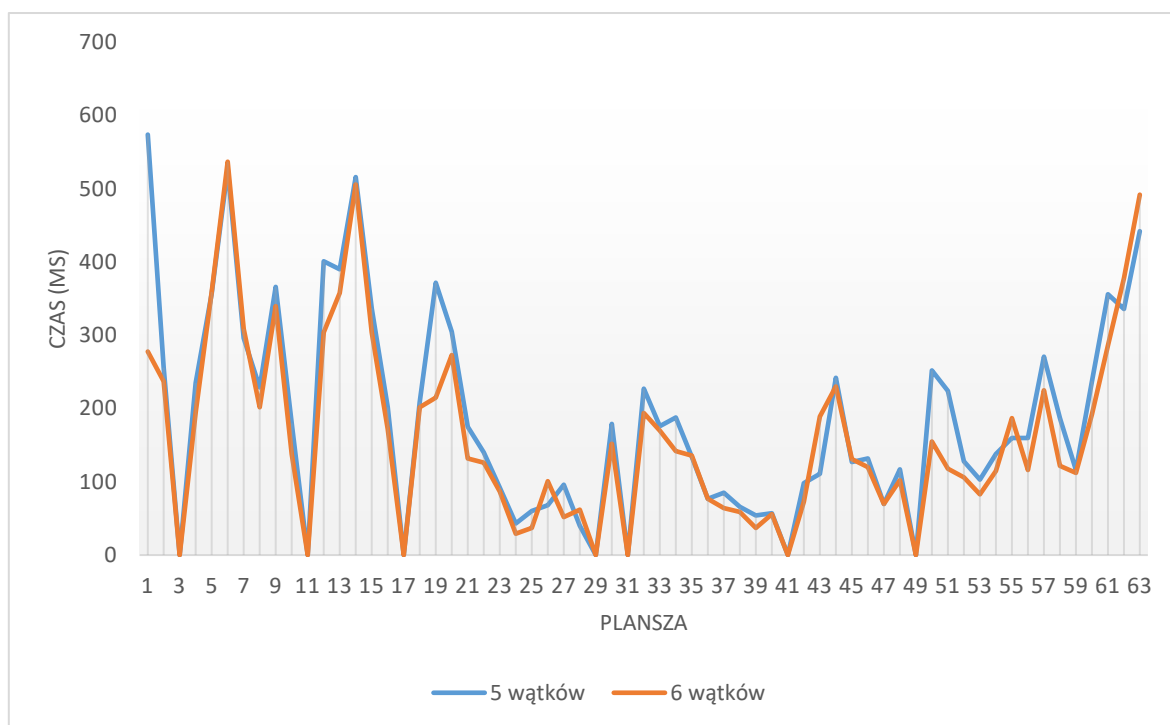


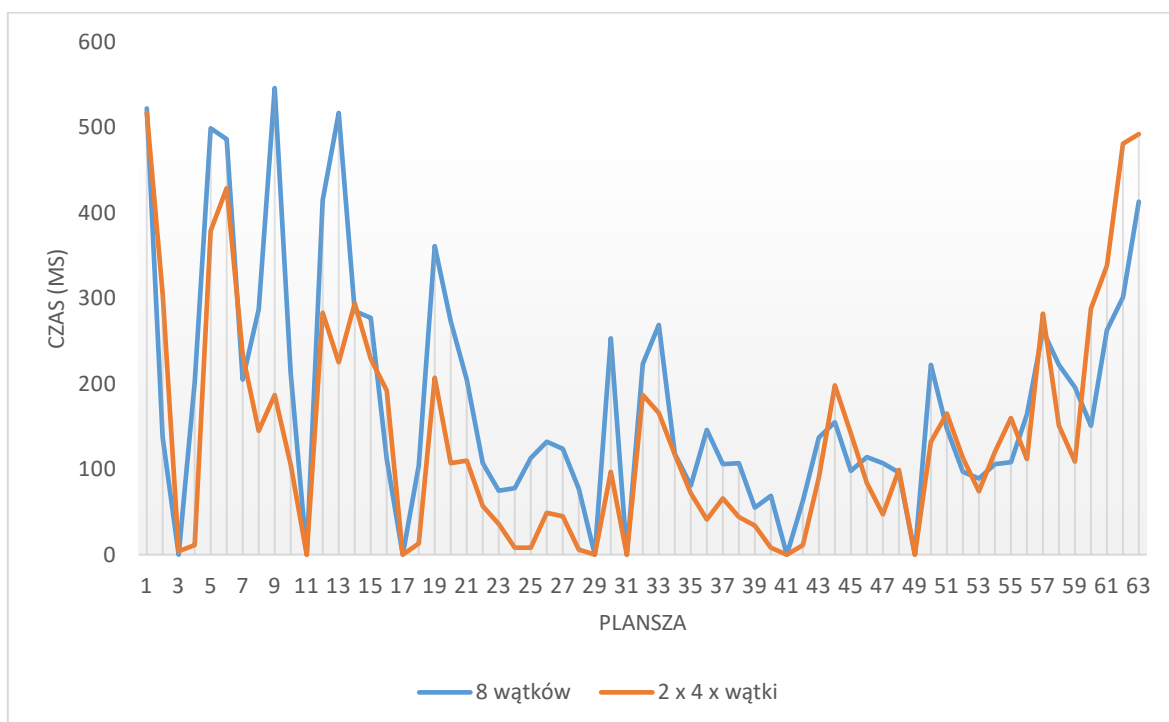
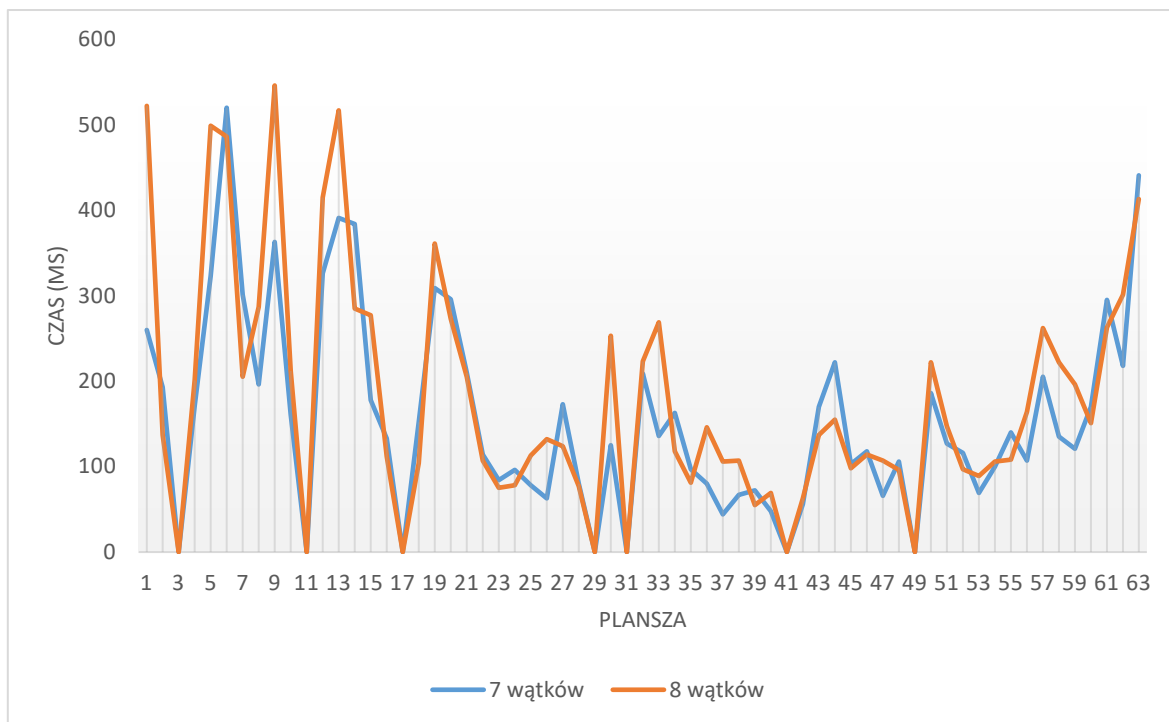
Wyniki testów dla łatwego poziomu trudności (głębokość minimax – 7)
na planszy 8x8 pól:



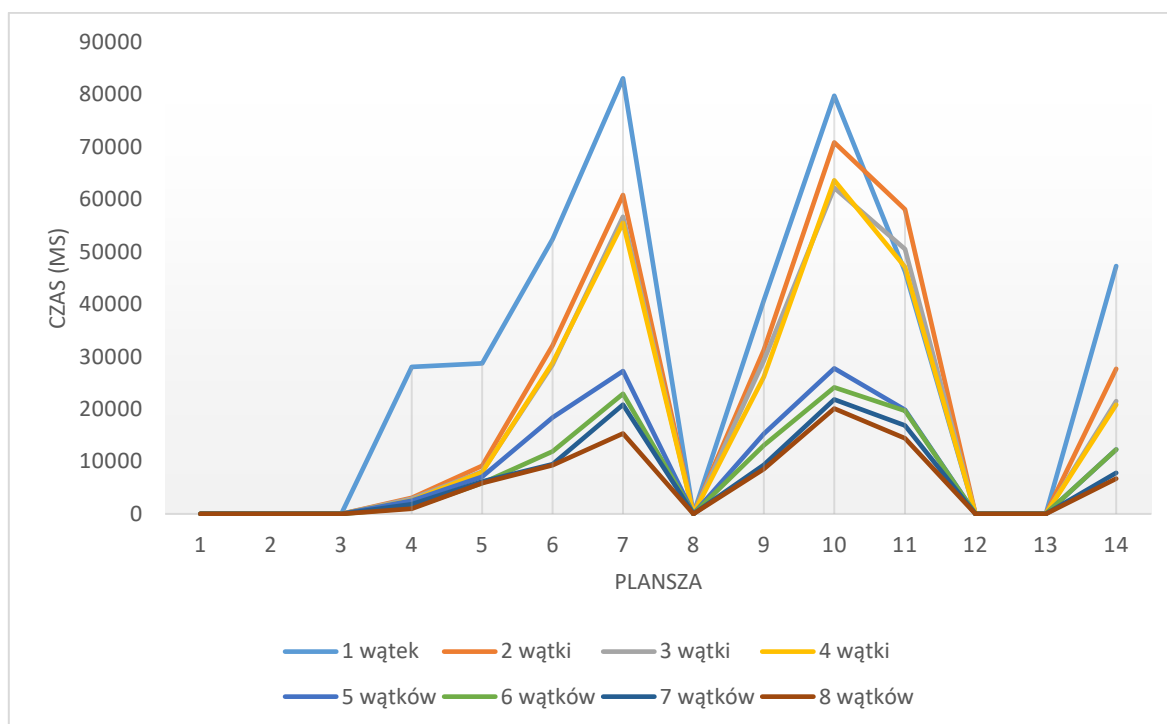








Wyniki testów dla łatwego poziomu trudności na planszy 8x8 pól
(6 damek białych – 6 damek czarnych):



6. Wnioski

Podczas tworzenia projektu poznaliśmy podstawowe możliwości technologii WCF, która znacząco ułatwia tworzenie aplikacji rozproszonych. W trakcie pisania logiki gry walczyliśmy z kilkoma mniejszymi problemami takimi jak np.: odnajdywanie wszystkich możliwych kombinacji ścieżek dla białych i czarnych. Głównym problemem było jednak opracowanie algorytmu znajdującego najlepszy ruch. Zapoznaliśmy się z działaniem algorytmu minimax i na tej podstawie opracowaliśmy własną, rozproszoną wersję. Myśląc o rozbudowie projektu można zmodyfikować aktualny algorytm gry tak, aby nie tworzył za każdym razem nowego drzewa gry tylko modyfikował wygenerowane wcześniej. Ogromny wpływ ma również funkcja oceniająca stan gry, którą też należałoby rozbudować.

7. Bibliografia

1. <https://pl.wikipedia.org/wiki/Warcaby>
2. http://www.google.pl/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0CDEQFjAC&url=http%3A%2F%2Fsstolin.kis.p.lodz.pl%2Fdane%2Fstudenci%2Fsiwpg%2FAlgorytmMM.pptx&ei=XKGCVZSfEoSxsQGipZTQBA&usg=AFQjCNFA8276FZzgYJYfPYbGW9o296mWYQ&sig2=LzqJdrsY1MwD_ohMyT6G_Q&bvm=bv.96041959,d.ZGU&cad=rja
3. https://pl.wikipedia.org/wiki/Algorytm_alfa-beta