**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 6 з дисципліни

«Проектування алгоритмів»

„**Пошук в умовах протидії, ігри з повною інформацією, ігри з елементом випадковості, ігри з неповною інформацією**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-11 Головня О.Р.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc86770239)

[2 Завдання 4](#_Toc86770240)

[3 Виконання 6](#_Toc86770241)

[3.1 Програмна реалізація алгоритму 6](#_Toc86770242)

[3.1.1 Вихідний код 6](#_Toc86770243)

[3.1.2 Приклади роботи 6](#_Toc86770244)

[Висновок 7](#_Toc86770245)

[Критерії оцінювання 8](#_Toc86770246)

# Мета лабораторної роботи

Мета роботи - вивчити основні підходи до формалізації алгоритмів знаходження рішень задач в умовах протидії. Ознайомитися з підходами до програмування алгоритмів штучного інтелекту в іграх з повною інформацією, іграх з елементами випадковості та в іграх з неповною інформацією.

# Завдання

Для ігор з повної інформацією, згідно варіанту (таблиця 2.1) реалізувати візуальний ігровий додаток для гри користувача з комп'ютерним опонентом. Для реалізації стратегії гри комп'ютерного опонента використовувати алгоритм альфа-бета-відсікань. Реалізувати три рівні складності (легкий, середній, складний).

Для ігор з елементами випадковості, згідно варіанту (таблиця 2.1) реалізувати візуальний ігровий додаток, з користувацьким інтерфейсом, не консольним, для гри користувача з комп'ютерним опонентом. Для реалізації стратегії гри комп'ютерного опонента використовувати алгоритм мінімакс.

Для карткових ігор, згідно варіанту (таблиця 2.1), реалізувати візуальний ігровий додаток, з користувацьким інтерфейсом, не консольним, для гри користувача з комп'ютерним опонентом. Потрібно реалізувати стратегію комп'ютерного опонента, і звести гру до гри з повною інформацією (див. Лекцію), далі реалізувати стратегію гри комп'ютерного опонента за допомогою алгоритму мінімаксу або альфа-бета-відсікань.

Реалізувати анімацію процесу жеребкування (+1 бал) або реалізувати анімацію ігрових процесів (роздачі карт, анімацію ходів тощо) (+1 бал).

Реалізувати варто тільки одне з бонусних завдань.

Зробити узагальнений висновок лабораторної роботи.

Таблиця 2.1 – Варіанти

|  |  |  |
| --- | --- | --- |
| **№** | **Варіант** | **Тип гри** |
| 7 | Dots and Boxes https://ru.wikipedia.org/wiki/Палочки\_(игра) | З повною інформацією |

# Виконання

## Програмна реалізація алгоритму

### Вихідний код

Main file:

from tkinter import \*

import numpy as np

from Bot import Bot

from typing import Optional

from GameState import GameState

from RandomBot import RandomBot

from MinimaxSearchBot import MinimaxSearchBot

from LocalSearchBot import LocalSearchBot

from time import time

#Настройка ткинтера

size\_of\_board = 500

number\_of\_dots = 4 # Кількість точок

symbol\_size = (size\_of\_board / 3 - size\_of\_board / 8) / 2

symbol\_thickness = 500

dot\_color = "black"

player1\_color = "#0000FF"

player1\_color\_light = "#6495ED"

player2\_color = "#CD3333"

player2\_color\_light = "#FF4040"

Green\_color = "#3D9140"

dot\_width = 0.25 \* size\_of\_board / number\_of\_dots

edge\_width = 0.1 \* size\_of\_board / number\_of\_dots

distance\_between\_dots = size\_of\_board / (number\_of\_dots)

BOT\_TURN\_INTERVAL\_MS = 100

LEFT\_CLICK = "<Button-1>"

class Dots\_and\_Boxes:

def \_\_init\_\_(self, bot1: Optional[Bot] = None, bot2: Optional[Bot] = None):

self.window = Tk()

self.window.title("Dots and Boxes")

self.canvas = Canvas(

self.window, width=size\_of\_board, height=size\_of\_board)

self.canvas.pack()

self.player1\_starts = True

self.refresh\_board()

self.bot1 = bot1

self.bot2 = bot2

self.play\_again()

def play\_again(self):

self.refresh\_board()

self.board\_status = np.zeros(

shape=(number\_of\_dots - 1, number\_of\_dots - 1))

self.row\_status = np.zeros(shape=(number\_of\_dots, number\_of\_dots - 1))

self.col\_status = np.zeros(shape=(number\_of\_dots - 1, number\_of\_dots))

self.pointsScored = False

# Введення від користувача у вигляді кліків

self.player1\_starts = not self.player1\_starts

self.player1\_turn = not self.player1\_starts

self.reset\_board = False

self.turntext\_handle = []

self.already\_marked\_boxes = []

#self.display\_turn\_text()

self.turn()

def mainloop(self):

self.window.mainloop()

def is\_grid\_occupied(self, logical\_position, type):

x = logical\_position[0]

y = logical\_position[1]

occupied = True

if type == "row" and self.row\_status[y][x] == 0:

occupied = False

if type == "col" and self.col\_status[y][x] == 0:

occupied = False

return occupied

def convert\_grid\_to\_logical\_position(self, grid\_position):

grid\_position = np.array(grid\_position)

position = (grid\_position - distance\_between\_dots / 4) // (

distance\_between\_dots / 2

)

type = False

logical\_position = []

if position[1] % 2 == 0 and (position[0] - 1) % 2 == 0:

x = int((position[0] - 1) // 2)

y = int(position[1] // 2)

logical\_position = [x, y]

type = "row"

# self.row\_status[c][r]=1

elif position[0] % 2 == 0 and (position[1] - 1) % 2 == 0:

y = int((position[1] - 1) // 2)

x = int(position[0] // 2)

logical\_position = [x, y]

type = "col"

return logical\_position, type

def pointScored(self):

self.pointsScored = True

def mark\_box(self):

boxes = np.argwhere(self.board\_status == -4)

for box in boxes:

if list(box) not in self.already\_marked\_boxes and list(box) != []:

self.already\_marked\_boxes.append(list(box))

color = player1\_color\_light

self.shade\_box(box, color)

boxes = np.argwhere(self.board\_status == 4)

for box in boxes:

if list(box) not in self.already\_marked\_boxes and list(box) != []:

self.already\_marked\_boxes.append(list(box))

color = player2\_color\_light

self.shade\_box(box, color)

def update\_board(self, type, logical\_position):

x = logical\_position[0]

y = logical\_position[1]

val = 1

playerModifier = 1

if self.player1\_turn:

playerModifier = -1

if y < (number\_of\_dots - 1) and x < (number\_of\_dots - 1):

self.board\_status[y][x] = (

abs(self.board\_status[y][x]) + val

) \* playerModifier

if abs(self.board\_status[y][x]) == 4:

self.pointScored()

if type == "row":

self.row\_status[y][x] = 1

if y >= 1:

self.board\_status[y - 1][x] = (

abs(self.board\_status[y - 1][x]) + val

) \* playerModifier

if abs(self.board\_status[y - 1][x]) == 4:

self.pointScored()

elif type == "col":

self.col\_status[y][x] = 1

if x >= 1:

self.board\_status[y][x - 1] = (

abs(self.board\_status[y][x - 1]) + val

) \* playerModifier

if abs(self.board\_status[y][x - 1]) == 4:

self.pointScored()

def is\_gameover(self):

return (self.row\_status == 1).all() and (self.col\_status == 1).all()

def make\_edge(self, type, logical\_position):

if type == "row":

start\_x = (

distance\_between\_dots / 2 +

logical\_position[0] \* distance\_between\_dots

)

end\_x = start\_x + distance\_between\_dots

start\_y = (

distance\_between\_dots / 2 +

logical\_position[1] \* distance\_between\_dots

)

end\_y = start\_y

elif type == "col":

start\_y = (

distance\_between\_dots / 2 +

logical\_position[1] \* distance\_between\_dots

)

end\_y = start\_y + distance\_between\_dots

start\_x = (

distance\_between\_dots / 2 +

logical\_position[0] \* distance\_between\_dots

)

end\_x = start\_x

if self.player1\_turn:

color = player1\_color

else:

color = player2\_color

self.canvas.create\_line(

start\_x, start\_y, end\_x, end\_y, fill=color, width=edge\_width

)

def display\_gameover(self):

player1\_score = len(np.argwhere(self.board\_status == -4))

player2\_score = len(np.argwhere(self.board\_status == 4))

if player1\_score > player2\_score:

# Гравець 1 виграє

text = "Переміг гравець 1"

color = "black"

elif player2\_score > player1\_score:

text = "Переміг гравець 2 "

color = "black"

else:

text = "Нічия"

color = "black"

self.canvas.delete("all")

self.canvas.create\_text(

size\_of\_board / 2,

size\_of\_board / 3,

font="cmr 40 bold",

fill=color,

text=text,

)

score\_text = "К-сть очок: \n"

self.canvas.create\_text(

size\_of\_board / 2,

5 \* size\_of\_board / 8,

font="cmr 40 bold",

fill=Green\_color,

text=score\_text,

)

score\_text = "Гравець 1 : " + str(player1\_score) + "\n"

score\_text += "Гравець 2 : " + str(player2\_score) + "\n"

# score\_text += 'НІчия : ' + str(self.tie\_score)

self.canvas.create\_text(

size\_of\_board / 2,

3 \* size\_of\_board / 4,

font="cmr 30 bold",

fill=Green\_color,

text=score\_text,

)

self.reset\_board = True

score\_text = "Клік, щоб зіграти ще раз \n"

self.canvas.create\_text(

size\_of\_board / 2,

15 \* size\_of\_board / 16,

font="cmr 20 bold",

fill="gray",

text=score\_text,

)

def refresh\_board(self):

for i in range(number\_of\_dots):

x = i \* distance\_between\_dots + distance\_between\_dots / 2

self.canvas.create\_line(

x,

distance\_between\_dots / 2,

x,

size\_of\_board - distance\_between\_dots / 2,

fill="black",

dash=(2, 2),

)

self.canvas.create\_line(

distance\_between\_dots / 2,

x,

size\_of\_board - distance\_between\_dots / 2,

x,

fill="black",

dash=(2, 2),

)

for i in range(number\_of\_dots):

for j in range(number\_of\_dots):

start\_x = i \* distance\_between\_dots + distance\_between\_dots / 2

end\_x = j \* distance\_between\_dots + distance\_between\_dots / 2

self.canvas.create\_oval(

start\_x - dot\_width / 2,

end\_x - dot\_width / 2,

start\_x + dot\_width / 2,

end\_x + dot\_width / 2,

fill=dot\_color,

outline=dot\_color,

)

def display\_turn\_text(self):

text = "Наступний хід: "

if self.player1\_turn:

text += "Гравець 1"

color = player1\_color

else:

text += "Гравець 2"

color = player2\_color

self.canvas.delete(self.turntext\_handle)

self.turntext\_handle = self.canvas.create\_text(

size\_of\_board - 5 \* len(text),

size\_of\_board - distance\_between\_dots / 8,

font="cmr 15 bold",

text=text,

fill=color,

)

def shade\_box(self, box, color):

start\_x = (

distance\_between\_dots / 2 + box[1] \*

distance\_between\_dots + edge\_width / 2

)

start\_y = (

distance\_between\_dots / 2 + box[0] \*

distance\_between\_dots + edge\_width / 2

)

end\_x = start\_x + distance\_between\_dots - edge\_width

end\_y = start\_y + distance\_between\_dots - edge\_width

self.canvas.create\_rectangle(

start\_x, start\_y, end\_x, end\_y, fill=color, outline=""

)

def click(self, event):

if not self.reset\_board:

grid\_position = [event.x, event.y]

logical\_position, valid\_input = self.convert\_grid\_to\_logical\_position(

grid\_position

)

self.update(valid\_input, logical\_position)

else:

self.canvas.delete("all")

self.play\_again()

self.reset\_board = False

def update(self, valid\_input, logical\_position):

if valid\_input and not self.is\_grid\_occupied(logical\_position, valid\_input):

self.window.unbind(LEFT\_CLICK)

self.update\_board(valid\_input, logical\_position)

self.make\_edge(valid\_input, logical\_position)

self.mark\_box()

self.refresh\_board()

self.player1\_turn = (

not self.player1\_turn if not self.pointsScored else self.player1\_turn

)

self.pointsScored = False

if self.is\_gameover():

# self.canvas.delete("all")

self.display\_gameover()

self.window.bind(LEFT\_CLICK, self.click)

else:

#self.display\_turn\_text()

self.turn()

def turn(self):

current\_bot = self.bot1 if self.player1\_turn else self.bot2

if current\_bot is None:

self.window.bind(LEFT\_CLICK, self.click)

else:

self.window.after(BOT\_TURN\_INTERVAL\_MS, self.bot\_turn, current\_bot)

def bot\_turn(self, bot: Bot):

action = bot.get\_action(

GameState(

self.board\_status.copy(),

self.row\_status.copy(),

self.col\_status.copy(),

self.player1\_turn,

)

)

self.update(action.action\_type, action.position)

from tkinter import ttk

def Easy():

game\_instance = Dots\_and\_Boxes(

None,

RandomBot()

)

def Normal():

game\_instance = Dots\_and\_Boxes(

None,

LocalSearchBot()

)

def Hard():

game\_instance = Dots\_and\_Boxes(

None,

MinimaxSearchBot()

)

if \_\_name\_\_ == "\_\_main\_\_":

root = Tk()

root.title("Вибір Складності")

root.geometry("250x200")

button = ttk.Button(text="Легко", command=Easy)

button.pack(anchor=CENTER, expand=1)

button1 = ttk.Button(text="Нормально", command=Normal)

button1.pack(anchor=CENTER, expand=1)

button2 = ttk.Button(text="Важко", command=Hard)

button2.pack(anchor=CENTER, expand=1)

minimax file:

from time import time

from Bot import Bot

from GameAction import GameAction

from GameState import GameState

from typing import List

import numpy as np

TIMEOUT = 1

class MinimaxSearchBot(Bot):

def \_\_init\_\_(self):

self.is\_player1 = True

self.global\_time = 0

# Отримати дію від бота

def get\_action(self, state: GameState) -> GameAction:

self.is\_player1 = state.player1\_turn

selected\_action: GameAction = None

self.global\_time = time() + TIMEOUT

row\_not\_filled = np.count\_nonzero(state.row\_status == 0)

column\_not\_filled = np.count\_nonzero(state.col\_status == 0)

for i in range(row\_not\_filled + column\_not\_filled):

# if time() >= self.global\_time:

# break

try:

actions = self.generate\_actions(state)

utilities = np.array([self.get\_minimax\_value(

state=self.get\_result(state, action), max\_depth=i + 1) for action in actions])

index = np.random.choice(

np.flatnonzero(utilities == utilities.max()))

selected\_action = actions[index]

except TimeoutError:

break

return selected\_action

# Створити список ігрових дій

def generate\_actions(self, state: GameState) -> List[GameAction]:

row\_positions = self.generate\_positions(state.row\_status)

col\_positions = self.generate\_positions(state.col\_status)

actions: List[GameAction] = []

# Евристику впорядкування ходів

for position in row\_positions:

actions.append(GameAction("row", position))

for position in col\_positions:

actions.append(GameAction("col", position))

return actions

# Створити дійову позицію

def generate\_positions(self, matrix: np.ndarray):

[ny, nx] = matrix.shape

positions: List[tuple[int, int]] = []

for y in range(ny):

for x in range(nx):

if matrix[y, x] == 0:

positions.append((x, y))

return positions

# Обновити дошку

def get\_result(self, state: GameState, action: GameAction) -> GameState:

type = action.action\_type

x, y = action.position

new\_state = GameState(

state.board\_status.copy(),

state.row\_status.copy(),

state.col\_status.copy(),

state.player1\_turn,

)

player\_modifier = -1 if new\_state.player1\_turn else 1

is\_point\_scored = False

val = 1

[ny, nx] = new\_state.board\_status.shape

# Перевірка, чи цей рух створить квадрат

if y < ny and x < nx:

new\_state.board\_status[y, x] = (

abs(new\_state.board\_status[y, x]) + val

) \* player\_modifier

if abs(new\_state.board\_status[y, x]) == 4:

is\_point\_scored = True

# Модифікація та перевірка стасусу рядків

if type == "row":

new\_state.row\_status[y, x] = 1

if y > 0:

new\_state.board\_status[y - 1, x] = (

abs(new\_state.board\_status[y - 1, x]) + val

) \* player\_modifier

if abs(new\_state.board\_status[y - 1, x]) == 4:

is\_point\_scored = True

# Модифікація та перевірка стасусу стовпців

elif type == "col":

new\_state.col\_status[y, x] = 1

if x > 0:

new\_state.board\_status[y, x - 1] = (

abs(new\_state.board\_status[y, x - 1]) + val

) \* player\_modifier

if abs(new\_state.board\_status[y, x - 1]) == 4:

is\_point\_scored = True

new\_state = new\_state.\_replace(

player1\_turn=not (new\_state.player1\_turn ^ is\_point\_scored)

)

return new\_state

def get\_minimax\_value(

self,

state: GameState,

depth: int = 0,

max\_depth: int = 0,

alpha: float = -np.inf,

beta: float = np.inf,) -> float:

if time() >= self.global\_time:

raise TimeoutError()

if self.terminal\_test(state) or depth == max\_depth:

return self.get\_utility(state)

# Якщо найкраще значення максимайзера дорівнює або перевищує найкраще значення

# мінімізатора (альфа більше ніж бета) то пошук сусідів можна зупинити,

# оскільки ми можемо бути впевнені, що мінімальне значення, яке ми шукаємо, є оптимальним рухом

if self.is\_player1 == state.player1\_turn:

value = -np.inf

actions = self.generate\_actions(state)

for action in actions:

value = max(

value,

self.get\_minimax\_value(

self.get\_result(state, action),

depth=depth + 1,

max\_depth=max\_depth,

alpha=alpha,

beta=beta

),

)

alpha = max(alpha, value)

if beta <= alpha:

break

return value

else:

value = np.inf

actions = self.generate\_actions(state)

for action in actions:

value = min(

value,

self.get\_minimax\_value(

self.get\_result(state, action),

depth=depth + 1,

max\_depth=max\_depth,

alpha=alpha,

beta=beta

),

)

beta = min(beta, value)

if beta <= alpha:

break

return value

# Перевірка чи термінальна вершина(лист) утворює квадрат

def terminal\_test(self, state: GameState) -> bool:

return np.all(state.row\_status == 1) and np.all(state.col\_status == 1)

# Функція корисності з абсолютним значенням 1, якщо сформований квадрат

def get\_utility(self, state: GameState) -> float:

[ny, nx] = state.board\_status.shape

utility = 0

# Підрахунок квадратів

box\_won = 0

box\_lost = 0

for y in range(ny):

for x in range(nx):

if self.is\_player1:

if state.board\_status[y, x] == -4:

utility += 1

box\_won += 1

elif state.board\_status[y, x] == 4:

utility -= 1

box\_lost += 1

else:

if state.board\_status[y, x] == -4:

utility -= 1

box\_lost += 1

elif state.board\_status[y, x] == 4:

utility += 1

box\_won += 1

# Ланцюгове правило

if self.chain\_count(state) % 2 == 0 and self.is\_player1:

utility += 1

elif self.chain\_count(state) % 2 != 0 and not self.is\_player1:

utility += 1

# Евристика виграшу/програшу

if box\_won >= 5:

utility = np.inf

elif box\_lost >= 5:

utility = -np.inf

return utility

# Підрахунок кількості ланцюгів

def chain\_count(self, state: GameState) -> int:

chain\_count = 0

chain\_list: List[List[int]] = []

for box\_num in range(9):

# Перевірка чи квадрат вже частина ланцюга

flag = False

for chain in chain\_list:

if box\_num in chain:

flag = True

break

if not flag:

chain\_list.append([box\_num])

self.add\_chain(state, chain\_list, box\_num)

for chain in chain\_list:

if len(chain) >= 3:

chain\_count += 1

return chain\_count

# Знаходження сусіднього квадрату, що можуть побудувати ланцюг

def add\_chain(self, state: GameState, chain\_list: List[List[int]], box\_num):

neighbors\_num = [box\_num - 1, box\_num - 3, box\_num + 1, box\_num + 3]

for idx in range(len(neighbors\_num)):

if (

neighbors\_num[idx] < 0

or neighbors\_num[idx] > 8

or (idx % 2 == 0 and neighbors\_num[idx] // 3 != box\_num // 3)

):

continue

flag = False

for chain in chain\_list:

if neighbors\_num[idx] in chain:

flag = True

break

if not flag and idx % 2 == 0:

reference = max(box\_num, neighbors\_num[idx])

if not state.col\_status[reference // 3][reference % 3]:

chain\_list[-1].append(neighbors\_num[idx])

self.add\_chain(state, chain\_list, neighbors\_num[idx])

if not flag and idx % 2 != 0:

reference = max(box\_num, neighbors\_num[idx])

if not state.row\_status[reference // 3][reference % 3]:

chain\_list[-1].append(neighbors\_num[idx])

self.add\_chain(state, chain\_list, neighbors\_num[idx])

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

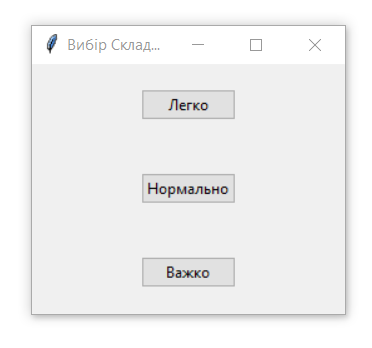


Рисунок 3.1 –

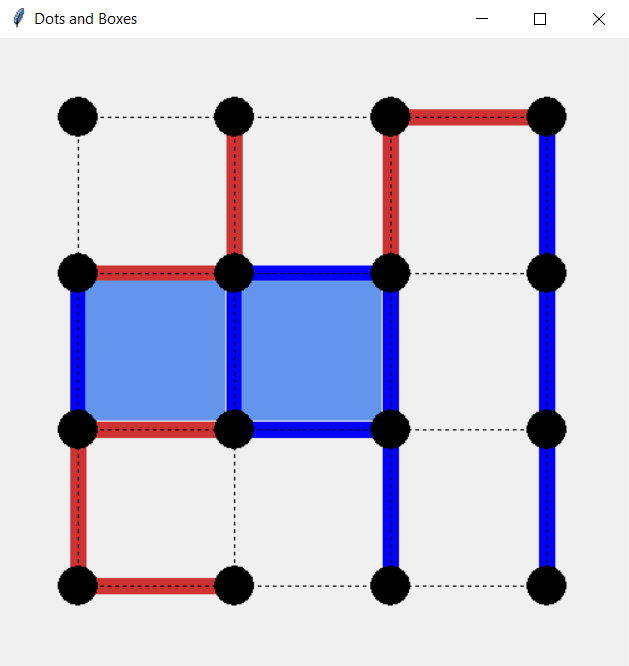
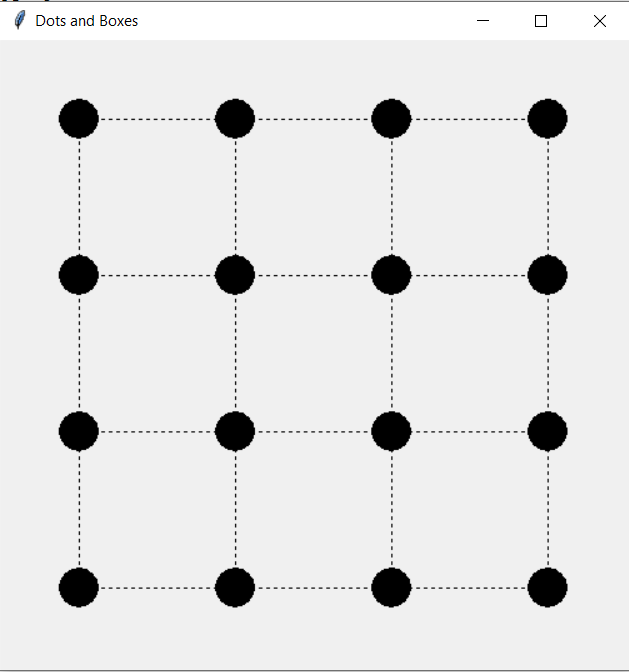


Рисунок 3.2 –

## 

Висновок

В рамках даної лабораторної роботи я вивчив основний підхід до формалізації алгоритмів знаходження рішень задач в умовах протидії. Ознайомився з підходами до програмування алгоритмів штучного інтелекту в іграх з повною інформацією, було реалізовано візуальний ігровий додаток для гри «Палочки». Для реалізації стратегії гри комп'ютерного опонента використовано алгоритм альфа-бета-відсікань.

Критерії оцінювання

При здачі лабораторної роботи до 25.12.2022 включно максимальний бал дорівнює – 5. Після 25.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* програмна реалізація – 95%;
* висновок – 5%.

+1 додатковий бал можна отримати за реалізацію анімації ігрових процесів (жеребкування, роздачі карт, анімацію ходів тощо).