Druga domaća zadaća

Igra Connect4

1. Implementacija

Rješenje sadrži sveukupno 9 razreda:

1. Player -> abstraktna klasa za modeliranje pojedinog igrača
2. HumanPlayer -> Implementira potez igrača, traži unos od korisnika
3. MinMaxPlayer -> Implementira algoritam pronalaska najboljeg mogućeg poteza za zadanu dubinu
4. Connect4 -> Modelira igru connect4, sastoji se od sljedećih bitnih metoda
    a. Get_available_moves() -> vraća sve dopuštene poteze za zadano stanje igre
    b. Undo_move(column) -> briše prethodni korak na zadanom stupcu
    c. Print_board() -> Estetski ispis ploče
    d. Make_move(column, mark) -> Upisuje potez na ploču u zadani stupac(column) s zadanom oznakom(mark)
    e. Check_game_status(column) -> provjerava stanje ploče u odnosu na zadnje odigrani potez koji se nalazi u stupcu (column) i vraća stringove draw, win i not finished
5. BoardSolver -> Razred u kojem se nalaze metode za evaluaciju poteza
    a. Evaluate(game, is_maximizing_player, last_column_idx,depth, max_player_mark, min_player_mark) -> rekurzivna funkcija za evaluaciju poteza
    b. Solve_task(task) -> riješava zadatak
    c. Resolve_node_score(node, is_max_player_turn) -> određuje vrijednost čvora – to se koristi u slijednom prolazu kad dođemo do dijela kad se izračunaju svi čvorovi na razini task_depth.
6. Node -> Razred koji predstavlja čvor u stablu, bitni atributi
    a. Node_id -> random string koji predstavlja id čvora.
    b. Game -> trenutno stanje igre
    c. Score -> trenutna vrijednost čvora
    d. Next_move -> u koji stupac ide sljedeći potez
    e. Is_finished -> da li je igra gotova
    f. Children -> djeca, isto Node instance
7. Task -> Implementira zadatak u MPI okruženju kad je broj procesa > 1
    a. Node -> čvor koji obrađuje
    b. Is_max_player_turn -> govori čiji je potez sljedeći
    c. Max i min player sign
    d. Remaining_depth -> koliko u dubinu još mora ići
8. TaskResult -> Razred koji vraćaju radnici nakon završetka zadatka
    a. Node_id -> da znamo koji je čvor odradio

b. Score -> nakon što prođe zadani broj dubini, koja je evaluacija, odnosno vrijednost čvora

9. Worker -> Razred koji ima ulogu oponašanja radnika
   a. Do_task -> Čeka poruka i kad primi na temelju tag-a poruke određuje šta će raditi

Evaluacija ploče

```
def evaluate(game: Connect4, is_maximizing_player: bool, last_column_idx:
int, depth: int, max_player_mark: str,
             min_player_mark: str) -> float:
    if game.check_game_status(last_column_idx) == 'win':
        if is_maximizing_player:
            return 1
        else:
            return -1

    if game.check_game_status(last_column_idx) == 'draw':
        return 0

    if depth == 0:
        return 0

    is_maximizing_player = not is_maximizing_player
    player_mark = max_player_mark if is_maximizing_player else
min_player_mark
    depth -= 1

    total = 0
    num_of_moves = 0

    is_all_lose = True
    is_all_win = True

    for column in game.get_available_moves():
        num_of_moves += 1

        game.make_move(column, player_mark)
        result = evaluate(game, is_maximizing_player, column, depth,
max_player_mark, min_player_mark)
        game.undo_move(column)

        if result > -1:
            is_all_lose = False
        if result != 1:
            is_all_win = False

        if result == 1 and is_maximizing_player:
            return 1
        if result == -1 and not is_maximizing_player:
            return -1

        total += result

    if is_all_win:
        return 1
    if is_all_lose:
        return -1

    return total / num_of_moves
```

Dio za evaluaciju poteza. Logika je ista kao i kod dostupnog kod-a na Intranetu. Tako da nemam tu šta posebno objašnjavat.

Worker klasa

```
import pickle
import mpi4py
from game.BoardSolver import solve_task
from parallel.TaskResult import TaskResult


class Worker:
    def __init__(self, comm: mpi4py.MPI.COMM_WORLD):
        self.comm = comm

    def do_task(self):
        while True:
            status = mpi4py.MPI.Status()
            task = self.comm.recv(source=0, tag=mpi4py.MPI.ANY_TAG, sta-
tus=status)
            if status.tag == 1:
                task = pickle.loads(task)
                node = task.node
                best_value = solve_task(task)
                task_result = TaskResult(score=best_value,
node_id=node.node_id)
                self.comm.send(pickle.dumps(task_result), dest=0, tag=1)

            elif status.tag == 2:
                print("Worker done for today")
                return
            else:
                raise ValueError(f"Invalid tag {status.tag}!")
```

Klasa za radnike, gdje prima poruku i provjerava tag.

- Tag == 1
  - S pickle.loads se serijalizira instanca razreda Task
  - Pošaljemo task metodi solve_task koja vrati evaluaciju čvora
  - Napravimo TaskResult instancu i deserijaliziramo je i pošaljemo ju voditelju.
- Tag == 2
  - Radnik je gotov s poslom, igra je gotova te s return izlazi iz beskonačne petlje
- Inače
  - Baca iznimku

Stvaranje zadataka

```
is_max_player_turn = False
start_time = time.time()
for search_depth in range(1, self.task_depth + 1):
    is_max_player_turn = not is_max_player_turn
    if search_depth not in tree:
        tree[search_depth] = []

    all_parent_nodes = tree[search_depth - 1]
    for parent in all_parent_nodes:
        if parent.is_finished:
            continue

        for column_move in parent.game.get_available_moves():
            copy_of_game = copy.deepcopy(parent.game)
            new_node: Node = self.make_node(copy_of_game, is_max_player_turn,
column_move)
            tree[search_depth].append(new_node)
            parent.children.append(new_node)

            if search_depth == self.task_depth and not new_node.is_finished:
                node_tree_search[new_node.node_id] = new_node
                task = Task(node=new_node, is_max_player_turn=not
is_max_player_turn,
                            max_player_sign=self.player_mark,
min_player_sign=self.other_mark,
                            remaining_depth=self.depth - self.task_depth)
                job_queue.put(task)
```

Bitne strukture:

- Tree -> mapa gdje je ključ razina, a vrijednost je polje čvorova, služi za simulaciju razina stabala. Razina 0 je korijenski čvor
- Node_tree_search -> Mapa kojoj je ključ node_id a vrijednost je instanca razreda Node. Ova struktura služi za lakši pronalazak čvora kojem treba dodijelit score nakon obavljenog zadatka
- Job_queue -> red s zadatcima
- Workers_destinations -> red s destinacijama radnika, znači od 1 do broja procesa – 1

Za svaki task_depth (dubina računanja) stvaramo prvo ključ u mapi te onda za svaki roditeljski čvor radimo sljedeće:

- Kopiramo objekt kako ne bi bilo problema s referencama
- Radimo novi čvor
    - Napravi se novi potez column, te ako je igra gotova već u tom potezu onda se postavlja odgovarajući score u čvor
- Ako se nalazimo na zadanoj dubini računanja i ako taj čvor nije već u završenom stanju:
    - Radimo novi zadatak (Task) i spremamo u red
    - Dodajemo čvor u mapu node_tree_search

Podjela poslova

```
num_of_workers_not_done = 0
while not job_queue.empty():
    if not workers_destinations.empty():
        self.send_job(job=job_queue.get(), dest=workers_destinations.get(),
tag=1)
        num_of_workers_not_done += 1
    else:
        task_result, worker = self.check_and_receive(tag=1)
        if task_result is None:
            task = job_queue.get()
            best_value = solve_task(task=task)
            node_tree_search[task.node.node_id].score = best_value
        else:
            node_tree_search[task_result.node_id].score = task_result.score
            self.send_job(job=job_queue.get(), dest=worker, tag=1)

for i in range(num_of_workers_not_done):
    task_result = self.comm.recv(source=mpi4py.MPI.ANY_SOURCE, tag=1)
    task_result = pickle.loads(task_result)
    node_tree_search[task_result.node_id].score = task_result.score
```

Prvo svim dostupnim procesima dodijelimo zadatak.

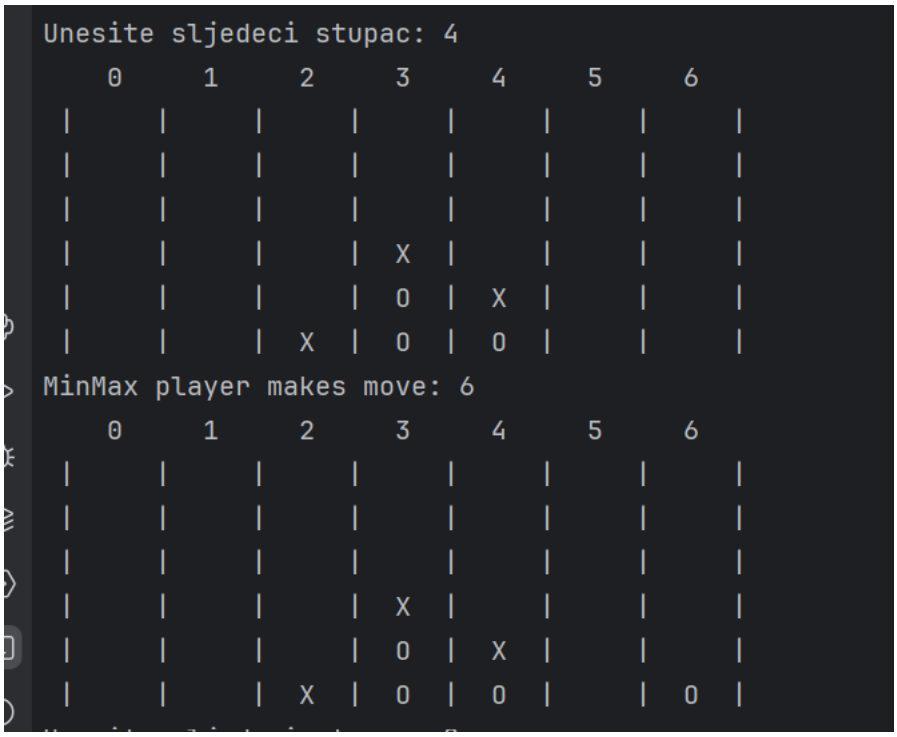
Nakon toga prvo provjeravamo da li imamo spreman neki rezultat i to nam radi self.check_and_receive()

- Ako postoji poruka za zadani tag, primit ćemo poruku i vratit ćemo serijalizirani task i indeks radnika koji je poslao rezultat. Onda dobiveni rezultat pridodamo čvoru pomoću naše mape node_tree_search i šaljemo novi zadatak tom radniku
- Ako nema poruke, voditelj sam odradi jedan zadatak

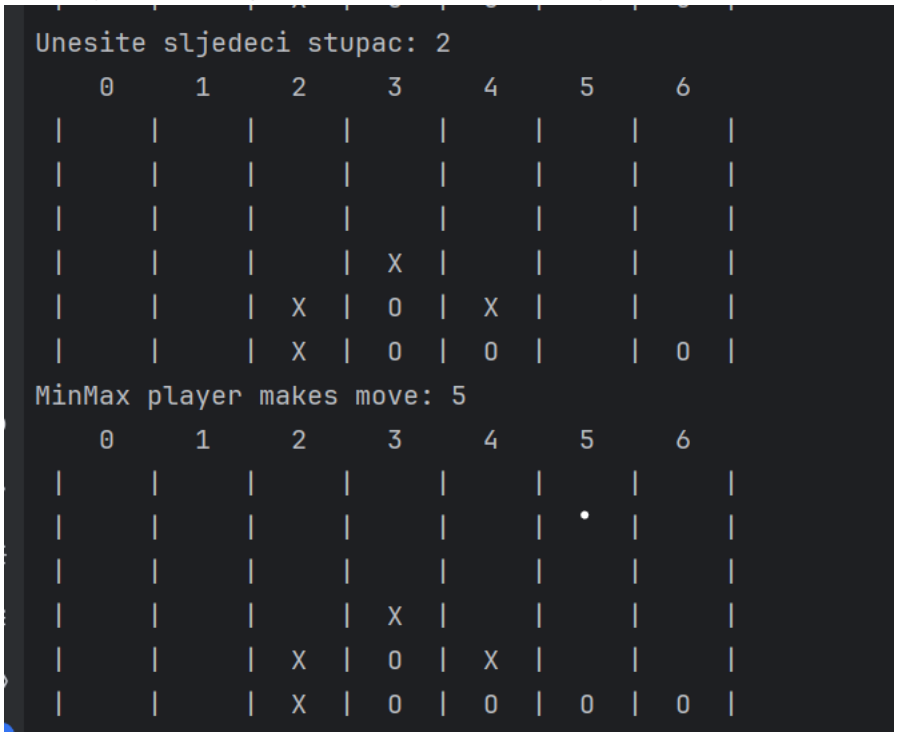
I tako sve dok ima zadataka u redu.

Kad nema više zadataka, sa petljom čekamo rezultate ostalih radnika.

Zadnja 2 koraka



Postavljam X na stupac 4 dok računalo odabire potez 6



Odabirem potez 2, računalo odabire potez 5 i pobjeđuje.

2. Kvanitativna analiza

Execution Time vs Number of Processors for Task Depth 1

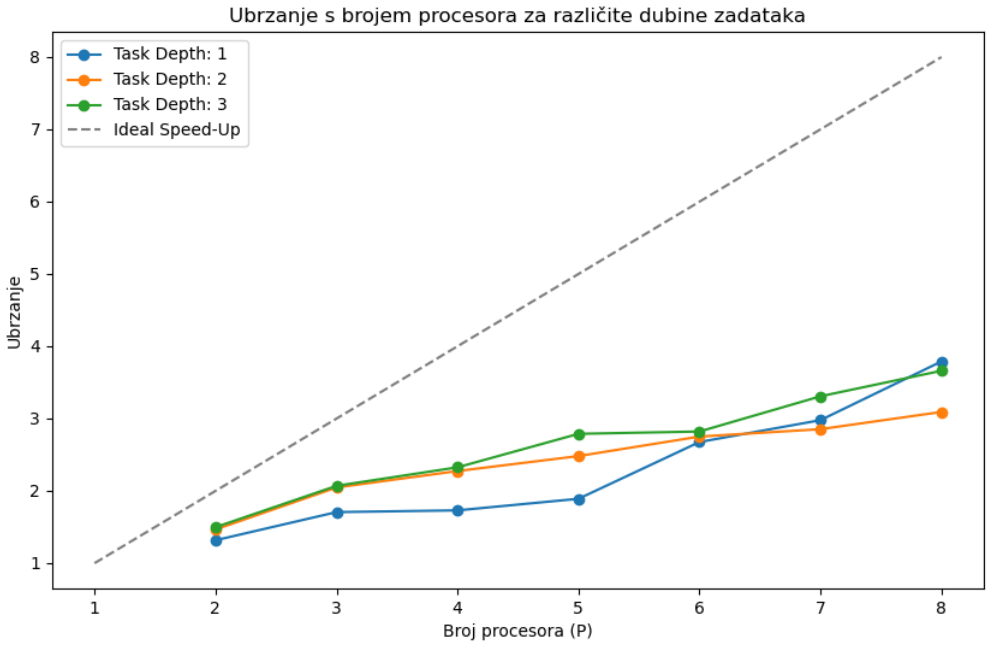
	1	2	3	4	5	6	7	8
6	25.873852	18.500662	14.279654	14.080455	12.887252	9.105185	8.180584	6.428453

Execution Time vs Number of Processors for Task Depth 2

	1	2	3	4	5	6	7	8
6	23.557743	16.617527	11.902372	10.711956	9.81809	8.855868	8.538484	7.880591

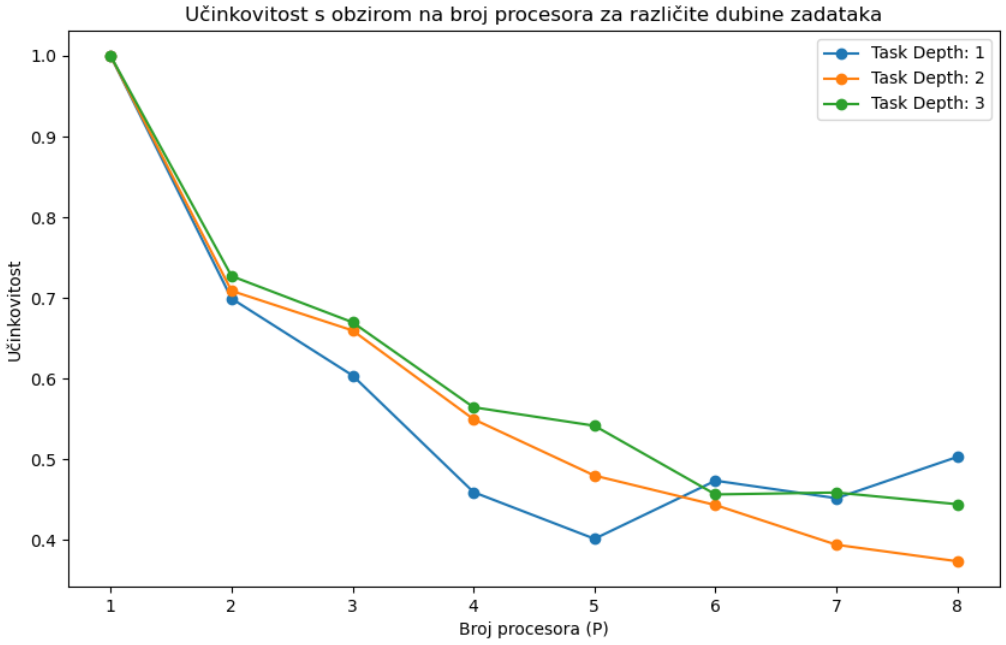
Execution Time vs Number of Processors for Task Depth 3

	1	2	3	4	5	6	7	8
6	23.65759	16.26727	11.771967	10.47571	8.735561	8.636438	7.36476	6.654469



U mom slučaju ispada da je dubina 3 najbolja kada imamo broj procesora manji od 8.

Razlog leži u tome što koristimo sve procese i podijelili smo na dovoljan broj zadataka, a i preostala dubina koju trebaju radnici izračunati nije previše zahtjevna. Razlog zašto je za 8 procesa najbolji ovaj s dubinom 1,je taj što u tom slučaju imamo 7 zadataka i 7 radnika i oni obave posao u jednakom vremenu, te ne gubimo vrijeme na komunikaciju i prijenosom nepotrebnih podataka. Ali to je ovdje ispalo ovako jer sam mjerio vrijeme prvog poteza kad su svi zadatci jednake duljine. Svakako je bolje i korisnije podijelit na više zadataka što se i može vidjeti na grafu.



Vidimo da učinkovitost uglavnom pada s brojem procesa.

Razlog tome leži zbog overhead-a komunikacije, poprilična količina podataka se prenosi u mom riješenju zadatka te to dosta utječe na učinkovitost.

Isto tako neki proces je možda završio ali pošto voditelj obrađuje zadatak, proces nema druge nego da čeka da proces voditelj završi i krene opet gledat da li ima poruka o rezultatu i da uposli tog procesa.