

# Proiect PS

Petre Robert Cristian

January 14, 2025

# Contents

1	Scop proiect	1
2	Descrierea jocului	1
	Tabla de joc — 1 • Harta — 1 • Marginea tableei — 2 • Industrie — 2 • Actiuni — 3	
3	Descriere matematica	5
	Obiectiv — 6	
4	Implementare	6
	Joc — 6 • MCTS — 8 • Dificultati intampinate — 10	
5	Possible imbunatatiri	11
	Performanta este abismala — 11 • Vizualizarea datelor — 11 • Modul in care e parcurs arborele — 11 • Stilul codului — 12 • Alte idei — 12	
6	Concluzii	12
7	Bibliografie	13

## 1 Scop proiect

Am creat acest proiect pentru a descoperi strategii noi care pot fi aplicate in jocul Brass Birmingham.

Fiind vorba de un joc complex, cred ca tehniciile necesare pentru a indeplini scopul proiectului pot fi utilizate si pentru a rezolva alte tipuri de probleme.

## 2 Descrierea jocului

Pot fi de la 2 la 4 jucatori. Castigatorul este cel cu cele mai multe puncte.

### 2.1 Tabla de joc

Tabla jocului este alcătuită dintr-o hartă a unor orașe și o piață, unde se află 2 tipuri de resurse: carbune și fier.

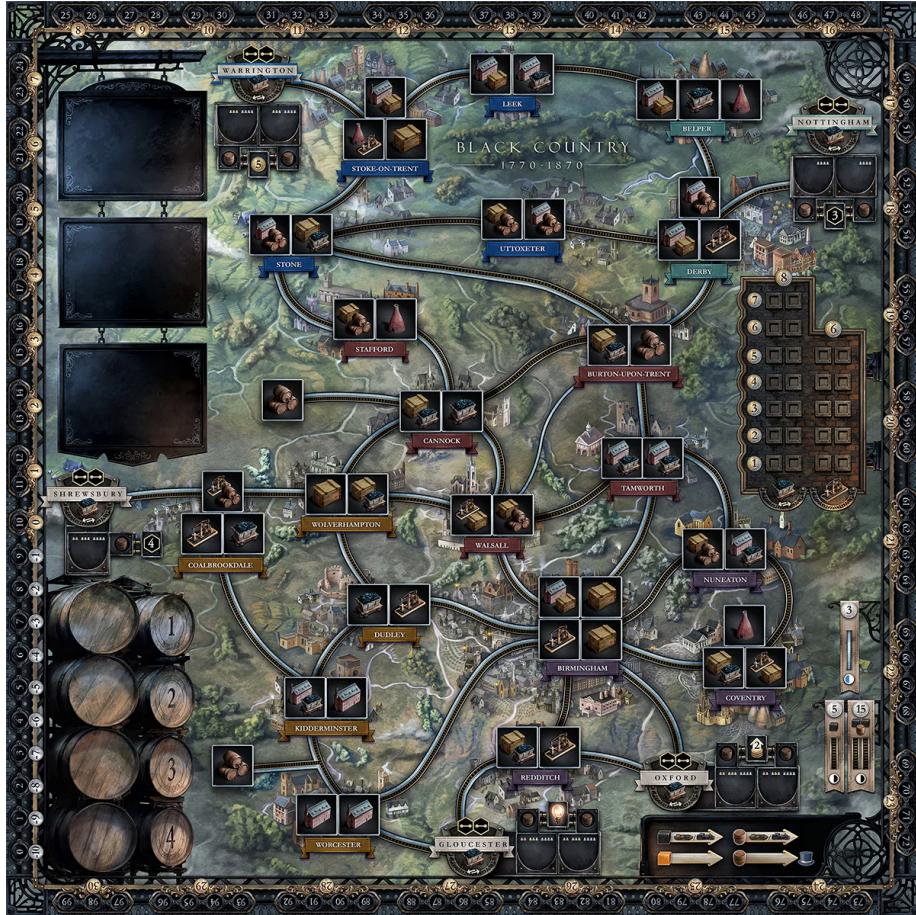


Figure 1: Tabla de joc

## 2.2 Harta

Harta ocupa cea mai mare parte a tablei de joc. Poate fi reprezentata sub forma unui graf nedirectionat. Iar fiecare nod, are in alcatuirea sa mai multe spatii, in care poti amplasa cladiri dintr-o anumita industrie.

In dreapta, se afla piata, unde exista un numar limitat de spatii pentru carbune si fier. Resursele sunt consumate de sus in jos, si trebuie utilizate imediat dupa achizitie(pentru a impiedica fenomenul de 'hoarding').

## 2.3 *Marginea tabelei*

Pe marginea tablei se vor afla informatii cu privire la jucatori. Fiecare jucator va pune un pion in dreptul punctelor acumulate si un alt pion in dreptul venitului sau.

In dreapta jos sunt niste informatii cu privire la joc, de exemplul costul conexiunilor si moduri de transport al resurselor.

In stanga jos, in dreptul butoaielor numerotate, se amplaseaza cartonase ce reprezinta jucatorii, acestea determina ordinea de joc, initial e aleasa arbitrar si apoi se schimba in functie de banii consumati de fiecare jucator in runda precedenta(cel care a consumat cel mai putin e primul, si asa mai departe, daca exista o egalitate, se pastreaza ordinea relativa).

## 2.4 Industrie

Exista 6 tipuri de industrie: fiecare difera putin de celelalte si aceste diferente pot influenta strategia jucatorului:

- Mina de carbune

**Detalii:**

ieftina, profitabila, ofera putine puncte. Nu trebuie vanduta

- Mina de fier

**Detalii:**

putin mai scumpa, mai putin profitabila, ofera mai multe puncte. Nu trebuie vanduta

- Berarie -

**Detalii:**

si mai scumpa, mai profitabila decat o mina de fier, ofera multe puncte, mai ales daca ai conexiuni la orasul in care se afla. Nu trebuie vanduta

- Moara de bumbac

**Detalii:**

scumpa, putin profitabila, ofera multe puncte daca o dezvolti. Trebuie vanduta

- Atelier de olarie

**Detalii:**

cele de nivel impar iti ofera foarte multe puncte si un profit decent, scumpa, putine locatii disponibile pentru a o construi. Trebuie vanduta

- Manufactura

**Detalii:**

destul de scumpa in general(depinde de nivelul de dezvoltare), variaza mult ca punctaj si profit in functie de nivel, unele iti ofera multe puncte daca ai conexiuni la orasul in care se afla, altele deloc. Trebuie vanduta

Cand spun ca o industrie trebuie vanduta, ma refer la faptul ca trebuie sa iei o actiune explicita pentru a vinde cladirea respectiva. Unele industrii, in schimb, se vand automat odata ce nu mai produc resurse.

## 2.5 Actiuni

Exista 6 tipuri de actiuni pe care un jucator le poate executa:

- constructie de industrie
- constructie de conexiuni intre orase(in prima era canale de navigatie, in a doua sine de tren)
- imprumut
- dezvoltare de industrie mai avansata
- cautare de zone noi(scout)
- vanzare industrie (pentru a obtine venit si puncte, industria trebuie vanduta)

Pentru a efectua actiunile, player-ul are nevoie de carti de joc, fiecare actiune necesita decartarea unei carti de joc, la alegere.

**Observatie:-**

Pot alege sa nu efectuezi o actiune, dar tot esti nevoit sa decartezi o carte.



Figure 2: Tipuri de carti

Exista 2 tipuri de carti: carti ce specifica un oras, si carti ce specifica una sau mai multe industrii.

Cu o carte ce specifica un oras, poti construi in acel oras orice industrie disponibila.

Cu o carte ce specifica o industrie, poti construi in orice oras conectat la reteaua ta de conexiuni, ce are un spatiu disponibil pentru industria respectiva.



Figure 3: Alte tipuri de carti

Cartile de deasupra se numesc 'wildcards'. Cea din stanga iti permite sa construiesti in orice oras, iar cea din dreapta sa construiesci orice industrie in orice oras conectat la reteaua ta. Sunt practic versiunile cele mai "puternice" ale cartilor normale. Si sunt utile cand nu esti multumit cu cartile din mana ta.

### 3 Descriere matematica

Avem:

1. un spatiu  $S$  ce reprezinta toate starile care pot fi obtinute pe parcursul unui joc. O stare este alcautuita din:
  - starea tablei de joc la momentul respectiv (constructii, conexiuni, cele 2 piete)
  - mainile jucatorilor (fiecare jucator are maximum 8 carti de joc pe care le poate utiliza), scorul fiecarui jucator, punctele, banii stransii
  - era in care are loc jocul (prima 1770 - 1820, sau a doua, 1820 - 1870)
  - o valoare ce indica player-ul ce urmeaza sa faca o miscare
2. un spatiu  $A$  ce reprezinta actiunile ce pot fi efectuate din starea curenta. Exista 6 actiuni posibile:
  - constructia de industrie

- dezvoltarea de industrie noua
  - constructia de conexiuni intre orase (canale in prima era, linii de tren in a doua)
  - imprumut (primesti 30 de bani, si platesti 3 bani pe tura)
  - vanzare industrie (industria poate fi valorificata doar daca o vinzi)
  - recunoastere (dai discard la 3 carti si primesti 2 carti de tip 'wildcard' care iti permit sa te amplasezi unde doresti)
3. O functie de tranzitie  $T(s, a, s')$  unde  $s \in S$ ,  $a \in A$ ,  $s' \in S$
  4. O functie de recompensa  $\mathcal{R}$  care calculeaza punctele unei stari  $s$

### 3.1 Obiectiv

Obiectivul e sa maximizam recompensa totala  $P$  obtinuta pe parcursul jocului:

$$P = \mathbb{E} \sum_{t=0}^H \gamma^t \mathcal{R}(s_t, a_t, s_{t+1})$$

unde

- $s_t$  este starea la timpul  $t$
- $a_t$  este actiunea aleasa in starea  $s_t$
- $\gamma$  factorul de reducere, in cazul asta are valoarea 1
- $H$  numarul de ture totale

## 4 Implementare

Voi prezenta cele mai importante clase si functii. Codul este voluminos(mai ales partea care se ocupa de logica jocului), asa ca nu-l pot acoperi pe tot intr-un numar rezonabil de pagini.

### 4.1 Joc

```
class Building:
    def __init__(self, level, industry_type, stats, price, beers=0, resources=0):
        self.level = level
        self.industry_type = industry_type
        self.stats = stats
        self.price = price
        self.beers = beers # number of beers required for it to be sold
        self.resources = resources

    def __str__(self):
        return f"{self.industry_type} {self.level} stats: {self.stats} price: {self.price} {self.beers} produces: {self.resources}"

    def __repr__(self):
        return f"{self.industry_type} {self.level} stats: {self.stats} price: {self.price} {self.beers} produces: {self.resources}"
```

Figure 4: clasa 'Building'

Utilizata de cele mai complexe actiuni, 'sell', 'build', 'develop'. Fiecare player creeaza un numar limitat de obiecte 'Building' la inceput, pe baza unui tabel. Cand construiesti ceva pe tabla de joc, este ales un obiect specific 'Building', care este encapsulat in clasa 'BuildingInstance'.

```
class BuildingInstance:  
    def __init__(self, building, player_id):  
        self.building = building  
        self.player_id = player_id  
        self.sold = False
```

Figure 5: clasa 'BuildingInstance'

```

class Player:
    def __init__(self, id, state, environment):
        self.id = id
        self.iron_works = []
        self.coal_mines = []
        self.breweries = []
        self.potteries = []
        self.manufactories = []
        self.cotton_mills = []

        self.buildings_on_board = []

        self.createBuildings()
        self.victory_points = 0
        self.income = 10
        self.coins = 17

        self.has_industry_wildcard = False
        self.has_city_wildcard = False
        self.cards = []
        self.discard_pile = [] # could be useful
        self.state = state # reference to the game state
        self.environment = environment
        self.available_cities = []
        # reference to cities connected to the network
        # these cities are the only ones that can be built
        self.links = [] # links placed by player

```

Figure 6: clasa 'Player'

Contine toate informatiile despre un player, cat si functii precum: 'build', 'develop', 'sell', 'network', 'scout', 'loan', care reprezinta cele 6 actiuni pe care le poate efectua un player(excluzand actiunea 'pass', care nu face nimic).

De asemenea, exista functii ajutatoare utilizate si de alte clase care au nevoie de acces la datele unui player. (precum cele care preiau cladirile de pe tabla de joc)

## 4.2 MCTS

```
class MCTSNode:  
    def __init__(self, state, parent=None):  
        self.state = state  
        self.parent = parent  
        self.explored_moves = []  
        self.unexplored_moves = state.getLegalMoves()  
        self.children = []  
        self.visits = 0  
        self.total_reward = 0
```

Figure 7: clasa 'MCTSNode'

Contine tot ce e necesar pentru a crea si explora un arbore de stari.

Functiile 'explored\_moves' si 'unexplored\_moves' sunt folosite de functia 'expand' pentru a face tranzitia de la node-ul actual la un nod nou mai rapida.

```
class State: # the state consists of the current board, and  
    def __init__(self, number_of_players, current_player):  
        self.cards = []  
        self.players = []  
        self.legal_moves = None  
        self.actions_taken = 0  
        self.last_action = None  
        self.current_player = current_player  
        self.number_of_players = number_of_players  
        self.board = board.Board(number_of_players)  
        self.reached_end = False
```

Figure 8: clasa 'State' in care se afla toata informatia necesara pentru MCTS

```

def search(self, root_state, iterations=100):
    root_node = MCTSNode(state=root_state)

    for i in range(iterations):
        print(i)
        # Selection
        node = self._select(root_node)

        # Expansion
        if not node.state.isTerminal():
            child = self._expand(node)

        # Simulation
        reward = self._simulate(child.state)

        # Backpropagation
        self._backpropagate(child, reward)

    self.visualize_tree(root_node)

    best_child = root_node.bestChild(exploration_weight=0)
    return best_child.state.getLastAction()

```

Figure 9: Functia principala

Functie preluata din cartea 'Artificial Intelligence A Modern Approach', cu mici modificari. La final afisam arborele de stari.

```

def _select(self, node):
    while not node.state.isTerminal() and node.isFullyExpanded():
        node = node.bestChild(self.exploration_weight)

    return node

def _expand(self, node):
    unexplored_moves = node.unexplored_moves
    move = random.choice(unexplored_moves)
    node.unexplored_moves.remove(move)
    node.explored_moves.append(move)
    new_state = node.state.applyMove(move)
    return node.addChild(new_state)

def _simulate(self, state):
    current_state = state.clone()
    while not current_state.isTerminal():
        if current_state.legal_moves is None:
            legal_moves = current_state.getLegalMoves()
        else:
            legal_moves = current_state.legal_moves
            random_move = random.choice(legal_moves)
            current_state = current_state.applyMove(random_move)

    return current_state.getReward(current_state.getPlayer())

def _backpropagate(self, node, reward):
    while node is not None:
        node.visits += 1
        node.total_reward += reward
        node = node.parent

```

Figure 10: Functiile ajutatoare

### 4.3 Dificultati intampinate

Cea mai dificila parte a fost si cea mai voluminoasa. Ma refer la partea care implementeaza logica jocului. 'player.py', 'environment.py', au avut nevoie de mult debugging si probabil inca mai au nevoie de mult debugging. 'board.py' a fost mai usor de implementat si testat, ceea ce nu e surprinzator, se ocupa in principal de creat un graf, nu are multe parti miscatoare, cum s-ar spune.

Consider ca 'mcts.py' ar trebui testat mai mult, nu am observat ceva neinregula cu arborii generati, cel mai ingrijorator lucru fiind doar adancimea arborelui respectiv, insa cand ai un branching factor de ordinul zecilor si un numar relativ mic de iteratii e de asteptat.

## 5 Posibile imbunatatiri

### 5.1 Performanta este abismala

Pe un Ryzen 5 3600 dureaza aproape o ora sa executi 10 000 de iteratii. Evident, lucrurile ar decurge mult mai bine daca m-as putea folosi de placa video pentru a parurge aborele de stari. Si asta ar fi imbunatatirea evidenta.

Insa, cred ca implementarea actuala poate fi imbunatatita si in alte moduri. Ar trebui sa fac niste profiling intai, dar ma astept ca cel mai mare bottleneck sa fie actiunea 'build' si functiile utilizate de aceasta. Ma refer la faptul ca de fiecare data cand vrei sa construiesti ceva, trebuie sa executi un BFS, apoi sa parcurgi o lista, apoi sa sortezi rezultatele, ca intr-un final sa parcurgi iar doua liste sortate.

### 5.2 Vizualizarea datelor

Probabil cea mai neglijata parte a proiectului, datorita timpului limitat.

Trebuie imbunatatit modul de afisare al arborelui la final. De asemenea, ar trebui sa alese niste metriki specifice jocului pe care sa le afisez separat.

### 5.3 Modul in care e parcurs arborele

Un joc intre 2 playeri ar trebui sa dureze 11 runde, iar fiecare runda are cate 2 actiuni. Deci 22 de actiuni in total. Prin urmare arborele ar trebui sa aiba o adancime de 22. Insa, in fiecare runda ai zeci de actiuni posibile. Pe scurt, nu putem parurge tot arborele. Pentru a ajunge la o adancime de 3, am avut nevoie de 10 000 de iteratii, pentru o adancime de 4 probabil ne apropiem de 100 000 de iteratii.

Daca imbunatatim semnificativ viteza cu care parcurgem aborele probabil putem ajunge la o adancime de 5 intr-un timp rezonabil(cateva ore), o adancime de 6 cu rabdare.

O alta posibila solutie ar fi sa executam un numar fix de iteratii (sa zicem 1000), alegem nodurile mai promitatoare de la adancimea cea mai mare si repetam asta pana ajungem la finalul jocului.

#### Observatie:-

O estimare a numarului de actiuni posibile(branching factor):

- imprumut - 1
- scout - 1
- pass - 1
- dezvoltare - [0, 42]
- vanzare industrie - [1,  $\approx$  5] (in jur de 5 daca cumva reupesti sa vinzi toate cladirile deodata, improbabil totusi)
- constructie conexiuni - [2,  $\approx$  30] (in jur de 30 de optiuni daca construiesti industrie in orase mari si amanii construcia de conexiuni pentru mai tarziu)
- constructie industrie - [0, 49]

In total ai undeva intre 6 si aproape 130 de optiuni, in functie de conditiile jocului. Iar branching factorul e in general pe undeva la mijloc. De ordinul zecilor.

Si daca am avea un branching factor de 6 mereu, asta ar rezulta in  $13,366,680,000,000,000,000,000$  de stari. Daca reusim cumva sa eliminam stari duplicate, poate ar fi de cateva ori mai mic.

In orice caz, nu putem parcurge toate starile arborelui.

## 5.4 Stilul codului

Fiind primul meu proiect major in Python, probabil am omis multe moduri prin care as fi putut face codul mai eficient si mai usor de inteles.

## 5.5 Alte idei

MCTS + RL

As putea incerca sa emulez strategia folosita pe AlphaZero si sa combin MCTS cu Reinforcement Learning pentru a obtine rezultate mai bune.

### Functiile de recompensa

Momentan functie de recompensa ia in calcul diferența de punctaj intre player-ul actual si oponentul cu cel mai mare punctaj.

Poate o alta functie de recompensa utila ar fi una care incurajeaza playerii sa acumuleze cat mai multe puncte.

### Alegerea cartilor de joc

Momentan playerii aleg cartile de joc in mod arbitrar, daca as include si cartile ca factor branching factor-ul probabil ar ajunge la cateva sute. Un branching factor asemanator cu jocul Go.

## 6 Concluzii

MCTS pur nu este optim pentru un joc precum Brass Birmingham deoarece numarul de stari posibile este imens, crescand exponential pentru fiecare posibila actiune a unui jucator.

In prezent, simulam doar una dintre cele doua ere iar jocul implica doar doi jucatori, pentru patru jucatori numarul de stari ar fi fost de cateva ori mai mare.

De asemenea, daca cartile de joc nu erau alese arbitrar cand un player executa o actiune, numarul de stari ar fi fost mult mai mare. Dar in acelasi timp acest lucru ne-ar fi ajutat sa observam ce influenta au cartile asupra jocului, cat de mult tine de noroc si cat de mult tine de capacitatea jucatorului.

Desi jocul a fost simplificat enorm, performanta tot lasa de dorit. Pentru a putea atinge scopul proiectului, acela de a descoperi strategii noi, avem nevoie de niste tehnici noi.

## 7 Bibliografie

Artificial Intelligence A Modern Approach (4th edition) - Stuart Russel & Peter Norvig