Departamentul de Calculatoare
Universitatea Tehnică din Cluj-Napoca

Inteligență artificială

Activitate de laborator

Nume: Bozdog Raluca - Delia,
Lazea Dragoș - Bogdan
Grupa: 30233
Email: ralucabozdog@gmail.com,
ldragosbogdan@gmail.com

Profesor de laborator: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

A1: Căutare

1.1 Introducere

Capitolul de față prezintă funcționalitățile Pacman pe care am decis să le dezvoltăm, cu scopul de a pune în valoare o selecție de algoritmi de căutare, cum ar fi:

- Depth-First Search & Breadth-First Search - algoritmi pentru care am ales o abordare "nonconformistă"
- Uniform-Cost Search - algoritm complet care găsește costul optim al deplasării
- A* Search - algoritm ce introduce ideea de euristică pentru o mai bună performanță a căutării
- Weighted A* - bazat pe găsirea unui echilibru între costul deplasării și euristica aleasă
- Iterative-Deepening Search - variantă modificată a algoritmului din *Artificial Intelligence - A Modern Approach* de Stuart Russel și Peter Norvig, pentru rezultate mai bune particular pe cazurile de utilizare studiate în contextul Pacman

Pe lângă problema clasică în care Pacman caută o singură "capsulă" de mâncare, poziționată în colțul din stânga jos al grilei de joc, am ales să explorăm și problema de căutare AnyFoodSearchProblem, în care Pacman își îndeplinește misiunea atunci când mănâncă oricare "porție" de mâncare din labirint.

Pentru a analiza importanța unei euristici portivite, am studiat următoarele euristici:

- Distanța Manhattan
- Distanța Euclidiană
- Distanța Cebîșev
- Octile Distance

1.2 Depth-First Search

Ideea care stă la baza acestei variante de implementare a algoritmului de căutare în adâncime este aceea de a evita definirea unei noi clase pentru a reține starea anterioară fiecărei stări pe care o traversează Pacman. Această informație este critică, fiindcă algoritmul pornește de la poziția inițală a lui Pacman și se oprește la starea *goalState*, însă pentru a trimite secvența de acțiuni pe care Pacman trebuie să le execute ca să ajungă de la starea de pornire la cea finală, este nevoie de a analiza traseul în sens invers, de la ultima stare la prima, traversând legăturile de tip "părinte" dintre stări. Informația reținută în mod tradițional în câmpul de "părinte" a fost modelată aici cu ajutorul tuplelor, mai precis cu ajutorul tuplelor imbricate: în stiva folosită pentru a reține frontiera problemei se introduce de fiecare dată o tuplă cu două elemente: starea curentă și starea anterioară (starea "părinte").

```
def depthFirstSearch(problem):
    visited = []
    st = util.Stack()
    st.push(((problem.getStartState(), ()), None))

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux[0][0])):
            break
        if (aux[0][0] not in visited):
            visited.append(aux[0][0])
            for x in problem.getSuccessors(aux[0][0]):
                st.push((x, aux))

    moves = []

    while (aux[0][1]):
        #print "Next step:", aux[0][1]
        moves.append(aux[0][1])
        aux = aux[1]
    moves.reverse()
    return moves
```

Code Listing 1.1: Depth-First Search

1.3 Breadth-First Search

Algoritmul Breadth-First Search diferă ca implementare de căutarea în adâncime doar prin faptul că utilizează structura de coadă pentru reținerea stărilor frontieră, în loc de stivă. La fel ca în cazul DFS, s-a folosit referința către stările anterioare prin tuple, fără a mai fi nevoie de clase anexe.

```
def breadthFirstSearch(problem):
    visited = []
    st = util.Queue()
    st.push(((problem.getStartState(), ()), None))

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux[0][0])):
            break
```

```
        if (aux[0][0] not in visited):
            visited.append(aux[0][0])
            for x in problem.getSuccessors(aux[0][0]):
                st.push((x, aux))

    moves = []

    while (aux[0][1]):
        #print "Next step:", aux[0][1]
        moves.append(aux[0][1])
        aux = aux[1]
    moves.reverse()
    return moves
```

Code Listing 1.2: Breadth-First Search

1.4 Uniform-Cost Search

Pentru algoritmul Uniform-Cost Search s-a introdus noțiunea de "nod" prin definirea clasei *Node*, a cărei implementare și structură se va detalia în secțiunea următoare. S-a luat această decizie fiindcă actualizarea costului unei stări pe măsură ce se explorează noi drumuri devenea greoaie și ineficientă în lipsa existenței unor referințe de "părinte" ușor de urmărit. Algoritmul este asemănător cu BFS, însă folosește o coadă de priorități în locul cozii clasice. Astfel, elementul scos din frontieră la fiecare pas este cel al cărui cost are valoarea minimă. De asemenea, spre deosebire de BFS, unde un nod vizitat o dată nu mai este apoi deloc analizat, în cazul UCS, dacă un nod a mai fost deja vizitat, se actualizează costul nodului vizitat dacă la cea mai recentă iterație s-a obținut un cost mai bun. Pentru a realiza această comparație s-a folosit metoda *betterCost*, prezentată mai jos.

```
def uniformCostSearch(problem):
    visited = []
    st = util.PriorityQueue()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n, n.cost)

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux.state)):
            break
        if (aux.state not in visited):
            visited.append(aux.state)
            for x in problem.getSuccessors(aux.state):
                node = Node(x[0], x[1], aux.cost + x[2], aux)
                if (node.state not in visited):
                    st.push(node, node.cost)
                elif betterCost(st, node):
                    st.update(node, node.cost)

    moves = []

    while (aux.parent):
        print "Next step:", aux.action, aux.cost
        moves.append(aux.action)
        aux = aux.parent
    moves.reverse()
    return moves
```

Code Listing 1.3: Uniform-Cost Search

```
def betterCost(list, node):
    while (list.isEmpty == False):
        cell = list.pop()
        if (cell.state == node.state and cell.cost > node.cost):
            return 1
    return 0;
```

Code Listing 1.4: metoda *betterCost*

1.4.1 Clasa *Node*

Clasa *Node* a fost definită pentru a reprezenta cât mai clar spațiul stărilor pentru problema de căutare, în cazul algoritmilor Uniform-Cost Search, A* Search, Weighted A* Search și Iterative-Deepening Search. Fiecare nod are o stare (coloana si linia la care se regăsesc pe grila de joc), o acțiune (mutarea prin care s-a ajuns la starea nodului curent), un cost și un părinte (nodul anterior nodului curent la explorare). S-au definit metode getter pentru fiecare dintre aceste proprietăți.

```
class Node:
    def __init__(self, state, action, cost, parent):
        self.state = state
        self.action = action
        self.cost = cost
        self.parent = parent

    @property
    def state(self):
        return self.__state

    @property
    def action(self):
        return self.__action

    @property
    def cost(self):
        return self.__cost

    @property
    def parent(self):
        return self.__parent
```

Code Listing 1.5: Clasa *Node*

1.5 A* Search

A* Search este un algoritm asemănător cu Uniform-Cost Search, care, spre deosebire de acesta, pe lângă calcularea costului efectiv de deplasare la un nod din spațiul stărilor, ia în calcul și o estimare aleasă de dezvoltator pentru a prezice costul pe care agentul care se deplasează îl va mai "plăti" pentru a ajunge la destinație. Această estimare poartă denumirea de *euristică* și stă la baza Inteligenței Artificiale. Alegerea unei euristici potrivite problemei ce se dorește a fi rezolvată este un lucru esențial pentru a valorifica la adevăratul potențial căutarea A*.

În tabelul de mai jos s-au înregistrat rezultatele testării algoritmului A* Search folosind diferite euristici a căror analiză detaliată se va realiza în secțiunea 1.9. Se poate observa că euristica *distanța Manhattan* este cea mai performantă pentru A* atât ca număr de noduri expandate, cât și ca timp de execuție. Costul de deplasare este același pentru toate euristicile, fiindcă alegerea oricărei euristici admisibile generează un rezultat optim al costului. Grid-ul pe care s-au făcut testele este *mediumMaze*, cu problema de căutare *PositionSearchProblem*.

Euristica	Noduri Expandate	Timp de execuție (sec)	Cost
distanța Manhattan	221	0.03455	68
distanța euclidiană	226	0.05411	68
distanța Cebîșev	228	0.04193	68
Octile Distance	223	0.05694	68

```
def aStarSearch(problem, heuristic):
    visited = []
    st = util.PriorityQueue()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n, n.cost)

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux.state)):
            break
        if (aux.state not in visited):
            visited.append(aux.state)
            for x in problem.getSuccessors(aux.state):
                node = Node(x[0], x[1], aux.cost + x[2], aux)
                if (node.state not in visited):
                    st.push(node, node.cost + heuristic(node.state, problem))
                elif betterCost(st, node):
                    st.update(node, node.cost + heuristic(node.state, problem))

    moves = []

    while (aux.parent):
        print "Next step:", aux.action, aux.cost
        moves.append(aux.action)
        aux = aux.parent
    moves.reverse()
    return moves
```

Code Listing 1.6: A* Search

1.6 Weighted A* Search

Algoritmul Weighted A*, propus de Pohl în anul 1970, se bazează pe aceeași idee ca și A*, cu deosebirea că evaluarea în calcularea costului total, valoarea euristicii este ponderată cu un număr supraunitar. Astfel, funcția cost devine:

$$f(n) = g(n) + (1 + \epsilon)\, h(n), \quad \epsilon \geq 0$$

Algoritmul Weighted A* s-a dovedit a fi mult mai eficient decât A* clasic în multe situații din practică, ponderarea valorii evaluate de euristica adoptată având adesea drept rezultat o scădere a timpului de căutare, testele rulate evidențiind faptul că, odată cu creșterea valorii ϵ, scade numărul de noduri expandate. Spre exemplu, la rularea algoritmului pe layout-ul mediumMaze, s-au obținut următoarele rezultate pentru valorile lui ϵ indicate în tabel:

ϵ	Euristica	Noduri Expandate	Timp de execuție (sec)	Cost
0,5	distanța Manhattan	229	0.06889	68
2,25	distanța Manhattan	212	0.06662	68
0,5	distanța euclidiană	228	0.07090	68
2,25	distanța euclidiană	222	0.06854	68
0,5	distanța Cebîșev	228	0.05941	68
2,25	distanța Cebîșev	223	0.06076	68
0,5	octile distance	227	0.04736	68
2,25	octile distance	209	0.04545	68

Prin urmare, în urma testelor rulate, se poate observa că, pentru valori mai mari ale ponderii ϵ, numărul de noduri expandate scade, indiferent de euristica utilizată. Totodată, analizând rezultatele obținute, este de menționat faptul că euristica octile distance s-a dovedit a fi cea mai eficientă din punctul de vedere al numărului de noduri expandate în timpul căutării.

```
def weightedAStarSearch(problem, heuristic):
    visited = []

    st = util.PriorityQueue()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n, n.cost)

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux.state)):
            break
        if (aux.state not in visited):
            visited.append(aux.state)
            for x in problem.getSuccessors(aux.state):
                node = Node(x[0], x[1], aux.cost + x[2], aux)
                eps = 2.25
                if (node.state not in visited):
                    st.push(node, node.cost + (1 + eps) * heuristic(node.state,
    problem))
                elif betterCost(st, node):
                    st.update(node, node.cost + (1 + eps) * heuristic(node.state,
     problem))

    moves = []

    while (aux.parent):
```

```
        #print "Next step:", aux.action, aux.cost
        moves.append(aux.action)
        aux = aux.parent
    moves.reverse()
    return moves
```

Code Listing 1.7: Weighted A* Search

1.7 Iterative-Deepening Search - adaptare după pseudocodul AIMA

Algoritmul *Iterative-Deepening Search*, a cărui implementare poate fi regăsită mai jos, funcționează pe următoarea idee: caut soluția problemei la o adâncime minimă stabilită (aici, 0); dacă nu găsesc soluția la adâncimea curentă, avansez progresiv la adâncime tot mai mare până când ajung la starea destinație; dacă ajung la adâncimea maximă stabilită (în cazul de față setată la valoarea 300, fiindcă niciun grid Pacman deja construit nu rezolvă o problemă al cărei cost să ajungă la 300) și totuși nu am găsit soluția, mă opresc și returnez NIL (Pacman nu va executa nicio mișcare).

IDS folosește o stivă pentru a reține stările frontieră, motiv pentru care se aseamănă în execuție algoritmului Depth-First Search. În fapt, costul de deplasare al Iterative-Deepening Search coincide cu cel al căutării clasice în adâncime.

După cum se poate observa din cod, Iterative-Deepening Search folosește o funcție numită aici *depthLimitedSearch*, care face căutarea la o adâncime dată (funcția utilă, care execută efectiv căutarea), și o funcție învelitoare care apelează *depthLimitedSearch* pe rând pentru fiecare valoare a parametrului *depth*.

```
def iterativeDeepeningSearch(problem):
    moves = []
    for depth in range(0, 300):
        result = depthLimitedSearch(problem, depth)
        if(result.parent != None):
            break

    while (result.parent):
        #print "Next step:", result.state, result.action, result.cost
        moves.append(result.action)
        result = result.parent
    moves.reverse()
    return moves


def depthLimitedSearch(problem, depth):
    st = util.Stack()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n)
    result = Node((0,0), (), 0, None)
    visited = []

    while(st.isEmpty() == False):
        aux = st.pop()
        if (aux.state not in visited):
            visited.append(aux.state)
            if(problem.isGoalState(aux.state)):
                return aux
```

```
            if(aux.cost > depth):
                return result
            else:
                for x in problem.getSuccessors(aux.state):
                    node = Node(x[0], x[1], aux.cost + x[2], aux)
                    st.push(node)

    return result
```

Code Listing 1.8: Iterative-Deepening Search

1.8 AnyFoodSearchProblem

O nouă abordare, ușor diferită față de cea în care câștigul era reprezentat de sosirea avatarului Pacman în poziția caracterizată de coordonatele (1, 1), prezentă în cazul problemei PositionSearchProblem, este cea dezvoltată în cadrul problemei AnyFoodSearchProblem, unde țelul final al jocului îl reprezintă găsirea de către Pacman a unei bucăți oarecare de mâncare. Singura diferență existentă între cele două maniere de concretizare a câștigului este reprezentată de testarea succesului, căutarea realizându-se identic în cazul ambelor probleme menționate. Prin urmare, diferențele de implementare dintre cele două abordări au fost înregistrare în cadrul funcției isGoalState, care are drept scop verificarea finalității jocului. Dacă în cazul problemei PositionSearchProblem funcția returna o valoare adevărată când avatarul ajungea în poziția ce caracteriza colțul din stânga-jos al frame-ului atașat jocului, pentru soluționarea problemei AnyFoodSearchProblem valoarea True va fi returnată de funcția isGoalState în momentul în care Pacman întâlnește o bucată de mâncare. Ideea de implementare este următoarea: valoarea variabilei booleene, isGoal, returnate de funcția isGoalState devine True în momentul în care, în grid-ul ce memorează pozițiile pe care se găsește mâncarea, la poziția determinată de starea curentă a avatarului prin coordonatele (x, y) se găsește o valoare True, indicând astfel faptul că Pacman s-a poziționat pe o bucată de mâncare. Codul Python ce conține implementarea propriu zisă a întregii probleme AnyFoodSearchProblem poate fi regăsit mai jos.

```
class AnyFoodSearchProblem(PositionSearchProblem):
    def __init__(self, gameState):
        "Stores information from the gameState.  You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()
        self.data = gameState.data
        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self.visualize = True
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT
   CHANGE

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        isGoal = False
        if self.food[state[0]][state[1]]:
            isGoal = True

        # For display purposes only
```

```
        if isGoal and self.visualize:
            self._visitedlist.append(state)
            import __main__
            if '_display' in dir(__main__):
                if 'drawExpandedCells' in dir(__main__._display): #
    @UndefinedVariable
                    __main__._display.drawExpandedCells(self._visitedlist) #
    @UndefinedVariable

        return isGoal
```

Code Listing 1.9: AnyFoodSearchProblem

1.8.1 Adaptarea codului primit la problema aleasă

O problemă întâmpinată în dezvoltarea *AnyFoodSearchProblem* a fost faptul că această problemă nu este neapărat o problemă clasică de căutare, adică pe grila de joc se pot găsi mai multe "capsule" de mâncare, nu doar una singură. Problema intervine din faptul că în modul clasic de joc, modelat de clasele *PacmanRules* și *ClassicGameRules*, scopul jocului este ca Pacman să nu întâlnească fantomele și să mănânce toată mâncarea de pe grila de joc. Dacă pe grilă avem o singură poziție ocupată cu mâncare, atunci când Pacman ajunge pe poziția respectivă, el mănâncă automat mâncarea, și cum nu mai există mâncare de căutat, și-a îndeplinit misiunea și a câștigat. Dacă, însă, oferim ca suport un grid nou în care să existe mai multe poziții cu mâncare, după ce Pacman consumă prima "capsulă" de mâncare, în cazul problemei *AnyFoodSearchProblem* ne-am aștepta ca acesta să câștige, fiindcă și-a îndeplinit misiunea propusă. Totuși, din cauză că pe grila de joc mai rămâne mâncare neconsumată de Pacman, după regulile clasice, Pacman nu a câștigat. Am rezolvat acest lucru prin consumarea artificială a tuturor "capsulelor" de mâncare în momentul când Pacman consumă prima "capsulă", mai exact setând la valoarea *False* întreaga matrice care reține dacă se află sau nu mâncare pe fiecare poziție din grid, prin intermediul metodei *getNumFood(self)* a clasei *GameState*.

```
def getNumFood( self ):
        for i in range (0, self.data.food.width):
            for j in range (0, self.data.food.height):
                self.data.food[i][j] = False
        return self.data.food.count()
```

Code Listing 1.10: metoda *getNumFood(self)* a clasei *GameState*

1.9 Euristici explorate

1.9.1 Distanța Manhattan

Distanța Manhattan reprezintă distanța de la un punct la altul, atunci când deplasarea se face strict pe drepte paralele cu axele de coordonate. Aceasta s-a dovedit a fi cea mai performantă euristică pentru algoritmul A* Search.

```
def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
```

Code Listing 1.11: Manhattan Distance Heuristic

1.9.2 Distanța Euclidiană

Distanța Euclidiană reprezintă distanța matematică dintre două puncte (lungimea segmentului ce le unește).

```
def euclideanHeuristic(position, problem, info={}):
    "The Euclidean distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5
```

Code Listing 1.12: Euclidean Distance Heuristic

1.9.3 Distanța Cebîșev

Distanța Cebîșev reprezintă un caz special al distanței diagonale, o euristică ce estimează deplasarea minimă pe diagonală de la poziția curentă spre poziția finală. Astfel, distanța diagonală depinde de doi parametri: D, reprezentând costul unei deplasări nediagonale (orizontale sau verticale) și D2, care exprimă costul unei deplasări pe diagonală. În cazul distanței Cebîșev, valorile parametrilor sunt egale între ele și egale cu 1 (D = D2 = 1). Practic, în definirea distanței Cebîșev se pornește de la premisa că atât costul deplasării nediagonale, cât și cel al deplasării pe diagonală sunt egale și au valoarea 1. Codul Python ce implementează această euristică este următorul:

```
def chebyshevHeuristic(position, problem, info={}):
    "The Chebyshev distance heuristic"
    xy1 = position
    xy2 = problem.goal
    return max(xy1[1] - xy2[1], xy1[0] - xy2[0])
```

Code Listing 1.13: Chebyshev Distance Heuristic

1.9.4 Octile Distance

Octile distance reprezintă, de asemenea, un caz particular al distanței diagonale. Prin setarea parametrilor la valorile D = 1 și D2 = $\sqrt{2}$ se obține octile distance. Ideea de calcul a acestui tip de distanță este determinarea costului total în cazul în care deplasarea se face strict orizontal și vertical, din care se va scădea numărul pașilor evitați prin deplasarea pe diagonală, implementarea fiind următoarea:

```
def octileHeuristic(position, problem, info={}):
    "The Octile distance heuristic"
    xy1 = position
    xy2 = problem.goal
    dx = abs(xy1[0] - xy2[0])
    dy = abs(xy1[1] - xy2[1])
    D = 1
    D2 = math.sqrt(2)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Code Listing 1.14: Octile Distance Heuristic

1.10 Compararea performanțelor. Concluzii

Pentru a putea realiza o comparație cât mai relevantă între algoritmii studiați, și pentru a concluziona care dintre aceștia este cel mai performant pentru problema de căutare Pacman s-a analizat performanța tuturor algoritmilor pentru grid-ul *mediumMaze*.

Pentru algoritmul A* Search s-a ales ca euristică distanța Manhattan, cea mai performantă variantă a acestui algoritm pe cazul analizat (vezi secțiunea 1.5).

Pentru algoritmul Weighted A* Search s-a ales ca euristică Octile Distance cu parametrul $\epsilon = 2.25$, cea mai performantă variantă a acestui algoritm pe cazul particular analizat (vezi secțiunea 1.6).

Rezultatele testelor sunt înregistrate în tabelul de mai jos:

Algoritm	Noduri Expandate	Timp de execuție (sec)	Cost
Depth-First	146	0.03928	130
Breadth-First	269	0.03727	68
Uniform-Cost	269	0.05950	68
A*	221	0.03455	68
Weighted A*	209	0.04545	68
IDS	9046	0.20780	130

Se poate concluziona clar că algoritmul cel mai ineficient este Iterative-Deepening Search, căruia dacă nu i se indică exact adâncimea la care să caute soluția problemei, pierde foarte mult timp expandând la fiecare iterație aceleași noduri pe care le-a analizat și anterior, de unde este explicabil atât timpul mult mai mare de execuție, cât și numărul de zeci de ori mai mare al nodurilor expandate.

Din punctul de vedere al costului deplasării, nici Depth-First Search nu este un algoritm preferabil. El nu găsește calea cea mai scurtă (mai puțin costisitoare), nefiind un algoritm optimal. Totuși, de menționat numărul sesizabil mai mic de noduri pe care le expandează DFS spre deosebire de ceilalți algoritmi de căutare. Așadar, dacă scopul unei probleme este de a

găsi o soluție, indiferent de costul drumului, cu explorare cât mai redusă a spațiului stărilor (pentru minimizarea rolului memoriei), căutarea în adâncime poate fi o variantă de dorit.

Breadth-First Search și Uniform-Cost Search expandează același număr de noduri (UCS este, în cazul de față în care toate stările au un cost de deplasare de 1, echivalent ca rezultat lui BFS), și ambele găsesc soluția optimă a problemei, cu o performanță în timp mai bună pentru BFS (sunt, deci, sesizabile operațiile UCS suplimentare de calcul și de actualizare a costului nodurilor).

Introducerea ideii de euristică în calcule se dovedește a fi justificată, algoritmii A* și Weighted A* având performanțe foarte bune atât din punct de vedere al numărului de noduri expandate, cât și în ceea ce privește timpul de execuție, cu mențiunea că euristica aleasă trebuie să se potrivească problemei, iar în cazul Weighted A* ponderea care se acordă euristicii în calcule joacă un rol extrem de important. Dacă atât ponderea cât și euristica sunt alese în mod potrivit, se poate observa că Weighted A* expandează cele mai puține noduri, un indiciu clar al performanței, cu atât mai important cu cât crește dimensiunea problemei.

Chapter 2

A2: Logică

2.1 Introducere

Capitolul de față prezintă implementarea noastră în First Order Logic a jocului de societate *Mafia*, precum și modelarea unor probleme complexe de tipul zebra-puzzle.

Pentru rezovarea problemelor propuse am ales folosirea Mace4, un program care caută modele finite ale formulelor de ordinul întâi. Implicațiile logice vor fi atent și complet explicate, în timp ce transpunerea din limbaj natural în logică de ordinul întâi se dorește a fi cât mai intuitivă și la îndemână.

Subiectele discutate în acest capitol:

- Mafia - joc de societate adus în lumea FOL
- Ghicitoarea lui Einstein - problemă ce se crede a fi propusă de Einstein, și despre care acesta ar fi susținut că doar 2% din populația lumii o poate rezolva
- Problema donării de sânge - o extindere a Einstein Riddle

2.2 Mafia

În acest subcapitol se va modela în First Order Logic (FOL), folosind Mace4, și se va simula corespunzător o variantă a jocului de societate Mafia, cunoscut și sub numele de Killer, adaptată la regulile logicii de ordinul 1. Acest joc presupune identificarea unui criminal dintr-un grup de personaje, având punct de plecare diverse indicii privind persoanele care au fost ucise și relațiile dintre acestea.

În cele ce urmează vor fi prezentate succint rolurile tuturor personajelor prezente în joc și regulile de bază ale jocului.

Cadrul fictiv în care are loc acțiunea jocului îl reprezintă un sat în care toți locuitorii se cunosc, având însă fiecare atribuții și interese diferite. Printre acești cetățeni de rând se află un criminal al cărui scop îl reprezintă eliminarea forțelor binelui pentru a putea pune el însuși stăpânire pe întregul sat în care acesta își caută victimele. De menționat este faptul că, deși se cunosc între ei, acești cetățeni nu își cunosc decât propriile atribuții, fără a avea cunoștință care dintre ceilalți este doctorul, sau polițistul, sau mai ales criminalul. Așadar, locuitorii satului imaginar se împart în două tabere: binele, reprezentat prin intermediul polițistului, al doctorului și al cetățenilor de rând, al căror scop este identificarea și pedepsirea celui care le pune viața în pericol, și răul, avându-l drept reprezentant pe criminalul infiltrat în comunitate. De asemenea, jocul necesită prezența unui personaj aparte, asemeni unui narator din operele

literare, care nu ia parte efectiv la cele întâmplate, ci supraveghează și coordonează fluxul evenimentelor derulate pe parcursul episoadelor jocului. Acțiunea se desfășoară pe parcursul mai multor zile și nopți, până în momentul în care una dintre cele două tabere se poate declara învingătoare.

În fiecare noapte, în timp ce reprezentanții binelui dorm, criminalul alege un personaj dintre cei rămași în viață pe care să îl omoare. După înfăptuirea crimei, acesta adoarme, moment în care se trezește medicul, alegând o persoană pe care să o salveze de la moarte, neavând însă cunoștință de alegerea făcută de către criminal. În momentul în care medicul adoarme din nou, polițistul își intră în atribuții, alegând să aresteze unul dintre cetățenii rămași în viață, pe care îl suspectează de înfăptuirea crimei. La sosirea zorilor, toți cetățenii se trezesc, iar, prin intermediul personajului narator, află despre cele întâmplate în noaptea ce tocmai s-a încheiat.

Prin urmare, rolurile personajelor prezente în cadrul jocului se pot defini după cum urmează:

- Criminalul (The Killer) - ucide în fiecare noapte câte un cetățean, pentru a-și îndeplini scopul de a elimina forțele binelui care stau în calea înfăptuirii dezideratului său, acela de a prelua conducerea satului; nu cunoaște identitățile cetățenilor, neștiind care dintre aceștia este polițist sau medic; în momentul în care acesta este arestat de către polițist forțele binelui se declară învingătoare;

- Polițistul (The Policeman) - luptă pentru a asigura siguranța cetățenilor și arestează cetățenii pe care îi suspectează, neștiind însă adevărata sa identitate și având drept scop identificarea criminalului și pedepsirea acestuia pentru toate fărădelegile pe care le-a comis; poate aresta câte un singur cetățean în fiecare noapte, iar dacă acesta se dovedește a fi nevinovat, este eliberat la sosirea dimineții; întrucât el reprezintă singura speranță a cetățenilor de duce o viață liniștită, victoria sa, reprezentată prin demascarea și arestarea criminalului, este identificată cu victoria forțelor binelui, în timp ce uciderea acestuia de către criminal este asociată, în final, cu câștigul răului;

- Medicul (The Doctor) - alege în fiecare noapte o singură persoană pe care dorește să o salveze de la moarte; dacă persoana aleasă de către doctor este întocmai persoana ucisă în noaptea respectivă de către criminal, aceasta este readusă miraculos la viață, însă dacă alegerea sa nu coincide cu alegerea făcută de către criminal, cetățeanul ucis nu mai poate fi salvat;

- Cetățenii (Citizens) - trăiesc în satul în care criminalul își alege victimele și își doresc reinstaurarea liniștii și a siguranței în comunitatea lor; toate personajele speciale (criminal, polițist, medic) sunt și cetățeni;

Pentru a asigura corectitudinea variantei finale de program, în dezvoltarea codului pentru modelarea jocului Mafia am traversat succesiv următoarele etape:

- Definirea rolului de killer
- Introducerea rolului de polițist
- Introducerea rolului de doctor
- Definitivarea mecanismului de stabilire a câștigătorului

2.2.1 Definirea rolului de killer

Reguli

Pentru a trasa atribuțiile killer-ului, s-a stabilit că în fiecare sat există exact un killer, prin următoarele afirmații logice:

```
killer(x) -> (killer(y) <-> x = y).
exists x killer(x).
```

Code Listing 2.1: Exact un killer

Funcția pick(x, y), care va fi folosită și în alte scopuri în următoarele variante ale jocului, are aici rolul de a-i permite killer-ului să aleagă persoana pe care dorește să o ucidă.

Este important de stabilit diferența dintre *ucis (murdered)* și *mort (dead)*:

- Un personaj *murdered* este cineva care trăia la începutul rundei și a fost ucis în timpul rundei curente de către killer
- Un personaj *dead* este deja mort la începutul rundei de joc, adică a fost ucis într-o rundă anterioară

În cadrul acestei variante incipiente de joc, doar killer-ul poate alege

```
pick(x, y) -> killer(x).
```

Code Listing 2.2: Doar killer-ul poate alege

El poate alege doar o persoană pe rundă, și nu poate alege pe cineva care deja este mort

```
pick(x, y) -> (pick(x, z) <-> y = z).
pick(x, y) -> -dead(y).
```

Code Listing 2.3: Alege o singură persoană care nu este deja moartă

Killer-ul nu se poate alege pe sine

```
pick(x, y) -> x != y.
```

Code Listing 2.4: Nu se poate alege pe sine

Jucătorul ales de killer este ucis, și doar acela

```
pick(x, y) & killer(x) -> murdered(y) & (all z ((y != z) -> -murdered(z))).
```

Code Listing 2.5: Doar jucătorul ales este ucis

Exact un jucător este ucis în fiecare rundă

```
murdered(x) -> (murdered(y) <-> x = y).
exists x murdered(x).
```

Code Listing 2.6: Exact un jucător este ucis

Jucătorul ucis nu poate fi killer-ul

```
murdered(x) -> -killer(x).
```

Code Listing 2.7: Killer-ul nu poate fi ucis

Jucătorul ucis a fost ales de către killer

```
exists y exists x (killer(y) & pick(y, x) & murdered(x)).
```

Code Listing 2.8: Cel ucis a fost ales de către killer

Exemplu

Pornind de la presupunerea că există doi jucători ai acestei variante de Mafia, Ann și Brad, și că niciunul nu este mort la începutul rundei, dacă știm că Brad a fost ucis, avem toate informațiile necesare pentru a modela tabloul rolurilor:

Se exclude varianta unei sinucideri, deci Brad a fost ucis de unicul jucător în afară de el, adică de Ann, care este, invariabil, killer.

```
formulas(assumptions).

-dead(Ann).
-dead(Brad).

murdered(Brad).

end_of_list.
```

Code Listing 2.9: Exemplu de presupuneri pentru rulare

2.2.2 Introducerea rolului de polițist

Reguli

Se stabilește inițial faptul că există un singur polițist:

```
policeman(x) -> (policeman(y) <-> x = y).
exists x policeman(x).
```

Code Listing 2.10: Exact un polițist

Pentru orice persoană este ușor de înțeles că un killer nu poate fi polițist, dar acest lucru trebuie tradus în logică de ordinul întâi și pentru interpretorul folosit:

```
killer(x) -> -policeman(x).
policeman(x) -> -killer(x).
```

Code Listing 2.11: Un singur rol

Odată cu introducerea rolului de polițist se extinde și semantica funcției pick(x, y). Acum această funcție poate avea ca prim operand fie killer-ul, fie polițistul – killer-ul alege pe cine să ucidă, polițistul alege pe cine să aresteze. Cel care alege trebuie să nu fie nici mort, nici ucis (precizare care se adresează, de fapt, doar polițistului, deoarece killer-ul nu ajunge niciodată în starea dead sau murdered).

```
pick(x, y) -> killer(x) | policeman(x).
pick(x, y) -> (pick(x, z) <-> y = z).
pick(x, y) -> -dead(x) & -murdered(x).
pick(x, y) -> -dead(y).
pick(x, y) -> x != y.
```

Code Listing 2.12: Extinderea funcției pick

Doar jucătorul ales de către polițist este arestat

```
pick(x, y) & policeman(x) -> arrested(y) & (all z ((y != z) -> -arrested(z))).
```

Code Listing 2.13: Numai cel ales de polițist este arestat

Dacă polițistul este mort sau ucis nu se mai pot face arestări

```
dead(x) & policeman(x) -> (all z -arrested(z)).
murdered(x) & policeman(x) -> (all z -arrested(z)).
```

Code Listing 2.14: Polițistl nu poate face arestări dacă este mort sau ucis

Cel mult un jucător este arestat (exact unul dacă polițistul trăiește, niciunul în caz contrar)

```
arrested(x) -> (arrested(y) <-> x = y).
exists y (policeman(y) & -dead(y)) -> exists x arrested(x).
```

Code Listing 2.15: Maxim un jucător arestat

Cel arestat nu poate fi polițistul

```
arrested(x) -> -policeman(x).
```

Code Listing 2.16: Polițistul nu se poate aresta pe sine

Cel arestat a fost ales de către polițist

```
arrested(x) & policeman(y) -> pick(y, x).
```

Code Listing 2.17: Arestarea înseamnă alegerea de către polițist

Exemple

Știu că există trei jucători, Ann, Brad și Chris, și că niciunul dintre ei nu este mort inițial. Ann este killer, Chris este ucis și Ann este arestată. Pot deduce imediat că Brad este polițistul (Ann ar fi putut să fie arestată de către Brad sau de către Chris; unul dintre ei este polițist; din moment ce Chris este ucis, dacă el ar fi fost polițist atunci Ann nu ar fi putut să fie arestată; dar cum Ann este arestată, înseamnă că nu Chris este Polițistul, ci Brad).

```
formulas(assumptions).

-dead(Ann).
-dead(Brad).
-dead(Chris).

killer(Ann).
murdered(Chris).
arrested(Ann).

end_of_list.
```

Code Listing 2.18: Exemplul 1 de presupuneri pentru rulare

Dacă în același caz în care Ann, Brad și Chris sunt jucători, niciunul mort inițial știu că Ann este killer, Chris este polițist și Ann este arestată, ce concluzie pot trage despre Brad?

Aparent nicio concluzie, dar Ann este killer. Ea este arestată și Chris este polițist, deci Chris a arestat-o pe Ann, ceea ce înseamnă că nu Chris a fost ucis. Dar totuși Ann a ucis pe cineva, ceea ce înseamnă că Brad a fost ucis.

```
formulas(assumptions).

-dead(Ann).
-dead(Brad).
```

```
-dead(Chris).

killer(Ann).
policeman(Chris).
arrested(Ann).

end_of_list.
```

Code Listing 2.19: Exemplul 2 de presupuneri pentru rulare

2.2.3 Introducerea rolului de doctor

Reguli

Se impune condiția existenței unui singur medic, prin următoarele afirmații logice:

```
doctor(x) -> (doctor(y) <-> x = y).
exists x doctor(x).
```

Code Listing 2.20: Exact un doctor

Se extinde condiția atribuirii unui rol unic fiecărui personaj, conform definirii noului rol, acela de medic:

```
doctor(x) -> -killer(x) & -policeman(x).
killer(x) -> -doctor(x) & -policeman(x).
policeman(x) -> -killer(x) & -doctor(x).
```

Code Listing 2.21: Un singur rol

Se extinde, de asemenea, și condiția efectuării unei singure alegeri în fiecare rundă a jocului de către cele trei personaje care au această posibilitate (criminal, polițist, medic):

```
pick(x, y) -> killer(x) | doctor(x) | policeman(x).
pick(x, y) -> (pick(x, z) <-> y = z).
```

Code Listing 2.22: Extinderea funcției pick

Se specifică faptul că o persoană care alege poate să fie ucisă dar salvată(ex: un polițist)

```
pick(x, y) -> -dead(x) & -murdered(x) | -dead(x) & murdered(x) & saved(x).
pick(x, y) -> -dead(y).
pick(x, y) -> x != y.
```

Code Listing 2.23: Cineva ucis dar salvat are dreptul să aleagă

Se definește rolul propriu-zis al doctorului, specificând clar faptul că singurul personaj salvat de la moarte în runda curentă a jocului este cel ales de către medic, toți ceilalți fiind expuși pericolului de a fi uciși de către criminal:

```
pick(x, y) & doctor(x) -> saved(y) & (all z ((y != z) -> -saved(z))).
```

Code Listing 2.24: Doctorul salvează o singură persoană

Precum în cazul polițistului, se impune constrângerea ca atunci când medicul este mort sau ucis de către criminal, acesta nu mai poate salva alți cetățeni:

```
murdered(x) & doctor(x) -> (all z -saved(z)).
dead(x) & doctor(x) -> (all z -saved(z)).
```

Code Listing 2.25: Doctorul poate salva doar dacă trăiește

Se specifică faptul că o singură persoană poate fi salvată la un moment dat, iar acea persoană nu poate fi medicul însuși, ea fiind aleasă de către acesta pentru a fi salvată:

```
saved(x) -> (saved(y) <-> x = y).
saved(x) -> -doctor(x).
saved(x) & doctor(y) -> pick(y, x).
```

Code Listing 2.26: Doctorul poate salva o singură persoană și nu pe sine însuși

Se tratează în mod separat cazul în care polițistul este ucis și nu este salvat, caz în care nu se mai pot face arestări

```
murdered(x) & policeman(x) & -saved(x)-> (all z -arrested(z)).
```

Code Listing 2.27: Dacă polițistul este ucis și nu este salvat nu se mai pot face arestări

Se identifică și se definește explicit cazul special în care polițistul este ucis, dar fiind salvat de medic, el poate alege în continuare un personaj pentru a-l aresta, iar dacă acest personaj nu este chiar criminalul căutat, jocul continuă, nefiind declarat niciun câștigător:

```
killer(x) & -arrested(x) & murdered(y) & policeman(y) & saved(y)-> (all z -win(z
    )).
```

Code Listing 2.28: Dacă polițistul este ucis dar salvat și nu îl arestează pe killer

Exemple

În joc sunt prezente trei personaje: Anna, Brad și Chris. Presupunând că niciunul dintre aceștia nu este mort la începutul rundei curente a jocului și că Ann este criminalul, Brad este doctorul, iar Chris polițistul, dacă Ann alege să îl ucidă pe Chris, însă Chris o arestează mai apoi pe Ann, care sunt concluziile care pot fi desprinse?

Întrucât Chris, deși ucis de către Ann, încă este capabil de a o aresta pe aceasta, concluzia evidentă este că Brad l-a salvat pe Chris de la moarte, acesta arestând întocmai criminalul căutat și înfăptuind astfel victoria forțelor binelui. Ipoteza mai sus menționată evidențiază întocmai cazul special în care polițistul, deși ucis de către criminal, este readus la viață de către medic și poate aresta personajul pe care îl suspectează de înfăptuirea crimelor.

```
formulas(assumptions).

-dead(Ann).
-dead(Brad).
-dead(Chris).

killer(Ann).
doctor(Brad).
policeman(Chris).

pick(Ann, Chris).
pick(Brad, Chris).
pick(Chris, Ann).
end_of_list.

formulas(goals).
end_of_list.
```

Code Listing 2.29: Exemplul 1 de presupuneri pentru rulare

Având aceeași listă de personaje și atribuții ale acestora și presupunând de această dată că, la finalul rundei de joc Chris este mort, ce se poate afirma despre alegerea făcută de către medic?

Deoarece Brad este medic și el nu putea să se salveze pe el însuși, acesta ar fi putut să aleagă între a o salva pe Ann și a-l salva pe Chris. Din moment ce Chris este ucis, acesta nu a fost salvat de către Brad, singura opțiune pentru alegerea lui Brad rămânând chiar criminalul, Ann.

```
formulas(assumptions).

-dead(Ann).
-dead(Brad).
-dead(Chris).

killer(Ann).
doctor(Brad).
policeman(Chris).

pick(Ann, Chris).
pick(Brad, Ann).

end_of_list.
```

Code Listing 2.30: Exemplul 2 de presupuneri pentru rulare

2.2.4 Definitivarea mecanismului de stabilire a câștigătorului

Reguli

Se păstrează filosofia conform căreia doar killer-ul sau polițistul pot câștiga:

```
win(x) -> killer(x) | policeman(x).
```

Code Listing 2.31: Killer-ul sau polițistul pot câștiga

Cel mult o persoană poate câștiga o rundă

```
win(x) -> (win(y) <-> x = y).
```

Code Listing 2.32: Cel mult o persoană poate câștiga

Polițistul câștigă dacă arestează killer-ul

```
policeman(x) & arrested(y) & killer(y) -> win(x).
```

Code Listing 2.33: Cazul în care polițistul câștigă

Dacă polițistul arestează pe altcineva în afară de killer, sau dacă mai există alți oameni care trăiesc, în afară de killer, atunci nu câștigă nimeni

```
(policeman(x) & arrested(y) & -killer(y)) |
(killer(x) & (exists y (-dead(y) & -murdered(y) & -killer(y))) & -arrested(x))->
    (all z -win(z)).
```

Code Listing 2.34: Nu câștigă nimeni

Este foarte important de detaliat modul de gândire în scrierea celui de-al doilea termen al disjuncției de mai sus. De ce nu se ia în calcul și cazul în care cineva este ucis, dar este salvat.

Cu siguranță este un caz în care killer-ul nu ucide pe toată lumea, deci dacă polițistul nu a arestat killer-ul, acest caz ar trebui inclus aici pentru a declara că ne există niciun câștigător al acestei runde. Logica pentru care acest caz nu este inclus explicit în cod este următoarea: în cazul în care un personaj x ar fi -dead(x) & murdered(x) & saved(x), înseamnă că există cu siguranță un personaj y, x != y, care l-a salvat pe x, și este, deci, -dead(y) & -murdered(y). Pe scurt, existența unui x -dead & murdered & saved implică existența unui y -dead & -murdered.

Cazul în care killer-ul câștigă este acela când ucide ultima persoană rămasă în viață:

```
killer(x) &
(exists y exists z exists u exists v
(dead(y) & -killer(y) & dead(z) & -killer(z) & dead(u) & -killer(u) & murdered(v
    ) & -killer(v) &
y != z & y != u & y != v & z != u & z != v & u != v))-> win(x).
```

Code Listing 2.35: Cazul în care câștigă killer-ul

Exemple

Dacă dintre cele 5 personaje – Ann, Brad, Chris, Dave și Eliza – doar Ann și Dave nu sunt morți la începutul rundei, Ann este killer, Brad a fost doctor, Chris a fost polițist, iar Dave este ucis, cine a câștigat?

În acest caz, killer-ul ucide și ultima persoană rămasă în viață, ceea ce înseamnă că Ann, killer-ul, câștigă.

```
formulas(assumptions).

-dead(Ann).
dead(Brad).
dead(Chris).
-dead(Dave).
dead(Eliza).

killer(Ann).
doctor(Brad).
policeman(Chris).

murdered(Dave).

end_of_list.
```

Code Listing 2.36: Exemplul 1 de presupuneri pentru rulare

Dacă, în schimb, nici Eliza nu a murit cândva înainte de începerea acestei runde, killer-ul nu poate câștiga, fiindcă mai există persoane rămase în viață după uciderea lui Dave. În acest caz nu există niciun câștigător.

```
formulas(assumptions).

-dead(Ann).
dead(Brad).
dead(Chris).
-dead(Dave).
-dead(Eliza).

killer(Ann).
doctor(Brad).
```

```
policeman(Chris).

murdered(Dave).

end_of_list.
```

Code Listing 2.37: Exemplul 2 de presupuneri pentru rulare

2.3 Ghicitoarea lui Einstein

Se spune că acest puzzle a fost creat de Albert Einstein, care bănuia că doar 2% din populația globului l-ar putea rezolva:

Există cinci case de culori diferite una lângă cealaltă. În fiecare casă locuiește un singur bărbat. Fiecare bărbat are o naționalitate unică, o băutură preferată doar de el, fumează o marcă distinctă de țigări și îngrijește un tip specific de animale.

Folosind indiciile de mai jos, găsește răspunsul la întrebarea *Cine îngrijește pești?*:

- Britanicul locuiește în casa roșie
- Suedezul îngrijește câini
- Danezul bea ceai
- Casa verde este exact în stânga casei albe
- Proprietarul casei verzi bea cafea
- Persoana care fumează Pall Mall îngrijește păsări
- Proprietarul casei galbene fumează Dunhill
- Bărbatul care locuiește în casa din mijloc bea lapte
- Norvegianul locuiește în prima casă
- Bărbatul care fumează Blends locuiește lângă cel care îngrijește pisici
- Bărbatul care îngrijește cai locuiește lângă cel care fumează Dunhill
- Bărbatul care fumează Blue Master bea bere
- Neamțul fumează Prince
- Norvegianul locuiește lângă casa albastră
- Bărbatul care fumează Blends are un vecin care bea apă

În rezolvarea puzzle-ului fără ajutorul computerului, pașii logici sunt următorii:

1. Norvegianul locuiește în casa 1
2. Băutura asociată casei 3 este laptele
3. Norvegianul locuiește lângă casa albastră & 1) $\implies$ casa 2 este albastră

4. Casa verde este exact în stânga casei albe $\Longrightarrow$ casa verde este casa 3 sau casa 4 (a) (a) & Proprietarul casei verzi bea cafea & 2) $\Longrightarrow$ casa 3 nu poate fi verde (b) (b) & (a) $\Longrightarrow$ casa 4 este verde $\Longrightarrow$ casa 5 este albă & băutura casei 4 este cafeaua

5. Britanicul locuiește în casa roșie & 1) & 3) & 4) $\Longrightarrow$ casa 3 este roșie & britanicul locuiește în casa 3 & casa 1 este galbenă

6. Proprietarul casei galbene fumează Dunhill & 5) $\Longrightarrow$ Proprietarul casei 1 fumează Dunhill

7. Bărbatul care îngrijește cai locuiește lângă cel care fumează Dunhill & 6) $\Longrightarrow$ caii sunt animalele asociate casei 2

8. Bărbatul care fumează Blue Master bea bere & 6) $\Longrightarrow$ proprietarul casei 1 nu bea bere (*) Danezul bea ceai & 1) $\Longrightarrow$ proprietarul casei 1 nu bea ceai (**) (*) & (**) & 2) & 4) $\Longrightarrow$ proprietarul casei 1 bea apă

9. Bărbatul care fumează Blends are un vecin care bea apă & 8) $\Longrightarrow$ Proprietarul casei 2 fumează Blends

10. Bărbatul care fumează Blue Master bea bere & 9) $\Longrightarrow$ proprietarul casei 2 nu bea bere $\Longrightarrow$ proprietarul casei 2 bea ceai $\Longrightarrow$ proprietarul casei 5 bea bere

11. Danezul bea ceai & 10) $\Longrightarrow$ Danezul locuiește în casa 2

12. Bărbatul care fumează Blue Master bea bere & 10) $\Longrightarrow$ proprietarul casei 5 fumează Blue Master

13. 1) & 11) & 5) $\Longrightarrow$ Neamțul locuiește în casa 4 sau în casa 5 (x) Neamțul fumează Prince & 12) $\Longrightarrow$ Neamțul nu locuiește în casa 5 (y) (x) & (y) $\Longrightarrow$ Neamțul locuiește în casa 4 $\Longrightarrow$ Suedezul locuiește în casa 5 $\Longrightarrow$ Proprietarul casei 4 fumează Prince $\Longrightarrow$ Proprietarul casei 3 fumează Pall Mall

14. Persoana care fumează Pall Mall îngrijește păsări & 13) $\Longrightarrow$ Proprietarul casei 3 îngrijește păsări

15. Suedezul îngrijește câini & 13) $\Longrightarrow$ Proprietarul casei 5 îngrijește câini

16. Bărbatul care fumează Blends locuiește lângă cel care îngrijește pisici & 9) $\Longrightarrow$ Proprietarul casei 1 sau proprietarul casei 3 îngrijește pisici (u) (u) & 14) $\Longrightarrow$ Proprietarul casei 1 îngrijește pisici $\Longrightarrow$ Proprietarul casei 4, neamțul, îngrijește pești

Lucrurile devin mult mai simple pentru operatorul uman odată ce introduce datele problemei într-un program pentru Mace4.

Fiecare caracteristică a unei case – culoare, naționalitate, băutură, marcă de țigări, animal – este reprezentată printr-o listă de elemente declarate ca fiind distincte:

```
list(distinct).
    [Blue, Green, Red, White, Yellow].
    [Brit, Dane, German, Norwegian, Swede].
    [Beer, Coffee, Milk, Tea, Water].
    [Blends, BlueMaster, Dunhill, PallMall, Prince].
    [Birds, Cats, Dogs, Horses, Fish].
end_of_list.
```

Code Listing 2.38: Listele caracteristicilor

Fiecare casă va fi reprezentată de un număr de la 0 la 4, după cum urmează:

- 0 = casa 1
- 1 = casa 2
- 2 = casa 3
- 3 = casa 4
- 4 = casa 5

Cum fiecare element din fiecare listă va lua o valoare de la 0 la 4, o tuplă de 5 caracteristici cu aceeași valoare numerică va descrie în mod complet o casă. De exemplu: Yellow = 0, Norwegian = 0, Water = 0, Dunhill = 0, Cats = 0 se traduce în limbaj natural în: Prima casă este galbenă. Proprietarul său este un norvegian care fumează Dunhill, îngrijește pisici, și a cărui băutură preferată este apa.

Pentru modelarea relațiilor specificate în indiciile problemei am definit funcțiile:

- to_the_left(x, y) – x este vecinul imediat din stânga al lui y
- next_to(x, y) – x și y au case alăturate

```
formulas(utils).
    to_the_left(x,y) <-> x + 1 = y.
    next_to(x,y) <-> x + 1 = y | y + 1 = x.
end_of_list.
```

Code Listing 2.39: Funcțiile definite

Indiciile reprezintă, de fapt, egalități care se stabilesc între variabilele ce descriu caracteristici ale celor cinci case, sau funcții aplicate anumitor variabile, care, la bază, descriu tot egalități. De exemplu:

- *Britanicul locuiește în casa roșie* se exprimă în cod prin egalitatea Brit = Red;
- *Bărbatul care locuiește în casa din mijloc bea lapte* se transpune prin relația 2 = Milk;
- *Casa verde este exact în stânga casei albe* devine to_the_left(Green, White).
- *Bărbatul care îngrijește cai locuiește lângă cel care fumează Dunhill* este o afirmație echivalentă cu next_to(Horses, Dunhill).

Codul care modelează toate indiciile primite este:

```
formulas(assumptions).

Brit = Red.
Swede = Dogs.
Dane = Tea.
to_the_left(Green, White).
Green = Coffee.
PallMall = Birds.
Yellow = Dunhill.
2 = Milk.
0 = Norwegian.
next_to(Blends, Cats).
```

```
next_to(Horses, Dunhill).
BlueMaster = Beer.
German = Prince.
next_to(Norwegian, Blue).
next_to(Blends, Water).

end_of_list.
```

Code Listing 2.40: Indiciile primite

Dacă se dorește determinarea modelului returnat de codul scris, adică felul în care sunt mapate caracteristicile pentru fiecare casă, atunci apelul din linia de comandă este

```
mace4 -c -f einsteinRiddle.in
```

Code Listing 2.41: Comanda pentru rulare

Dimensiunea domeniului de valori a fost setată la 5 prin comanda *assign(domain_size, 5).*, ceea ce înseamnă că variabilele din cod vor primi valori de la 0 la 4.

Se dorește returnarea tuturor modelelor posibile pentru problema propusă, fapt exprimat prin linia *assign(max_models,-1).*

În exprimarea relațiilor de vecinătate *to_the_left* și *next_to* se folosesc operatori aritmetici, cum ar fi operatorii + și = . Pentru ca acești operatori să fie recunoscuți de Mace4 este nevoie de comanda *set(arithmetic).*

2.4 Problema donării de sânge

Un puzzle inspirat de ghicitoarea lui Einstein este cel al donării de sânge.

Cinci femei donatoare de sânge sunt așezate una lângă cealaltă. Fiecare dintre acestea este caracterizată de o culoare diferită a tricoului, un nume unic, o grupă de sânge specifică, vârsta diferită de a oricărei alte donatoare, greutate diferită și o meserie specifică.

La fel ca în cazul problemei anterior descrise, caracteristicile sunt reprezentate prin liste de variabile declarate distincte:

```
list(distinct).
    [Black, Blue, Green, Purple, Red].
    [Andrea, Brooke, Kathleen, Meghan, Nichole].
    [APlus, ABPlus, BPlus, BMinus, OMinus].
    [Age25, Age30, Age35, Age40, Age45].
    [Lb120, Lb130, Lb140, Lb150, Lb160].
    [Actress, Chef, Engineer, Florist, Policewoman].
end_of_list.
```

Code Listing 2.42: Listele caracteristicilor

Fiecare donatoare de sânge va avea un număr asignat de la 0 la 4, în conformitate cu ordinea în care sunt așezate (prima donatoare = 0, a cincea donatoare = 4). Variabila corespunzătoare fiecărei caracteristici particulare dintr-o listă va lua o valoare de la 0 la 4. Dacă variabila Engineer are valoarea 3, înseamnă ca a patra donatoare este inginer (se va ține cont de indexarea de la 0). Alegând din toate cele 6 liste de caracteristici variabilele a căror valoare este x, se va putea caracteriza complet a (x+1) – a donatoare.

Pentru a facilita modelarea indiciilor problemei s-au definit următoarele funcții:

- exactly_to_the_left(x, y) – x ocupă exact locul din stânga lui y
- somwhere_to_the_left(x, y) – x ocupă oricare dintre locurile din stânga lui y

- somwhere_to_the_right(x, y) – x ocupă oricare dintre locurile din dreapta lui y
- somwhere_between(x, y, z) – x ocupă oricare din locurile dintre x și z, în această ordine
- next_to(x, y) – x ocupă un loc exact lângă y
- at_the_ends(x) – x ocupă fie primul, fie ultimul loc

```
formulas(utils).
    exactly_to_the_left(x,y) <-> x + 1 = y.
    somwhere_to_the_left(x, y) <-> x + 1 = y | x + 2 = y | x + 3 = y | x + 4 = y.
    somwhere_to_the_right(x, y) <-> y + 1 = x | y + 2 = x | y + 3 = x | y + 4 = x
    .
    somwhere_between(x,y,z) <-> (y + 1 = x | y + 2 = x | y + 3 = x) & (x + 1 = z
    | x + 2 = z | x + 3 = z).
    next_to(x,y) <-> x + 1 = y | y + 1 = x.
    at_the_ends(x) <-> x = 0 | x = 4.
end_of_list.
```

Code Listing 2.43: Funcțiile definite

Indiciile oferite pentru a determina caracteristicile fiecăreia dintre cele cinci donatoare, alături de transpunerea acestora în cod sunt:

1. Donatoarea A+ este lângă donatoarea B+

```
next_to(APlus, BPlus).
```

Code Listing 2.44: Indiciul 1

2. Brooke se află la unul dintre capete

```
at_the_ends(Brooke).
```

Code Listing 2.45: Indiciul 2

3. Cea care poartă tricou negru este undeva la stânga femeii care cântărește 150 pounds

```
somwhere_to_the_left(Black, Lb150).
```

Code Listing 2.46: Indiciul 3

4. Actrița este lângă bucătar

```
next_to(Actress, Chef).
```

Code Listing 2.47: Indiciul 4

5. Kathleen are 40 de ani

```
Kathleen = Age40.
```

Code Listing 2.48: Indiciul 5

6. Florăreasa se află undeva la dreapta femeii care poartă tricou mov

```
somwhere_to_the_right(Florist, Purple).
```

Code Listing 2.49: Indiciul 6

7. Cea mai vârstnică donatoare cântărește 130 pounds

```
Age45 = Lb130.
```

Code Listing 2.50: Indiciul 7

8. Brooke este lângă Nichole

```
next_to(Brooke, Nichole).
```

Code Listing 2.51: Indiciul 8

9. Femeia în vârstă de 35 de ani se află exact în stânga femeii în vârstă de 30 de ani

```
exactly_to_the_left(Age35, Age30).
```

Code Listing 2.52: Indiciul 9

10. Donatoarea care cântărește 120 pounds se află undeva între donatoarea 0- și cea care cântărește 150 pounds, în această ordine

```
somwhere_between(Lb120, OMinus, Lb150).
```

Code Listing 2.53: Indiciul 10

11. Kathleen se află la unul dintre capete

```
at_the_ends(Kathleen).
```

Code Listing 2.54: Indiciul 11

12. Femeia care poartă tricou mov se află undeva în dreapta celei care poartă tricou verde

```
somwhere_to_the_right(Purple, Green).
```

Code Listing 2.55: Indiciul 12

13. Donatoarea B+ cântărește 140 pounds

```
BPlus = Lb140.
```

Code Listing 2.56: Indiciul 13

14. Cea mai tânără femeie se află lângă cea în vârstă de 30 de ani

```
next_to(Age25, Age30).
```

Code Listing 2.57: Indiciul 14

15. Femeia considerată primitor universal se află exact în stânga donatoarei A+

```
exactly_to_the_left(ABPlus, APlus).
```

Code Listing 2.58: Indiciul 15

16. Meghan se află undeva în dreapta femeii care poartă tricou mov

```
somwhere_to_the_right(Meghan, Purple).
```

Code Listing 2.59: Indiciul 16

17. Femeia care poartă tricou verde se află undeva între actriță și femeia care poartă tricou roșu, în această ordine

```
somwhere_between(Green, Actress, Red).
```

Code Listing 2.60: Indiciul 17

18. La unul dintre capete se află femeia care cântărește 130 pounds

```
at_the_ends(Lb130).
```

Code Listing 2.61: Indiciul 18

19. Donatoarea universală are 35 de ani

```
OMinus = Age35.
```

Code Listing 2.62: Indiciul 19

20. Florăreasa se află undeva între actriță și inginer, în această ordine

```
somwhere_between(Florist, Actress, Engineer).
```

Code Listing 2.63: Indiciul 20

21. Femeia care poartă tricou albastru se află undeva în stânga femeii care poartă tricou roșu

```
somwhere_to_the_left(Blue, Red).
```

Code Listing 2.64: Indiciul 21

22. Donatoarea AB+ se află lângă cea mai tânără femeie

```
next_to(ABPlus, Age25).
```

Code Listing 2.65: Indiciul 22

Dacă se dorește determinarea modelului returnat de codul scris, adică felul în care sunt mapate caracteristicile pentru fiecare casă, atunci apelul din linia de comandă este

S-a apelat comanda *set(arithmetic).* pentru a putea folosi operatori aritmetici în descrierea funcțiilor folosite pentru modelare.

Specificarea din cod a dimensiunii domeniului de valori al variabilelor – *assign(domain_size, 5).* – și a faptului că se doresc toate modelele posibile ale problemei – *assign(max_models,-1).* – face ca apelul din linia de comandă pentru rularea programului să se rezume la:

```
mace4 -c -f bloodDonation.in
```

Code Listing 2.66: Comanda pentru rulare

Chapter 3

A3: Planificare

Appendix A

Codul nostru original

A.0.1 A1: Căutare

```
def depthFirstSearch(problem):
    visited = []
    st = util.Stack()
    st.push(((problem.getStartState(), ()), None))

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux[0][0])):
            break
        if (aux[0][0] not in visited):
            visited.append(aux[0][0])
            for x in problem.getSuccessors(aux[0][0]):
                st.push((x, aux))

    moves = []

    while (aux[0][1]):
        #print "Next step:", aux[0][1]
        moves.append(aux[0][1])
        aux = aux[1]
    moves.reverse()
    return moves
```

Code Listing A.1: Depth-First Search

```
def breadthFirstSearch(problem):
    visited = []
    st = util.Queue()
    st.push(((problem.getStartState(), ()), None))

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux[0][0])):
            break
        if (aux[0][0] not in visited):
            visited.append(aux[0][0])
            for x in problem.getSuccessors(aux[0][0]):
                st.push((x, aux))

    moves = []

    while (aux[0][1]):
```

```
        #print "Next step:", aux[0][1]
        moves.append(aux[0][1])
        aux = aux[1]
    moves.reverse()
    return moves
```

Code Listing A.2: Breadth-First Search

```
class Node:
    def __init__(self, state, action, cost, parent):
        self.state = state
        self.action = action
        self.cost = cost
        self.parent = parent

    @property
    def state(self):
        return self.__state

    @property
    def action(self):
        return self.__action

    @property
    def cost(self):
        return self.__cost

    @property
    def parent(self):
        return self.__parent
```

Code Listing A.3: Clasa *Node*

```
def uniformCostSearch(problem):
    visited = []
    st = util.PriorityQueue()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n, n.cost)

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux.state)):
            break
        if (aux.state not in visited):
            visited.append(aux.state)
            for x in problem.getSuccessors(aux.state):
                node = Node(x[0], x[1], aux.cost + x[2], aux)
                if (node.state not in visited):
                    st.push(node, node.cost)
                elif betterCost(st, node):
                    st.update(node, node.cost)

    moves = []

    while (aux.parent):
        print "Next step:", aux.action, aux.cost
        moves.append(aux.action)
        aux = aux.parent
```

```
    moves.reverse()
    return moves
```

Code Listing A.4: Uniform-Cost Search

```
def betterCost(list, node):
    while (list.isEmpty == False):
        cell = list.pop()
        if (cell.state == node.state and cell.cost > node.cost):
            return 1
    return 0;
```

Code Listing A.5: metoda *betterCost*

```
def aStarSearch(problem, heuristic):
    visited = []
    st = util.PriorityQueue()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n, n.cost)

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux.state)):
            break
        if (aux.state not in visited):
            visited.append(aux.state)
            for x in problem.getSuccessors(aux.state):
                node = Node(x[0], x[1], aux.cost + x[2], aux)
                if (node.state not in visited):
                    st.push(node, node.cost + heuristic(node.state, problem))
                elif betterCost(st, node):
                    st.update(node, node.cost + heuristic(node.state, problem))

    moves = []

    while (aux.parent):
        print "Next step:", aux.action, aux.cost
        moves.append(aux.action)
        aux = aux.parent
    moves.reverse()
    return moves
```

Code Listing A.6: A* Search

```
def weightedAStarSearch(problem, heuristic):
    visited = []

    st = util.PriorityQueue()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n, n.cost)

    while (st.isEmpty() == False):
        aux = st.pop()
        if(problem.isGoalState(aux.state)):
            break
        if (aux.state not in visited):
            visited.append(aux.state)
```

```
            for x in problem.getSuccessors(aux.state):
                node = Node(x[0], x[1], aux.cost + x[2], aux)
                eps = 2.25
                if (node.state not in visited):
                    st.push(node, node.cost + (1 + eps) * heuristic(node.state,
    problem))
                elif betterCost(st, node):
                    st.update(node, node.cost + (1 + eps) * heuristic(node.state,
     problem))

    moves = []

    while (aux.parent):
        #print "Next step:", aux.action, aux.cost
        moves.append(aux.action)
        aux = aux.parent
    moves.reverse()
    return moves
```

Code Listing A.7: Weighted A* Search

```
def iterativeDeepeningSearch(problem):
    moves = []
    for depth in range(0, 300):
        result = depthLimitedSearch(problem, depth)
        if(result.parent != None):
            break

    while (result.parent):
        #print "Next step:", result.state, result.action, result.cost
        moves.append(result.action)
        result = result.parent
    moves.reverse()
    return moves


def depthLimitedSearch(problem, depth):
    st = util.Stack()
    n = Node(problem.getStartState(), (), 0, None)
    st.push(n)
    result = Node((0,0), (), 0, None)
    visited = []

    while(st.isEmpty() == False):
        aux = st.pop()
        if (aux.state not in visited):
            visited.append(aux.state)
            if(problem.isGoalState(aux.state)):
                return aux
            if(aux.cost > depth):
                return result
            else:
                for x in problem.getSuccessors(aux.state):
                    node = Node(x[0], x[1], aux.cost + x[2], aux)
                    st.push(node)

    return result
```

Code Listing A.8: Iterative-Deepening Search

```
    def isGoalState(self, state):
        isGoal = False
        if self.food[state[0]][state[1]]:
            isGoal = True
        return isGoal
```

Code Listing A.9: stabilirea *goalState* în metoda *isGoalState* a clasei *AnyFoodSearchProblem*

```
def getNumFood( self ):
        for i in range (0, self.data.food.width):
            for j in range (0, self.data.food.height):
                self.data.food[i][j] = False
        return self.data.food.count()
```

Code Listing A.10: modificările aduse la metoda *getNumFood(self)* a clasei *GameState*

```
def chebyshevHeuristic(position, problem, info={}):
    "The Chebyshev distance heuristic"
    xy1 = position
    xy2 = problem.goal
    return max(xy1[1] - xy2[1], xy1[0] - xy2[0])
```

Code Listing A.11: Chebyshev Distance Heuristic

```
def octileHeuristic(position, problem, info={}):
    "The Octile distance heuristic"
    xy1 = position
    xy2 = problem.goal
    dx = abs(xy1[0] - xy2[0])
    dy = abs(xy1[1] - xy2[1])
    D = 1
    D2 = math.sqrt(2)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Code Listing A.12: Octile Distance Heuristic

A.0.2 A2: Logică

```
assign(max_seconds, 30).
assign(domain_size, 2).
assign(max_models,-1).
set(binary_resolution).
set(print_gen).



list(distinct).
[Ann, Brad].
end_of_list.



formulas(utils).
```

```
% POATE EXISTA DOAR UN KILLER
killer(x) -> (killer(y) <-> x = y).
exists x killer(x).

% KILLER-UL POATE ALEGE O SINGURA DATA PE RUNDA UN ALT JUCATOR IN AFARA DE EL,
    CARE NU ESTE DEJA MORT
pick(x, y) -> killer(x).
pick(x, y) -> (pick(x, z) <-> y = z).
pick(x, y) -> -dead(y).
pick(x, y) -> x != y.

% JUCATORUL ALES DE KILLER ESTE UCIS. TOTI CEILALTI NU SUNT UCISI
pick(x, y) & killer(x) -> murdered(y) & (all z ((y != z) -> -murdered(z))).

%EXACT UN JUCATOR ESTE UCIS
murdered(x) -> (murdered(y) <-> x = y).
exists x murdered(x).

%CEL UCIS NU POATE FI KILLER-UL
murdered(x) -> -killer(x).

%CEL UCIS A FOST ALES DE CATRE KILLER
exists y exists x (killer(y) & pick(y, x) & murdered(x)).

end_of_list.



formulas(assumptions).

-dead(Ann).
-dead(Brad).

%murdered(Brad).

end_of_list.



formulas(goals).
end_of_list.
```

Code Listing A.13: mafiaKiller.in - varianta 1 simplificată a jocului Mafia - doar killer

```
assign(max_seconds, 30).
assign(domain_size, 3).
assign(max_models,-1).
set(binary_resolution).
set(print_gen).



list(distinct).
[Ann, Brad, Chris].
end_of_list.



formulas(utils).
```

```
% POATE EXISTA DOAR UN KILLER
killer(x) -> (killer(y) <-> x = y).
exists x killer(x).

% POATE EXISTA UN SINGUR POLITIST
policeman(x) -> (policeman(y) <-> x = y).
exists x policeman(x).

% FIECARE JUCATOR POATE AVEA CEL MULT UN ROL
killer(x) -> -policeman(x).
policeman(x) -> -killer(x).

% KILLER-UL SI POLITISTUL POT ALEGE O SINGURA DATA PE RUNDA UN ALT JUCATOR CARE
    NU ESTE MORT
% POLITISTUL TREBUIE SA FIE IN VIATA PENTRU A ALEGE
pick(x, y) -> killer(x) | policeman(x).
pick(x, y) -> (pick(x, z) <-> y = z).
pick(x, y) -> -dead(x) & -murdered(x).
pick(x, y) -> -dead(y).
pick(x, y) -> x != y.

% JUCATORUL ALES DE KILLER ESTE UCIS. TOTI CEILALTI NU SUNT UCISI
pick(x, y) & killer(x) -> murdered(y) & (all z ((y != z) -> -murdered(z))).

% JUCATORUL ALES DE POLITIST ESTE ARESTAT. TOTI CEILALTI NU SUNT ARESTATI
pick(x, y) & policeman(x) -> arrested(y) & (all z ((y != z) -> -arrested(z))).

%DACA POLITISTUL ESTE MORT, NU SE MAI POT FACE ARESTARI
dead(x) & policeman(x) -> (all z -arrested(z)).

%DACA POLITISTUL ESTE UCIS, NU SE MAI POT FACE ARESTARI
murdered(x) & policeman(x) -> (all z -arrested(z)).

%EXACT UN JUCATOR ESTE UCIS
murdered(x) -> (murdered(y) <-> x = y).
exists x murdered(x).

%CEL UCIS NU POATE FI KILLER-UL
murdered(x) -> -killer(x).

%CEL UCIS A FOST ALES DE CATRE KILLER
exists y exists x (killer(y) & pick(y, x) & murdered(x)).

%CEL MULT UN JUCATOR ESTE ARESTAT
arrested(x) -> (arrested(y) <-> x = y).

%CEL ARESTAT NU POATE FI POLITISTUL
arrested(x) -> -policeman(x).

%CEL ARESTAT A FOST ALES DE POLITIST
arrested(x) & policeman(y) -> pick(y, x).

%DOAR O SINGURA PERSOANA POATE CASTIGA
win(x) -> (win(y) <-> x = y).

%DOAR POLITISTUL SAU KILLERUL POT CASTIGA
win(x) -> killer(x) | policeman(x).

%POLITISTUL CASTIGA DACA ARESTEAZA KILLER-UL
policeman(x) & arrested(y) & killer(y) -> win(x).
```

```
%DACA POLITISTUL ARESTEAZA PE ALTCINEVA IN AFARA DE KILLER NU CASTIGA NIMENI
policeman(x) & arrested(y) & -killer(y) -> (all z -win(z)).

%KILLER-UL CASTIGA DACA UCIDE POLITISTUL
killer(x) & murdered(y) & policeman(y) -> win(x).
end_of_list.



formulas(assumptions).

-dead(Ann).
-dead(Brad).
-dead(Chris).

killer(Ann).
policeman(Chris).
arrested(Ann).

end_of_list.



formulas(goals).
end_of_list.
```

Code Listing A.14: mafiaPoliceman.in - varianta 2 simplificată a jocului Mafia - killer & policeman

```
assign(max_seconds, 30).
assign(domain_size, 3).
assign(max_models,-1).
set(binary_resolution).
set(print_gen).



list(distinct).
[Ann, Brad, Chris].
end_of_list.



formulas(utils).

% POATE EXISTA DOAR UN KILLER
killer(x) -> (killer(y) <-> x = y).
exists x killer(x).

% POATE EXISTA UN SINGUR DOCTOR
doctor(x) -> (doctor(y) <-> x = y).
exists x doctor(x).

% POATE EXISTA UN SINGUR POLITIST
policeman(x) -> (policeman(y) <-> x = y).
exists x policeman(x).

% FIECARE JUCATOR POATE AVEA CEL MULT UN ROL
```

```
doctor(x) -> -killer(x) & -policeman(x).
killer(x) -> -doctor(x) & -policeman(x).
policeman(x) -> -killer(x) & -doctor(x).

% KILLER-UL, DOCTORUL SI POLITISTUL POT ALEGE O SINGURA DATA PE RUNDA UN ALT
    JUCATOR
pick(x, y) -> killer(x) | doctor(x) | policeman(x).
pick(x, y) -> (pick(x, z) <-> y = z).
pick(x, y) -> -dead(x) & -murdered(x) | -dead(x) & murdered(x) & saved(x).
pick(x, y) -> -dead(y).
pick(x, y) -> x != y.

% JUCATORUL ALES DE KILLER ESTE UCIS. TOTI CEILALTI NU SUNT UCISI
pick(x, y) & killer(x) -> murdered(y) & (all z ((y != z) -> -murdered(z))).

% JUCATORUL ALES DE DOCTOR ESTE SALVAT. TOTI CEILALTI NU SUNT SALVATI
pick(x, y) & doctor(x) -> saved(y) & (all z ((y != z) -> -saved(z))).

% JUCATORUL ALES DE POLITIST ESTE ARESTAT. TOTI CEILALTI NU SUNT ARESTATI
pick(x, y) & policeman(x) -> arrested(y) & (all z ((y != z) -> -arrested(z))).

%DACA POLITISTUL ESTE MORT, NU SE MAI POT FACE ARESTARI
dead(x) & policeman(x) -> (all z -arrested(z)).

%DACA DOCTORUL ESTE MORT, NU SE MAI POT FACE SALVARI
dead(x) & doctor(x) -> (all z -saved(z)).

%DACA POLITISTUL ESTE UCIS SI NU ESTE SALVAT, NU SE MAI POT FACE ARESTARI
murdered(x) & policeman(x) & -saved(x)-> (all z -arrested(z)).

%DACA DOCTORUL ESTE UCIS, NU SE MAI POT FACE SALVARI
murdered(x) & doctor(x) -> (all z -saved(z)).

%EXACT UN JUCATOR ESTE UCIS
murdered(x) -> (murdered(y) <-> x = y).
exists x murdered(x).

%CEL UCIS NU POATE FI KILLER-UL
murdered(x) -> -killer(x).

%CEL UCIS A FOST ALES DE CATRE KILLER
exists y exists x (killer(y) & pick(y, x) & murdered(x)).

%CEL MULT UN JUCATOR ESTE ARESTAT
arrested(x) -> (arrested(y) <-> x = y).

%CEL ARESTAT NU POATE FI POLITISTUL
arrested(x) -> -policeman(x).

%CEL ARESTAT A FOST ALES DE POLITIST
arrested(x) & policeman(y) -> pick(y, x).

%CEL MULT UN JUCATOR ESTE SALVAT
saved(x) -> (saved(y) <-> x = y).

%CEL SALVAT NU POATE FI DOCTORUL
saved(x) -> -doctor(x).

%CEL SALVAT A FOST ALES DE DOCTOR
saved(x) & doctor(y) -> pick(y, x).
```

```
%DOAR O SINGURA PERSOANA POATE CASTIGA
win(x) -> (win(y) <-> x = y).

%FIE KILLER-UL, FIE POLITISTUL CASTIGA
win(x) -> killer(x) | policeman(x).

%POLITISTUL CASTIGA DACA ARESTEAZA KILLER-UL
policeman(x) & arrested(y) & killer(y) -> win(x).

%DACA POLITISTUL ARESTEAZA PE ALTCINEVA IN AFARA DE KILLER NU CASTIGA NIMENI
policeman(x) & arrested(y) & -killer(y) -> (all z -win(z)).

%KILLER-UL CASTIGA DACA UCIDE POLITISTUL
killer(x) & murdered(y) & policeman(y) & -saved(y)-> win(x).

%DACA POLITISTUL ESTE UCIS, DAR ESTE SALVAT DE MEDIC, IAR APOI NU ARESTEAZA
    KILLER-UL, NU CASTIGA NIMENI
killer(x) & -arrested(x) & murdered(y) & policeman(y) & saved(y)-> (all z -win(z
    )).
end_of_list.



formulas(assumptions).

-dead(Ann).
-dead(Brad).
-dead(Chris).

killer(Ann).

saved(Chris).
arrested(Ann).

end_of_list.



formulas(goals).
end_of_list.
```

Code Listing A.15: mafiaDoctor.in - varianta 3 simplificată a jocului Mafia - killer & policeman & doctor

```
assign(max_seconds, 30).
assign(domain_size, 5).
assign(max_models,-1).
set(binary_resolution).
set(print_gen).



list(distinct).
[Ann, Brad, Chris, Dave, Eliza].
end_of_list.
```

```
formulas(utils).

% POATE EXISTA DOAR UN KILLER
killer(x) -> (killer(y) <-> x = y).
exists x killer(x).

% POATE EXISTA UN SINGUR DOCTOR
doctor(x) -> (doctor(y) <-> x = y).
exists x doctor(x).

% POATE EXISTA UN SINGUR POLITIST
policeman(x) -> (policeman(y) <-> x = y).
exists x policeman(x).

% FIECARE JUCATOR POATE AVEA CEL MULT UN ROL
doctor(x) -> -killer(x) & -policeman(x).
killer(x) -> -doctor(x) & -policeman(x).
policeman(x) -> -killer(x) & -doctor(x).

% KILLER-UL, DOCTORUL SI POLITISTUL POT ALEGE O SINGURA DATA PE RUNDA UN ALT
    JUCATOR
pick(x, y) -> killer(x) | doctor(x) | policeman(x).
pick(x, y) -> (pick(x, z) <-> y = z).
pick(x, y) -> -dead(x) & -murdered(x) | -dead(x) & murdered(x) & saved(x).
pick(x, y) -> -dead(y).
pick(x, y) -> x != y.

% JUCATORUL ALES DE KILLER ESTE UCIS. TOTI CEILALTI NU SUNT UCISI
pick(x, y) & killer(x) -> murdered(y) & (all z ((y != z) -> -murdered(z))).

% JUCATORUL ALES DE DOCTOR ESTE SALVAT. TOTI CEILALTI NU SUNT SALVATI
pick(x, y) & doctor(x) -> saved(y) & (all z ((y != z) -> -saved(z))).

% JUCATORUL ALES DE POLITIST ESTE ARESTAT. TOTI CEILALTI NU SUNT ARESTATI
pick(x, y) & policeman(x) -> arrested(y) & (all z ((y != z) -> -arrested(z))).

%DACA POLITISTUL ESTE MORT, NU SE MAI POT FACE ARESTARI
dead(x) & policeman(x) -> (all z -arrested(z)).

%DACA DOCTORUL ESTE MORT, NU SE MAI POT FACE SALVARI
dead(x) & doctor(x) -> (all z -saved(z)).

%DACA POLITISTUL ESTE UCIS SI NU ESTE SALVAT, NU SE MAI POT FACE ARESTARI
murdered(x) & policeman(x) & -saved(x) -> (all z -arrested(z)).

%DACA DOCTORUL ESTE UCIS, NU SE MAI POT FACE SALVARI
murdered(x) & doctor(x) -> (all z -saved(z)).

%EXACT UN JUCATOR ESTE UCIS
murdered(x) -> (murdered(y) <-> x = y).
exists x murdered(x).

%CEL UCIS NU POATE FI KILLER-UL
murdered(x) -> -killer(x).

%CEL UCIS A FOST ALES DE CATRE KILLER
exists y exists x (killer(y) & pick(y, x) & murdered(x)).

%CEL MULT UN JUCATOR ESTE ARESTAT
arrested(x) -> (arrested(y) <-> x = y).
```

```
%CEL ARESTAT NU POATE FI POLITISTUL
arrested(x) -> -policeman(x).

%CEL ARESTAT A FOST ALES DE POLITIST
arrested(x) & policeman(y) -> pick(y, x).

%CEL MULT UN JUCATOR ESTE SALVAT
saved(x) -> (saved(y) <-> x = y).

%CEL SALVAT NU POATE FI DOCTORUL
saved(x) -> -doctor(x).

%CEL SALVAT A FOST ALES DE DOCTOR
saved(x) & doctor(y) -> pick(y, x).

%DOAR O SINGURA PERSOANA POATE CASTIGA
win(x) -> (win(y) <-> x = y).

%FIE KILLER-UL, FIE POLITISTUL CASTIGA
win(x) -> killer(x) | policeman(x).

%POLITISTUL CASTIGA DACA ARESTEAZA KILLER-UL
policeman(x) & arrested(y) & killer(y) -> win(x).

%DACA POLITISTUL ARESTEAZA PE ALTCINEVA IN AFARA DE KILLER, SAU DACA MAI EXISTA
    ALTI OAMENI CARE TRAIESC, IN AFARA DE KILLER, NU CASTIGA NIMENI
(policeman(x) & arrested(y) & -killer(y)) | (killer(x) & (exists y (-dead(y) & -
    murdered(y) & -killer(y))) & -arrested(x))-> (all z -win(z)).

%DACA TOTI CEILALTI SUNT MORTI, KILLER-UL CASTIGA
killer(x) &
(exists y exists z exists u exists v
(dead(y) & -killer(y) & dead(z) & -killer(z) & dead(u) & -killer(u) & murdered(v
    ) & -killer(v) &
y != z & y != u & y != v & z != u & z != v & u != v))-> win(x).

end_of_list.



formulas(assumptions).

-dead(Ann).
-dead(Brad).
-dead(Chris).
-dead(Dave).
-dead(Eliza).

killer(Ann).
doctor(Brad).
policeman(Chris).

murdered(Chris).
saved(Chris).
arrested(Ann).

end_of_list.
```

```
formulas(goals).
end_of_list.
```

Code Listing A.16: mafia3.1.in - jocul Mafia

```
assign(max_seconds, 30).
assign(domain_size, 5).
assign(max_models,-1).
set(arithmetic).
set(binary_resolution).
set(print_gen).


list(distinct).
    [Blue, Green, Red, White, Yellow].
    [Brit, Dane, German, Norwegian, Swede].
    [Beer, Coffee, Milk, Tea, Water].
    [Blends, BlueMaster, Dunhill, PallMall, Prince].
    [Birds, Cats, Dogs, Horses, Fish].
end_of_list.

formulas(utils).
    to_the_left(x,y) <-> x + 1 = y.
    next_to(x,y) <-> x + 1 = y | y + 1 = x.
end_of_list.

formulas(assumptions).

Brit = Red.
Swede = Dogs.
Dane = Tea.
to_the_left(Green, White).
Green = Coffee.
PallMall = Birds.
Yellow = Dunhill.
2 = Milk.
0 = Norwegian.
next_to(Blends, Cats).
next_to(Horses, Dunhill).
BlueMaster = Beer.
German = Prince.
next_to(Norwegian, Blue).
next_to(Blends, Water).

end_of_list.

formulas(goals).
end_of_list.
```

Code Listing A.17: einsteinRiddle.in - Ghicitoarea lui Einstein

```
assign(max_seconds, 30).
assign(domain_size, 5).
assign(max_models,-1).
set(arithmetic).
set(binary_resolution).
set(print_gen).
```

```
list(distinct).
    [Black, Blue, Green, Purple, Red].
    [Andrea, Brooke, Kathleen, Meghan, Nichole].
    [APlus, ABPlus, BPlus, BMinus, OMinus].
    [Age25, Age30, Age35, Age40, Age45].
    [Lb120, Lb130, Lb140, Lb150, Lb160].
    [Actress, Chef, Engineer, Florist, Policewoman].
end_of_list.

formulas(utils).
    exactly_to_the_left(x,y) <-> x + 1 = y.
    somwhere_to_the_left(x, y) <-> x + 1 = y | x + 2 = y | x + 3 = y | x + 4 = y.
    somwhere_to_the_right(x, y) <-> y + 1 = x | y + 2 = x | y + 3 = x | y + 4 = x
    .
    somwhere_between(x,y,z) <-> (y + 1 = x | y + 2 = x | y + 3 = x) & (x + 1 = z
    | x + 2 = z | x + 3 = z).
    next_to(x,y) <-> x + 1 = y | y + 1 = x.
    at_the_ends(x) <-> x = 0 | x = 4.
end_of_list.

formulas(assumptions).

next_to(APlus, BPlus).
at_the_ends(Brooke).
somwhere_to_the_left(Black, Lb150).
next_to(Actress, Chef).
Kathleen = Age40.
somwhere_to_the_right(Florist, Purple).
Age45 = Lb130.
next_to(Brooke, Nichole).
exactly_to_the_left(Age35, Age30).
somwhere_between(Lb120, OMinus, Lb150).
at_the_ends(Kathleen).
somwhere_to_the_right(Purple, Green).
BPlus = Lb140.
next_to(Age25, Age30).
exactly_to_the_left(ABPlus, APlus).
somwhere_to_the_right(Meghan, Purple).
somwhere_between(Green, Actress, Red).
at_the_ends(Lb130).
OMinus = Age35.
somwhere_between(Florist, Actress, Engineer).
somwhere_to_the_left(Blue, Red).
next_to(ABPlus, Age25).


end_of_list.

formulas(goals).
end_of_list.
```

Code Listing A.18: bloodDonation.in - Problema donării de sânge