Дополнителен материјал за прв колоквиум по предметот Вештачка интелигенција

Материјалот содржи решени задачи со чекори на решавање за следните теми:

1. Неинформирано пребарување
2. Информирано пребарување
3. Проблеми кои исполнуваат услови

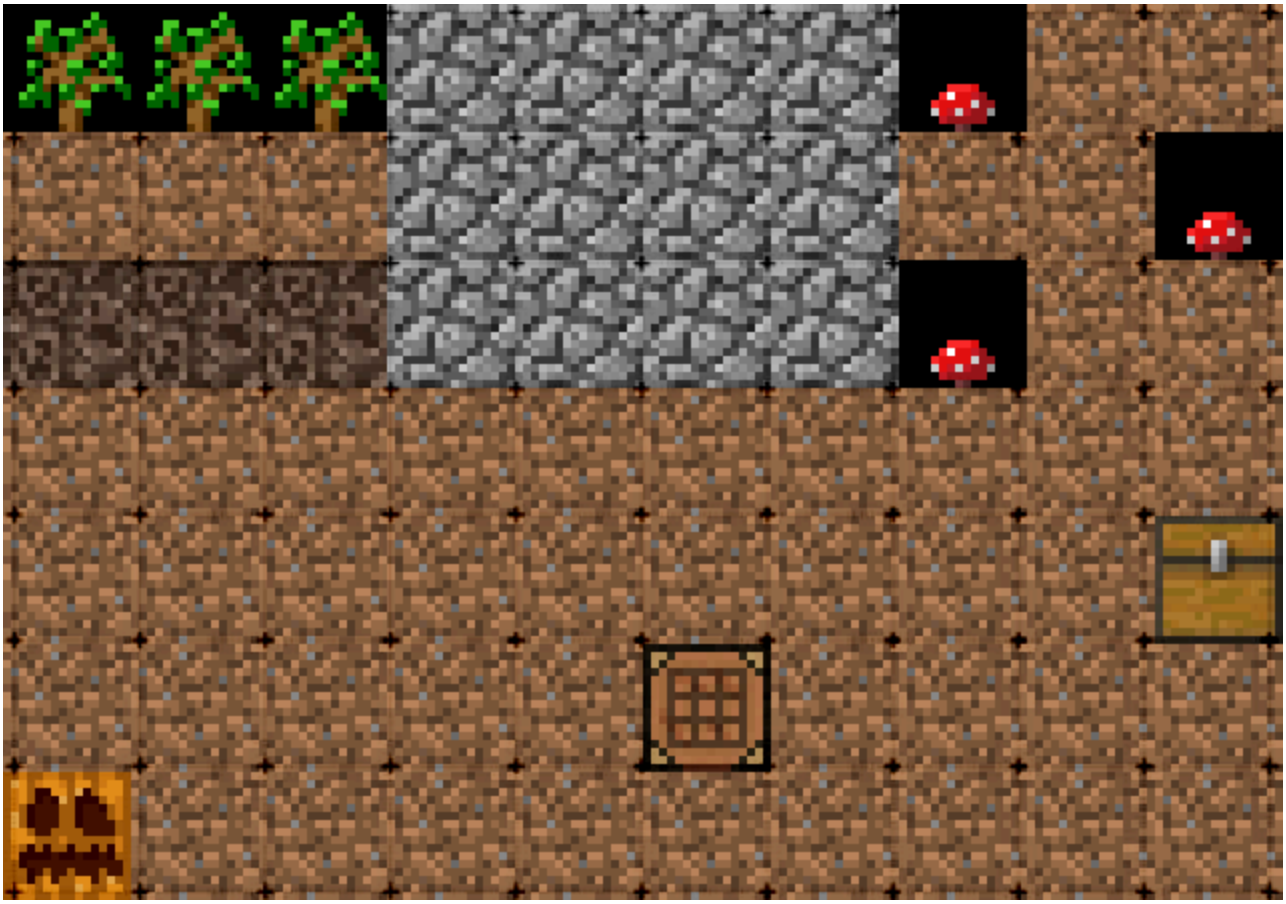
1. Неинформирано пребарување

Во проблеми на пребарување, дефинираме состојби (`states`) и премини помеѓу состојбите (`actions`). Целта е од почетна состојба да стигнеме до целна состојба. Преку Задача 1.1. ќе разгледаме како да ги моделираме состојбите и премините меѓу нив.

Задача 1.1.

(Задачите ќе бидат посложени од задачите од аудиториски/лабораториски со цел да опфатиме повеќе сценарија за моделирање.)

Слика 1: Состојба на игра.



Мајнкрафт играч сака да направи чорба од печурки. Потребно е да собере 2 печурки, да сече 3 дрва со секира, и материјалите да ги подготви на дадената маса.

Печурките се наоѓаат на позиции `(7, 4)`, `(7, 6)` и `(9, 5)`.
Дрвата се наоѓаат на позиции `(0, 6)`, `(1, 6)` и `(2, 6)`.
Маса за припрема на чорба од печурки се наоѓа на позиција `(5, 1)`.
Жив песок се наоѓа на позиции `(0, 4)`, `(1, 4)` и `(2, 4)`.

Позициите на играчот и ковчегот се внесуваат преку стандарден влез, пример

```
0,0
9,2
```

За дадениот влез, играчот се наоѓа на позиција `(0, 0)` додека ковчегот со секира се наоѓа на позиција `(9, 2)`.

(Ќе надополнуваме и добро-дефинираме дополнителни ограничувања за проблемот во деловите што следат)

Дел 1. Моделирање на состојбата на проблемот

Состојбата на проблемот (`state`) е торка од вредности кои се менуваат во текот на проблемот, т.е. еднозначно ја определуваат состојбата на проблемот во даден момент.

Вредности кои треба да ги чуваме во секоја состојба т.е. торка се:

- **Позицијата на играчот** `player = (px, py)` - се менува во текот на проблемот.
- **Позицијата на печурките** `mushrooms = ((mx1, my1), (mx2, my2), ...)` - се менуваат во текот на проблемот (позициите на собраните печурки ќе бидат отстранети).
- **Број на собрани печурки** `num_mushrooms` - се менува во текот на проблемот.
- **Позицијата на дрвата** `trees = ((tx1, ty1), (tx2, ty2), ...)` - се менуваат во текот на проблемот (позициите на сечените дрва ќе бидат отстранети).
- **Број на сечени дрва** `num_trees` - се менува во текот на проблемот.
- **Има секира** `has_axe` - се менува во текот на проблемот.
- **Заглавен во жива песок** `stuck` - во ситуации кога играчот стапнал на жива песок, треба да знаеме дали прв пат стапнал таму и е заглавен, или презел акција за да се одглави од песокта.

Значи состојбата на игра ја претставуваме како:

```
state = (player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck)
```

(*Ова не е минимална репрезентација на состојбата, но во рамките на практичниот дел од предметот, нема да обрнеме внимание на сложеноста на репрезентацијата на состојбата.*)

Од друга страна, имаме вредности кои ни се важни, но тие се статички во текот на целиот проблем т.е. се исти за сите состојби, па нема потреба да ги внесуваме во состојбите кои ги моделираме. Такви вредности се:

- **Позицијата на сандакот** `(cx, cy)` - за даден влез, не се менува во текот на проблемот.
- **Позицијата на живата песок** `((qsx1, qsy1), (qsx2, qsy2), ...)` - не се менува во текот на проблемот.
- **Позицијата на масата** `(tx1, tx2)` - не се менува во текот на проблемот.

На следниот начин може да ги дефинираме променливите:

```
if __name__ == '__main__':
    player = list(map(int, input().split(',')))
    chest = list(map(int, input().split(',')))

    mushrooms = ((7, 4), (7, 6), (9, 5))
```

```
    trees = ((0, 6), (1, 6), (2, 6))
    crafting_table = (5, 1)
    quicksand = ((0, 4), (1, 4), (2, 4))

    initial_state = (tuple(player), mushrooms, 0, trees, 0, False, False)
```

Иницијалната состојба е позицијата на играчот чии координати се внесени од тастатура, па позициите на печурките, па бројот на собрани печурки кој на почеток е `0`, па позициите на дрвата, па бројот на собрани дрва кој на почеток е `0`, па `False` дека играчот иницијално нема секира (таа се наоѓа во ковчегот), па `False` бидејќи играчот иницијално не е заглавен во живата песок.

Важно: Забележи како сите елементи во `initial_state` се или торки, вредности кои се претвораат во торки, или константи. За правилно функционирање на алгоритмите за пребарување, не смеат да содржат состојбите објекти како листи или други типови на објекти кои не може да се хешираат.

Дел 2. Дефинирање на класа за проблемот

За решавање на проблеми со неинформирано пребарување, ќе ги користиме скриптите достапни на курсот на предметот. Овој код мора да стои на почетокот на скриптата:

```
from searching_framework import Problem, breadth_first_graph_search
```

За секој проблем е потребно да дефинираме посебна класа која ќе наследува од класата `Problem` и ќе имплементира одредени функционалности.

Нека ја именуваме класата `Minecraft` со следните функции:

```
class Minecraft(Problem):
    def __init__(self, initial, ...):
        ...
    def successor(self, state):
        ...
    def actions(self, state):
        ...
    def result(self, state, action):
        ...
    def goal_test(self, state):
        ...
```

Ако овие 5 функции се веќе имплементирани, тогаш го решаваме проблемот на следниот начин:

```
initial_state = (tuple(player), mushrooms, 0, trees, 0, False, False)

problem = Minecraft(initial_state, chest, crafting_table, quicksand)
solution = breadth_first_graph_search(problem)

if solution is not None:
    print(solution.solution())
else:
    print('No Solution!')
```

Како што е прикажано на кодот, дефинираме објект од посебно имплементираната класа (во овој случај `Minecraft`). Како прв аргумент во конструкторот ја ставаме почетната состојба (која ги содржи сите вредности кои се менуваат во текот на проблемот), а следните аргументи се сите вредности кои не се менуваат во текот на проблемот, како сандакот, масата и живата песок.

Со помош на функцијата `breadth_first_graph_search` користиме `BFS` за наоѓање на најкраткото решение.

Крајно го печатиме решението ако тоа постои, инаку печатиме дека нема решение.

Да се вратиме назад и да ја дискутираме имплементацијата на функциите во рамките на класата `Minecraft`.

Дел 2.1. Дефинирање конструктор

Секоја класа мора да си дефинира сопствен конструктор. Да забележиме дека ни е потребно да го повикаме конструкторот како `Minecraft(initial_state, chest, crafting_table, quicksand)`.

За таа цел, пишуваме:

```
class Minecraft(Problem):
    def __init__(self, initial, chest, crafting_table, quicksand):
        super().__init__(initial)
        self.chest = chest
        self.crafting_table = crafting_table
        self.quicksand = quicksand
```

Иницијалната состојба секогаш ја проследуваме во конструкторот на надкласата `Problem` (на објектот `super()`) која е потребна за правилно функционирање на алгоритмите за пребарување, додека останатите променливи ги чуваме на ниво на класата `Minecraft`, па затоа и клучниот збор `self` пред променливите.

Дел 2.2. Дефинирање премини помеѓу состојби

Преку следните три функции, имено `successor`, `actions` и `result`, ги моделираме премините меѓу состојбите на проблемот. Всушност ќе видиме дека кодовите за функциите `actions` и `result` се многу кратки бидејќи целата логика се заснова зад функцијата `successor`.

Функцијата `successor` има за цел да земе состојба `state`, и да врати речник од парови `{'name_of_action': new_state}` т.е. да врати нови состојби добиени со примена на одредена акција над состојбата `state`.

Прво, да започнеме со пишување на функцијата

```
def successor(self, state):
    successors = {}

    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck = state
```

Креираме речник `successors` кој ќе биде речникот од парови акција-нова состојба.

За да ги дефинираме следните состојби, потребно е да работиме со податоците од тековната состојба.

Важно: Не правиме промена на податоците `player`, `mushrooms`, итн. Тие мора да останат непроменети. Само нивните вредности ги користиме за дефинирање нови променливи во новите состојби.

Да размислиме кои акции се можни. Можно е играчот да се придвижи во една од осумте насоки (горе, горе-десно, десно, ...). Тоа е можно само ако не е заглавен во живата песок, инаку ако играч заглави во живата песок, треба да дефинираме акција „Одглави“ за да излезе од песокта. Другите акции како земање секира од сандак, собирање печурки, сечење дрва и припрема на чорбата од печурки, ќе ги направиме автоматски со промената на движењето на играчот во една од насоките. (За вежба може да дефинираш дополнителни акции како „подготви чорба“ која ќе ги потроши ресурсите над масата. Размисли дали ќе треба да направиш промена на состојбата ако ги воведуваш овие акции.)

Да ја разгледаме акција „Движи десно“ и како таа прави промени на новата состојба:

- Ако при движење, играчот во новата состојба се најде на ковчег, треба да ја ажурираме вредноста на `has_axe` во `True`.
- Ако при движење, играчот во новата состојба се најде на печурка, треба да го зголемеме бројот на собрани печурки за `1` и да ја отстраниме печурката во новата

состојба.

- Ако при движење, играчот во новата состојба се најде на дрво, акцијата е валидна само ако играчот има секира (инаку оваа состојба нема да ја додадеме во речникот), и притоа го зголемуваме бројот на собрани дрва за `1` и го отстрануваме дрвото во новата состојба.
- Ако при движење, играчот во новата состојба се најде на жива песок, треба да ја ажурираме вредноста на `stuck`.

Откако ќе ги направиме потребните промени, треба да провериме дали оваа состојба е валидна. Ако е валидна, ја додаваме во речникот.

Состојбата не смее да се додаде, т.е. не е валидна ако било кој од следните услови е исполнет:

- Играчот се наоѓа надвор од гридот или се наоѓа во камениот ѕид.
- Играчот се наоѓа на позиција на дрво или печурка (не смее да се наоѓа на исто место, тогаш дрвото/печурката треба да е собрана).
- Играчот е заглавен во жива песок, но не се наоѓа на поле на жива песок (односно е преземена акција за движење која не е валидна).

Најчесто за проверка на состојбата се дефинира посебна функција именувана `check_valid`.

```
def check_valid(self, state):
    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck = state
    px, py = player

    if px < 0 or px > 9 or py < 0 or py > 6:
        return False

    if 3 <= px <= 6 and 4 <= py <= 6:
        return False

    if player in trees or player in mushrooms:
        return False

    if stuck and player not in self.quicksand:
        return False

    return True
```

Ако играчот е надвор од гридот, врати `False`.
Ако играчот е во ѕидот, врати `False`.

Ако играчот е на исто поле со дрво или печурка, врати `False`.
Ако играчот е заглавен во жива песок, но се придвижил надвор од поле без жива песок, тоа е невалиден премин на состојби, врати `False`.
Инаку, состојбата е валидна, врати `True`.

Да ја разгледаме сега функцијата `successor` со додадената акција:

```
def successor(self, state):
    successors = {}

    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck = state
    px, py = player

    # Move right
    npx, npy = px + 1, py
    nmushrooms = mushrooms
    nnum_mushrooms = num_mushrooms
    ntrees = trees
    nnum_trees = num_trees
    nhas_axe = has_axe
    nstuck = stuck

    if (npx, npy) == tuple(self.chest):
        nhas_axe = True
    elif (npx, npy) in mushrooms:
        nnum_mushrooms = num_mushrooms + 1
        nmushrooms = tuple(mushroom for mushroom in mushrooms if mushroom !=
(npx, npy))
    elif (npx, npy) in trees and has_axe:
        nnum_trees = num_trees + 1
        ntrees = tuple(tree for tree in trees if tree != (npx, npy))
    elif (npx, npy) in self.quicksand:
        nstuck = True

    new_state = ((npx, npy), nmushrooms, nnum_mushrooms, ntrees, nnum_trees,
nhas_axe, nstuck)

    if self.check_valid(new_state):
        successors['Move right'] = new_state
```

Дефинираме нови променливи со додадена буква `n` пред имињата кои означуваат дека ќе припаѓаат во новата состојба.

Ако позицијата на новиот играч е над ковчегот, ја ажурираме вредноста на `nhas_axe`.
Ако позицијата на новиот играч е над печурките, го зголемуваме бројот на собрани

печурки и ја отстрануваме печурката која е на иста позиција со играчот од сите печурки.
Ако играчот има секира и се наоѓа на позиција на дрво, го зголемуваме бројот на собрани дрва и го отстрануваме дрвото.
Ако играчот се позиционирал над жива песок, ја ажурираме вредноста на `nstuck`.

По овие промени, дефинираме нова состојба `new_state`.

Ако новата состојба е валидна, ја додаваме во речникот `successors` со клучот `Move right`.

Да забележиме дека логиката за движење на човекот во било која насока е идентична, па и кодот за движење на човечето би бил сличен со единствени разлики во следните сегменти:

```
npx, npy = px + 1, py
```

и

```
successors['Move right'] = new_state
```

За други насоки на движење, ќе треба поинаку да го поместуваме играчот. Многу јасно дека другите акции ќе треба да ги именуваме поинаку, т.е. `Move up-right`, `Move up`, итн. Но, се друго е исто. Па така може да дефинираме `for` циклус на следниот начин:

```
actions = ['Move right', 'Move up-right', 'Move up', 'Move up-left', 'Move
left', 'Move down-left', 'Move down', 'Move down-right']
directions = [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1,
-1)]

for action, (dx, dy) in zip(actions, directions):
    npx, npy = px + dx, py + dy
    nmushrooms = mushrooms
    nnum_mushrooms = num_mushrooms
    ntrees = trees
    nnum_trees = num_trees
    nhas_axe = has_axe
    nstuck = stuck

    if (npx, npy) == tuple(self.chest):
        nhas_axe = True
    elif (npx, npy) in mushrooms:
        nnum_mushrooms = num_mushrooms + 1
        nmushrooms = tuple(mushroom for mushroom in mushrooms if mushroom !=
```

```
(npx, npy))
    elif (npx, npy) in trees and has_axe:
        nnum_trees = num_trees + 1
        ntrees = tuple(tree for tree in trees if tree != (npx, npy))
    elif (npx, npy) in self.quicksand:
        nstuck = True

    new_state = ((npx, npy), nmushrooms, nnum_mushrooms, ntrees, nnum_trees,
nhas_axe, nstuck)

    if self.check_valid(new_state):
        successors[action] = new_state
```

Дефинираме променливи `actions`, т.е. кои насоки ќе ги разгледуваме, и `directions` т.е. придвижувањата на играчот за еден чекор во дадената насока. `For` јамката ги изминува соодветните парови на акции и насоки (`zip` ги спојува соодветните елементи од двете листи).

Дополнително, треба да ја дефинираме и акцијата „Одглави“ која се случува само ако играчот е на жива песок и променливата `stuck` има вредност `True`.

```
if stuck and (px, py) in self.quicksand:
    successors['Unstuck'] = (player, mushrooms, num_mushrooms, trees,
num_trees, has_axe, False)

return successors
```

Новата состојба при акцијата „Одглави“ е всушност останување на играчот во исто место, но со променета вредност на променливата `stuck` во `False`.

Со претходното ги дефиниравме сите акции и може функцијата да го врати речникот `successors`.

Да резимираме уште еднаш што прави функцијата:

```
def successor(self, state):
    successors = {}

    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck = state
    px, py = player

    actions = ['Move right', 'Move up-right', 'Move up', 'Move up-left', 'Move
left', 'Move down-left', 'Move down', 'Move down-right']
    directions = [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1),
```

```
(1, -1)]

    for action, (dx, dy) in zip(actions, directions):
        npx, npy = px + dx, py + dy
        nmushrooms = mushrooms
        nnum_mushrooms = num_mushrooms
        ntrees = trees
        nnum_trees = num_trees
        nhas_axe = has_axe
        nstuck = stuck

        if (npx, npy) == tuple(self.chest):
            nhas_axe = True
        elif (npx, npy) in mushrooms:
            nnum_mushrooms = num_mushrooms + 1
            nmushrooms = tuple(mushroom for mushroom in mushrooms if mushroom
!= (npx, npy))
        elif (npx, npy) in trees and has_axe:
            nnum_trees = num_trees + 1
            ntrees = tuple(tree for tree in trees if tree != (npx, npy))
        elif (npx, npy) in self.quicksand:
            nstuck = True

        new_state = ((npx, npy), nmushrooms, nnum_mushrooms, ntrees,
nnum_trees, nhas_axe, nstuck)

        if self.check_valid(new_state):
            successors[action] = new_state

    if stuck and (px, py) in self.quicksand:
        successors['Unstuck'] = (player, mushrooms, num_mushrooms, trees,
num_trees, has_axe, False)

    return successors
```

- Креира речник во кој ќе имаме парови на акции и нови состојби.
- Ја зема тековната состојба.
- За секоја акција посебно, генерира нова состојба (преку тековната).
- Проверува дали новата состојба е валидна, и ја додава во речникот.
- Го враќа речникот.

Да забележиме дека оваа функција, за дадена состојба, враќа валидни следни акции, заедно со резултатот (новите состојби) од применетите акции.

Па така, функциите `actions` и `result` секогаш се сведуваат на следните кодови:

```
def actions(self, state):
    return self.successor(state).keys()

def result(self, state, action):
    return self.successor(state)[action]
```

Дел 2.3. Проверка на целна состојба

Последната функција во класата `Minecraft` е функцијата `goal_test`. Ако ни е дадена некоја состојба, ние треба да определиме дали таа состојба ни го решава нашиот проблем, и да вратиме `True` доколку тоа е така.

За нашиот проблем, играчот победува ако ги има соодветните материјали (барем 2 собрани печурки и барем 3 собрани дрва) и се наоѓа над масата за припрема на чорба.

```
def goal_test(self, state):
    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck = state

    return num_mushrooms >= 2 and num_trees >= 3 and player ==
self.crafting_table
```

Функцијата враќа `True` само ако се исполнети сите од наведените услови.

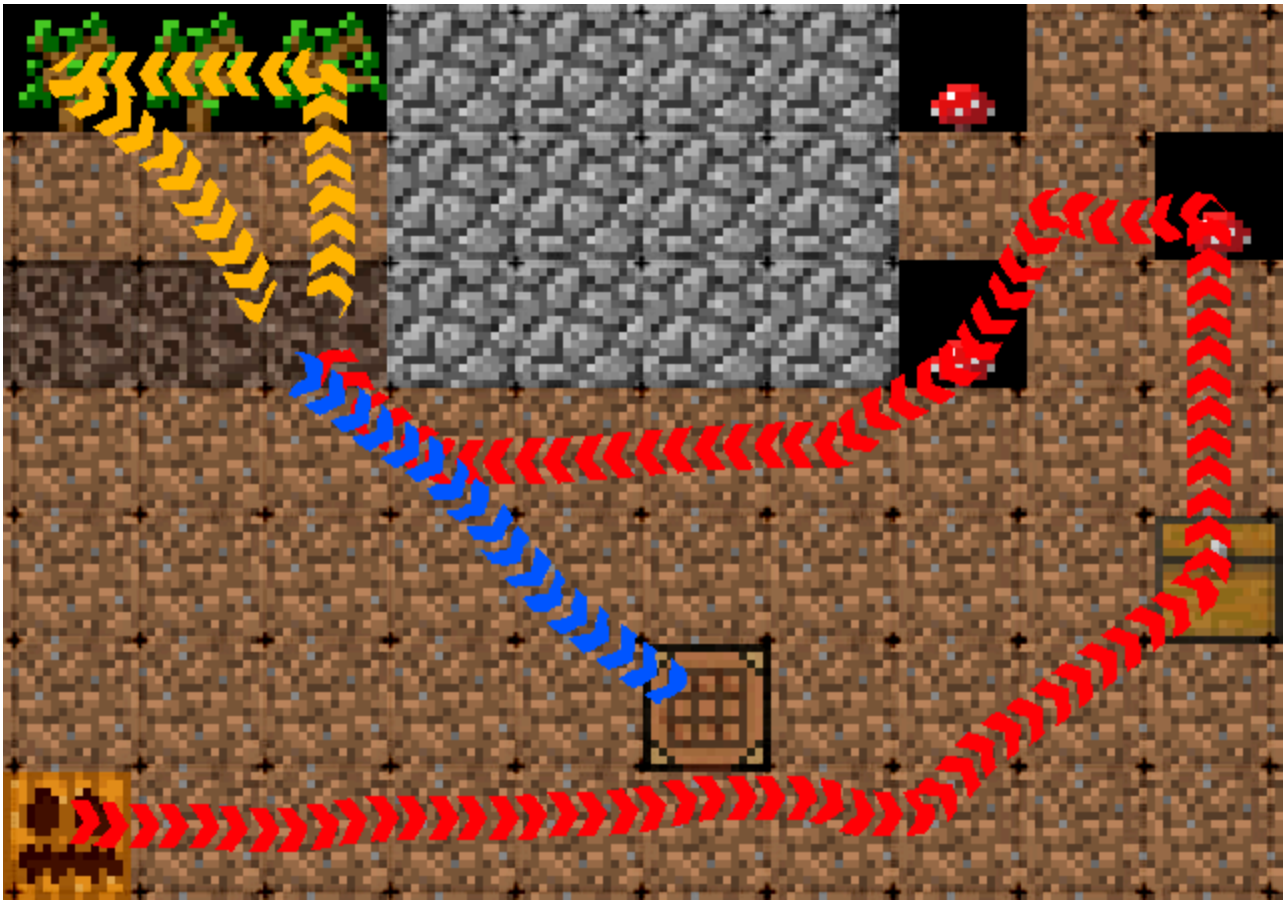
По имплементација на сите пет функции во класата, за влезот

```
0,0
9,2
```

го добиваме решението

```
['Move right', 'Move right', 'Move right', 'Move right', 'Move right', 'Move
right', 'Move right', 'Move up-right', 'Move up-right', 'Move up', 'Move up',
'Move up', 'Move left', 'Move down-left', 'Move down-left', 'Move left', 'Move
left', 'Move left', 'Move up-left', 'Unstuck', 'Move up', 'Move up', 'Move
left', 'Move left', 'Move down-right', 'Move down-right', 'Unstuck', 'Move
down-right', 'Move down-right', 'Move down-right']
```

Слика 2. Визуализација на решение на проблемот.



Други тест примери без визуализација:

```
1,5
0,5
```

```
['Move left', 'Move up', 'Move right', 'Move right', 'Move down-left', 'Move down-right', 'Unstuck', 'Move down-right', 'Move right', 'Move right', 'Move right', 'Move up-right', 'Move right', 'Move up-right', 'Move down-left', 'Move down-left', 'Move down-left', 'Move down-left']
```

```
9,2
0,0
```

```
['Move up', 'Move up', 'Move up', 'Move left', 'Move down-left', 'Move down-left', 'Move left', 'Move left', 'Move left', 'Move down-left', 'Move down-left', 'Move down-left', 'Move up-right', 'Move up-right', 'Move up-right', 'Move up-left', 'Unstuck', 'Move up', 'Move up', 'Move left', 'Move left',
```

```
'Move down-right', 'Move down-right', 'Unstuck', 'Move down-right', 'Move
down-right', 'Move down-right']
```

Целосниот код:

```
from searching_framework import Problem, breadth_first_graph_search


class Minecraft(Problem):
    def __init__(self, initial, chest, crafting_table, quicksand):
        super().__init__(initial)
        self.chest = chest
        self.crafting_table = crafting_table
        self.quicksand = quicksand

    def check_valid(self, state):
        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
state
        px, py = player

        if px < 0 or px > 9 or py < 0 or py > 6:
            return False

        if 3 <= px <= 6 and 4 <= py <= 6:
            return False

        if player in trees or player in mushrooms:
            return False

        if stuck and player not in self.quicksand:
            return False

        return True
    def successor(self, state):
        successors = {}

        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
state
        px, py = player

        actions = ['Move right', 'Move up-right', 'Move up', 'Move up-left',
'Move left', 'Move down-left', 'Move down', 'Move down-right']
        directions = [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0,
-1), (1, -1)]
```

```
        for action, (dx, dy) in zip(actions, directions):
            npx, npy = px + dx, py + dy
            nmushrooms = mushrooms
            nnum_mushrooms = num_mushrooms
            ntrees = trees
            nnum_trees = num_trees
            nhas_axe = has_axe
            nstuck = stuck

            if (npx, npy) == tuple(self.chest):
                nhas_axe = True
            elif (npx, npy) in mushrooms:
                nnum_mushrooms = num_mushrooms + 1
                nmushrooms = tuple(mushroom for mushroom in mushrooms if
mushroom != (npx, npy))
            elif (npx, npy) in trees and has_axe:
                nnum_trees = num_trees + 1
                ntrees = tuple(tree for tree in trees if tree != (npx, npy))
            elif (npx, npy) in self.quicksand:
                nstuck = True

            new_state = ((npx, npy), nmushrooms, nnum_mushrooms, ntrees,
nnum_trees, nhas_axe, nstuck)

            if self.check_valid(new_state):
                successors[action] = new_state

        if stuck and (px, py) in self.quicksand:
            successors['Unstuck'] = (player, mushrooms, num_mushrooms, trees,
num_trees, has_axe, False)

        return successors

    def actions(self, state):
        return self.successor(state).keys()

    def result(self, state, action):
        return self.successor(state)[action]

    def goal_test(self, state):
        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
state

        return num_mushrooms >= 2 and num_trees >= 3 and player ==
self.crafting_table
```

```python
if __name__ == '__main__':
    player = list(map(int, input().split(',')))
    chest = list(map(int, input().split(',')))

    mushrooms = ((7, 4), (7, 6), (9, 5))
    trees = ((0, 6), (1, 6), (2, 6))
    crafting_table = (5, 1)
    quicksand = ((0, 4), (1, 4), (2, 4))

    initial_state = (tuple(player), mushrooms, 0, trees, 0, False, False)

    problem = Minecraft(initial_state, chest, crafting_table, quicksand)
    solution = breadth_first_graph_search(problem)

    if solution is not None:
        print(solution.solution())
    else:
        print('No Solution!')
```

2. Информирано пребарување

Проблемите кои ги решаваме со информирано пребарување го следат истиот шаблон како и оние проблеми кои ги решававме со неинформирано пребарување. Она што е ново - во овие проблеми ќе треба да вклучиме евристики.

Евристики

За да го забрзаме процесот на пребарување низ состојби, може да дефинираме евристики - функции кои за дадена состојба даваат проценка во колку чекори ќе стигнеме до решението односно целната состојба.

Важно е да се напомене дека евристиките мора да се **допустливи/прифатливи** за алгоритмите да го вратат точното решение. Евристика е допустлива ако нејзината проценка за бројот на чекори да се реши проблемот е помал или еднаков на вистинскиот број чекори за решавање на проблемот. Допустливите евристики се оптимистички оценувачи.

Најидеална евристика е онаа која го враќа точниот број на потребни чекори за премин на тековната состојба до целната состојба, но поради сложеноста на проблемите, ова не секогаш е можно. Најчесто потребата на евристиките е да даваат брзи проценки, не да заглавуваат на сложени пресметувања над една состојба.

Ако работиме со евристики, прво е потребно да го вклучиме алгоритмот за пребарување A^*:

```
from searching_framework import astar_search
```

и соодветно да ја повикаме оваа функција над дадениот проблем како:

```
problem = Minecraft(initial_state, chest, crafting_table, quicksand)
solution = astar_search(problem)
```

Евристиките се дефинираат во рамките на самите класи кои наследуваат од класата `Problem`. Нивното име мора да биде означено со `h`. Тие примаат аргумент `node` кој е еден јазел од дрвото на пребарување. Само `node.state`, состојбата во дадениот јазел, ќе ја користиме.

```
class Minecraft(Problem):
    def h(self, node):
        state = node.state
        ...
```

(Во наредниот дел од текстот ќе разгледаме модификации на Задача 1.1. Да напоменеме дека евристиките кои ќе ги дефинираме не се единствени, може да постојат и други проценки за состојбите.)

Задача 1.2.

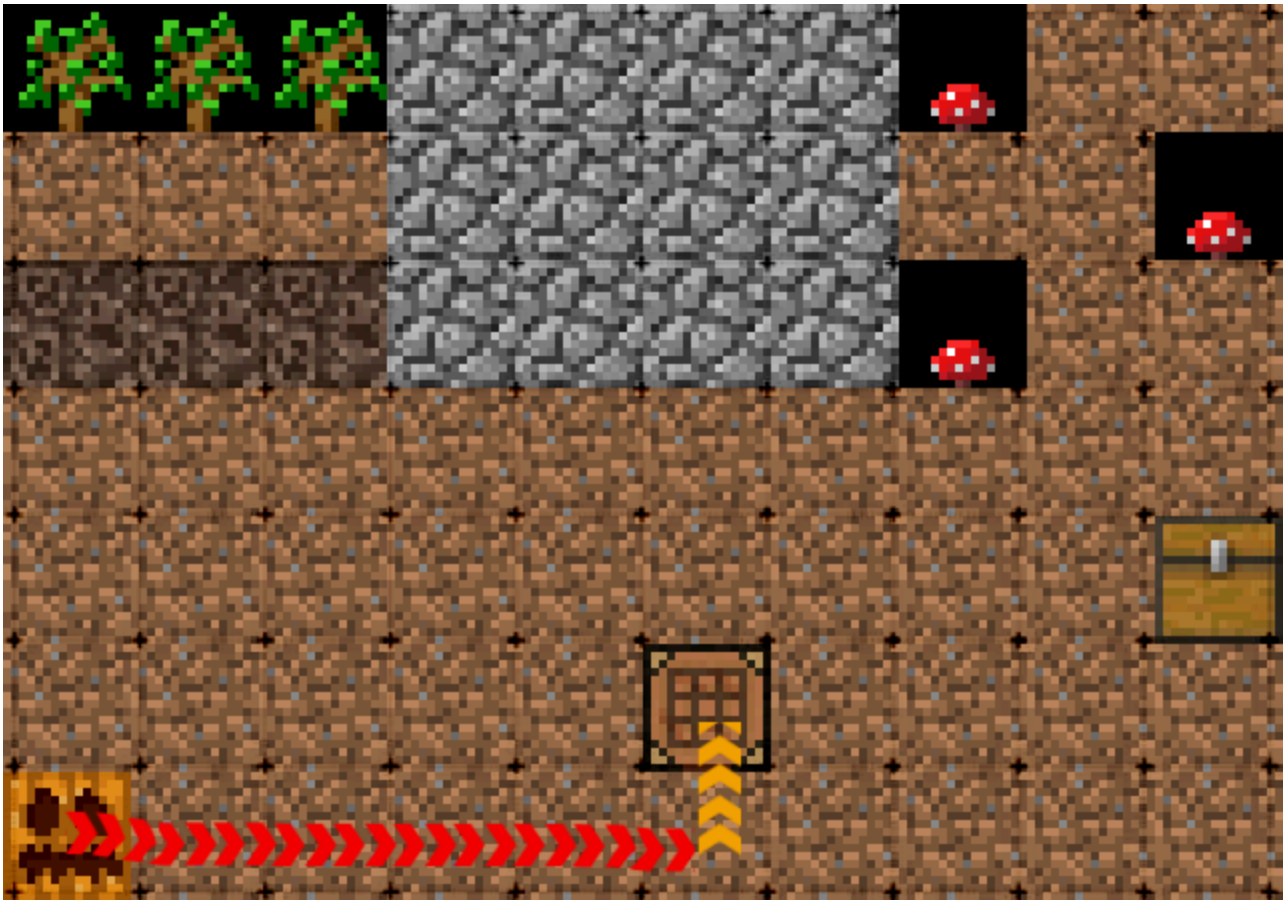
Во оваа верзија, нека играчот смее да се движи само горе, долу, лево и десно. На гридот нема жива песок. Целта е играчот само да стигне над масата за припрема (не треба да собира материјали).

Евристиката која е најпогодна за овој проблем може да го користи растојанието на Менхетен. Растојанието на Мехнетен е дефинирано на следниот начин:

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|.$$

Интерпретацијата на ова растојание е збир од хоризонталното и вертикалното растојание, на Слика 3. претставено како бројот на чекори на црвениот и портокаловиот пат.

Слика 3. Растојание на Менхетен.



Во код функцијата за пресметување на ова растојание може да го запишеме на следниот начин:

```
def manhattan_distance(coords1, coords2):
    x1, y1 = coords1
    x2, y2 = coords2
    return abs(x1 - x2) + abs(y1 - y2)
```

па така, евристиката ја дефинираме како:

```
def h(self, node):
    state = node.state
    player = state[0]
    crafting_table = self.crafting_table
    return manhattan_distance(player, crafting_table)
```

Задача 1.3.

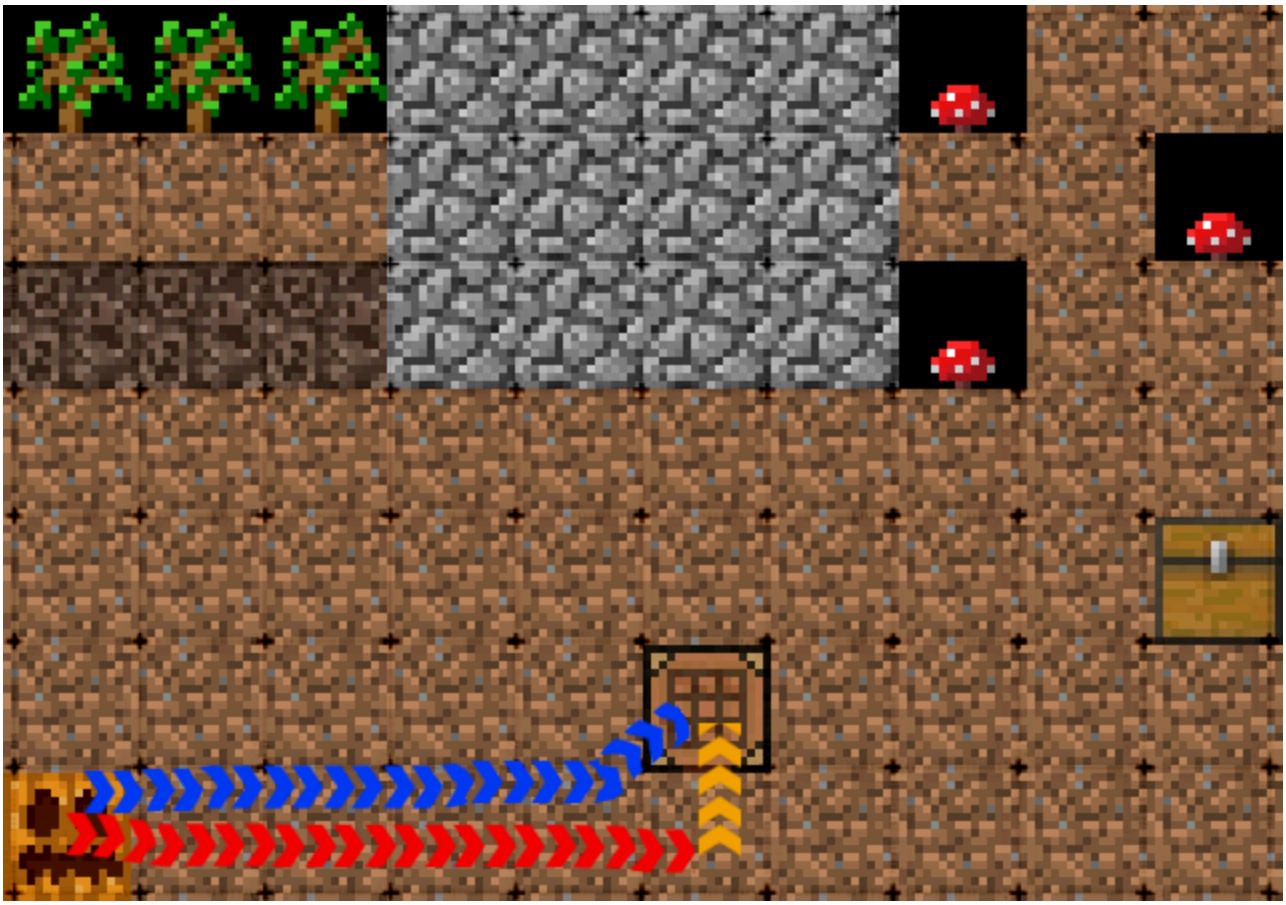
Во оваа верзија, играчот смее да се движи во сите 8 насоки. Како и претходно, нема жива песок и целта е играчот да стигне над масата.

Земајќи го предвид движењето на играчот, погодна евристика за решавање на проблемот го користи растојанието на Чебишов

$$d((x_1, y_1), (x_2, y_2)) = \max\{|x_1 - x_2|, |y_1 - y_2|\}.$$

Интерпретацијата на ова растојание е земање на поголемото од хоризонталното и вертикалното растојание, претставено со црвен и портокалов пат на Слика 4. Да забележеме дека бројот на чекори кое играчот го поминува (претставено со синиот пат) има ист број на чекори како и подолгиот од црвениот и портокаловиот пат (кој на Слика 4. е црвениот пат т.е. хоризонталното растојание меѓу позицијата на играчот и позицијата на масата).

Слика 4. Растојание на Чебишов. Должината на синиот пат (во чекори) е иста како и должината од подолгиот од останатите патишта (во овој случај црвениот пат).



Растојанието на Чебишов претставено во код:

```
def chebyshev_distance(coords1, coords2):
    x1, y1 = coords1
    x2, y2 = coords2
    return max(abs(x1 - x2), abs(y1 - y2))
```

и соодветната евристика:

```
def h(self, node):
    state = node.state
    player = state[0]
    crafting_table = self.crafting_table
    return chebyshev_distance(player, crafting_table)
```

Задача 1.4.

Во оваа верзија, играчот смее да се движи во сите 8 насоки, но има жива песок. Играчот започнува на позиција `(0, 5)`, а масата на позиција `(5, 1)`.

Она што се забележува е дека за играчот од позиција `(0, 5)` да стигне до позицијата на масата, задолжително мора да премине по барем едно поле со жив песок. Оптимистичка проценка за ова растојание е да се игнорира живата песок како препрека, што како резултат би вратила оцена која е за `1` помала од вистинската вредност на патот.

Но, може да додадеме вредност `1` во крајниот резултат во оценката за да ја добиеме следната евристика:

```
def h(self, node):
    state = node.state
    player = state[0]
    crafting_table = self.crafting_table
    return chebyshev_distance(player, crafting_table) + 1
```

Задача 1.5.

Во оваа верзија, играчот треба да ги собере сите печурки и да се врати на масата за припрема на материјали.

Бидејќи играчот треба да ги собере сите печурки, тогаш мора да ја собере и најдалечната од сите печурки релативно на играчот. Тогаш, една можна евристика е бројот на чекори на играчот да стигне до најдалечната печурка плус бројот на чекори од таа печурка до масата за припрема.

```
def h(self, node):
    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
node.state
    crafting_table = self.crafting_table

    if len(mushrooms) == 0:
        return chebyshev_distance(player, crafting_table)
```

```
    max_mushroom_distance = 0
    result = 0
    for mushroom in mushrooms:
        if chebyshev_distance(mushroom, player) > max_mushroom_distance:
            max_mushroom_distance = chebyshev_distance(mushroom, player)
            result = max_mushroom_distance + chebyshev_distance(mushroom,
crafting_table)

    return result
```

Доколку нема печурки кои треба да се соберат, се враќа растојанието на играчот до масата. Ако има печурки кои треба да се соберат, најголемото растојание од играчот до печурките го чуваме во `max_mushroom_distance`, и соодветно го ажурираме резултатот `result` - сумата од растојанието играч-најдалечна печурка и печурка-маса, кое потоа го враќаме.

Задача 1.6.

Во оваа верзија, играчот треба само да ги собере сите дрва и да се врати на масата за припрема на материјали. Ова значи дека прво играчот мора да оди над сандакот каде се наоѓа секирата.

Решението е слично како кај Задача 1.5. ако играчот веќе ја има секирата. Во случај кога играчот ја нема секирата, прво го пресметуваме растојанието од играчот до сандакот, а потоа ја применуваме идејата во Задача 1.5.

```
def h(self, node):
    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
node.state
    crafting_table = self.crafting_table
    chest = self.chest

    if len(trees) == 0:
        return chebyshev_distance(player, crafting_table)

    if has_axe:
        max_tree_distance = 0
        result = 0
        for tree in trees:
            if chebyshev_distance(tree, player) > max_tree_distance:
                max_tree_distance = chebyshev_distance(tree, player)
                result = max_tree_distance + chebyshev_distance(tree,
crafting_table)
    else:
```

```
        max_tree_distance = 0
        result = 0
        for tree in trees:
            if chebyshev_distance(tree, chest) > max_tree_distance:
                max_tree_distance = chebyshev_distance(tree, chest)
                result = max_tree_distance + chebyshev_distance(tree,
crafting_table)
        result += chebyshev_distance(player, chest)

    return result
```

Ако сите дрва се собрани, врати го растојанието од играчот до масата. Ако играчот има секира, врати го растојанието играч-дрво-маса. Ако играчот нема секира, врати го растојанието играч-ковчег плус растојанието ковчег-дрво-маса.

Задача 1.7.

Оваа верзија е идентична како оригиналната задача, т.е. Задача 1.1.

Бидејќи задачата содржи многу подвижни делови и е тешко да се одреди редоследот како играчот треба да ги посетува сите елементи во проблемот, ќе ја моделираме евристиката на друг начин.

За да го доловиме моментот дека играчот е потребно да ги собере потребните материјали, ќе ги користиме променливите `num_mushrooms` и `num_trees` за да ја дефинираме евристиката.

```
def h(self, node):
    player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
node.state
    crafting_table = self.crafting_table

    return max((3 - num_trees) + max(2 - num_mushrooms, 0),
chebyshev_distance(player, crafting_table))
```

Како резултат враќаме максимум од растојанието на играчот од масата, и бројот на потребни преостанати ресурси.
Ако имаме собрано `num_trees` број на дрва, тогаш ни преостанува да собереме уште `3 - num_trees` број на дрва.
Ако имаме собрано `num_mushrooms` број на печурки, тогаш ни преостануваат уште `2 - num_mushrooms` број на печурки кои треба да ги собереме, а доколку `num_mushrooms` го надмине потребниот број на собрани печурки, поставуваме предодредена вредност на нула.

Ова е само еден начин да се реши проблемот, каде идејата е да се потенцира употребата на променливите `num_trees` и `num_mushrooms`. Може да се дефинираат и други евристики кои би биле исто допустливи и доминантни во однос на таа што ја прикажавме, но тоа ќе го оставиме за вежба на читателот ☻.

Целосниот код од овој дел:

```
from searching_framework import Problem, breadth_first_graph_search
from searching_framework import astar_search


def manhattan_distance(coords1, coords2):
    x1, y1 = coords1
    x2, y2 = coords2
    return abs(x1 - x2) + abs(y1 - y2)


def chebyshev_distance(coords1, coords2):
    x1, y1 = coords1
    x2, y2 = coords2
    return max(abs(x1 - x2), abs(y1 - y2))


class Minecraft(Problem):
    def __init__(self, initial, chest, crafting_table, quicksand):
        super().__init__(initial)
        self.chest = chest
        self.crafting_table = crafting_table
        self.quicksand = quicksand

    def check_valid(self, state):
        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
state
        px, py = player

        if px < 0 or px > 9 or py < 0 or py > 6:
            return False

        if 3 <= px <= 6 and 4 <= py <= 6:
            return False

        if player in trees or player in mushrooms:
            return False

        if stuck and player not in self.quicksand:
```

```
            return False

        return True
    def successor(self, state):
        successors = {}

        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
state
        px, py = player

        actions = ['Move right', 'Move up-right', 'Move up', 'Move up-left',
'Move left', 'Move down-left', 'Move down', 'Move down-right']
        directions = [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0,
-1), (1, -1)]

        for action, (dx, dy) in zip(actions, directions):
            npx, npy = px + dx, py + dy
            nmushrooms = mushrooms
            nnum_mushrooms = num_mushrooms
            ntrees = trees
            nnum_trees = num_trees
            nhas_axe = has_axe
            nstuck = stuck

            if (npx, npy) == tuple(self.chest):
                nhas_axe = True
            elif (npx, npy) in mushrooms:
                nnum_mushrooms = num_mushrooms + 1
                nmushrooms = tuple(mushroom for mushroom in mushrooms if
mushroom != (npx, npy))
            elif (npx, npy) in trees and has_axe:
                nnum_trees = num_trees + 1
                ntrees = tuple(tree for tree in trees if tree != (npx, npy))
            elif (npx, npy) in self.quicksand:
                nstuck = True

            new_state = ((npx, npy), nmushrooms, nnum_mushrooms, ntrees,
nnum_trees, nhas_axe, nstuck)

            if self.check_valid(new_state):
                successors[action] = new_state

        if stuck and (px, py) in self.quicksand:
            successors['Unstuck'] = (player, mushrooms, num_mushrooms, trees,
num_trees, has_axe, False)
```

```
        return successors

    def actions(self, state):
        return self.successor(state).keys()

    def result(self, state, action):
        return self.successor(state)[action]

    def goal_test(self, state):
        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
state

        return num_mushrooms >= 2 and num_trees >= 3 and player ==
self.crafting_table

    def h(self, node):
        player, mushrooms, num_mushrooms, trees, num_trees, has_axe, stuck =
node.state
        crafting_table = self.crafting_table

        return max((3 - num_trees) + max(2 - num_mushrooms, 0),
chebyshev_distance(player, crafting_table))


if __name__ == '__main__':
    player = list(map(int, input().split(',')))
    chest = list(map(int, input().split(',')))

    mushrooms = ((7, 4), (7, 6), (9, 5))
    trees = ((0, 6), (1, 6), (2, 6))
    crafting_table = (5, 1)
    quicksand = ((0, 4), (1, 4), (2, 4))

    initial_state = (tuple(player), mushrooms, 0, trees, 0, False, False)

    problem = Minecraft(initial_state, chest, crafting_table, quicksand)
    solution = astar_search(problem)

    if solution is not None:
        print(solution.solution())
    else:
        print('No Solution!')
```

3. Проблеми кои исполнуваат услови

За работа со проблеми кои исполнуваат услови (`Constraint satisfaction problems`) потребно е да го преземеме следниот модул:

```
pip install python-constraint
```

Во проблемите кои исполнуваат услови е потребно добро да се моделираат променливите (`variables`), доменот на вредности (`domain`) кои се доделуваат на променливите, и условите (`constraints`) кои ограничуваат кои вредности од доменот може да се доделат на променливите.

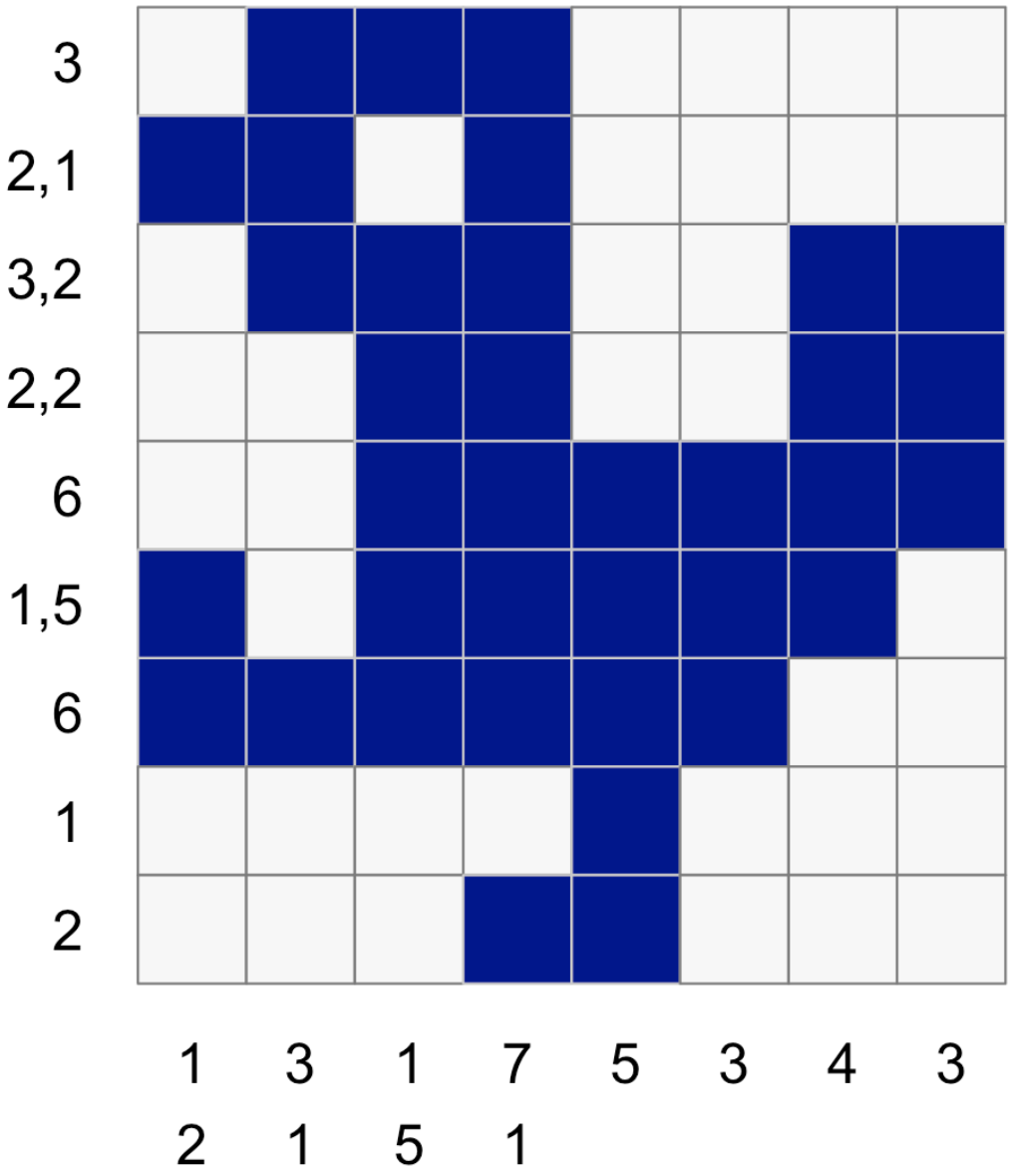
Овие елементи ќе ги разгледаме преку следните задачи.

Задача 2.

Нонограми претставуваат логички загатки во кои ќелиите во даден грид мора да бидат обоени или оставени празни според броевите на рабовите на гридот за да се открие скриена слика.

Слика 5. Пример за нонограм. Првата редица го содржи бројот `3` и соодветно гледаме 3 последователно обоени ќелии. Првата колона ги содржи броевите `1, 2`, па гледаме една обоена ќелија, проследена со барем едно празно место, па потоа две последователно обоени ќелии.

Duck



(За големи вредности на гридот, пребарувањето на алгоритмите може да биде многу споро.)

За дадени вредности во секоја редица и колона, потребно е да го обоиме гридот т.е. да означиме кои ќелии треба да се обојат.

На влез се прима број `n` кој е големината на $n \times n$ таблата. Понатаму во следните `n` редици се проследуваат вредностите за секој ред. Потоа во наредните `n` редици се проследуваат вредностите за секоја колона.

Пример влез:

```
2
1
0
0
1
```

Визуализација на влезот, со обоена ќелија претставена со `X`.

1	.	X
0	.	.
	0	1

Дел 1. Дефинирање на променливи

Целта на задачата е да го обоеме гридот, односно за секоја ќелија да одлучиме дали ќе ѝ доделиме боја.

Така, променливи (`variables`) ни се сите ќелии $c_{ij} \equiv (i, j)$ кои ќе ни бидат торки од нивната позиција.

Доменот на вредности (`domain`) ни се можните вредности за секоја од променливите, па може со `1` да означиме дека ќе ставаме боја во таа ќелија, додека `0` ни значи спротивното. Оттука $D = \{0, 1\}$.

Во код:

```python
from constraint import *


if __name__ == '__main__':
    n = int(input())
    rows = [tuple(map(int, input().split(','))) for _ in range(n)]
    cols = [tuple(map(int, input().split(','))) for _ in range(n)]

    variables = [(i, j) for i in range(n) for j in range(n)]
    domain = [0, 1]
```

За да дефинираме проблем кој исполнува услови, ќе ја користиме класата `Problem` (оваа класа доаѓа од `constraint` библиотеката, не од `searching_framework`) на следниот начин.

```
problem = Problem(BacktrackingSolver())
problem.addVariables(variables, domain)
```

Дел 2. Дефинирање на услови/ограничувања

Во овој дел е потребно да се ограничат дел од променливите преку одредени правила. Правилата во оваа игра се едноставни, за дадена редица/колона, провери дали вредностите на променливите во таа редица/колона се такви, така што соодветните вредности во `rows/cols` соодветствуваат на вредностите на променливите.

Односно, ако во дадена редица имаме вредности `1 0 1 1`, во соодветниот дел од `rows` треба да постои торка `(1, 2)`.

Да го дефинираме ова ограничување на следниот начин:

```
def check_line(values, variables):
    blocks = []

    continuous = False
    temp = 0
    for variable in variables:
        if variable == 1:
            continuous = True
            temp += 1
        if variable == 0 and continuous:
            continuous = False
            blocks.append(temp)
            temp = 0
    if temp > 0:
        blocks.append(temp)
    if not blocks:
        blocks.append(0)

    return values == tuple(blocks)
```

Ограничувањето `check_line` како аргументи прима `values` кои се торките од броеви на последователно обоени ќелии, пример `(1, 2)`. Променливата `variables` е листа од вредностите на променливите кои се проследени. Пример ако се проследени ќелиите од првата редица како $c_{00}, c_{10}, \ldots$, променливата `variables` е листа од нивните бои т.е. `[1, 0, ...]`.

Понатаму следува код кој за дадена листа од вредности, пример `[1, 0, 0, 1, 1, 0, 0, 1]`, ја трансформира истата во променлива `blocks` која е листа од бројот на

последователни единици, т.е. `blocks = [1, 2, 1]`.

Ако за дадената линија (ред/колона), `blocks` и `values` имаат исти елементи, тогаш условот е задоволен, во спротивно условот е прекршен.

Следно треба да ги додадеме овие ограничувања во самиот проблем. Ограничувањата се однесуваат на сите редици и колони, па за дадена редица/колона, потребно е да ги преземеме сите променливи од таа редица/колона, како и соодветната вредност од `rows/cols` и да го додадеме ограничувањето.

```
for i in range(n):
    problem.addConstraint(lambda *x, col=i: check_line(cols[col], x), [(i, j)
for j in range(n)])
    problem.addConstraint(lambda *x, row=i: check_line(rows[row], x), [(j, i)
for j in range(n)])
```

`problem.addConstraint` е функција која како прв аргумент очекува ограничување, а како втор аргумент листа од променливи за кои треба да важи ограничувањето. Ограничувањето може да е и функција.

За секое `i` го правиме следното:

- Ги земаме сите променливи кои се наоѓаат во колоната `i`, имено `[(i, j) for j in range(n)]`.
- Сите променливи ги ставаме во функцијата `check_line`.
  - `lambda` изразот дефинира ламбда функција, функција која може да ја дефинираме во една линија. Од левата страна на симболот `:` стојат влезните аргументи. Од десната страна стои излезот од ламбда функцијата.
  - `*x` дефинира променлива `x` која ќе ги содржи сите влезни аргументи кои ќе и ги проследиме на ламбда функцијата. Наместо да пишуваме функција од `n` променливи (колку што има во редица/колона), `*x` ни овозможува елегантно да ги агрегираме сите променливи.
  - `col=i` е екстремно важен дел кога користиме ламбда изрази/функции. Ламбда изразите по природа се *мрзеливи* (lazy evaluation), па ако не ја прецизираме во самиот влез со која колона ќе работиме, ламбда изразот може да земе вредност за `i` во понатамошното извршување на програмата, што ќе даде погрешен резултат. (Значи, ако го отстраневме `col=i` и напишевме `cols[i]` ќе се земе погрешна колона поради мрзеливата евалуација)
  - `check_line(cols[col], x)` излезот на функцијата ќе биде булеан. Булеанот ќе соодветствува на тоа дали променливите агрегирани во `x` (пример `[1, 0, 0, 1,`

`1]`) се наместени на тој начин што соодветствуваат на последователно обоени ќелии зададени со вредностите од `cols[col]` (пример `(1, 2)`).

- Го правиме истото и за сите редици.

Откако ги додадовме сите променливи и ограничувања, ги наоѓаме сите решенија како

```
solutions = problem.getSolutions()
print(solutions)
```

или доколку сакаме само едно решение

```
solution = problem.getSolution()
```

Променливата `solution` ни претставува речник од парови променлива-вредност

```
for key, value in solution.items():
    print(key, value)
```

Целосно решение:

```
from constraint import *


def check_line(values, variables):
    blocks = []

    continuous = False
    temp = 0
    for variable in variables:
        if variable == 1:
            continuous = True
            temp += 1
        if variable == 0 and continuous:
            continuous = False
            blocks.append(temp)
            temp = 0
    if temp > 0:
        blocks.append(temp)
    if not blocks:
        blocks.append(0)

    return values == tuple(blocks)
```

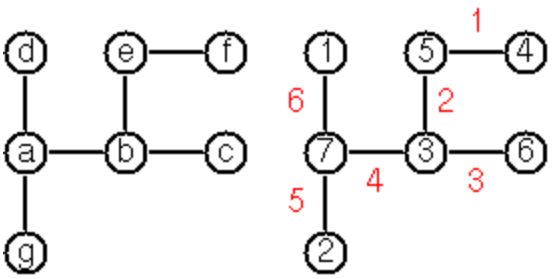
```
if __name__ == '__main__':
    n = int(input())
    rows = [tuple(map(int, input().split(','))) for _ in range(n)]
    cols = [tuple(map(int, input().split(','))) for _ in range(n)]

    variables = [(i, j) for i in range(n) for j in range(n)]
    domain = [0, 1]

    problem = Problem(BacktrackingSolver())
    problem.addVariables(variables, domain)

    for i in range(n):
        problem.addConstraint(lambda *x, col=i: check_line(cols[col], x), [(i,
j) for j in range(n)])
        problem.addConstraint(lambda *x, row=i: check_line(rows[row], x), [(j,
i) for j in range(n)])

    solutions = problem.getSolutions()
    print(solutions)

    solution = solutions[0]
    for key, value in solution.items():
        print(key, value)
```

Задача 3.

За дадено дрво кое се состои од N темиња и $N-1$ ребра, да се нумерираат темињата со различни вредности од 1 до N и ребрата со различни вредности од 1 до $N-1$ за да важи следното:

> За секое ребро K важи дека апсолутната разлика на темињата од кое е формирано реброто е исто K.

Слика 6. Дрво пред нумерирање (лево). Дрво со валидно нумерирање (десно).



Влезот го задаваме на следнот начин - во првата линија се чита бројот на темиња n. Темињата ќе ги означуваме како $v_1, v_2, \ldots, v_n$. Во следните $n-1$ линии се читаат ребрата

во формат `i,j` што означува дека постои ребро меѓу темето v_i и v_j.

Еден пример за влез на програмата согласно Слика 6. е:

```
7
1,2
1,4
1,7
2,3
2,5
5,6
```

Прво ги дефинираме променливите. Тоа се темињата и ребрата на графот.

Имаме точно n променливи за темињата и тоа $v_1, v_2, \ldots, v_n$. Наместо да ги пишуваме темињата како `v1`, ќе пишуваме само стрингот `1`. Ова **не** значи дека темето `v1` има вредност `1`. Пример темето `1` може да има вредност "5".
Имаме точно $n-1$ променливи за ребрата и тоа $e_1, e_2, \ldots e_{n-1}$. Ребрата ќе ги пишуваме како пар од темиња одделени со запирки, т.е. "1,2".

Доменот на вредности за темињата е множеството $D_V = \{1, 2, \ldots, n\}$.
Доменот на вредности за ребрата е множеството $D_E = \{1, 2, \ldots, n-1\}$.

Еве неколку примери за доделување на вредности - темето "1" ја прима вредноста 2; темето "2" ја прима вредноста 1; реброто "1,2" ја прима вредноста 1.

Во код претходното го моделираме на следниот начин:

```python
from constraint import *


if __name__ == '__main__':
    n = int(input())
    edges = [input() for _ in range(n - 1)]

    problem = Problem(BacktrackingSolver())

    vertices = [f'{i + 1}' for i in range(n)]
    DV = [i + 1 for i in range(n)]
    DE = [i + 1 for i in range(n - 1)]

    problem.addVariables(vertices, DV)
    problem.addVariables(edges, DE)
```

Ограничувањата кои треба да ги додадеме се следни:

1. Секое теме/ребро мора да има уникатна вредност.
2. Апсолутната разлика на вредностите на темињата од кое е составено едно ребро мора да биде еднаква на вредноста на тоа ребро.

За да го исполнеме првиот услов, веќе постои ограничување кое може да го искористиме во рамките на модулот `constraint` и се нарекува `AllDifferentConstraint`.

```
problem.addConstraint(AllDifferentConstraint(), vertices)
problem.addConstraint(AllDifferentConstraint(), edges)
```

За да го исполнеме вториот услов, ја дефинираме следната функција:

```
def edge_value_constraint(value1, value2, edge):
    return edge == abs(value1 - value2)
```

Функцијата враќа точно доколку вредноста на апсолутната разлика на вредностите на темињата е еднаква со вредноста на реброто.

Ова ограничување го додаваме на секое ребро на следниот начин:

```
for edge in edges:
    vertex1, vertex2 = edge.split(',')
    problem.addConstraint(edge_value_constraint, [vertex1, vertex2, edge])
```

За секое ребро се гледа од кои темиња е составено. Променливите од листата од `vertex1`, `vertex2` и `edge` имплицитно се праќаат како аргументи на `edge_value_constraint` ограничувањето.

Важно: Во `addConstraint` функцијата ние ги проследуваме имињата на променливите, но во `edge_value_constraint` функцијата ќе се искористат ***вредностите*** на тие променливи, не нивните имиња. Пример ако сме ја пратиле листата `['1', '5', '1,5']` т.е. ознаките на првото и петтото теме, како и реброто меѓу нив, кога тие ќе се проследат во функцијата `edge_value_constraint(value1, value2, edge)`, променливите `value1`, `value2` и `edge` нема да бидат стрингови, туку вредности од домените D_V и D_E соодветно.

Целосното решение:

```
from constraint import *
```

```
def edge_value_constraint(value1, value2, edge):
    return edge == abs(value1 - value2)


if __name__ == '__main__':
    n = int(input())
    edges = [input() for _ in range(n - 1)]

    problem = Problem(BacktrackingSolver())

    vertices = [f'{i + 1}' for i in range(n)]
    DV = [i + 1 for i in range(n)]
    DE = [i + 1 for i in range(n - 1)]

    problem.addVariables(vertices, DV)
    problem.addVariables(edges, DE)

    problem.addConstraint(AllDifferentConstraint(), vertices)
    problem.addConstraint(AllDifferentConstraint(), edges)

    for edge in edges:
        vertex1, vertex2 = edge.split(',')
        problem.addConstraint(edge_value_constraint, [vertex1, vertex2, edge])

    print(problem.getSolutions())
```