Комп'ютерний практикум №3

Інтелектуальний агент з моделями управління на основі станів

Бригада 14: Мартинюк Ростислав

Група: ІК-02

Мета роботи: ознайомитись з моделями на основі станів та класичними методами пошуку в просторі станів в детермінованих і стохастичних середовищах; дослідити їх використання для управління інтелектуальним агентом в типовому середовищі.

***З*авдання:** обрати середовище моделювання та задачу, що містить агента, який може бути навчений відповідними методом. В обраному середовищі реалізувати модель інтелектуального агента, що реалізує найкращу стратегії досягнення цільового стану: для детермінованої та стохастичної версії одного і того ж середовища, використовуючи методи і вимоги свого варіанту. Проаналізувати результати та особливості використаних методів пошуку.

Номер варіанту: 41

Завдання для варіанту:

Номер студента/ бригади	Форма карти	Алгоритм для детермінованого середовища	Алгоритм для стохастичного середовища
41	лабіринт	RBFS	Value iteration

Середовище: Лабіринт 30x30(як виглядає можна побачити у прикладі роботи). Можливі стани: стіни(куди агент не може потрапити), проходи(куди агент може перейти з поточного стану). Вартість клітинки обраховується як, вартість = відстань до фінішу + 5*кількість відвідувань цієї клітинки. Агент обирає наступну клітинку з найменшою вартістю. Врахування кількості відвідувань було додано щоб агент не зациклювався у тупіках.

Алгоритм для детермінованого середовища: RBFS, або Recursive Best-First Search, це алгоритм пошуку в графах, який використовує стратегію кращого першого вибору. Він працює рекурсивно, розглядаючи найкращий вузол на поточному рівні перед тим, як переходити до наступного рівня. Кожен вузол зберігається в стеку, але тільки частково розгорнуті вузли знаходяться в пам'яті, що дозволяє ефективно обходити графи з великою кількістю вузлів.

Евристична функція оцінки: Вартість клітинки обраховується як, вартість = відстань до фінішу + 5*кількість відвідувань цієї клітинки.

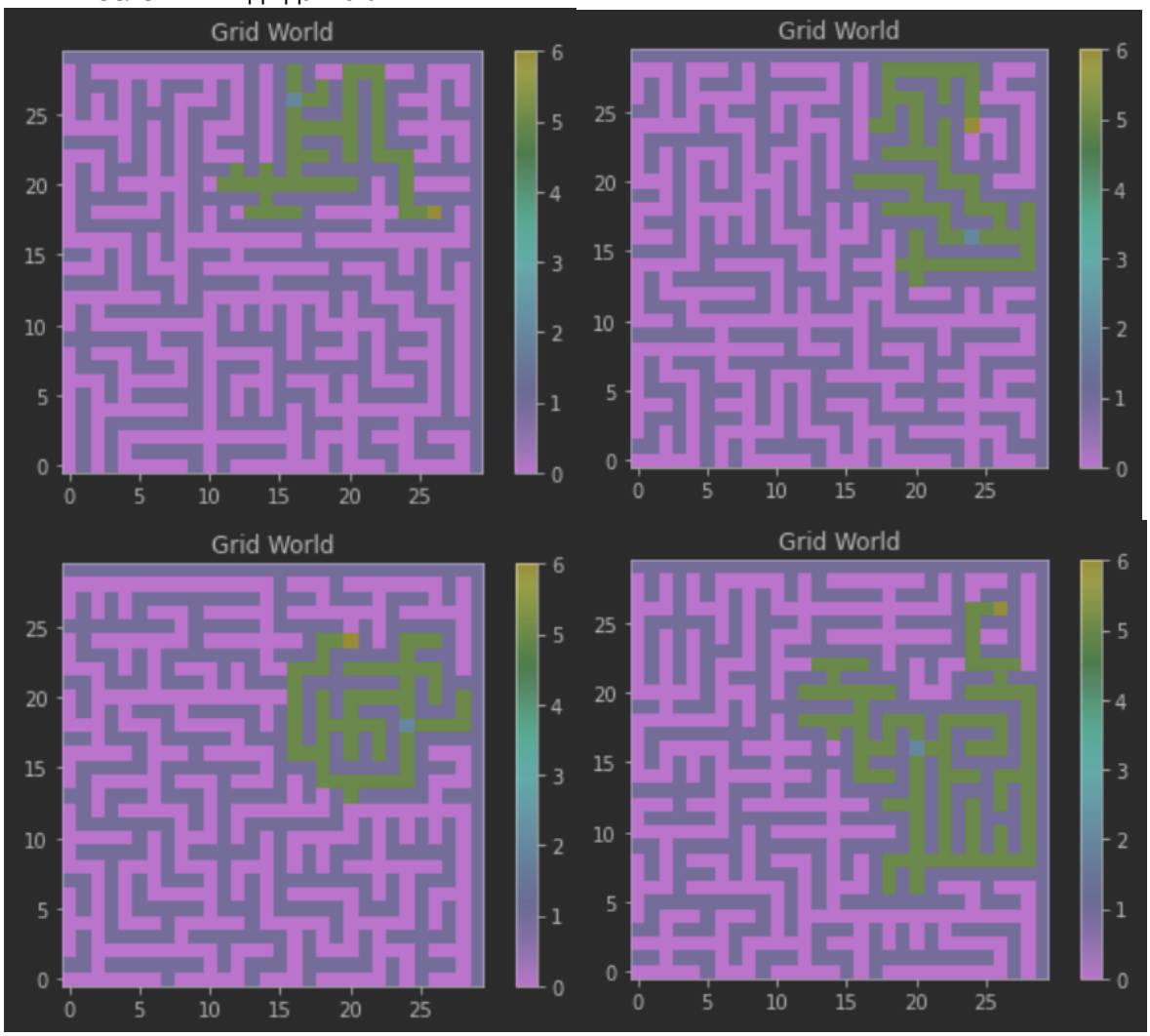
Програмна реалізація алгоритму інформованого пошуку:

Функція реалізації алгоритму RBFS:

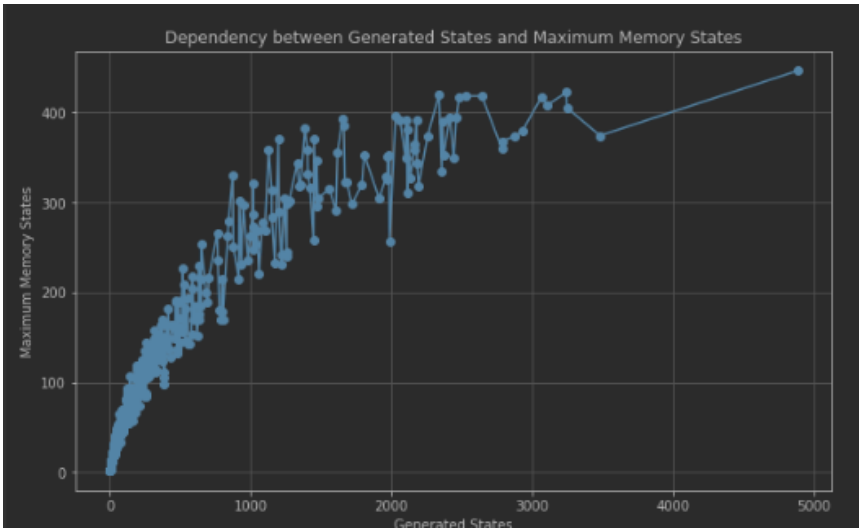
```
def rbfs(self, state, g, f_limit, path):
    if self.environment.is_goal_state(state):
        return path

    self.generated_states += 1
    self.max_memory_states = max(self.max_memory_states,
len(self.visited states))

    self.visit_count[state] = self.visit_count.get(state, 0) + 1
    self.visited_states.add(state)
    self.environment.visited_states.add(state)

    successors = self.environment.get_possible_actions(state)
    if not successors:
        return None

    previous_state = path[-1].state if path else state
    successors = [action for action in successors if action != previous_state]

    f_values = [g + self.heuristic(successor) + 5 *
self.visit_count.get(successor, 0) for successor in successors]
    path_nodes = [Node(successor, None, g + self.heuristic(successor) + 5 *
self.visit_count.get(successor, 0)) for successor in successors]

    while successors:
        best_index = f_values.index(min(f_values))
        best_f = f_values.pop(best_index)
        best_successor = successors.pop(best_index)
        best_path_node = path_nodes.pop(best_index)

        if best_f > f_limit:
            return None

        alternative_f = min(f_values) if f_values else float('inf')

        result = self.rbfs(best_successor, g + 1, min(f_limit, alternative_f),
path + [best_path_node])
        if result:
            return result
```

Приклад роботи розробленої програми:
Значення кольорів клітинок:

- Рожевий – проходи у лабіринті
- Фіолетовий – стіни
- Блакитний – початковий стан агенту (старт)
- Жовтий – кінцевий стан (фініш)
- Зелений – відвідані стани



Результати використання алгоритмів: Було проведено 1000 запусків та побудовано графік залежності максимальної кількісті станів від загальної кількісті згенерованих станів під час пошуку. Бачимо, що графік набуває логарифмічного вигляду, це означає, що при роботі з більшою кількістю станів алгоритм буде більш залежним від обчислювальної потужності ніж від пам’яті.



Стохастична версія середовища: Агент може з заданою імовірністю потрапити у випадковий з доступних на поточному кроці станів.

Метод для стохастичного середовища: Value iteration - основна ідея полягає в тому, щоб ітеративно оцінювати функцію цінності для кожного стану в середовищі. Після кожної ітерації алгоритм оновлює значення функції цінності для кожного стану, використовуючи максимальне очікуване нагороду за можливі дії. Процес триває до того часу, поки функція цінності не збіжиться до оптимального значення. Value iteration використовує офлайн стратегію, оскільки він повністю оцінює функцію цінності перед вибором оптимальної стратегії. Тобто, він не взаємодіє з середовищем під час оцінювання, а лише аналізує його структуру. Дослідження середовища може бути важливим етапом перед використанням value iteration. Агент повинен мати відомість про можливі стани і можливі дії в цих станах. Цю інформацію можна отримати як експертно, так і експериментально, проводячи дослідження в середовищі. Реалізація дослідження може включати в себе збір даних про стани, нагороди і переходи між станами.

Реалізація методу:

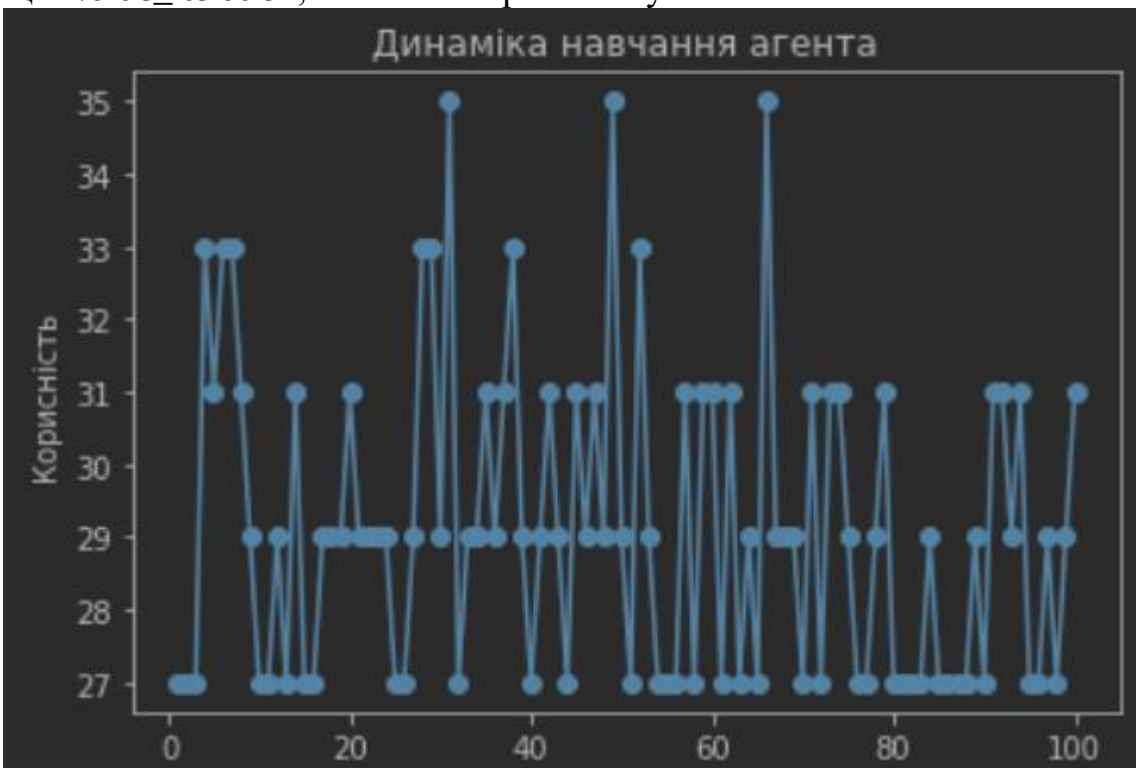
```
def value_iteration(self, max_iterations=100):
    for   in range(max_iterations):
        new_values = np.copy(self.values)
        for i in range(self.environment.size[0]):
            for j in range(self.environment.size[1]):
                if (i, j) == self.environment.goal:
                    continue

                next_states = self.environment.get_possible_actions((i, j))
                expected_rewards = []

                for next_state in next_states:
                    reward = self.environment.get_reward((i, j), next_state)
                    expected_rewards.append(self.environment.slip_probability *
(reward + self.discount_factor * self.values[next_state]) +
                                            (1 -
self.environment.slip_probability) * (reward + self.discount_factor *
self.values[(i, j)]))
                if expected_rewards:
```

```
                    new_values[(i, j)] = max(expected_rewards)

        self.values = new_values
```

Метод яким агент обирає наступний крок:
slip_probability – імовірність потрапити у випадковий стан

```
def make_decision(self):
    possible_actions =
self.environment.get_possible_actions(self.current_state)

    if np.random.rand() < self.environment.slip_probability:
        random_index = np.random.choice(len(possible_actions))
        return possible_actions[random_index]

    best_action = None
    best_value = float('-inf')

    for action in possible_actions:
        reward = self.environment.get_reward(self.current_state, action)
        value = self.environment.slip_probability * (reward +
self.discount_factor * self.values[action]) + \
                (1 - self.environment.slip_probability) * (reward +
self.discount_factor * self.values[self.current_state])

        if value > best_value:
            best_value = value
            best_action = action

    return best_action
```

Результати застосування розробленого методу: Було проведено тренування агента зі 100 епізодів. Тобто на початку кожного епізоду виконується 100 ітерацій value_iteration, і агент повертається у початковий стан.



Оцінка результатів:
RBFS : Графік залежності максимальної кількості станів від загальної кількості згенерованих станів може вказувати на ефективність алгоритму при роботі з великою кількістю станів. Логарифмічний вигляд графіка може свідчити про те,

що RBFS добре справляється з великими просторами станів, але при цьому необхідно враховувати обчислювальну потужність.
Value Iteration: Основні показники корисністі лежать у межах від 27 до 31 може бути позначено як відносно непогана, але також може вказувати на можливість покращення стратегії. Але якщо згадати що агент з 10% імовірністю потрапляє у випадковий з доступних станів, то можна сказати що метод добре працює.

Висновки:

RBFS:

Переваги:

- Ефективний для роботи з великими просторами станів, може швидко знаходити оптимальні шляхи.
- Працює з різноманітними та складними просторами станів.

Недоліки:

- Вимагає значних обчислювальних ресурсів, особливо при великій кількості станів.
- Може не бути оптимальним для просторів станів з великою кількістю можливих шляхів.

Value Iteration:

Переваги:

- Добре підходить для задач з марківськими випадковими процесами прийняття рішень (MDP).
- Може знаходити оптимальні стратегії в умовах невизначеності та помилкових дій.

Недоліки:

- Може бути обчислювально витратним, особливо при великій кількості станів та епізодів.
- Вимагає точних даних про ймовірності переходів та винагород.