2. ЛАБОРАТОРНАЯ РАБОТА № 2 «ИССЛЕДОВАНИЕ НЕИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»

2.1. Цель работы

Исследование неинформированных методов поиска решений задач в пространстве состояний, приобретение навыков программирования интеллектуальных агентов, планирующих действия на основе методов слепого поиска решений задач.

2.2. Краткие теоретические сведения

2.2.1. Обучающая среда AI Pacman

Обучающая среда AI Pacman была разработана Джоном ДеНеро и Дэном Кляйном [6] для обучения искусственному интеллекту. Оригинальная видео игра ***Pac-Man*** была разработана японской компанией Namco в1980 году.

Игра Pacman проста: Pacman (герой игры) должен пройти лабиринт (рисунок 2.1) и съесть все (маленькие) гранулы, не будучи съеденным злобными привидениями, которые охотятся на него. Если Pacman съест одну из (больших) энергетических гранул, он становится невосприимчивым к привидениям на определенный период времени и получает способность поедать призраков, зарабатывая очки.

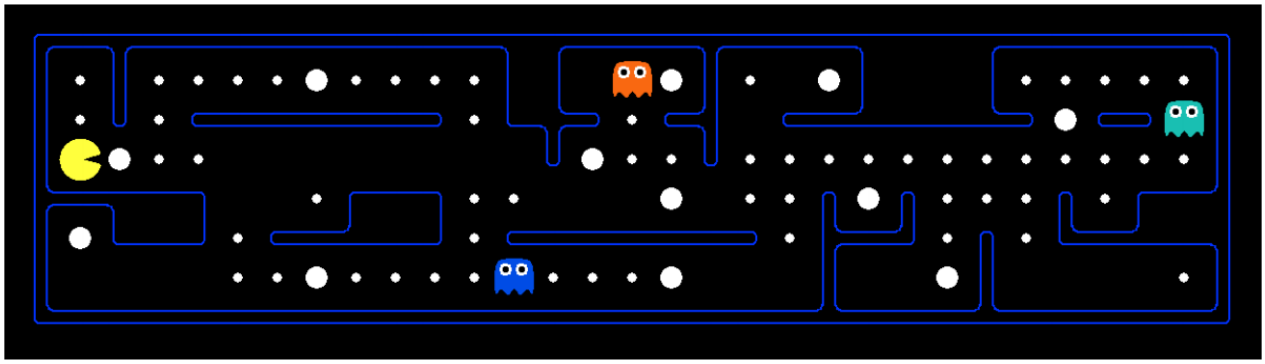

Рисунок 2.1 – Экран обучающей среды AI Pacman

2.2.2. Агенты

Основной задачей искусственного интеллекта является создание **рационального агента** – сущности, которая имеет цели, воспринимает среду и пытается выполнить серию **действий**, которые приведут к наилучшему/оптимальному ожидаемому результату с учетом этих целей. Рациональные агенты существуют в **среде**, специфичной для заданного агента. Например, для шашечного агента среда – это виртуальная шашечная доска, на которой он играет против оппонентов, где ходы фигур – это действия. Вместе среда и агенты создают некоторый **мир**.

Рефлекторный агент – это агент, который не оценивает последствия своих действий, и, скорее, выбирает действие, основываясь исключительно на текущем состоянии мира. Такие агенты обычно проигрывают агентам, **планирующим** свои

действия, которые поддерживают некоторую модель мира и используют её для симуляции выполнения различных действий. Кроме этого, агент может строить гипотезы о предполагаемых последствиях действий и выбирать лучшие из них. Таким образом, агент моделирует одну из интеллектуальных функций – продумывание действий наперед.

2.2.3. Формулировка задач поиска в пространстве состояний

Чтобы создать рационального агента, планирующего действия, нужен способ математического описания среды, в которой функционирует агент. Для этого мы должны формально представить задачу поиска, учитывая текущее состояние агента и то, как агент может перейти в новое состояние, которое наилучшим образом удовлетворяет цели функционирования агента. Представление задачи в такой постановке соответствует поиску в пространстве состояний и определяется совокупностью четырех составляющих

$$(S_0, S, F, G), \qquad (2.1)$$

где S — множество (пространство) состояний задачи; S_0 — множество начальных состояний, $S_0 \in S$; F — множество функций, преобразующих одни состояния в другие; G — множество целевых состояний, $G \in S$ [1].

Каждая функция $f \in F$, отображает одно состояние в другое — $s_j = f(s_i)$, где s_i, $s_j \in S$. Решением задачи является последовательность функций (действий) f_i , преобразующих начальные состояния в конечные, т.е. $f_n \; f_{n-1}(\ldots(f_2(f_1(S_0)))\ldots)) \in G$. Если такая последовательность не одна и задан критерий оптимальности, то возможен поиск оптимальной последовательности.

Поиск решения в пространстве состояний представляют в виде ***графа состояний.*** Множество вершин графа соответствует состояниям задачи, а множество дуг (ребер) — операторам. Тогда решение задачи может интерпретироваться как поиск пути на графе из исходного состояния задачи в целевое состояние.

Формулировка задачи поиска в пространстве состояний требует определения всех составляющих выражения (2.1):

- **пространство состояний (state space)** — множество всех состояний, которые возможны мире агента;
- **начальное состояние (start state)** — состояние, в котором агент существует изначально;
- **функция-преемника** (**successor function**) — функция, которая принимает на вход состояние и действие, вычисляет стоимость выполнения этого действия и определяет состояние-преемник (**successor**), в котором находился бы агент, если бы выполнил это действие.
- **целевогой тест** — **функция (goal test)**, которая принимает на вход состояние и определяет, является ли оно целевым состоянием.

По сути, проблема поиска решается следующим образом: сначала рассматривается начальное состояние, затем исследуется пространство состояний с помощью

функции-преемника, которая итеративно вычисляет преемников различных состояний, пока мы не достигнем целевого состояния, после чего определяется путь из начального состояния в целевое состояние (обычно его называется планом).

Процесс применения **функций-приемников** к некоторой вершине графа с целью получения всех ее дочерних вершин (приемников) называется ***раскрытием вершины*.**

2.2.4. Формулировки задач поиска в среде AI Pacman

Рассмотрим вариант игры, в которой имеется лабиринт, Pacman и пищевые гранулы. В этом случае можно сформулировать две отдельные задачи поиска: нахождение пути в лабиринте и поедание всех гранул. В первом случае мы будем пытаться решить проблему оптимального перехода из позиции (x1; y1) в позицию (x2; y2) в лабиринте, в то время как при решении второй задачи наша цель будет заключаться в поедании всех пищевых гранул в кратчайшие сроки. Ниже для этих задач определяются: состояния, действия, функция-преемник и целевой тест.

Нахождение пути: - *состояния*: координаты (x, y) местоположений; - *действия*: Север, Юг, Восток, Запад; - *преемник*: обновить только местоположение; - *проверка цели*: (x, y) = END?	**Поедание всех гранул**: - *состояния*: координаты (x, y) местоположений, логические значения точек расположения гранул (true/false); - *действия*: Север, Юг, Восток, Запад; - *преемник*: обновить местоположение и логические значения точек; - *проверка цели*: все ли логические значения точек ложны?

Обратите внимание, что для задачи нахождения пути состояния содержат меньше информации, чем состояния для задачи поедания всех гранул. Так как во втором случае мы должны дополнительно поддерживать массив логических значений, соответствующих каждой грануле, независимо от того, была ли она съедена в данном состоянии или нет. **Состояние мира (world state) агента** может содержать дополнительную информацию, например, информацию о расстоянии, пройденным агентом, или информацию о всех позициях, посещенных агентом, кроме текущего (x, y) местоположения агента и логических значений точек.

2.2.5. Методы поиска решений задач в пространстве состояний

Методы поиска в пространстве состояний подразделяются на две группы: методы неинформированного (слепого) и информированного (упорядоченного, эвристического) поиска. В методах “слепого” поиска выполняется полный просмотр всего пространства состояний, что может приводить к проблеме *комбинаторного взрыва*. К методам слепого поиска относят: поиск в ширину, поиск на основе алгоритма равных цен, различные виды поиска в глубину. Стратегии поиска в пространстве состояний обычно сравнивают с помощью критериев, указанных ниже [4].

Полнота. Гарантируется ли обнаружение решения, если оно существует?

Оптимальность. Стратегию называют оптимальной, если она обеспечивает нахождение решения, которое не обязательно будет наилучшим, но известно, что оно принадлежит подмножеству решений, обладающих некоторыми заданными свойствами, согласно которым мы относим их к оптимальным.

Минимальность. Стратегию называют минимальной, если она гарантирует нахождение наилучшего решения, т.е. минимальность является более сильным случаем оптимальности.

Временная сложность. Время (число операций), необходимое для нахождения решения.

Пространственная сложность. Объем памяти, необходимый для решения задачи.

Рассматриваемые ниже методы поиска используют два списка: список открытых вершин и список закрытых вершин. В *списке открытых вершин* **OPEN** (часто в программах такой список именуют как **FRINGE** или **FRONTIER**) находятся вершины, подлежащие раскрытию; *в списке закрытых вершин* (**CLOSED**) находятся уже раскрытые вершины. Список **CLOSED** позволяет запоминать рассмотренные вершины с целью исключения их повторного раскрытия.

2.2.6. Методы неинформированного (слепого) поиска

Поиск в ширину

Рассмотрим алгоритм поиска в ширину (**BFS- Breadth First Search).** В начале поиска список **CLOSED** пустой, а **OPEN** содержит только начальную вершину. На каждой итерации из списка **OPEN** выбирается для раскрытия первая вершина. Если эта вершина не целевая, то она перемещается в список **CLOSED**, а ее дочерние вершины помещаются в ***конец*** списка **OPEN**, т.е. принцип формирования списка **OPEN** соответствует ***очереди***. Далее процесс повторяется.

Согласно этой стратегии, при построении дерева поиска вершины с глубиной ***k*** раскрываются только после того, как будут раскрыты все вершины предыдущего уровня ***k-1***. В этом случае фронт поиска (fringe, frontier) растет в ширину. Для построения обратного пути (из целевой вершины в начальную вершину) все дочерние вершины снабжаются указателями на соответствующие родительские вершины.

Ниже приведен псевдокод алгоритма поиска в ширину на графе состояний. Функция **pop()** извлекает из списка (в данном случае очереди) **OPEN** первую вершину, функция **push()** выполняет вставку вершины в конец очереди. Вызов функции **problem.getStartState()** обеспечивает получение стартовой вершины (состояния) решаемой задачи **problem**. Параметр **problem** содержит компьютерное представление описания задачи в соответствии с формулой (2.1).

```
def  breadthFirstSearch (problem):
   Определить стартовую вершину: start = problem.getStartState()
   Поместить стартовую вершину в список OPEN: OPEN.push(start)
   CLOSED = [ ]
   Путь = [ ]
```

```
while not OPEN.isEmpty() :
        node = OPEN.pop()
        If node == 'целевая вершина': return Путь
        CLOSED = CLOSED.append(node)
        Раскрыть node и поместить все дочерние вершины, отсутствующие в
        списке CLOSED или OPEN, в конец списка OPEN, связав с каждой до-
        черней вершиной указатель на node
return 'НЕУДАЧА'
```

Если целевая вершина найдена, то алгоритм должен вернуть **Путь** от стартовой вершины до целевой. Значение переменной **Путь** может быть определено на основе дополнительного анализа списка **CLOSED** либо, в простых случаях, путь к каждой вершине накапливается и хранится в самой вершине.

Если повторяющиеся вершины не исключаются из рассмотрения (т.е. не выполняется проверка принадлежности дочерних вершин списку **CLOSED** или **OPEN**), то алгоритм строит не граф поиска, а **дерево поиска**! В этом случае можно легко оценить временную и пространственную сложности алгоритма.

Если каждая вершина дерева поиска имеет B дочерних вершин, то при остановке поиска на глубине, равной d, максимальное число раскрытых вершин будет равно $B+B^2+B^3+\ldots+B^d+(B^{d+1}-B)$ [4]. Обычно вместо этой формулы употребляют её обозначение $O(B^{d+1})$, которое называют ***экспоненциальной оценкой сложности***. Если полагать, что раскрытие каждой дочерней вершины требует одной единицы времени, то $O(B^{d+1})$ является ***оценкой временной сложности***. При этом каждая вершина должна сохраняться в памяти до получения решения. Поэтому ***оценка пространственной сложности*** будет также равна $O(B^{d+1})$. Это приводит к тому, что стратегия поиска в ширину может использоваться только для задач, которые характеризуются пространством поиска небольшой размерности. Но при этом она удовлетворяет ***критерию полноты и минимальности*** (обеспечивает нахождение целевого состояния, которое находится на минимальной глубине дерева поиска, т.е. поиск в ширину обеспечивает нахождение ***самого «поверхностного» решения***).

Поиск по критерию стоимости (алгоритм равных цен)

Поиск в ширину является оптимальным, если стоимости раскрытия всех вершин равны. Если с каждой вершиной связать стоимость её раскрытия и на каждом шаге из списка **OPEN** выбирать для ракрытия вершину с наименьшей стоимостью, то рассмотренная выше процедура будет обеспечивать нахождение **оптимального** решения по критерию стоимости. Стоимость раскрытия $g(V_i)$ некоторой вершины V_i равна

$$g(V_i)=g(V)+c(V,V_i), \qquad (2.2)$$

где V — родительская вершина для вершины V_i; $c(V,V_i)$ — стоимость пути из вершины V в вершину V_i; $g(V)$ — стоимость раскрытия родительской вершины (для начальной вершины $g(V)=0$).

Рассмотренная стратегия гарантирует **полноту** поиска, если стоимость каждого участка пути положительная величина. Так как поиск в этом случае направляется стоимостью путей, то ***временная и пространственная сложности*** (при построении дерева поиска) в наихудшем случае будут пропорциональны $O(B^{1+C/c})$, где C — стоимость оптимального решения, c — средняя минимальная стоимость действия. Эта оценка может быть больше $O(B^d)$. Это связано с тем, что процедура поиска по критерию стоимости часто обследует поддеревья поиска, состоящие из мелких участков небольшой стоимости, прежде чем перейти к исследованию путей, в которые входят крупные, но возможно более полезные участки [4].

Поиск в глубину

При поиске в глубину всегда раскрывается самая глубокая вершина из текущего фронта поиска. Процедура поиска в глубину отличается от процедуры поиска в ширину тем, что дочерние вершины, получаемые при раскрытии вершины **node**, помещаются в **начало** списка **OPEN,** т.е. принцип формирования списка **OPEN** соответствует **стеку**.

Поиск в глубину требует хранения только одного пути от корня до листового узла. Для дерева поиска с коэффициентом ветвления B и максимальной глубиной m поиск в глубину требует хранения $Bm+1$ узлов, т.е ***пространственная сложность*** соответствует $O(Bm)$, что намного меньше по сравнению с рассмотренными выше стратегиями поиска [4]. При **поиске в глубину с возвратами** потребуется еще меньше памяти. В этом случае каждый раз формируется только одна из дочерних вершин и запоминается информация о том, какая вершина должна быть сформирована следующей. Таким образом, требуется только $O(m)$ ячеек памяти.

Однако поиск в глубину ***не является полным*** (в случае не ограниченной глубины) и ***оптимальным*** (не обеспечивает гарантированное нахождение наиболее поверхностного целевого узла). В наихудшем случае ***временная сложность*** пропорциональна $O(B^m)$, где m — максимальная глубина залегания решения (m может быть гораздо больше по сравнению с d — глубиной самого поверхностного решения).

Проблему деревьев поиска неограниченной глубины можно решить, предусматривая применение во время поиска заранее определенного ***предела глубины L.*** Это означает, что вершины на глубине L рассматриваются таким образом, как если бы они не имели дочерних вершин. Такая стратегия поиска называется **поиском с ограничением глубины**. Однако при этом вводится дополнительный источник **неполноты**, если будет выбрано $L<d$, т.е. самая поверхностная цель находится за пределами глубины. А при выборе $L>d$ поиск с ограничением глубины будет **неоптимальным** (не гарантируется получение самого поверхностного решения).

Поиск в глубину с итеративным углублением. Эта стратегия поиска позволяет найти наилучший предел глубины. Для этого применяется процедура поиска с ограничением по глубине. При этом предел глубины постепенно увеличивается (в начале он равен 0, затем 1, затем 2 и т.д.) до тех пор пока не будет найдена цель. Это происходит, когда предел глубины достигает значения d — глубины са-

мого поверхностного решения. Конечно, здесь допускается повторное обследование одних и тех же состояний. Однако такие повторные операции не являются слишком дорогостоящими, и временная сложность оценивается значением $O(B^d)$ [3].

В поиске с итеративным углублением (по дереву поиска) сочетаются преимущества поиска в ширину (является **полным**) и поиска в глубину (малое значение **пространственной сложности**, равное $O(Bd)$).

2.2.7 Общие сведения о программировании в среде AI Pacman

Код среды AI Pacman, используемый в данной лабораторной работе, находится в нескольких файлах. Некоторые из файлов необходимо будет прочитать и понять, чтобы выполнить задание, а некоторые из них можно игнорировать. Весь необходимый код и вспомогательные файлы лабораторной работы находятся в архиве **МиСИИ_лаб2_3_2024.zip:**

Файлы для редактирования:	
search.py	Здесь будут размещаться все алгоритмы поиска, которые Вы запрограммируете.
searchAgents.py	Здесь будут находиться все Ваши поисковые агенты.
Файлы, которые необходимо просмотреть	
pacman.py	Основной файл, из которого запускают Pacman. Этот файл описывает тип Pacman GameState, который используется в лабораторных работах.
game.py	Логика, лежащая в основе мира Pacman. Этот файл описывает несколько поддерживающих типов, таких как AgentState, Agent, Direction и Grid.
util.py	Полезные структуры данных для реализации алгоритмов поиска.
Поддерживающие файлы, которые можно игнорировать:	
graphicsDisplay.py	Графика Pacman
graphicsUtils.py	Графические утилиты
textDisplay.py	ASCII графика Pacman
ghostAgents.py	Агенты, управляющие привидениями
keyboardAgents.py	Интерфейс клавиатуры для управления игрой
layout.py	Код для чтения файлов схем и хранения их содержимого
autograder.py	Автооцениватель
testParser.py	Парсер тестов автооценивателя и файлы решений
testClasses.py	Общие классы автооценивателя
test_cases/	Папка, содержащая тесты для каждого из заданий (вопросов)
searchTestClasses.py	Специальные тестовые классы автооценивателя для данной лабораторной работы

Во время выполнения данной лабораторной работы необходимо будет дописать код файлов **search.py** и **searchAgents.py**. Пожалуйста, не изменяйте другие файлы в дистрибутиве лабораторной работы.

После загрузки кода (**МиСИИ_лаб2_3_2024.zip**), его распаковки и перехода в соответствующий каталог вы сможете играть в Pacman, набрав в командной строке Python следующее (если используется среда IPython, то набор команды начинается со знака “**!**”):

```
python pacman.py
```

Pacman живет в мире разветвленных лабиринтов и пищевых гранул. Эффективная навигация в этом мире будет первым шагом Pacman в освоении свой области обитания. Самый простой агент, находящийся в **searchAgents.py** и называемый **GoWestAgent**, всегда идет на запад (тривиальный рефлекторный агент). Этот агент может иногда выигрывать:

```
python pacman.py --layout testMaze --pacman GoWestAgent
================
Pacman emerges victorious! Score: 503
Average Score: 503.0
Scores:       503.0
Win Rate:     1/1 (1.00)
Record:       Win
```

Здесь параметр **--layout**, определяют сложность лабиринта, а параметр **-- pacman** – тип используемого агента. Для агента **GoWestAgent** всё становится плохо, когда требуется поворот: он упирается в стену и останавливается. Например,

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

Если Pacman застрял, вы можете выйти из игры, нажав **CTRL+c** на своем терминале. Необходимо будет научить агента преодолевать не только лабиринт **tinyMaze**, но и любой лабиринт по вашему желанию.

Обратите внимание, что **pacman.py** поддерживает ряд параметров, каждый из которых может быть задан длинным (например, **--layout**) или коротким (например, **-l**) способом. Можно просмотреть список всех параметров вызова Pacman и их значения по умолчанию, выполнив команду:

```
python pacman.py –h
```

Все команды вызова Pacman, которые используются в этой лабораторной работе и рассматриваются далее в заданиях, также содержатся в файле **commands.txt** для облегчения копирования и вставки.

2.3. Задания для выполнения

Задание 1. Поиск в глубину

Воспользуемся поисковым агентом **SearchAgent**, реализованным в **searchAgents.py**, который планирует путь в лабиринте, а затем выполняет его пошагово. Ваша задача состоит в том, чтобы реализовать алгоритмы поиска пути для этого агента.

Сначала проверьте правильность работы **SearchAgent,** выполнив команду:

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

Приведенная выше команда указывает **SearchAgent** использовать **tinyMazeSearch** в качестве алгоритма поиска, который реализован в **search.py.** Pacman должен успешно перемещаться по лабиринту.

Вам необходимо написать полноценные универсальные функции поиска, которые помогут Pacman планировать маршруты. Помните, что поисковый узел должен содержать не только состояние, но и информацию, необходимую для восстановления пути (плана), который приводит в это состояние.

Указание 1: функции поиска должны возвращать список действий, которые приведут агента из стартового состояния в целевое состояние. Все эти действия должны быть разрешенными (должны использоваться допустимые направления, запрет на перемещение через стены).

Указание 2: обязательно используйте структуры данных **Stack, Queue** и **PriorityQueue**, предоставленные в **util.py**. У реализаций этих структур данных есть определенные свойства, которые требуются для совместимости с автооценивателем.

Подсказка: все алгоритмы поиска очень похожи. Алгоритмы для DFS, BFS, UCS отличаются только деталями управления списком **OPEN**. Сконцентрируйтесь на правильном написании кода алгоритма DFS, а остальное должно быть относительно простым. Действительно вместо DFS, BFS, UCS можно реализовать только один обобщенный метод поиска, который настраивается стратегией организации порядка обработки списка **OPEN**, зависящей от алгоритма. (Ваша реализация не обязательно должна быть в этой обобщенной форме).

Реализуйте алгоритм поиска в глубину (DFS) в функции **depthFirstSearch** в файле **search.py.** Напишите версию DFS для поиска пути на графе, которая избегает раскрытия любых уже посещенных состояний. Псевдокод функции **depthFirstSearch** аналогичен псевдокоду функции **breadthFirstSearch**, который был рассмотрен в разделе 2.2.6. При этом список **OPEN** представляет собой не очередь, а стек. Для создания пустого стека используйте вызов **OPEN =util.Stack().**

Написанный код **depthFirstSearch** должен быстро найти решение для следующих лабиринтов:

python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent

Схема лабиринта Pacman (рисунок 2.2) показывает наложение обследованных состояний и порядок, в котором эти состояния обходятся агентом (более яркий красный означает более ранний обход). Ответьте на два вопроса: Вы ожидали такой порядок обхода? На самом ли деле Pacman проходит все обследованные позиции на пути к цели?

Подсказка: если вы используете стек в качестве структуры данных, то решение, найденное вашим алгоритмом DFS для **mediumMaze**, должно иметь длину 130

(при условии, что вставка преемников в **OPEN** выполняется в порядке, формируемом методом **getSuccessors**; вы можете получить длину 246, если будете использовать обратный порядок). Ответьте на два вопроса: Это решение наименьшей стоимости? Если нет, подумайте, что не так с поиском в глубину.

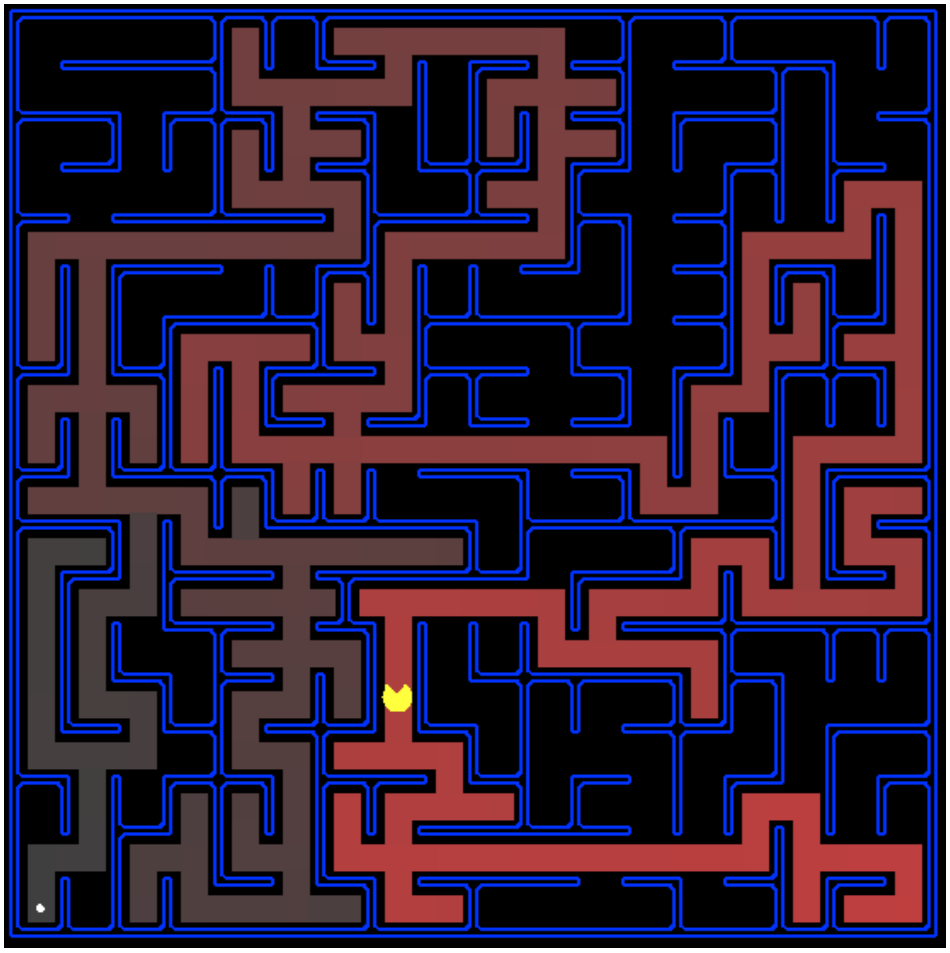

Рисунок 2.2 – Схема большого лабиринта (bigMaze)

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тестовые примеры автооценивателя:

python autograder.py -q q1

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценивателя внесите результаты в отчет

.

Задание 2. Поиск в ширину

Реализуйте алгоритм поиска в ширину (BFS) в функции **breadthFirstSearch** в файле **search.py.** Напишите алгоритм поиска пути на графе, который избегает повторного раскрытия вершин. Проверьте свой код так же, как и для поиска в глубину:

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

Ответьте на вопрос: найдет ли BFS решение с наименьшими затратами? Если нет, проверьте свою реализацию.

Подсказка: если Pacman движется слишком медленно, то попробуйте опцию **--frameTime 0**.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тестовые примеры автооценивателя:

python autograder.py -q q2

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценивателя внесите результаты в отчет

Задание 3. Поиск на основе алгоритма равных цен

Хотя BFS найдет путь к цели с наименьшим количеством действий, можно попытаться найти путь, который является «лучшим» с точкт зрения иных критериев. Рассмотрим лабиринты **mediumDottedMaze** и **mediumScaryMaze**.

Изменяя функцию стоимости, мы можем побудить Pacman находить разные пути. Например, мы можем назначать большую цену за опасные шаги в областях, рядом с призраками, или меньшую цену за шаги в областях, богатых едой, и рациональный агент Pacman должен корректировать свое поведение.

Реализуйте алгоритм равных цен для поиска пути на графе в функции **uniformCostSearch** в файле **search.py**. Рекомендуется просмотреть файл **util.py** для ознакомления с некоторыми структурами данных, которые могут быть полезны.

Теперь вы должны наблюдать успешное поведение агента во всех трех лабиринтах, где все агенты являются агентами, функционирующими на основе алгоритма UCS, которые отличаются только используемой функцией стоимости (агенты и функции стоимости уже написаны):

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

Примечание. Вы должны получить очень низкие и очень высокие стоимости путей для агентов **StayEastSearchAgent** и **StayWestSearchAgent**, соответственно, из-за их функций экспоненциальной стоимости (подробности см. в **searchAgents.py**).

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тестовые примеры автооценивателя:

python autograder.py -q q3

Если возникают ошибки, то исправьте их. После успешного прохождения тестов автооценивателя внесите результаты в отчет

2.4. Порядок выполнения лабораторной работы

2.4.1. Изучить по лекционному материалу и учебным пособиям [1-3] методы слепого поиска решений задач в пространстве состояний. Если Вы еще не распаковали архив **МиСИИ_лаб2_3_2024.zip** с файлами лабораторной работы, то разаривируйте его в отдельную папку.

2.4.2. Изучить структуры данных **Stack, Queue** и **PriorityQueue**, предоставленные в модуле **util.py**.

2.4.3. Изучить методы среды AI Pacman: **problem.getStartState()** , **problem.isGoalState(), problem.getSuccessors().** Для этого проверить выполнение команды

```
python pacman.py -l tinyMaze -p SearchAgent
```

для случая, когда реализация функции **depthFirstSearch** содержит только вызовы операторов печати

```
print("Start:", problem.getStartState())
print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
print("Start's successors:", problem.getSuccessors(problem.getStartState()))
```

Разобраться с типами значений, возвращаемых методами и использовать их при написании кода, реализующего алгоритмы поиска DFS, BFS, UCS. Обратите внимание, что состояние-приемник, возвращаемое методом **getSuccessors()**, представляет собой кортеж **((x,y), действие, стоимость),** содержащий: **(x,y)** – координаты узла-приемника; **действие** – команда (‘East’, 'West’, 'North’, 'South’), обеспечивающая переход в состояние приемник; **стоимость** - стоимость перехода в состояние-приемник.

2.4.4. Определить в соответствии с заданиями 1-3 раздела 2.3 функции, реализующие алгоритмы поиска DFS, BFS, UCS. При реализации алгоритмов поиска рекомендуется использовать псевдокод из раздела 2.2.6. Для реализации списка **OPEN** в алгоритме UCS следует использовать очередь с приоритетами **PriorityQueue**. Рекомендуется для редактирования функций и выполнения кода использовать интегрированную среду (IDE). Следует выполнять вставку кода, определяемых функций, после строки:

```
"*** ВСТАВЬТЕ ВАШ КОД СЮДА***"
```

Никакие другие функции архива **МиСИИ_лаб2_3_2024.zip** не менять.

2.4.5. Зафиксировать результаты использования функций для всех лабиринтов, указанных в заданиях 1-3. Ответить письменно на предлагаемые в заданиях 1-3 вопросы.

2.4.6. Выполнить с помощью **autograder.py** автооценивание заданий 1-3. При обнаружении ошибок отредактировать код. Результаты автооценивания внести в отчет.

2.4.7. Оценить эффективность используемых методов поиска по критериям временной и пространственной сложности.

2.5. Содержание отчета

Цель работы, краткий обзор методов слепого поиска решений задач в пространстве состояний, описание представления задачи в AI Pacman (описание состояний, операторов, начального и конечного состояний, критериев достижения цели), представление пространства состояний в виде графа, тексты реализованных функций с комментариями, полученные результаты для разных лабиринтов и их анализ, результаты автооценивания, выводы по проведенным экспериментам с разными алгоритмами слепого поиска и разными лабиринтами.

2.6. Контрольные вопросы

2.6.1. Назовите основные способы представления задач в ИИ?

2.6.2. Определите состояния, операторы преобразования состояний, функции стоимости для следующих задач:

а) задача о коммивояжере;

б) задача о миссионерах и каннибалах;

в) задача о раскраске карт.

2.6.3. Сформулируйте для задачи поиска пути и задачи поедания всех пищевых гранул в AI Pacman, что представляют собой состояния, действия, функция-приемник, функция проверки цели.

2.6.4. Для задачи о двух кувшинах емкостью 5 литров и 2 литра, построить дерево поиска, если требуется налить во второй кувшин ровно 1 литр воды.

2.6.5. Напишите на псевдоязыке процедуры поиска в ширину и глубину, объясните их различие с алгоритмической точки зрения.

2.6.6. Напишите на псевдоязыке процедуру поиска в соответствии с алгоритмом равных цен.

2.6.7. Объясните назначение функций **problem.getStartState()**, **problem.isGoalState(), problem.getSuccessors()** среды AI Pacman и приведите примеры значений, возвращаемых этими функциями.

2.6.8. Напишите на языке Python функцию, реализующую поиск в глубину применительно к среде AI Pacman.

2.6.9. Сформулируйте принципы поиска, используемые в алгоритмах поиска в глубину: с возвратом, с ограничением глубины и с итеративным углублением.

2.6.10. Определите критерии полноты, оптимальности, минимальности, пространственной и временной сложности.

2.6.11. Сравните основные алгоритмы слепого поиска в пространстве состояний по критериям полноты, оптимальности, пространственной и временной сложности.