

### 3. ЛАБОРАТОРНАЯ РАБОТА № 3

## «ИССЛЕДОВАНИЕ ИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»

#### **3.1. Цель работы**

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

#### **3.2. Краткие теоретические сведения**

##### **3.2.1 Общая характеристика методов информированного поиска**

Основная идея таких методов состоит в **использовании дополнительной информации для ускорения процесса поиска**. Эта дополнительная информация формируется на основе эмпирического опыта, догадок и интуиции исследователя, т.е. **эвристик**. Использование эвристик позволяет **сократить количество просматриваемых вариантов при поиске решения** задачи, что ведет к более быстрому достижению цели.

В алгоритмах эвристического поиска **список открытых вершин упорядочивается** по возрастанию некоторой **оценочной функцией**, формируемой **на основе эвристических правил**. Оценочная функция может включать **две составляющие**, одна из которых называется **эвристической** и характеризует близость текущей и целевой **вершин**. Чем меньше значение эвристической составляющей оценочной функции, тем “ближе” рассматриваемая вершина к целевой вершине. В зависимости от **способа формирования оценочной функции** выделяют следующие **алгоритмы эвристического поиска**: алгоритм “подъема на гору”, алгоритм глобального учета соответствия цели, А-алгоритм [1-3]. Наиболее общим является А-алгоритм.

##### **3.2.2. А - алгоритм**

А-алгоритм похож на алгоритм равных цен, но в отличие от него **учитывает при раскрытии вершин, как уже сделанные затраты, так и предстоящие затраты**. В этом случае **оценочная функция** имеет вид:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

где  $\hat{g}(n)$  – оценка стоимости пути из начальной вершины в вершину  $n$ , которая вычисляется в соответствии с (2.2);  $\hat{h}(n)$  – **эвристическая оценка стоимости кратчайшего пути из вершины  $n$  в целевую вершину** (предстоящие затраты). Чем меньше значение  $\hat{h}(n)$ , тем перспективнее путь, на котором находится вершина  $n$ . В ходе поиска **раскрываются вершины с минимальным значением оценочной функции**  $\hat{f}(n)$ .

Готовых рецептов в отношении построения эвристической составляющей оценочной функции не существует. При решении каждой конкретной задачи используется ранее накопленный опыт решения подобных задач. Например, для игры в восемь это может быть количество фишек, которые находятся не на своем месте или сумма расстояний каждой фишке от текущего до целевого расположения.

Схема A\*-алгоритма соответствует алгоритму равных цен. При этом оценки  $\hat{f}(n)$  могут менять свои значения в процессе поиска. Это приводит к тому, что вершины из списка CLOSED могут перемещаться обратно в список OPEN. Ниже приведен псевдокод функции, выполняющей поиск на графе в соответствии с A-алгоритмом [1]:

```
def aStarSearch (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.append(node)
        Раскрыть node и для всех дочерних вершин  $n_i$  вычислить оценку
        
$$\hat{f}(n, n_i) = \hat{g}(n, n_i) + \hat{h}(n_i)$$

        Поместить все дочерние вершины, отсутствующие в списке CLOSED
        или OPEN, в список OPEN, связав с каждой дочерней вершиной указатель на node
        Для дочерних вершин  $n_i$ , которые уже содержатся в OPEN, сравнить
        оценки  $\hat{f}(n, n_i)$  и  $\hat{f}(n_i)$ , если  $\hat{f}(n, n_i) < \hat{f}(n_i)$ , то связать с вершиной  $n_i$ 
        новую оценку  $\hat{f}(n, n_i)$  и указатель на вершину node
        Если вершина  $n_i$  содержится в списке CLOSED и  $\hat{f}(n, n_i) < \hat{f}(n_i)$ , то
        связать с вершиной  $n_i$  новую оценку  $\hat{f}(n, n_i)$ , переместить её в список
        OPEN и установить указатель на node;
        Упорядочить список OPEN по возрастанию  $\hat{f}(n_i)$ ;
    return 'НЕУДАЧА'
```

### 3.2.3. Свойства А-алгоритма

Свойства А-алгоритма существенно зависят от условий, которым удовлетворяет или не удовлетворяет эвристическая часть оценочной функции  $\hat{f}(n)$  [1]:

- 1) А-алгоритм соответствует алгоритму равных цен, если  $h(n)=0$ ;

- 2) А-алгоритм **гарантирует оптимальное решение**, если  $\hat{h}(n) \leq h(n)$ ; в этом случае он называется **A\* – алгоритмом**. A\*-алгоритм недооценивает затраты на пути из промежуточной вершины в целевую вершину или оценивает их правильно, но никогда не переоценивает;
- 3) А-алгоритм обеспечивает однократное раскрытие вершин, если выполняется **условие монотонности**  $\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j)$ , где  $n_i$  – родительская вершина;  $n_j$  – дочерняя вершина;  $c(n_i, n_j)$  – стоимость пути между вершинами  $n_i$  и  $n_j$ ;
- 4) алгоритм  $A_1^*$  является эвристически более сильным, чем алгоритм  $A_2^*$  при условии  $\hat{h}_1(n) > \hat{h}_2(n)$ . Эвристически более сильный алгоритм  $A_1^*$  в большей степени сокращает пространство поиска;
- 5) A\*-алгоритм **полностью информирован**, если  $\hat{h}(n) = h(n)$ . В этом случае никакого поиска не происходит и приближение к цели идет по оптимальному пути;
- 6) при  $\hat{h}(n) > h(n)$  А-алгоритм **не гарантирует получение оптимального решения**, однако часто решение получается быстро.

Эффективность поиска с помощью A\*-алгоритма может снижаться из-за того, что вершина, находящаяся в списке **CLOSED**, может попадать обратно в список **OPEN** и затем повторно раскрываться.

Чтобы A\*-алгоритм не раскрывал несколько раз одну и ту же вершину необходимо, чтобы выполнялось **условие монотонности**

$$\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j),$$

где  $n_i$  – родительская вершина;  $n_j$  – дочерняя вершина;  $c(n_i, n_j)$  – стоимость пути между вершинами  $n_i$  и  $n_j$ . Монотонность предполагает, что не только не переоценивается стоимость  $\hat{h}(n)$  оставшегося пути из  $n$  до цели, но и не переоцениваются стоимости ребер между двумя соседними вершинами.

Условие монотонности поглощает условие гарантированности (допустимости). Если условие монотонности соблюдается для всех дочерних вершин, то можно доказать, что в тот момент, когда раскрывается некоторая вершина  $n$ , оптимальный путь к ней уже найден. Следовательно, оценочная функция для данной вершины в дальнейшем не меняет своих значений, и никакие вершины из списка CLOSED в список OPEN не возвращаются.

Если соблюдается условие монотонности, то значения оценочной функции  $f(n)$  вдоль любого пути являются **неубывающими**. Тот факт, что стоимости вдоль любого пути являются неубывающими, означает что в пространстве состояний могут быть очерчены **контуры равных стоимостей**. Поэтому A\*-алгоритм проверяет все узлы в контуре меньшей стоимости, прежде чем перейдет к проверке узлов следующего контура.

**A\*-алгоритм** (при выполнении условия монотонности) является **полным, оптимальным и оптимально эффективным** (не гарантируется развертывание меньшего кол-ва узлов с помощью любого иного оптимального алгоритма).

**Временная сложность A\*-алгоритма** по-прежнему остается **экспоненциальной**, т.е количество раскрываемых узлов в пределах целевого контура пространства состояний все еще зависит экспоненциально от глубины решения. По этой причине на практике стремление находить оптимальное решение не всегда оправдано. Иногда вместо этого целесообразно использовать варианты A-поиска, позволяющие быстро находить квазиоптимальные решения, или разрабатывать более точные эвристические функции, но не строго допустимые.

Так как при A\*-поиске хранятся все раскрытия узлы, фактические **ресурсы пространства** исчерпаются гораздо раньше, чем временные ресурсы. С этой целью применяют **A\*-алгоритм с итеративным углублением (IDA\*)**. Применимым условием остановки здесь служит f-стоимость, а не глубина. Преодолеть проблему пространственной сложности за счет небольшого увеличения времени выполнения позволяют также алгоритмы **RBFS** и **MA\*** [4].

### 3.3. Задания для выполнения

#### Задание 1. A\*- поиск

Реализуйте A\*-алгоритм на графе состояний, используя шаблон функции **aStarSearch** в файле **search.py**. Функция **aStarSearch** принимает в качестве аргумента эвристическую функцию. Эвристическая функция имеет два аргумента: **состояние агента** (основной аргумент) и **задача** (**problem**) (для справочной информации). Функция **nullHeuristic** в **search.py** является тривиальным примером эвристической функции.

Протестируйте свою реализацию A\*-поиска на задаче поиска пути в лабиринте, используя **эвристику манхэттенского расстояния** (уже реализованную как **manhattanHeuristic** в **searchAgents.py**):

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

Вы должны увидеть, что A\*- алгоритм находит оптимальное решение **немного быстрее**, чем поиск в соответствии с алгоритмом равных цен (он раскрывает около 549 узлов по сравнению с 620 узлами, из-за учета приоритета узлов числа могут немного отличаться).

Проверьте результаты поиска на лабиринте **openMaze**. Что можно сказать о различных стратегиях поиска?

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация A\*-поиска все тестовые примеры автооценивателя:

```
python autograder.py -q q4
```

#### Задание 2. Поиск всех углов

Настоящая сила А\*-поиска станет очевидной только при решении более сложной задачи. Сформулируем новую проблему и разработаем эвристику для ее решения.

В углах лабиринта есть четыре точки, по одной в каждом углу. Необходимо найти кратчайший путь, который связан с посещением всех четырех углов (независимо от того, есть ли в лабиринте еда или нет). Обратите внимание, что для некоторых лабиринтов, таких как **tinyCorners**, кратчайший путь не всегда ведет к ближайшей грануле в первую очередь!

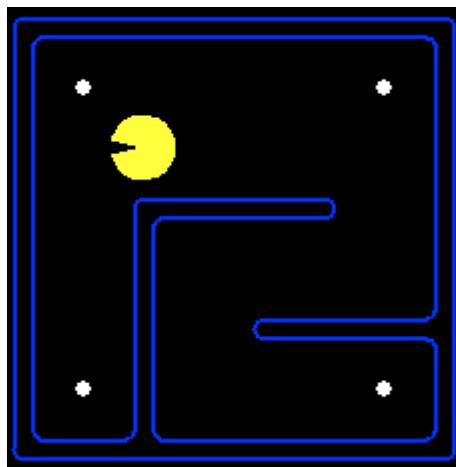


Рисунок 2.3. – Задача поиска углов

**Подсказка:** кратчайший путь через **tinyCorners** состоит из 28 шагов.

**Примечание.** Обязательно выполните задание 2 предыдущей лабораторной работы, прежде чем решать это задание

Реализуйте задачу поиска углов, дописав участки кода в определении класса **CornersProblem** в файле **searchAgents.py**. Вам нужно будет выбрать такое представление состояния, которое кодирует всю информацию, необходимую для определения достижения цели: посетил ли агент все четыре угла.

Протестируйте поискового агента, выполнив команды:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Чтобы получить высокую оценку за выполнение задания, вам необходимо определить абстрактное представление состояния, которое не содержит несущественную информацию (например, положение призраков, где находится дополнительная еда и т. п.). В частности, не используйте Pacman **GameState** в качестве состояния поиска. Ваш код будет очень медленным.

**Подсказка 1.** Единственные части игрового состояния, на которые вам нужно ссылаться в своей реализации, – это начальная позиция Pacman и расположение четырех углов.

**Подсказка 2:** при написании кода **getSuccessors** не забудьте добавить потомков в список преемников со стоимостью 1.

Наша реализация **breadthFirstSearch** расширяет почти 2000 поисковых узлов на задаче **mediumCorners**. Однако эвристика (используемая в A\*-поиске) может уменьшить этот объем.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация агента, выполняющего поиск углов, все тесты автооценивателя:

```
python autograder.py -q q5
```

### Задание 3. Эвристика для задачи поиска углов

Реализуйте нетривиальную монотонную эвристику для задачи поиска углов в методе **cornersHeuristic** класса **CornersProblem**. Проверьте реализацию, выполнив команду:

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Здесь **AStarCornersAgent** - это сокращение (ярлык) для

**-p SearchAgent -a fn = aStarSearch, prob = CornersProblem, heuristic = cornersHeuristic**

**Гарантированность (admissibility) против монотонности (consistency)**: помните, эвристики - это просто функции, которые принимают состояния поиска и возвращают числа, которые дают прогноз стоимости достижения ближайшей цели. Более эффективная эвристика вернет значения, близкие к фактическим затратам. Чтобы быть **гарантирующими** (допустимыми), эвристические значения должны быть **нижними границами фактических затрат** кратчайшего пути к **ближайшей цели** (и неотрицательными). Чтобы быть **монотонными** (согласованными), они должны дополнительно обеспечивать выполнение условия: **если действие имеет стоимость  $c$ , то выполнение этого действия может вызвать только снижение значения эвристики не более чем на  $c$ .**

Помните, что гарантированности (допустимости) недостаточно, чтобы обеспечить правильность поиска по графу - **вам нужно более строгое условие монотонности**. Однако допустимые эвристики часто также монотонны (согласованы). Поэтому обычно проще всего начать с мозгового штурма допустимых эвристик. Если у вас есть допустимая эвристика, которая хорошо работает, вы также можете проверить, действительно ли она согласована. Единственный способ проверить согласованность - это предъявить доказательство. Однако несогласованность часто можно обнаружить, убедившись, что для каждого расширяемого узла его последующие узлы имеют равные или большие значения  $f$ . Кроме того, если алгоритмы UCS и A\* когда-либо возвращают пути разной длины, ваша эвристика не согласована.

**Тривиальные эвристики** - это те, которые возвращают ноль **везде** (UCS), и **эвристики, которые вычисляют истинную стоимость**. Первая не снизит временную сложность, а вторая приведет к таймауту автооценивателя. Вам нужна **нетривиальная эвристика**, которая **сокращает общее время вычислений**, хотя для этого задания

автооцениватель будет проверять только количество узлов (помимо обеспечения разумного ограничения по времени).

**Оценивание:** ваша эвристика должна быть нетривиальной, неотрицательной и согласованной, чтобы получить какие-либо баллы за выполнение задания. Убедитесь, что ваша эвристика возвращает 0 при каждом целевом состоянии и никогда не возвращает отрицательное значение. В зависимости от того, сколько узлов раскроет ваша эвристика, вы получите следующие оценки за выполнение задания:

Число раскрываемых узлов	Оценка
более 2000	0/3
максимум 2000	1/3
максимум 1600	2/3
максимум 1200	3/3

Помните: если ваша эвристика не будет монотонной, вы не получите баллов!

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация эвристической функции все тесты автооценивателя

```
python autograder.py -q q6
```

#### Задание 4. Задача поедания всех гранул

Теперь мы решим сложную задачу поиска: агент должен съесть всю еду за минимальное количество шагов. Для этого нам понадобится новое определение задачи поиска, которое формализует поедание всех пищевых гранул. Эта задача реализуется классом **FoodSearchProblem** в **searchAgents.py**.

Решение определяется как путь, вдоль которого собирается вся еда в мире **Pacman**. Для данного задания не учитываются призраки или энергетические гранулы; решения зависят только от расположения стен, пищевых гранул и агента. (Конечно, призраки могут ухудшить решения! Мы вернемся к этому в следующих лабораторных работах.) Если будут правильно написаны общие методы поиска, то **A\*- поиск с нулевой эвристикой** (эквивалент поиска с равномерной стоимостью) должен быстро найти оптимальное решение для **testSearch** без изменения кода с вашей стороны (общая стоимость пути 7). Проверьте:

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Здесь **AStarFoodSearchAgent** - это сокращение (ярлык) для

**-p SearchAgent -a fn = astar, prob = FoodSearchProblem, эвристика = foodHeuristic**

Вы должны обнаружить, что алгоритм UCS начинает замедляться даже в случае простого лабиринта **tinySearch**.

**Примечание.** Обязательно выполните задание 1, прежде чем работать над заданием 4.

Допишите код в функции `foodHeuristic` в файле `searchAgents.py`, определив монотонную (согласованную) эвристику для класса `FoodSearchProblem`. Проверьте работу агента на сложной задаче поиска (требует времени):

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Любая нетривиальная неотрицательная согласованная эвристика, разработанная Вами получит 1 балл. Убедитесь, что ваша эвристика возвращает 0 при каждом целевом состоянии и никогда не возвращает отрицательное значение. В зависимости от того, сколько узлов будет раскрывать ваша эвристика, вы получите дополнительные баллы:

Число раскрываемых узлов	Оценка
более 15000	1/4
максимум 15000	2/4
максимум 12000	3/4
максимум 9000	4/4
максимум 7000	5/4 (трудно)

Помните: если ваша эвристика немонотонна, вы не получите баллов. Сможете ли вы решить `mediumSearch` за короткое время? Если да, то это либо впечатляющий результат, либо ваша эвристика немонотонна.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тесты автоценивателя:

```
python autograder.py -q q7
```

## Задание 5. Субоптимальный поиск

Иногда даже с помощью А\*-поиска при хорошей эвристике найти оптимальный путь через все точки накладно. В таких случаях можно просто выполнить быстрый поиск “достаточно” хорошего пути. В этом задании необходимо реализовать агента, который всегда жадно ест ближайшую гранулу. Такой агент `ClosestDotSearchAgent` реализован в файле `searchAgents.py`, но в нем отсутствует ключевая функция, которая находит путь к ближайшей точке.

Реализуйте функцию `findPathToClosestDot` в `searchAgents.py`. Проверьте решение:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Агент выполнит поиск субоптимального пути в этом лабиринте со стоимостью пути 350.

Подсказка: самый быстрый способ завершить `findPathToClosestDot` - это заполнить в классе `AnyFoodSearchProblem` функцию проверки достижения цели `isGoalState`. А затем завершить определение `findPathToClosestDot` с помощью соответствующей функции поиска, написанной ранее. Решение должно получиться очень коротким!

Ваш агент **ClosestDotSearchAgent** не всегда будет находить кратчайший путь через лабиринт. Убедитесь, что вы понимаете, почему, и попробуйте придумать небольшой пример, где многократный переход к ближайшей точке не приводит к нахождению кратчайшего пути для съедания всех точек.

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

```
python autograder.py -q q8
```

### 3.4. Порядок выполнения лабораторной работы

3.4.1. Изучить по лекционному материалу или учебным пособиям [1-3] методы информированного поиска решений задач в пространстве состояний.

3.4.2. Использовать для выполнения лабораторной работы файлы из архива **МиСИИ\_лаб2\_3\_2024.zip**. Изучить структуру данных, соответствующую очереди с приоритетами **PriorityQueue**, реализованную в модуле **util.py**.

3.4.3. Изучите эвристическую функцию, вычисляющую манхэттенское расстояние:

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

Функция вычисляет сумму абсолютных разностей координат текущей позиции и целевой позиции для задачи **PositionSearchProblem**.

3.4.4. Для реализации А\*- алгоритма в соответствии с заданием 1 используйте псевдокод из раздела 3.2.2. При этом создайте список открытых вершин в виде экземпляра очереди с приоритетом элементов: **OPEN = util.PriorityQueue()**. Для реализации части псевдокода, связанной со вставкой состояний-приемников в список **OPEN** используйте метод **OPEN.update**, определенный в классе **PriorityQueue()**. Метод **update** выполняет вставку или обновление элементов в очереди с учетом их приоритетов. В качестве приоритета используйте значение оценочной функции *f*. В этом случае не нужно будет выполнять упорядочение **OPEN** по возрастанию оценочной функции. Поиск состояний-приемников для узла **node** реализуется с использованием метода **problem.getSuccessors(node)**.

3.4.5 Для реализации задания 2 необходимо дописать код следующих методов в классе **CornersProblem** в файле **searchAgents.py**: **\_\_init\_\_**, **getStartState**, **isGoalState**, **getSuccessors**. При этом необходимо выбрать подходящее представление состояний для задачи поиска углов. Удобно представить состояние **state** в виде кортежа  $((x,y),((x1,y1),(x2,y2),\dots))$ , где  $x,y$  – координаты текущей позиции агента,  $x1,y1$  и т.д – координаты посещенных углов.

Рекомендуется добавить в конструктор класса **CornersProblem** присваивание значения атрибуту **self.startingGameState= startingGameState**.

Добавьте в метод `getStartState` возврат начального состояния `self.startingPosition` и пустого кортежа `()`. Пустой кортеж в дальнейшем будет заполняться позициями-кортежами посещенных углов.

Метод `isGoalState` для выбранного представления состояния задачи должен просто проверять длину второго кортежа представления. Если она будет равна 4, то текущее состояние – целевое состояние.

При реализации метода `getSuccessors` необходимо внимательно прочитать следующий комментарий внутри цикла `for` по возможным действиям (направлениям перемещения) агента:

```
# Добавьте состояние-приемник в список приемников, если действие является
# допустимым
# Ниже фрагмент кода, который выясняет, не попадает ли новая позиция на
# стену лабиринта:
#   x,y = currentPosition
#   dx, dy = Actions.directionToVector(action)
#   nextx, nexty = int(x + dx), int(y + dy)
#   hitsWall = self.walls[nextx][nexty]
```

Здесь `currentPosition = state[0]`.

Необходимо воспользоваться этой частью кода. Если координаты новой позиции не попадают на стену (верно `not hitsWall`), то следует сформировать новое состояние в соответствии с выбранным представлением (см. выше):

```
new_state = ((nextx, nexty), state[1])
```

А если координаты новой позиции `nextx, nexty` соответствуют углу (т.е. содержатся в `self.corners`) и этот угол еще не посещался (т.е. отсутствуют в `state[1]`), то новое состояние должно получить следующее значение:

```
new_state = ((nextx, nexty), (state[1] + ((nextx, nexty), ))
```

В итоге если действие не приводит к столкновению со стеной, то новое состояние должно быть добавлено в список-приемников:

```
successors.append((new_state, action, 1))
```

3.4.6. В задании 3 необходимо определить эвристическую функцию `cornersHeuristic` для задачи поиска углов. Возможный вариант эвристической функции: находить непосещенные углы для заданного состояния

```
corners = problem.corners # координаты углов
position = state[0] # текущая позиция
touchedcorners = state[1] # посещенные углы
# список непосещенных углов -- разность 2-х множеств
untouchedcorners = list(set(corners).difference(set(touchedcorners)))
```

и вычислять манхэттенское расстояние ( $\text{abs}(\text{corner}[0] - \text{position}[0]) + \text{abs}(\text{corner}[1] - \text{position}[1])$ ) от позиции агента до ближайшего непосещенного угла, виртуально обходя таким образом ближайшие углы; в качестве значения эвристической функции возвращать сумму виртуальных ближайших расстояний.

Возможны и другие варианты эвристик, например, основанные на вычислении расстояний непосредственно по лабиринту с использованием уже реализованных алгоритмов поиска путей. (см. ниже п.3.4.7)

3.4.7. При решении задания 4 - поедание всех пищевых гранул – требуется определить нетривиальную монотонную эвристику в методе **foodHeuristic(state, problem)** класса **FoodSearchProblem**. Рекомендуется сначала придумать допустимую эвристику. Обычно допустимые эвристики также оказываются монотонными.

Если при использовании А\*-алгоритма будет найдено решение, которое хуже, чем поиск в соответствии алгоритмом равных цен, ваша эвристика немонотонная и, скорее всего, недопустима. С другой стороны, немонотонные эвристики могут найти оптимальные решения, поэтому будьте осторожны.

Состояние в рассматриваемой задаче представляется в виде кортежа (**pacmanPosition, foodGrid**), где **foodGrid** относится к типу **Grid** (см. **game.py**). Чтобы получить список координат точек размещения еды можно вызвать **foodGrid.asList()**.

Если нужен будет доступ к информации о стенах можно сделать запрос в виде **problem.walls**, который вернет объект типа **Grid** с расположением стен. Если необходимо будет сохранять информацию для повторного использования, то можно использовать словарь **problem.heuristicInfo**. Например, если вы хотите посчитать стены только один раз и сохранить это значение, используйте запрос: **problem.heuristicInfo ['wallCount'] = problem.walls.count ()**

Написание кода эвристики начните с простой проверки: если длина списка **foodGrid.asList()** равна нулю, то верните **0**. Если указанный список не пустой, то можно, например, вычислить “расстояния” от текущей позиции до каждой пищевой гранулы. Таким образом можно будет спрогнозировать затраты на достижение целевого состояния – поедание всех гранул. Помните, прогнозные затраты не должны превышать реальных затрат. Один из вариантов вычисления расстояний предоставляет функция **mazeDistance(point1, point2, gameState)** в файле **searchAgents.py**. Эта функция строит путь по лабиринту между точками **point** и возвращает его длину.

3.4.8. В задании 5 необходимо реализовать функцию субоптимального поиска **findPathToClosestDot**, обеспечивающей реализацию агента **ClosestDotSearchAgent**, осуществляющего жадный поиск.

Как указано в задании, сначала определите функцию **isGoalState(state)** класса **AnyFoodSearchProblem**. Нужно просто вернуть результат проверки принадлежности выбранной текущей точки x,y=state списку, возвращаемому методом food.asList() объекта типа AnyFoodSearchProblem.

После этого путь до ближайшей точки в **findPathToClosestDot** может быть найден, например, с помощью вызыва алгоритма равных цен для задачи AnyFoodSearchProblem(gameState).

3.4.9. Для всех заданий необходимо выполнить проверку тестов автооценивателя, результаты прохождения тестов внести в отчет.

### 3.5. Содержание отчета

Цель работы, краткий обзор методов информированного поиска решений задач в пространстве состояний, описание свойств A\*- алгоритма, тексты реализованных функций с комментариями в соответствии с заданиями 1-5, результаты выполнения поиска для разных задач и алгоритмов и их анализ, результаты автооценивания, выводы по проведенным экспериментам с разными алгоритмами информированного поиска.

### 3.6. Контрольные вопросы

3.6.1. Что называют эвристикой?

3.6.2. Объясните основной принцип построения процедур эвристического поиска. Запишите вид оценочной функции и объясните её составляющие.

3.6.3. Напишите на псевдоязыке процедуру поиска в соответствии с A-алгоритмом.

3.6.4. Сформулируйте алгоритм подъема в гору.

3.6.5. Сформулируйте алгоритм глобального выбора первой наилучшей вершины.

3.6.6. Почему в A-алгоритме возможен возврат вершин из списка закрытых вершин в список открытых вершин? Приведите примеры.

3.6.7. Сформулируйте и объясните свойства A-алгоритма.

3.6.8. Какой A-алгоритм называют гарантирующим (допустимым)?

3.6.9. Сформулируйте и объясните условие монотонности.

3.6.10. Сформулируйте эвристику манхэттенского расстояния.

3.6.11. Сравните алгоритмы слепого и эвристического поиска по критерию гарантированности получения результата и эффективности поиска.