ЗАДАНИЕ 7

Муравьиные алгоритмы

Построение модели • Наша цель — реализовать классический вариант муравьиного алгоритма[1]. Готовая модель должна работать с несколькими типами графов и визуализировать распределение феромона и количество проходов муравьев по ребрам графа, а также показывать лучший найденный маршрут (решение задачи коммивояжера).

[1] С некоторыми незначительными отличиями, обусловленными больше вопросами визуализации.

A Создаем новую модель. Оставляем настройки по умолчанию, но снимаем циклическое замыкание границ.

B Создаем новый вид черепах, соответствующий вершинам графа (`nodes` и `node`).

C Добавляем к интерфейсу модели слайдер graph-size, соответствующий количеству вершин графа, с диапазоном от 4 до 100 и с текущим значением 16.

D Добавляем к интерфейсу выпадающий список layout, обозначающий тип графа и имеющий три опции для выбора:

- `"random"` — случайное расположение вершин;
- `"circle"` — вершины графа располагаются на окружности;
- `"grid"` — вершины располагаются в узлах прямоугольной сетки.

E Создаем кнопку setup и соответствующую ей процедуру `setup`, в которой пока выполняем следующие действия: очищаем мир; меняем цвет патчей на белый; создаем заданное количество `graph-size` вершин графа (`nodes`), для каждой новой вершины вызываем процедуру настройки `setup-node`; сбрасываем таймер.

F Пишем код процедуры `setup-node`. Сначала устанавливаем визуальные характеристики текущей вершины: круглая форма, размер 1, цвет оранжевый. Затем определяем ее местоположение с учетом выбранного пользователем типа графа `layout`. Для типа `"random"` координаты определяем случайным образом. Для двух других типов (`"circle"` и `"grid"`) положение вершины должно вычисляться по ее номеру `who`[2]. Для режима `"circle"` значение `who` определяет угол `a` относительно вертикальной оси, под которым на окружности радиуса `max-pxcor - 1` располагается данная вершина:

```
if layout = "circle" [
  let a 360 / graph-size * who
  let r max-pxcor - 1
  setxy (r * sin a) (r * cos a)
]
```

[2] Это уникальный числовой идентификатор любой черепахи, самая первая черепаха (любого вида) получает номер 0, следующая — 1 и т.д. Следовательно, в нашей модели все вершины графа оказываются пронумерованными от 0 до `graph-size - 1`.

G Для типа `"grid"` сначала находим минимальный размер `n` квадратной решетки, в которой число узлов не меньше размера графа. Затем по значению `who` вычисляем номера строки и столбца решетки и координаты точки, где должна располагаться текущая вершина:

```
if layout = "grid" [
  let n ceiling sqrt graph-size
  let d (2 * max-pxcor - 2) / (n - 1)
  set xcor min-pxcor + 1 + d * (who mod n)
  set ycor min-pycor + 1 + d * floor (who / n)
]
```

Проверяем работу процедуры `setup` для всех трех типов графов (рис. 7.1).

РИС. 7.1 Три способа расположения вершин графа

H Ребра графа будем представлять с помощью так называемых связей (`links`) — еще одного типа агентов в NetLogo. Каждое ребро в нашем графе будет характеризоваться следующими атрибутами: `len` — длина ребра (на ее основе будет вычисляться длина маршрута в задаче коммивояжера); `ph` — уровень феромона, который мы будем ограничивать диапазоном от 0 до 1; `visits` — количество муравьев, прошедших по этому ребру на текущей итерации алгоритма; `best?` — логическое значение, показывающее, входит ли данное ребро в текущий оптимальный маршрут. Указанные атрибуты создаются по стандартной схеме:

```
links-own [len ph visits best?].
```

I Наш граф (независимо от типа расположения его вершин) будет *полным*, в котором каждая вершина связана со всеми остальными вершинами, и *геометрическим* — длина ребра равна евклидовому расстоянию между соответствующими вершинами. Поэтому в процедуре `setup` попросим каждую вершину связаться со всеми остальными вершинами[3] и для каждой новой связи устанавливаем значения ее атрибутов: цвет (встроенный атрибут), начальный уровень феромона (берем максимальное значение), длину ребра (используем нормированное значение — длина `link-length` связи, деленная на размер модели `max-pxcor`):

```
ask nodes [
  create-links-with other nodes [
    set color blue
    set ph 1
    set len link-length / max-pxcor
  ]
]
```

J Проверяем еще раз работу кнопки setup, примеры графов показаны на рис. 7.2.

K К интерфейсу добавляем слайдер ants-number, соответствующий размеру колонии муравьев. Создаем новый вид черепах — муравьи (`ants` и `ant`). В процедуре `setup` создаем заданное количество муравьев и делаем их невидимыми (в нашей модели муравьи не будут визуализироваться).

L Переходим к реализации собственно самого муравьиного алгоритма. Для этого к интерфейсу модели добавим кнопку go и три слайдера alpha, beta и Q, соответствующих параметрам алгоритма в формулах (7.1) и (7.2), с диапазоном от 0 до 5, шагом 0.1 и значением по умолчанию 1.

$$p_{uv}^k = \begin{cases} \dfrac{\tau_{uv}^\alpha \eta_{uv}^\beta}{\sum_{w \notin X_k} \tau_{uw}^\alpha \eta_{uw}^\beta}, & \text{если } v \notin X_k, \\ 0, & \text{если } v \in X_k, \end{cases} \tag{7.1}$$

$$\triangle \tau_{uv}^k = Q/mD_k. \tag{7.2}$$

Также для управления режимом просмотра модели добавим к интерфейсу переключатель show-best?.

M Создадим глобальные переменные: `best-len` — длина лучшего на данный момент маршрута; `sel`

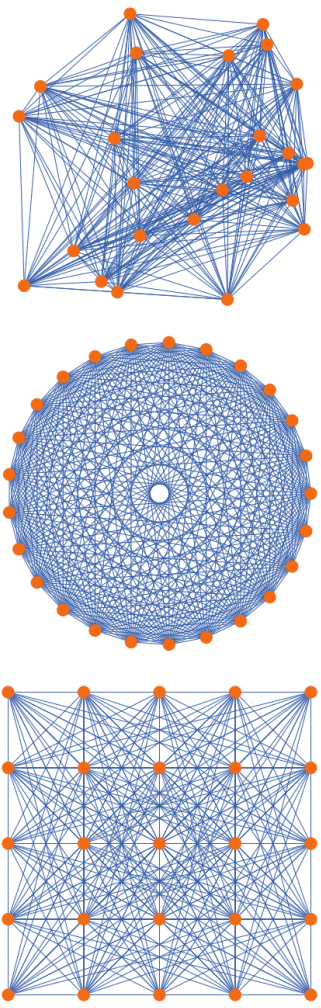

РИС. 7.2 Три типа графов в модели

[3] Команда `create-links-with` создает ненаправленную (двустороннюю) связь между двумя черепахами. Направленная связь создается командой `create-link-to`.

и rnd — две служебные переменные, которые нам потребуются в процедуре выбора муравьем новой вершины.

N Добавим следующие атрибуты муравьям:

- current — текущая вершина графа, где в данный момент находится муравей;
- to-visit — список вершин, еще не посещенных муравьем на данной итерации;
- visited-links — набор (agentset) связей, образующих текущий маршрут муравья;
- path-len — длина текущего маршрута.

O Создаем процедуру go, каждый вызов которой процедуры будет соответствовать одной итерации муравьиного алгоритма, на которой все муравьи сделают свой обход графа. Сначала проводим инициализацию муравьев — просим каждого муравья инициализировать свои атрибуты: текущая вершина current — нулевая[4]; список to-visit вершин для посещения — все, кроме нулевой, такой список можно получить командой

(range 1 graph-size)[5];

набор посещенных ребер visited-links — изначально пустой, создается командой no-links; длина текущего маршрута path-len равна 0. Также просим все связи установить нулевое значение атрибута visits. Реализуем цикл построения маршрута всеми муравьями. Число итераций в этом цикле равно размеру графа, т.к. номер итерации нам не важен, то цикл можно реализовать командой repeat:

```
repeat graph-size [ask ants [move-ant]].
```

Перемещение каждого муравья реализуется командой move-ant, код которой мы напишем чуть позже. После построения муравьями своих маршрутов просим их обновить феромон на посещенных ими ребрах графа (visited-links), сначала по формуле (7.2) вычисляем величину обновления dt, затем прибавляем эту величину к атрибуту ph всех ребер маршрута:

```
ask ants [
  let dt Q / path-len / graph-size
  ask visited-links [set ph ph + dt]
]
```

[4] Все муравьи будут начинать двигаться по графу из одной стартовой вершины.

[5] Скобки вокруг команды range обязательны, т.к. эта команда имеет переменное число аргументов, поэтому при вызове надо явно показывать, где заканчиваются эти аргументы.

С помощью вызова процедуры `check-best` проверяем, не нашли ли мы на данной итерации новое, более оптимальное решение. Затем вычисляем максимальный уровень феромона `max-ph` среди всех связей, просим все связи нормализовать свой уровень феромона (поделив значение `ph` на `max-ph`) и перенастроить свои визуальные параметры с помощью вызова процедуры `color-link`. Последней командой процедуры `go` обновляем таймер.

P Создаем процедуру `move-ant`. В ее коде сначала с помощью функции `choose-next` выбираем следующую вершину `next` для посещения. Исключаем эту вершину из списка `to-visit`; создаем локальную переменную `l`, в которой запоминаем связь из текущей вершины `current` в новую вершину; эту связь добавляем к набору ребер `visited-links`, образующих маршрут данного муравья:

```
set to-visit remove next to-visit
let l link current next
set visited-links (link-set l visited-links)
```

Увеличиваем атрибут `visits` связи `l` на 1. Увеличиваем атрибут `path-len` муравья на длину связи `[len] of l`. Наконец, делаем текущей вершиной `current` новую вершину `next`.

Q Создаем функцию `choose-next`, которая реализует выбор новой вершины маршрута муравья согласно формуле (7.1). Такой метод выбора одной из альтернатив с учетом их вероятностей называется *методом рулетки* из-за аналогии с игровой рулеткой (рис. 7.3). Реализуем этот метод следующим образом. Сначала составим список `weights` весов $\tau_{uv}^{\alpha}\eta_{uv}^{\beta}$ из формулы (7.1) для всех непосещенных городов из списка `to-visit`, используя команду `map`, которая генерирует новый список, применяя указанную функцию (первый аргумент) к каждому элементу заданного списка (второй аргумент):

`let weights map get-weight to-visit`.

Функция `get-weight` должна по заданному аргументу (идентификатору вершины) вернуть значение соответствующего веса:

```
to-report get-weight [c]
  let l link current c
  report [(ph ^ alpha) / (len ^ beta)] of l
end
```

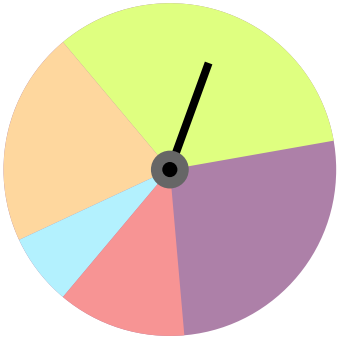

РИС. 7.3 Рулетка с секторами разного размера

R Следующим шагом в функции `choose-next` выбираем случайное действительное число `rnd` (это глобальная переменная) из диапазона от 0 до `sum weights`. Другой глобальной переменной `sel` присваиваем нулевое значение. Именно в этой переменной и будет храниться номер найденного продолжения маршрута. Вызываем функцию `foreach`, которая для каждой группы элементов заданных списков (первые аргументы) применяет указанную процедуру (последний аргумент):

```
(foreach to-visit weights select).
```

Сначала процедура `select` будет применена к паре первых элементов списков `to-visit` и `weights`, затем к паре вторых элементов и т.д. Процедура `select [c w]` проверяет, лежит ли значение `rnd` в диапазоне от 0 до `w`, если да, то значение `sel` полагается равным `c`. После этого значение `rnd` уменьшается на величину `w`. Последней командой процедуры `choose-next` мы возвращаем найденное значение `sel`. Причем если список `to-visit` оказался пустым (муравей обошел все вершины графа), то наша функция вернет нулевое значение, что означает возвращение в стартовую вершину, имеющую как раз нулевой идентификатор.

S Создаем процедуру `check-best`, которая проверяет, не нашли ли мы на данной итерации новое, лучшее решение. Сначала находим (лучшего) муравья `best-ant` с минимальным значением атрибута `path-len`. Затем запоминаем в переменной `bp` значение атрибута `path-len` найденного муравья `best-ant` (минимальная длина маршрута на данной итерации). Если наша итерация является первой (`tics = 0`) или длина `bp` меньше текущего минимума `best-len`, то обновляем лучшее решение: заменяем `best-len` на `bp`, просим *все* связи установить атрибут `best?` равным `false`, просим связи из набора `visited-links` муравья `best-ant` установить атрибут `best?` равным `true`.

T Пишем код последней процедуры `color-link`. Если включен режим просмотра лучшего решения `show-best?`, то устанавливаем толщину связи (атрибут `thickness`) равной 0.25, цвет — зеленым, атрибут `hidden?` — равным значению `not best?`. Таким образом, будут показаны только те связи, кото-

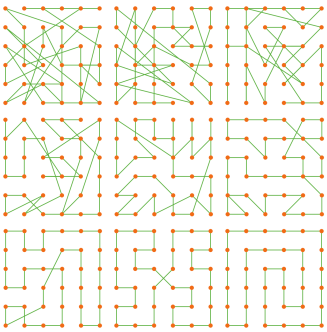

РИС. 7.4 Работа муравьиного алгоритма в режиме просмотра лучшего решения

рые входят в лучший найденный маршрут. Теперь можно проверить работу кнопки go (с включенным переключателем show-best?), пример построения оптимального маршрута для решетки размера 6×6 приведен на рис. 7.4.

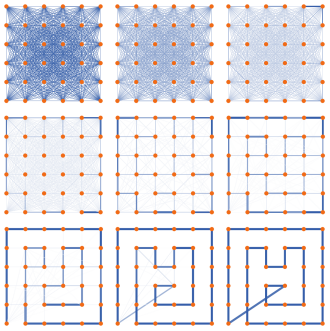

РИС. 7.5 Работа муравьиного алгоритма в режиме просмотра уровня феромона и количества посещений

U При выключенном режиме show-best? будем показывать количество феромона на ребрах графа (цвет) и сколько муравьев прошло по каждому ребру (толщина). Причем если уровень феромона на ребре меньше некоторого порога (например, равного 0.1) и число муравьев, прошедших по этому ребру, равно 0, то это ребро вообще не показывается (атрибут hidden? полагается равным false):

```
set hidden? visits = 0 and ph < 0.1
set thickness 0.5 * visits / ants-number
set color scale-color blue ph 2 0
```

На рис. 7.5 показано, как меняется распределение феромона на ребрах графа в процессе работы муравьиного алгоритма. Интересной особенностью данной (первоначальной) версии муравьиного алгоритма является то, что сформированный в итоге муравьями путь оказывается, как правило, не оптимальным, хотя в процессе его построения оптимальное решение часто возникает, но потом колония муравьев его «забывает». Одной из причин такого поведения является то, что построенный муравьями маршрут практически не отличается по длине от оптимального.

V Проверяем работу модели для разных значений ее параметров и на других типах графов, в том числе большего размера. Окончательный интерфейс модели показан на рис. 7.6.

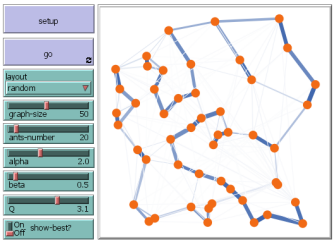

РИС. 7.6 Окончательный интерфейс модели

УПРАЖНЕНИЯ

1 Включите в интерфейс модели график (рис. 7.7), показывающий зависимость от номера итерации: длины лучшего найденного решения (best), длины лучшего решения на текущей итерации (min), средней длины на данной итерации (mean). На графике хорошо видно, как завершается работа алгоритма — кривая min отдаляется от кривой best и приближается к кривой mean.

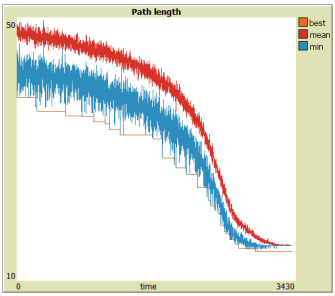

РИС. 7.7 График сходимости муравьиного алгоритма

2 Исследуйте, как влияет размер колонии на скорость сходимости алгоритма и на время его работы. Например, в оригинальном варианте авторы алгоритма предложили делать размер колонии `ants-number` равным размеру графа `graph-size`.

3 Проанализируйте поведение муравьиного алгоритма в двух крайних случаях, когда равен нулю либо параметр `alpha`, либо параметр `beta`. В первом случае алгоритм вырождается в вариант стохастического жадного алгоритма. Во втором случае на выбор муравьев полностью перестает влиять информация о длинах ребер.

4 Включите в модель показ лучшего решения, найденного на текущей итерации. Например, ребра, входящие в глобальное лучшее решение, надо изображать толстыми линиями, а входящие в локальное лучшее решение — тонкими линиями. Кроме того, можно варьировать цвет в зависимости от того, входит ли данное ребро в какой-нибудь оптимальный маршрут. Пример такой визуализации приведен на рис. 7.8 (линии темного цвета входят в оба оптимальных маршрута, и в глобальный, и в локальный).

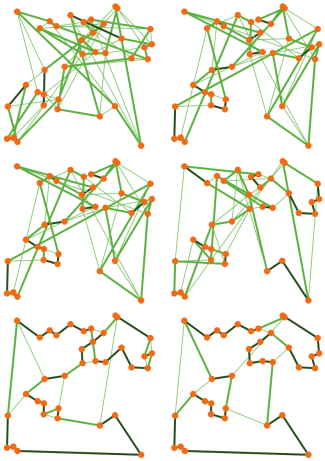

РИС. 7.8 Визуализация двух видов оптимальных маршрутов

5 Включите в модель другие типы геометрических графов (рис. 7.9, показаны оптимальные маршруты).

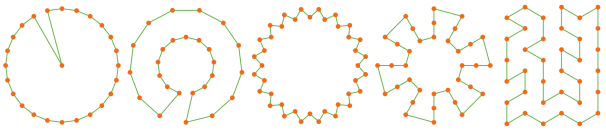

РИС. 7.9 Геометрические графы

6 Разработанную модель можно использовать для решения задачи поиска гамильтонова цикла[6] в заданном графе. Для этого строится вспомогательный *полносвязный* граф с тем же набором вершин, в котором ребра исходного графа получают вес (длину), равный 1, а ребра, отсутствующие в исходном графе, получают вес 10. Если гамильтонов цикл в исходном графе имеется, то он будет решением и задачи коммивояжера для вспомогательного графа. Тогда алгоритм оказывается следующим: запускаем модель на вспомогательном графе, если найденное решение имеет длину, равную размеру графа, то, значит, исходный граф имеет гамильтонов цикл, и мы его нашли. Если длина найденного маршрута больше числа вершин графа, то либо исходная задача не имеет решения, либо (в силу приближенности муравьиного алгоритма) мы его не смогли найти. На

[6] Цикл, содержащий все вершины графа ровно по одному разу.

рис. 7.10 показан пример поиска гамильтонова пути в случайном графе (каждое ребро получает вес 1 с вероятностью 25%, вес 10 с вероятностью 75%). Красным цветом выделены ребра в текущем решении, имеющие вес 10.

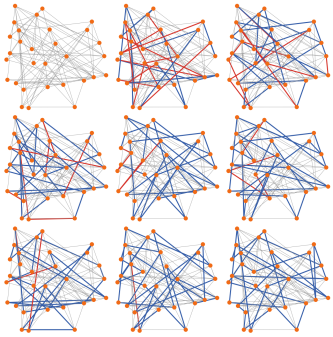

РИС. 7.10 Поиск гамильтонова цикла в случайном графе

7 Рассмотрите вариацию муравьиного алгоритма, в которой каждый муравей начинает строить свой маршрут со случайной вершины.

8 Реализуйте вариант муравьиного алгоритма, в котором муравьи помечают ребра феромоном сразу при их прохождении, а не после построения полного маршрута (более естественный подход с биологической точки зрения). Чтобы короткие пути получали преимущество, количество откладываемого феромона должно зависеть как от длины ребра (чем длиннее ребро, тем меньше феромона), так и от длины уже построенной части маршрута (чем она длиннее, тем меньше феромона).

9 Практически сразу после изобретения муравьиного алгоритма его авторами было предложено несколько методик, направленных на ускорение сходимости алгоритма[7]. Предлагалось, во-первых, обновлять феромоном только лучший на данной итерации маршрут. Во-вторых, использовать схему псевдослучайного пропорционального выбора: если случайное число $q \in [0,1]$ меньше заданного параметра q_0, то выбирается ребро с максимальным уровнем феромона, в противном случае для выбора используется метод рулетки. В-третьих, помимо (глобального) обновления феромона после завершения каждой итерации алгоритма, предлагалось использовать дополнительное *локальное* обновление, выполняемое каждым муравьем сразу же после прохода по ребру:

$$\tau_{uv} \leftarrow (1-\rho)\tau_{uv} + \rho\tau_0,$$

где $\rho \in [0,1]$ — параметр, τ_0 — начальное значение феромона. Целью локального обновления является *ослабление* ребра, по которому только что прошел муравей, для того чтобы следующие за ним муравьи с большей вероятностью выбрали другие ребра для продолжения маршрута. Включите эти методики в модель и выполните их тестирование.

10 В другой известной вариации муравьиных алгоритмов — модели максиминной системы[8] — обновление феромона производится также только муравьем, нашедшим лучшее решение на текущей итерации. Кроме того, для избегания слишком быстрой сходимости

[7] M. Dorigo, L. M. Gambardella, *Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem*, IEEE Transactions on Evolutionary Computation, 1997, 1 (1), p. 53–66.

[8] T. Stutzle, H.H. Hoos, *MAX MIN Ant System*, Future Generation Computer Systems, 2000, Vol. 16, p. 889–914.

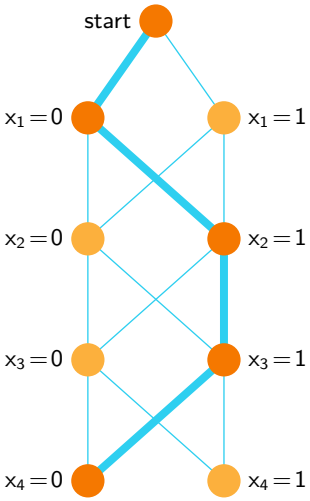

РИС. 7.11 Граф интерпретаций, показан путь, соответствующий двоичному вектору $x = 0110$

концентрация феромона на ребрах ограничивается заданным диапазоном от $\tau_{\min}$ до $\tau_{\max}$. Выполните реализацию этой модели и исследуйте ее эффективность.

11 Муравьиные алгоритмы применяются с успехом и при решении многих других задач дискретной оптимизации с помощью сведения их к задаче поиска кратчайшего пути в графе. Например, задача минимизации функции $f(x_1, \ldots, x_n)$ от n двоичных переменных может быть преобразована в задачу поиска кратчайшего пути в так называемом *графе интерпретаций*, показанном на рис. 7.11. Этот граф является ориентированным, его дуги в данном случае направлены сверху вниз. Цель — найти кратчайший путь из стартовой вершины *start* до одной из двух нижних вершин. Любой такой путь на i-м уровне графа проходит либо через левую вершину ($x_i = 0$), либо через правую ($x_i = 1$). После того как путь построен, определяется соответствующий ему вектор переменных x и вычисляется значение целевой функции $f(x)$, которое и служит обобщенной длиной пути. На рис. 7.12 показан пример применения такой вариации муравьиного алгоритма к решению задачи MinOne (минимизация числа единиц в двоичном векторе).

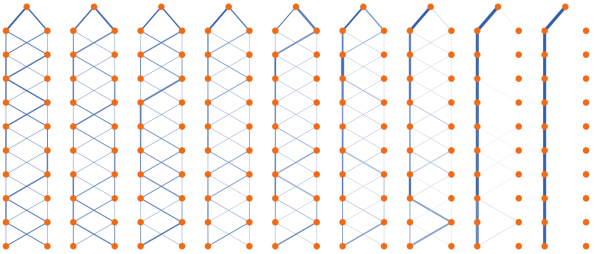

РИС. 7.12 Решение задачи MinOne с помощью муравьиного алгоритма