ЗАДАНИЕ 10

Генетические алгоритмы

Построение модели • Рассмотрим следующую модельную задачу для генетического алгоритма. Требуется проложить маршрут между двумя заданными точками на плоскости, состоящий ровно из n прямолинейных отрезков единичной длины. Параметрами маршрута являются углы поворота каждого звена искомой ломаной линии относительно предыдущего звена. Угол поворота первого звена отсчитывается от вертикального направления. Назовем последовательность таких углов планом маршрута. Исполнитель пытается выполнить движение согласно заданному плану. Если в процессе движения исполнитель оказывается перед препятствием (граница модели или закрытый для прохода патч), которое не дает совершить очередное перемещение, то процесс движения заканчивается досрочно. После завершения движения оценивается план, по которому оно выполнялось. Оценкой является евклидово расстояние от точки завершения движения до цели, чем оно меньше, тем лучше маршрут.

A Создаем новую модель, оставляем параметры по умолчанию, только убираем циклическое замыкание границ.

B Создаем три глобальные переменные: `n` — число звеньев искомой ломаной линии; `start` — патч, с которого начинаются все маршруты; `target` — патч, который должен быть достигнут.

C Каждая черепаха будет представлять отдельное решение нашей задачи, т.е. некоторый маршрут до цели. Черепахам приписываем два атрибу-

та: x — план маршрута, последовательность из n углов поворота; f — целевая функция, расстояние от последней точки маршрута до цели.

D Добавляем к интерфейсу кнопку setup и создаем соответствующую процедуру, в которой 1) очищаем модель; 2) вызываем команду setup-layout, которая должна определить стартовый и конечный патчи и расставить препятствия; 3) находим расстояние d от старта до цели и вычисляем количество звеньев n — расстояние d, умноженное на некоторый коэффициент[1]:

```
let d [distance target] of start
set n floor (1.3 * d)
```

4) вызываем процедуру create-population для создания популяции решений; 5) сбрасываем таймер.

[1] Больший 1, чтобы дать алгоритму некоторую свободу в построении маршрута.

E Создаем процедуру setup-layout. Сначала к интерфейсу модели добавляем выпадающий список layout, определяющий несколько схем расположения препятствий. Указываем в его настройках опции: "empty", "random", "bar" и "hole". Возвращаемся к процедуре setup-layout. Для указания типа патча будем использовать их цвет: свободные патчи будут белыми, патчи-препятствия — серыми (gray), стартовый патч — синим, целевой патч — красным. Сначала все патчи раскрашиваем в белый цвет. Затем расставляем препятствия согласно значению переменной layout (рис. 10.1): "empty" — нет препятствий; "random" — препятствиями являются 100 случайно выбранных патчей; "bar" — в центре модели создается горизонтальная полоса длины из 15 патчей; "hole" — горизонтальная полоса посередине модели со случайно расположенной дыркой (белого цвета) около центра. Объявляем стартовый и целевой патчи:

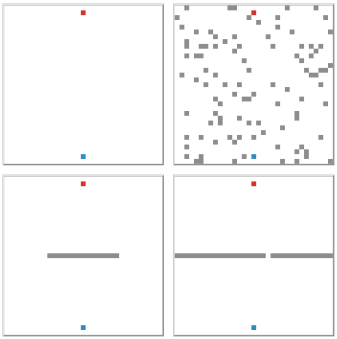

РИС. 10.1 Четыре типа расположения препятствий в модели

```
set start patch 0 (min-pycor + 1)
set target patch 0 (max-pycor - 1)
```

Раскрашивам их в соответствующие цвета.

F К интерфейсу добавляем слайдер pop-size, соответствующий размеру популяции, и пишем код процедуры create-population, работа которой заключается в создании указанного числа черепах и настройке их параметров. Визуальные параметры — форма "circle" и размер 0.8. Кроме того, создаем начальный случайный план маршрута x.

```
set x (list random 360)
repeat (n - 1) [
  set x lput (90 - random 180) x
```

Первой командой создаем новый список, состоящий из одного элемента — случайного числа, соответствующего первому углу поворота. Затем в цикле добавляем к списку (командой `lput`) оставшиеся случайные углы от −90 до 90 градусов. Последней командой просим черепаху построить (и нарисовать) маршрут по заданному плану с помощью команды `eval 1` и определить расстояние от его последней точки до цели. Аргумент этой команды означает толщину линии при прорисовке маршрута. Если этот параметр меньше нуля, то маршрут не рисуется (это свойство мы будем использовать дальше).

G Создаем процедуру `eval [w]`, которая отвечает за прокладывание и прорисовку маршрута движения по плану `x`. Сначала перемещаем черепаху в стартовую позицию: `move-to start` — и ориентируем ее вертикально: `set heading 0`. Если значение `w` положительно, то опускаем перо (команда `pd`) и устанавливаем его толщину: `set pen-size w`. Далее организуем цикл по углам `x` с помощью счетчика `k`. Т.к. прокладывание маршрута может закончиться досрочно при попадании в закрытую для прохода область, то введем дополнительный флаг `stop?` и инициализируем его значением `false`. Делаем цикл `while [k < n and not stop?]`, на каждой итерации которого выполняем поворот на угол `item k x` (`k`-й элемент списка `x`); проверяем состояние патча на один шаг вперед, если патча нет (граница) или он серый, то устанавливаем значение флага `stop?` равным `true`, иначе делаем движение вперед на 1; увеличиваем счетчик `k` на 1. После завершения цикла поднимаем перо (команда `pu`).

```
to eval [w]
  move-to start
  set heading 0
  if w > 0 [pd set pen-size w]
  let k 0
  let stop? false
  while [k < n and not stop?] [
    rt item k x
    let p-a patch-ahead 1
    ifelse p-a = nobody or [pcolor] of p-a = gray
```

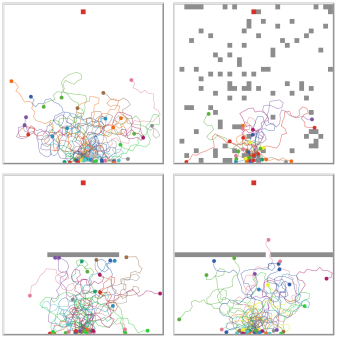

РИС. 10.2 Начальная популяция маршрутов для разных конфигураций layout (заметьте, что абсолютное большинство маршрутов заканчивается на нижней границе модели)

```
    [set stop? true] [fd 1]
    set k k + 1
  ]
  pu
  set f distance target
end
```

Проверяем работу кнопки setup (рис. 10.2).

H Добавляем к интерфейсу кнопку go и два переключателя hide-turtles? и show-best?, которые будут отвечать за режим просмотра модели. Включение первого переключателя будет скрывать черепах, оставляя только траектории их маршрутов, включение второго будет скрывать все траектории, кроме лучшей на данной итерации алгоритма.

I Создаем процедуру go, в которой будет реализована одна итерация генетического алгоритма плюс прорисовка и оценка полученных траекторий. Первой командой удаляем все нарисованные траектории: clear-drawing. Второй командой устанавливаем режим видимости всех черепах равным значению hide-turtles?. Затем последовательно с помощью трех отдельных команд ask просим всех черепах выполнить: отбор (команда select), скрещивание (команда crossover), мутацию (команда mutate). После выполнения трех генетических операторов производим прорисовку и оценку построенных маршрутов. Если значение переключателя show-best? ложно, то просим черепах выполнить оценку с прорисовкой: eval 1. В противном случае выполняем только оценку: eval -1, после чего просим черепаху с лучшим значением атрибута f прорисовать свою траекторию толстой линией:

```
ask min-one-of turtles [f] [eval 2].
```

Последней командой процедуры go обновляем таймер.

J Создаем процедуру select. Первой командой проверяем, выполнять ли отбор или нет (в данном случае мы используем значение $p_{sel} = 0.25$). Если отбор все-таки делаем, то выбираем еще одно случайное решение o-t для проведения турнира. Определяем, какое из двух решений лучше (переменная better?). Генерируем случайное число r для определения победителя. Лучшее решение в паре побеждает с вероятностью $p_{win} = 0.9$, поэтому

победителя определяем с помощью логической операции xor[2]. Победитель копирует свои атрибуты x и f в соответствующие атрибуты проигравшего.

```
to select
  if random-float 1 > 0.25 [stop]
  let o-t one-of other turtles
  let better? f < [f] of o-t
  ifelse better? xor (random-float 1 > 0.9) [
    ask o-t [
      set x [x] of myself
      set f [f] of myself
    ]
  ][
    set x [x] of o-t
    set f [f] of o-t
  ]
end
```

Проверяем работу кнопки go. С большой вероятностью из начальной популяции будет выживать только ее лучшее решение (рис. 10.3).

K По похожей схеме реализуем оператор скрещивания (процедура crossover). Проверяем, выполнять ли данной черепахе скрещивание (вероятность $p_{sel} = 0.5$). Если да, то выбираем другую черепаху o-t. Создаем новый пустой список t, создаем и обнуляем счетчик цикла i. Делаем цикл repeat n, на каждой итерации которого к списку t дописываем элемент с индексом i либо из списка x текущей черепахи (вероятность такого выбора сделаем равной 0.9), либо из списка x черепахи o-t; увеличиваем счетчик цикла на 1. Последним оператором процедуры копируем t в атрибут x.

```
to crossover
  if random-float 1 < 0.5 [stop]
  let o-t one-of other turtles
  let t (list)
  let i 0
  repeat n [
    ifelse random-float 1 < 0.8
    [set t lput (item i x) t]
    [set t lput (item i [x] of o-t) t]
    set i i + 1
  ]
  set x t
end
```

L Проверяем работу модели (рис. 10.4). Теперь генетический алгоритм демонстрирует более сложное поведение, пытаясь составить из имеющихся

[2] Результат бинарной операции xor является истинным, если истинен ровно один ее операнд.

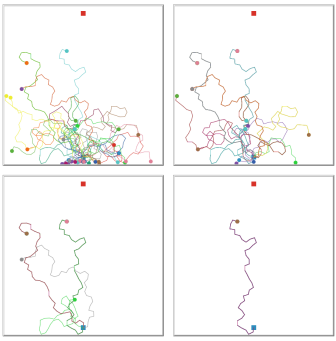

РИС. 10.3 Работа оператора отбора select, на последнем кадре все 50 решений представляют один и тот же маршрут

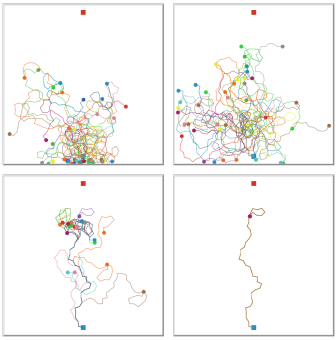

РИС. 10.4 Работа генетического алгоритма с оператором скрещивания

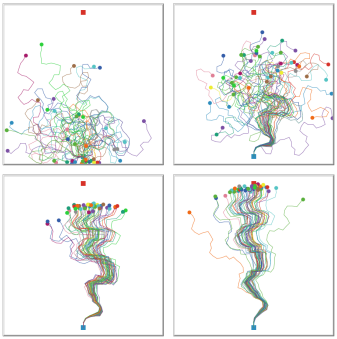

РИС. 10.5 Работа полного генетического алгоритма

решений более хорошее решение. Но рано или поздно вся популяция тем не менее сходится к одному и тому же маршруту (говорят, что в популяции происходит потеря генетического разнообразия), после чего оператор скрещивания перестает работать, а алгоритм перестает сходиться (т.е. улучшать решение).

M Создаем процедуру `mutate`. В ее начале проверяем, выполнять ли мутацию (вероятность мутации положим равной $p_{mut} = 0.2$). Далее создаем пустой список `t`, счетчик цикла `i` (как в процедуре `select`). Организуем цикл, на каждой его итерации дописываем в конец списка `t` очередной элемент из списка `x`, к которому прибавляется случайное число из интервала от -2 до 2 (градусов). В конце копируем `t` в `x`.

N Проверяем работу алгоритма со всеми тремя операторами (рис. 10.5). Теперь генетический алгоритм практически всегда находит оптимальное решение, правда, иногда для этого ему требуется выполнить много итераций.

O Для контроля сходимости алгоритма добавьте к интерфейсу монитор, показывающий минимальное значение атрибута `f` среди всех черепах. В конце процедуры `go` добавьте проверку достижения цели — если на целевом патче есть хотя бы одна черепаха, то останавливаем модель:

```
if [any? turtles-here] of target [stop].
```

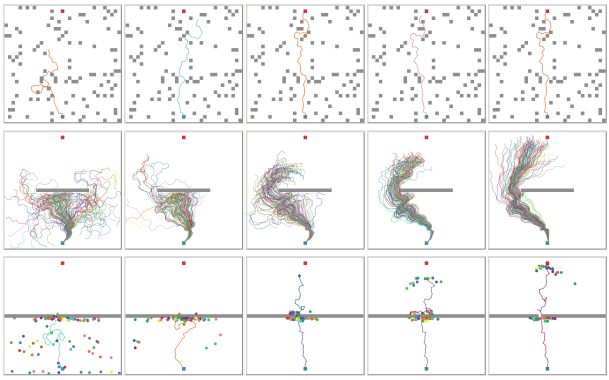

РИС. 10.6 Разные типы препятствий

P Тестируем модель на всех конфигурациях препятствий (рис. 10.6), используя разные режимы просмотра. Окончательный интерфейс модели показан на рисунке 10.7.

УПРАЖНЕНИЯ

1 Добавьте к интерфейсу модели слайдеры для динамического управления параметрами генетического алгоритма: p_{sel}, p_{win}, p_{cross}, p_{mut}. Исследуйте поведение модели при изменении этих параметров.

2 Одной из стандартных методик, применяемых в генетических алгоритмах для улучшения их работы, является имитация отжига для параметров мутации. На начальном этапе мутация делается высокой, а потом постепенно снижается до нулевого уровня. В нашей модели отжигу можно подвергать либо вероятность p_{mut}, либо величину максимального изменения угла ломаной, которая в рассмотренной модели была зафиксирована значением 2. Реализуйте и протестируйте такую схему отжига.

3 В построенной нами модели на каждой итерации есть ненулевой шанс потерять (за счет работы любого из трех генетических операторов) лучшее решение. Для сложных задач такое свойство можно сильно затруднить процесс поиска решения. Возможным выходом является применение *элитной* стратегии, когда лучшее текущее решение принудительно блокируется против его модификации. Дополнительно к этому при отборе и скрещивании можно использовать более агрессивную схему, когда с достаточно большой вероятностью (например, 0.1) в качестве второго решения выбирается не случайное решение, а лучшее в популяции.

4 Добавьте к модели другие типы препятствий, например показанные на рис. 10.8.

5 Реализуйте схему отбора на основе *метода рулетки*, в котором из текущего поколения решений составляется промежуточное поколение по следующему алгоритму. Выполняется цикл из m (размер популяции) итераций, на каждой из которых с вероятностью, *пропорциональной целевой функции* (которая в этом методе отбора должна быть неотрицательной и подлежащей максимизации), из текущей популяции выбирается одно решение и помещается в промежуточную популяцию (рис. 10.9). После завершения этого цикла текущая популяция замещается промежуточной популяцией.

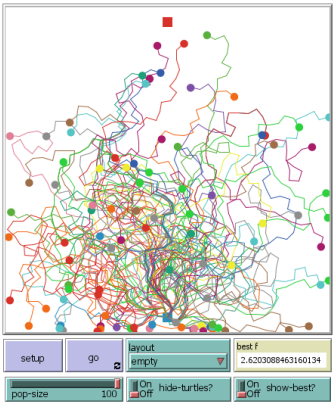

РИС. 10.7 Окончательный интерфейс модели

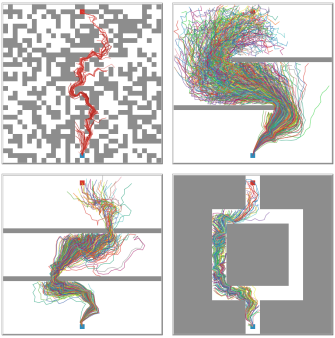

РИС. 10.8 Несколько других возможных конфигураций препятствий: `dense`, `zigzag`, `two holes`, `double bridge`

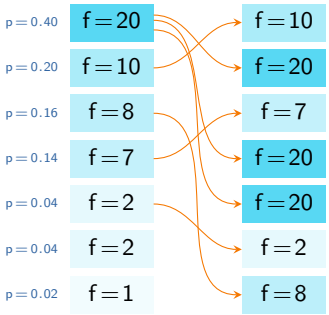

РИС. 10.9 Пример работы оператора отбора на основе метода рулетки

6 Слабым местом отбора методом рулетки, когда выживаемость решений пропорциональна функции их качества, является низкая скорость сходимости алгоритма при приближении к локальному экстремуму функции качества. В такой ситуации практически все решения имеют приблизительно одинаковое значение функции качества и отбор сводится к случайному выбору решений[3]. Чтобы избежать такого развития событий, применяют либо ранговый метод отбора, либо турнирный метод (реализованный нами в модели). В *ранговом методе* отбор производится на основе относительного расположения (ранга) решений в списке всех решений данной популяции, упорядоченном по возрастанию функции качества: чем лучше решение, тем выше ранг. Отбор производится тем же вероятностным способом, что и в методе рулетки, но вероятность отбора теперь пропорциональна рангу решения, а не функции качества. Таким образом, на протяжении всей работы ГА у лучших решений популяции (с более высоким рангом) будет сохраняться значительное конкурентное преимущество, даже если вся популяция имеет примерно равное значение функции качества.

[3] Т.е. генетический алгоритм фактически вырождается в метод Монте-Карло.

7 Реализуйте в модели и протестируйте два других метода скрещивания — одноточечное и двухточечное.

8 Примените генетический алгоритм к решению одномерной задачи оптимизации, рассмотренной в главе 5. Т.к. в этой задаче геном представляется всего одним числом, а не списком чисел, то стандартные операторы скрещивания в данной ситуации не работают. Для таких случаев имеется свой вариант скрещивания, называемый *арифметическим скрещиванием*, когда значение каждого гена потомка x' вычисляется как взвешенное среднее генов x и y своих родителей:

$$x' \leftarrow \alpha x + (1-\alpha)y,$$

где α — некоторое заданное число из интервала $[0,1]$.

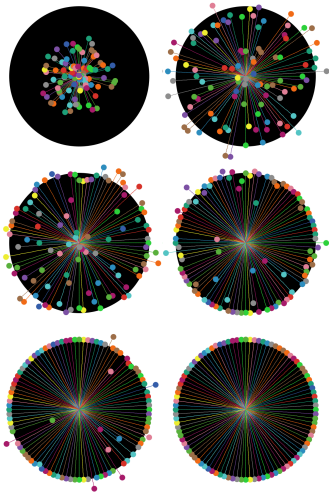

РИС. 10.10 Решение задачи о сумме элементов подмножества

9 Генетические алгоритмы с успехом применяются и в дискретной оптимизации. В простейшем случае решение ищется на множестве двоичных строк (последовательностей), т.е. каждый ген равен либо нулю, либо единице. Все рассмотренные нами генетические операторы работают и с таким кодированием решений. Существенно проще реализуется мутация — каждый ген с малой вероятностью инвертируется: $0 \leftrightarrow 1$. Примером NP-сложной оптимизационной задачи с двоичным кодированием является задача о сумме элементов подмножества. Задано множество A положительных чи-

сел и искомая сумма S. Требуется найти такое подмножество множества A, сумма элементов которого минимально отличалась бы от S. Т.к. решениями здесь являются подмножества, то естественным оказывается именно двоичное кодирование, когда каждый ген соответствует ровно одному элементу из A и указывает, входит этот элемент в подмножество (1) или нет (0). На рис. 10.10 приведен пример решения такой задачи с помощью генетического алгоритма. Каждое подмножество в данном случае представляется отрезком из начала координат, длина которого равна сумме этого подмножества. Черным цветом выделена окружность, соответствующая искомой сумме S.

10 Во многих задачах комбинаторной оптимизации естественным кодированием решений является кодирование перестановками из заданного числа элементов. Однако для перестановок не работает ни один из рассмотренных нами операторов скрещивания и мутации. Для реализации оператора мутации можно использовать схемы, показанные на рис. 5.19. Для скрещивания двух перестановок имеются свои методы, например метод Partially Mapped Crossover (PMX), который работает по следующей схеме[4]. Пусть заданы две перестановки x и y (см. пример на рис. 10.11). Делим их одинаковым образом на три части и центральную часть из y копируем в x' (как в двухточечном скрещивании). Далее перебираем все оставшиеся позиции в x (из первой и третьей частей). Если элемента в текущей позиции еще нет в x', то вставляем его на ту же позицию. Если есть, то смотрим, где он стоит в x', и анализируем по той же схеме соответствующий элемент из x (т.е. пытаемся вставить его в текущую позицию в x').

[4] J.-Y. Potvin, *Genetic algorithms for the traveling salesman problem*, Annals of Operations Research, 1996, 63, p. 339–370.

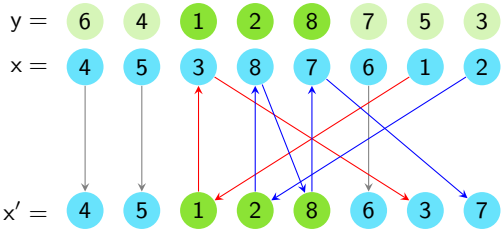

РИС. 10.11 Алгоритм скрещивания перестановок PMX

Реализуйте в NetLogo генетический алгоритм для решения задачи коммивояжера с использованием метода PMX, взяв за основу модель из главы 11.