Санкт-Петербургский государственный университет
Кафедра математической теории игр и статистических решений

Заковряшин Егор Михайлович

Выпускная квалификационная работа бакалавра

Многошаговые игры поиска

Направление 010400.62
Прикладная математика и информатика

Научный руководитель,
доктор физ.-мат. наук,
профессор
Петросян Л. А.

Санкт-Петербург

2016

Содержание

Введение

Поиск различных объектов издревле представляет собой одну из ключевых сторон человеческой активности. В работе [1] процесс поиска определяется как целенаправленное обследование конкретной области пространства для обнаружения находящегося там объекта. Обнаружением объекта можно считать получение информации о его местоположении путем установления с ним прямого энергетического контакта. Это вполне естественная проблема, с которой мы сталкиваемся на ежедневной основе. К примеру, перед тем как уходить утром на работу, необходимо собрать определенный набор вещей (ключи, кошелек, документы), которые в свою очередь необходимо локализовать. Их местоположение многообразно, и есть потребность свести количество времени, затраченное на поиск, к минимуму. Отходя от подобных бытовых проблем, можно расширить список задач до локализации поврежденных файлов компьютера, разведки месторождений нефти или поиска заблудившегося в лесу человека.

Одним из путей исследования поиска является построение и анализ математических моделей, отображающих объективные закономерности поиска и позволяющие установить связь между условиями его выполнения и его результатами, с целью дальнейшего анализа критериев эффективности поиска. Конечной целью теории поиска является установление оптимального способа ведения поисковых действий применительно к определенной обстановке и условиям проведения поиска. Выбор оптимального способа поиска основан на анализе математической модели, соответствующей поисковой ситуации, что позволяет локализовать объект поисков с минимальными затратами имеющихся ресурсов, ввиду их частой ограниченности.

В 1957 году в свет вышла работа Б. О. Купмана [7], в которой задачи поиска рассматривались как задачи оптимального распределения ре-

сурсов, направленных на поиск, т.е. односторонние задачи оптимизации. С момента выхода этой работы теория поиска получила широкое распространение, в результате чего появилось большое число публикаций, и стало общепринятым проводить разделение задач поиска на два класса: поиск неподвижного и поиск подвижного объектов. Также объект поиска может рассматриваться как пассивный (не оказывает сопротивления собственному обнаружению) и активный (оказывает сопротивление). Полное и систематическое изложение теории решения поисковых задач было приведено в работах [3, 6]. Проблематика оптимальности поиска затронута также в работах [4, 8].

В данной работе речь пойдет о многошаговой задаче поиска неподвижного пассивного объекта в дискретной среде, идеей для рассмотрения которой послужила статья [5]. Многошаговость задачи обусловлена повторяющейся процедурой поиска.

Постановка задачи

Опишем условия рассматриваемой задачи. Пусть дано дискретное множество местоположений объекта (в дальнейшем, коробок). Игрок H выбирает коробку и прячет в ней объект. Игрок S пытается обнаружить его. Событие, заключающееся в обнаружении объекта при просмотре верной коробки, не достоверное, т.е. объект может быть не найден, даже если он был в этой коробке. Каждый просмотр коробки требует затрат некоторых ресурсов (например, энергии или времени). Поиск продолжается до нахождения объекта или до исчерпания числа попыток. При нахождении предмета игрок S вознаграждается. Игрок S стремится свести затраты своих ресурсов к минимуму.

В ходе данной работы были поставлены следующие задачи:

1. Составить теоретико-игровую модель рассматриваемой задачи с обучающимся ищущим, т.е. ищущий располагает информацией, в каких коробках уже был произведен поиск, и на основе этой информации осуществляет следующий шаг. Поиск производится в одной коробке;
2. Аналитически решить данную задачу для случая двухшагового поиска объекта в двух коробках и проанализировать полученные результаты;
3. Обеспечить программную реализацию, способную решать данную задачу для k-шагового поиска объекта в N коробках;
4. Обобщить полученную модель на случай одновременного поиска объекта в нескольких коробках и разработать программное обеспечение для этого случая;

Глава 1. Теоретико-игровая модель простого поиска

1.1. Составление модели

Как говорилось выше, игрок H выбирает одну из N коробок, пронумерованных от 1 до N, и прячет в ней объект. Игрок S осуществляет поиск объекта в одной из них. Каждый просмотр i-й коробки, $i = 1, \ldots, N$, может быть осуществлен путем затрат ресурса $t > 0$. Также определим вероятность p_i, $0 < p_i \leq 1$, обнаружить объект в коробке i, при условии, что он в ней находится. Если игрок S находит объект, он получает вознаграждение в виде величины $a > 0$.

Смешанную стратегию игрока H обозначим за вектор $x = (x_1, \ldots, x_N)$, где $x_i \geq 0$, $i = 1, \ldots, N$; $\sum_{j=1}^{N} x_j = 1$. Каждое x_i обозначает вероятность того, что игрок H спрячет предмет в i-й коробке. Игрок S имеет представление о стратегии игрока H, т.е. ему известно значение вектора x. Зная его, он стремится пройти через серию поисков с минимальными затратами. Обозначим за $f_n(x)$ минимальные ожидаемые затраты игрока S на n-м шаге поиска и составим соответствующее нашим условиям уравнение минимальных ожидаемых затрат $f_n(x)$.

На каждом шаге игрок S платит ресурс t за право осуществить поиск в коробке под номером i. При этом он имеет возможность обнаружить в этой коробке искомый объект, при условии, что он там находится, и получить награду a с вероятностью $p_i x_i$. К ожидаемым затратам $f_n(x)$ следует добавить затраты предыдущего шага поиска $f_{n-1}(x)$, в том случае, если он оказался безрезультатным, вероятность чего равна $(1 - p_i x_i)$. Следуя принципу оптимальности Беллмана, изложенном в работе [2], будем искать минимальные ожидаемые затраты на шаге n в предположении оптимальности

всех последующих шагов. Таким образом имеем:

$$f_n(x) = \min_{1 \le i \le N} \Big[t - a p_i x_i + (1 - p_i x_i) f_{n-1}(C_i(x)) \Big], \qquad (1)$$

где $t > 0$, $0 < p_i \le 1$, $a > 0$ и $C_i(x) = (\xi_1, \ldots, \xi_N)$.

В данном случае вектор $C_i(x)$ играет роль обучения игрока S. Осуществляя неудачный поиск в какой-либо коробке, игрок запоминает, что в этой коробке предмета найдено не было и, в связи с этим, меняет приоритеты поиска в сторону остальных коробок. Строго говоря, вектор $C_i(x)$ являет собой апостериорную вероятность распределения объекта по коробкам при условии, что был произведен неудачный просмотр i-й коробки. Иными словами, вероятность того, что объект находится в i-й коробке после произведенного в ней неудачного поиска, уменьшится, в то время как вероятности нахождения объекта в остальных коробках увеличатся. Пользуясь формулой Байеса апостериорной вероятности, найдем значение вектора $C_i(x)$.

Для этого определим следующие события и найдем их вероятности:

- A_i - объект находится в коробке под номером i,

  $P(A_i) = x_i$;

- B - объект не был найден на текущем шаге при проверке i-ой коробки,

  $P(B) = 1 - p_i x_i$;

- $B \backslash A_i$ - объект не был найден на текущем шаге при проверке i-ой коробки, при условии, что находится в коробке под номером i,

  $P(B \backslash A_i) = 1 - p_i$,

  $P(B \backslash A_j) = 1, \qquad j \ne i.$

Событие, заключающееся в том, что предмет не был найден в коробке j при просмотре коробки i, достоверное. Этим объясняется последнее равенство.

Таким образом формула Байеса примет следующий вид:

$$\xi_k = P(A_k \backslash B) = \frac{P(B \backslash A_k) P(A_k)}{P(B)}, \quad k = 1, \ldots, N.$$

Подставив соответствующие значения вероятностей, определенных выше, найдем значение вектора $C_i(x) = (\xi_1, \ldots, \xi_N)$:

$$\begin{aligned} \xi_i &= x_i \frac{1 - p_i}{1 - p_i x_i}, \\ \xi_j &= x_j \frac{1}{1 - p_i x_i}, \qquad j \neq i \end{aligned} \tag{2}$$

Пользуясь уравнениями (1), (2) можно определить минимальные ожидаемые потери $f_n(x)$ игрока S на шаге n, в том случае, если он играет оптимально. Вместе со значением ожидаемых потерь можно также выяснить последовательность номеров коробок $i_1 \to i_2 \to \ldots \to i_n$, в которых производился поиск. Каждая такая последовательность номеров соответствует своему значению потерь игрока S, и наоборот. Значит, найдя минимальные ожидаемые потери игрока S, можно будет определить последовательность коробок, по которой необходимо производить поиски для получения этих потерь.

1.2. Аналитическое решение случая 2×2

Составим уравнения для двухшагового поиска объекта в двух коробках, воспользовавшись уравнениями (1), (2):

$$f_1(x) = \min_{1 \leq i \leq 2} \Big[t - a p_i x_i \Big];$$

$$f_2(x) = \min_{1 \leq i \leq 2} \Big[t - a p_i x_i + (1 - p_i x_i) f_1(C_i(x)) \Big],$$

где $f_1(x)$, $f_2(x)$ первый и второй шаги поиска соответственно, а вектор обучения имеет вид $C_i(x) = (\xi_1, \xi_2)$. Подставляя первое уравнение во второе,

получаем:

$$f_2(x) = \min_{1\le i\le 2}\Big[t - ap_ix_i + (1-p_ix_i)\min_{1\le i_1\le 2}(t - ap_{i_1}\xi_{i_1})\Big], \text{ где}$$
$$\xi_i = x_i\frac{1-p_i}{1-p_ix_i}, \tag{3}$$
$$\xi_j = x_j\frac{1}{1-p_ix_i}.$$

В зависимости от знака неравенства $t-ap_i\xi_i \vee t-ap_j\xi_j$ получаем два набора возможных значений функции (3).

Пусть $t-ap_i\xi_i > t-ap_j\xi_j$. Тогда игроку S выгодно будет открыть две разные коробки на протяжении поиска. Для этого случая имеем уравнение:

$$f_2(x) = \min_{1\le i\le 2}\Big[t - ap_ix_i + (1-p_ix_i)(t - ap_jx_j\frac{1}{1-p_ix_i})\Big] =$$
$$= \min_{1\le i\le 2}\Big[2t - p_ix_i(a+t) - ap_jx_j\Big] \tag{4}$$

В зависимости от того, при каком значении $i = \{1,2\}$ функция (4) принимает минимальное значение, получаем два возможных вида уравнения наименьших предполагаемых затрат:

$$f_2(x) = 2t - p_1x_1(a+t) - ap_2x_2, \tag{4'}$$
$$f_2(x) = 2t - p_2x_2(a+t) - ap_1x_1. \tag{4''}$$

В уравнении (4′) на первом шаге поиски велись во второй коробке, на втором — в первой. В уравнении (4″), соответственно, наоборот.

Пусть теперь $t-ap_i\xi_i < t-ap_j\xi_j$. В этом случае игроку S выгодно игнорировать одну из коробок в процессе поиска. Проведя аналогичные рассуждения, получаем уравнение наименьших предполагаемых затрат для этого случая:

$$f_2(x) = \min_{1\le i\le 2}\Big[t - ap_ix_i + (1-p_ix_i)(t - ap_ix_i\frac{1-p_i}{1-p_ix_i})\Big] =$$
$$= \min_{1\le i\le 2}\Big[2t - p_ix_i(2a+t) + ap_i^2x_i\Big] \tag{5}$$

И, соответствующие выбору $i = \{1, 2\}$ в функции (5), уравнения наименьших предполагаемых затрат:

$$f_2(x) = 2t - p_1 x_1 (2a + t) + a p_1^2 x_1 \tag{5'}$$

$$f_2(x) = 2t - p_2 x_2 (2a + t) + a p_2^2 x_2. \tag{5''}$$

В уравнении (5′) во время обоих шагов поиска игнорировалась вторая коробка. В уравнении (5″), соответственно, первая.

Пользуясь полученными выражениями, можно найти оптимальные стратегии игрока H. Для каждой функции $f_2(x)$ из уравнений (4′), (4″), (5′), (5″) найдем $\max\limits_x f_2(x)$ при условии, что $x_1 + x_2 = 1$:

1. $\max\limits_x f_2(x) = \max\limits_x (2t - p_1 x_1 (a+t) - a p_2 x_2)$. В этом случае, в зависимости от знака неравенства:

   $$p_1(a + t) - a p_2 \vee 0$$

   получаем оптимальные стратегии игрока H:

   $$x = (1, 0), \quad p_1(a + t) - a p_2 < 0;$$

   $$x = (0, 1), \quad p_1(a + t) - a p_2 > 0.$$

   Им соответствуют значения функции предполагаемых затрат:

   $$f_2(x) = 2t - p_1(a + t);$$

   $$f_2(x) = 2t - a p_2.$$

   Если же коэффициенты таковы, что получается равенство, то имеем бесконечное множество оптимальных стратегий:

   $$x_1 + x_2 = 1, \quad p_1(a + t) - a p_2 = 0;$$

2. $\max\limits_x f_2(x) = \max\limits_x(2t - p_2x_2(a+t) - ap_1x_1)$. Аналогично предыдущему пункту, в зависимости от знака неравенства

$$p_2(a+t) - ap_1 \vee 0$$

получаем оптимальные стратегии игрока H:

$$x = (0,1), \quad p_2(a+t) - ap_1 < 0.$$

$$x = (1,0), \quad p_2(a+t) - ap_1 > 0;$$

Им соответствуют значения функции предполагаемых затрат:

$$f_2(x) = 2t - p_2(a+t);$$

$$f_2(x) = 2t - ap_1.$$

Как и в предыдущем пункте, если получается равенство, то имеем бесконечное множество оптимальных стратегий:

$$x_1 + x_2 = 1, \quad p_2(a+t) - ap_1 = 0;$$

3. $\max\limits_x f_2(x) = \max\limits_x(2t - p_1x_1(2a+t) + ap_1^2x_1)$. Перепишем его в более наглядном виде:

$$\max_x f_2(x) = \max_x \big(2t - p_1x_1(a(2-p_1)+t)\big).$$

Видно, что коэффициент, стоящий у x_1 всегда неотрицательный. Значит, выбирая в качестве своей стратегии любой вектор x, за исключением вектора $(0,1)$, игрок H лишь преуменьшает свой выигрыш. Таким образом, получаем оптимальную стратегию $x = (0,1)$ и значение функции предполагаемых затрат принимает следующий вид:

$$f_2(x) = 2t.$$

4. $\max\limits_{x} f_2(x) = \max\limits_{x}(2t - p_2x_2(2a + t) + ap_2^2x_2)$. Аналогично предыдущему пункту, оптимальная стратегия принимает значение $x = (1, 0)$, а функция предполагаемых затрат:

$$f_2(x) = 2t.$$

Как видно из двух последних пунктов, итоговая функция прибыли игрока H зависит только от коэффициента t и ни от каких других. Более того, если игроку S безразлично в какой паре одинаковых коробок искать (серии поиска $1 \to 1$, $2 \to 2$), т.е. если выполняется равенство

$$ap_2^2x_2 - p_2x_2(2a + t) = ap_1^2x_1 - p_1x_1(2a + t),$$

то игроку H имеет смысл использовать смешанную стратегию:

$$x = (0.5, 0.5).$$

1.3. Программная реализация случая $N \times k$

Программная реализация поставленной задачи была произведена при помощи языка программирования C#. Основную сложность в процессе реализации составила способность к обучению игрока S, т.е. изменение вектора $x = (x_1, \ldots, x_N)$ распределения объекта по коробкам в ходе игры. Вследствие этого возникает большое количество различных векторов $C_i(x) = (\xi_1, \ldots, \xi_N)$, значение которых зависит не только от предыдущего выбора игрока S, но и от всех его действий с начала поиска. К примеру, для 3-го шага в задаче с N коробками приходится N^3 измененных значений x, каждое из которых связано с определенным значением $\check{x}$ со второго шага общей численностью в N^2, и так далее.

Самая очевидная интерпретация, приходящая на ум, это N-арные деревья высоты k, т.е. набор связных ациклических графов, из каждой

вершины которых выходит по N ветвей. Корневой вершиной i-го графа является i-я компонента вектора начальных смешанных стратегий игрока H: $x = (x_1, \ldots, x_N)$.

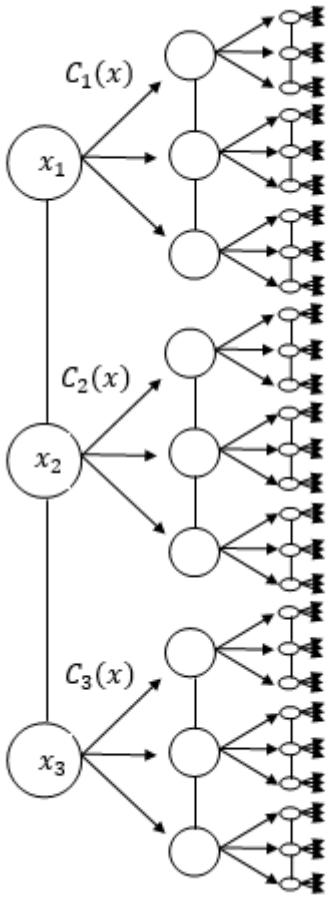

Рис. 1: N-арные деревья высоты 3, $N = 3$.

Сплошные вертикальные линии, соединяющие некоторые вершины графа на Рис.1, указывают на компоненты вектора $(x_1, \ldots, x_N)$, которые были задействованы в составлении нового вектора $(\xi_1, \ldots, \xi_N)$.

В основу программной реализации была положена именно эта идея, с целью избежания возможной путаницы в значениях векторов смешанных стратегий и их взаимосвязях.

На вход программы подаются следующие значения:

N — число задействованных в задаче коробок;

k — количество шагов;

$x = (x_1, \ldots, x_N)$ — смешанная стратегия игрока H;

$p = (p_1, \ldots, p_N)$ — вероятность обнаружить объект при просмотре;

$a > 0, t > 0$ — выигрыш и затраты на поиск.

Для удобства хранения информации о состояниях новых векторов $C_i(x) = (\xi_1, \ldots, \xi_N)$ был описан класс Probability:

```
public class Probability
{
    public double Value { get; set; }
    public short Level { get; set; }
    public short Index { get; set; }
    public Probability Parent { get; set; }
    public double Result { get; set; }
}
```

Каждый элемент этого класса — это компонента ξ_j вектора $C_i(x) = (\xi_1, \ldots, \xi_N)$. Свойства класса определяют следующие значения:

- $Value$ — значение компоненты ξ_j;
- $Level$ — номер шага k, на котором определяется значение компоненты ξ_j;
- $Index$ — значение индекса j, компоненты ξ_j;
- $Parent$ — выбор игрока S на предыдущем шаге. Этот параметр также является элементом класса Probability и определяет собой компоненту x_i, которая в свою очередь определяет значение текущего вектора $C_i(x) = (\xi_1, \ldots, \xi_N)$;
- $Result$ — значение функции (1) при подстановке компоненты ξ_j на k-ом шаге.

Хранение элементов класса происходит в списке probabilities. Создание дерева элементов класса осуществляется при помощи метода GenerateLevel:

```
public static void GenerateLevel(int level, int n)
{
    foreach (var c in probabilities.Where(x => x.Level == level).ToArray())
    {
        for (int i = 0; i < n; i++)
        {
            var newProbability = new Probability()
            {
                Level = (short)(level + 1),
                Index = (short)i,
                Parent = c,
            };
            probabilities.Add(newProbability);
        }
    }
}
```

На вход метода $GenerateLevel$ подаются два значения: $level$ — значение шага задачи k, для которого создаются вектора $C_i(x) = (\xi_1, \ldots, \xi_N)$ и n — количество коробок. После этого, для каждого элемента списка, чье значение параметра $Level$ соответствует переданному в метод, создаются N новых элементов с соответствующими значениями параметров $Index = 1, \ldots, N$ и значением $Level = k+1$. Каждому новому "дочернему" элементу назначается ссылка на "родительский" для сохранения связи между элементами. К примеру, в случае вектора $C_i(x) = (\xi_1, \ldots, \xi_N)$ элемент x_i является "родительским" для каждой из компонент ξ_j, $j = 1, \ldots, N$.

После создания дерева элементов происходит заполнение параметра $Value$ каждого элемента. Для этого используется метод FillLevel. Заполнение параметра $Value$ происходит в соответствии с формулами (2) и учитывает особенности хранения элементов класса Probability в списке probabilities. Если значения индексов "дочернего" и "родительского" элементов совпадают, то заполнение происходит по первой формуле с исполь-

зованием метода $CEquals$. В противном случае используется вторая формула и метод $CNotEquals$.

```
public static void FillLevel()
{
    for (int i = 1; i <= k; i++)
    {
        var i1 = i;
        foreach (var item in probabilities.Where(x => x.Level == i1))
        {
            if (item.Index == item.Parent.Index)
            {
                item.Value = CEquals(item);
            }
            else
            {
                item.Value = CNotEquals(item);
            }
        }
    }
}

public static double CEquals(Probability c)
{
    return c.Parent.Value * (1.0 - p[c.Parent.Index]) /
                / (1 - p[c.Parent.Index] * c.Parent.Value);
}


public static double CNotEquals(Probability c)
{
    var diff = c.Index - c.Parent.Index;
    var position = probabilities.IndexOf(c.Parent) + diff;
    return probabilities[position].Value /
                / (1 - p[c.Parent.Index] * c.Parent.Value);
}
```

Заполнение параметра $Result$ каждого элемента класса происходит в методе Main (см. Приложение 1). Для наибольшего значения параметра

$Level = k$ вычисляются все возможные значения функции (1) на первом шаге задачи. После чего для значения параметра $Level = k-1$ вычисляются значения данной функции на втором шаге, основываясь на минимальной комбинации "дочерних" к этому шагу элементов и т.д. Результатом данных операций будет являться значение минимальных ожидаемых затрат (1), вычисленное на последнем шаге задачи. После получения этого значения, алгоритм еще раз проходит по дереву элементов и собирает данные об оптимально-выбираемых коробках на каждом шаге.

Таким образом, на выходе программы имеются данные о последовательности оптимального выбора коробок $i_1 \to i_2 \to \ldots \to i_k$, в соответствии с которой вычисляется значение минимальных ожидаемых затрат $f_k(x)$ игрока S.

Первоначально программная реализация задумывалась в том числе и для численного исследования возможных свойств функции минимальных ожидаемых затрат, к примеру, ограниченности, монотонности, наличия предела и т.п. К сожалению, в ходе работы над программной реализацией от данной идеи пришлось отказаться ввиду чрезвычайной ресурсоемкости алгоритма программы. Для того чтобы решить поставленную задачу поиска объекта среди N коробок за k шагов, необходимо запоминать и оперировать $n = 5 \cdot \sum_{i=1}^{k} N^i$ количеством переменных. Множитель "5" отвечает за количество параметров элемента класса $Probability$. Как любая сумма геометрической прогрессии, n очень быстро возрастает, что приводит к неэффективной работе алгоритма по причине нехватки памяти. Например, при значениях $k = 9, N = 5$ число переменных достигает $n = 5 \cdot \sum_{i=1}^{9} 5^i = 2441405$.

Каким-либо образом сократить число переменных не представляется возможным, т.к. это повлечет за собой потерю части решения, в ко-

торой может оказаться искомое оптимальное значение $f_k(x)$. Таким образом, решить данную проблему можно только за счет увеличения мощности электронно-вычислительной машины либо путем большей оптимизации существующего алгоритма. За невозможностью этого, было решено не осуществлять численное исследование, а ограничиться расчетом нескольких примеров.

1.4. Результаты программной реализации

Ниже приведены результаты работы алгоритма для различных входных данных, в том числе и для данных проверки работы программы.

Пример 1. Две коробки, два шага

Для проверки правильности работы алгоритма программы использовались следующие входные данные:

$$t = 3, \quad a = 5, \quad p = (0.4, 0.7), \quad x = (0.6, 0.4).$$

На выходе программы было получено следующее значение минимальных ожидаемых затрат игрока S и соответствующая ему оптимальная последовательность проверяемых коробок:

$$f_2(x) = 2.56; \quad 1 \to 2,$$

что соответствует аналитическому решению данной задачи, рассчитанному по формуле (1). Совпадают также оба значения измененных векторов распределения объекта по коробкам:

$$C_1(x) = (0.474, 0.526);$$

$$C_2(x) = (0.833, 0.167),$$

вычислить которые не составляет труда, используя формулы (2).

На основе совпадения конечных, а также промежуточных значений можно сделать вывод о правильности работы алгоритма программы.

Пример 2. Три коробки, девять шагов

Рассмотрим задачу, где игрок S проводит поиски объекта среди трех коробок за девять шагов со следующими параметрами:

$$t = 5, \quad a = 8, \quad p = (0.8, 0.6, 0.7),$$

$$x = (0.15, 0.75, 0.1).$$

Следует заметить, что игрок S осведомлен, о том что объект наиболее вероятно расположен во 2-й коробке, и вероятность найти его там довольно высока. В результате работы программы получаем следующие значения минимальных затрат и оптимального пути поиска:

$$f_9(x) = 5.304; \quad 1 \to 2 \to 1 \to 2 \to 2 \to 2 \to 2 \to 2 \to 2.$$

По полученным результатам видно, что разумный игрок S отдаст предпочтение именно 2-ой коробке, иногда отвлекаясь на проверку 1-ой. 3-я коробка игнорируется ввиду малой вероятности найти в ней объект. Данные результаты ожидаемы и логичны.

Теперь поменяем исходные условия задачи, а именно, изменим вероятность обнаружения объекта во 2-й коробке:

$$p = (0.8, 0.1, 0.7).$$

Теперь объект в ней довольно трудно обнаружить, хотя игрок S знает, о том что он наиболее вероятно там. В результате работы программы получаем следующие значения:

$$f_9(x) = 26.389; \quad 1 \to 3 \to 1 \to 3 \to 1 \to 2 \to 1 \to 2 \to 1.$$

По полученным результатам видно, что, несмотря на довольно большую вероятность расположения объекта во 2-й коробке, разумный игрок S проверяет ее всего-лишь дважды, отдавая предпочтение 1-й коробке, в которой найти объект проще. Также следует обратить внимание на отличие минимальных ожидаемых затрат в обеих задачах. Во втором случае оно в 5 раз больше, что объясняется маленькой вероятностью получения награды на каждом шаге.

Пример 3. Шесть коробок, шесть шагов

Рассмотрим "шарлатанскую задачу", в которой игрок S будет точно знать, что объект находится в i-й коробке, но при этом найти его в ней будет невозможно, в то время как в других коробках он бы нашел объект практически моментально, будь он там. Условия следующие:

$$t = 4, \quad a = 2, \quad p = (0.96, 0.98, 0.9, 0, 0.99, 0.95),$$

$$x = (0, 0, 0, 1, 0, 0).$$

Результатом работы алгоритма программы имеем:

$$f_6(x) = 24; \quad 1 \to 1 \to 1 \to 1 \to 1 \to 1.$$

Алгоритм предложил просматривать 1-ую коробку раз за разом, лишь потому что она первая, а не из-за каких-либо преимуществ ее осмотра. Видно, что программа не способна дать логичный ответ в подобной, не имеющей смысла, задаче. Изменим условия задачи на чуть более честные:

$$p = (0.96, 0.98, 0.9, 0.01, 0.99, 0.95).$$

Теперь игрок S имеет возможность найти предмет в 4-й коробке, пусть весьма призрачную. В результате получим:

$$f_6(x) = 23.2909; \quad 4 \to 4 \to 4 \to 4 \to 4 \to 4.$$

Получив возможность обнаружить предмет в данной коробке, игрок S вполне логично начнет проверять лишь ее. Попробуем изменить начальную смешанную стратегию игрока H таким образом, чтобы игрок S изменил свой оптимальный путь поиска:

$$x = (0, 0.01, 0, 0.98, 0.01, 0).$$

И в результате получим полное изменение оптимального пути поиска:

$$f_6(x) = 23.2949; \quad 2 \to 5 \to 2 \to 5 \to 2 \to 5.$$

Примечательно, что в этом случае значение минимальных затрат больше на 0.004 единиц в сравнении с предыдущим.

Пример 4. Пять коробок, k шагов

Данный пример призван показать различие в выгоде игрока S в зависимости от количества шагов в задаче. Рассмотрим задачу со следующими начальными данными:

$$t = 5, \quad a = 4, \quad p = (0.62, 0.74, 0.75, 0.63, 0.69),$$

$$x = (0.23, 0.18, 0.17, 0.22, 0.2).$$

Произведем расчеты для значений $k = \overline{2,5}$, после чего изменим значение выигрыша игрока S и проведем аналогичные расчеты для $a = 32$. Полученные данные представлены в таблице 1. Отрицательное значение затрат игрока S будем называть его выгодой. Убывание или возрастание последовательности $\{f_k(x)\}_{k=2}^{5}$ зависит от значения a при прочих равных. К сожалению, вычислительной мощности не хватает для отслеживания поведения последовательности для значений $k > 5$. Однако уже для известных значений можно сказать, что существует такая задача с набором коэффициентов a, t, p, x, для которых игроку S оптимально "затягивать" процесс

Таблица 1: Значения $f_k(x)$ для различных a

	$a = 4$	$a = 32$
$f_2(x)$	8.16	0.29
$f_3(x)$	11.2	-0.53
$f_4(x)$	13.58	-1.89
$f_5(x)$	15.3	-3.73

поиска, т.е. игрок S получит больше выгоды, если будет искать объект дольше (т.к. затраты отрицательны и увеличиваются по модулю). Опять же, ввиду нехватки вычислительных возможностей, невозможно сказать, существует ли у игрока S ограничение по возможной выгоде, или, иначе, существует ли у убывающей последовательности $\{f_k(x)\}_{k=2}^{\inf}$ предел снизу и монотонна ли она.

Глава 2. Теоретико-игровая модель множественного поиска

Данная модель отличается от рассмотренной в первой главе тем, что в ней игрок S имеет право просматривать одновременно m коробок, т.е. за один шаг поиска игрок S ищет объект не в одной, а в m коробках. Для простоты понимания сначала рассмотрим вариант с одновременным поиском в двух коробках, после чего обобщим его на случай m.

2.1. Единовременный поиск в двух коробках

Основное уравнение предполагаемых затрат претерпевает не слишком радикальные изменения. Как и в случае с поиском в одной коробке, игрок S платит ресурс t за право просмотреть коробку под номером i. Так как игрок S имеет право поискать сразу в двух коробках под номерами $i, j = 1, \ldots, N$, $i \neq j$, он затратит $2t$ ресурсов за один шаг. При этом вероятность получить награду a на этом шаге возрастает до значения $p_i x_i + p_j x_j$. Если поиск оказался неудачным, игрок S запомнит это и изменит вероятностное распределение объекта по коробкам, используя функцию $C_{ij}(x)$. Добавляя затраты, полученные на предыдущем шаге поиска в случае его бесплодности, получаем уравнение минимальных ожидаемых затрат игрока S в этой задаче на n-ом шаге поиска:

$$f_n^2(x) = \min_{1 \leq i,j \leq N} \Big[2t - a(p_i x_i + p_j x_j) + P(B) f_{n-1}(C_{ij}(x)) \Big] \qquad (6)$$

Здесь P(B) - это вероятность того, что на текущем шаге объект не был найден за оба просмотра коробок i и j. Верхний индекс $f_n^2(x)$ указывает на то, что поиск ведется в двух коробках одновременно. Вектор $C_{ij}(x) = (\xi_1, \ldots, \xi_N)$ также остается вектором апостериорной вероятности распределения объекта по коробкам в случае неудачного шага поиска. Однако в отличие от простого поиска, выражения ξ_i, $i = 1, \ldots, N$, имеют куда

более громоздкий вид. Воспользуемся формулой Байеса апостериорной вероятности и определим с его помощью ξ_i.

Обусловим события:

- A_i - объект находится в коробке под номером i;
- B - объект не был найден на текущем шаге при поиске в коробоках i, j;
- $B\backslash A_i$ - объект не был найден на текущем шаге в коробках i, j, при условии, что находится в коробке под номером i;
- $A_i\backslash B$ - объект находится в коробке под номером i, при условии, что он не был найден на текущем шаге при поиске в коробках i, j;

Тогда, формула Байеса принимает вид:

$$\xi_k = P(A_k\backslash B) = \frac{P(B\backslash A_k)P(A_k)}{P(B)}, \quad k = 1, \ldots, N, \tag{7}$$

где $P(B) = \sum_{k=1}^{N} P(B\backslash A_k)P(A_k)$ - полная группа событий.
Найдем вероятность каждого из обусловленных событий:

- $P(A_i) = x_i$ - вероятность того, что объект находится в коробке i, $i = 1, \ldots, N$
- $P(B\backslash A_i) = 1 - p_i$ - вероятность не найти объект при поиске в i-й коробке;
- $P(B\backslash A_j) = 1 - p_j$ - вероятность не найти объект при поиске в j-й коробке;
- $P(B\backslash A_k) = 1$ - вероятность не найти объект при поиске в k-й коробке, $k \neq i, j$.

Событие, заключающееся в том, что объект не был найден в коробке k, в то время как поиск в ней не производился, достоверное. Иными словами, если не проверить коробку, вероятность найти в ней объект равняется нулю. Этим объясняется последнее равенство.

Пользуясь определенными выше вероятностями, можно найти вероятность неудачного поиска в коробках i, j на текущем шаге, т.е. $P(B)$:

$$P(B) = x_i(1 - p_i) + x_j(1 - p_j) + \sum_{k \neq i,j} x_k.$$

В итоге, подставив известные значения в (7), получим измененную вероятность распределения объекта по коробкам:

$$\xi_i = P(A_i \backslash B) = \frac{x_i(1 - p_i)}{x_i(1 - p_i) + x_j(1 - p_j) + \sum\limits_{k \neq i,j} x_k};$$

$$\xi_j = P(A_j \backslash B) = \frac{x_j(1 - p_j)}{x_i(1 - p_i) + x_j(1 - p_j) + \sum\limits_{k \neq i,j} x_k}; \tag{8}$$

$$\xi_k = P(A_k \backslash B) = \frac{x_k}{x_i(1 - p_i) + x_j(1 - p_j) + \sum\limits_{k \neq i,j} x_k}, \quad k \neq i, j$$

Перезапишем формулу (6), используя полученные обозначения:

$$f_n^2(x) = \min_{1 \leq i,j \leq N} \Big[2t - a(p_i x_i + p_j x_j) + \\ \big(x_i(1 - p_i) + x_j(1 - p_j) + \sum_{k \neq i,j} x_k\big) \cdot f_{n-1}^2\big(C_{ij}(x)\big)\Big], \tag{6'}$$

где вектор $C_{ij}(x) = (\xi_1, \ldots, \xi_N)$ определяется уравнениями (8).

2.2. Единовременный поиск в m коробках

Пользуясь данными полученными в предыдущем разделе, обобщим модель на случай единовременного поиска объекта в m коробках. Обозначим за $M \subset N$ подмножество коробок, в которых будет вестись поиск на

одном шаге, $M = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$. За один шаг поисков игрок S должен затратить mt ресурсов для поиска в m коробках, в то время как вероятность получить награду a равняется $\sum\limits_{i\,:\,x_i \in M} p_i x_i$. Если результаты поисков оказались бесплодными, игрок S изменит вероятностное распределение объекта по коробкам при помощи функции $C_M(x)$. Тогда функция предполагаемых минимальных затрат примет следующий вид:

$$f_n^m(x) = \min_{M \subset N} \Big[mt - a \sum_{i\,:\,x_i \in M} p_i x_i + P(B) f_{n-1}(C_M(x)) \Big]. \tag{9}$$

Как и в случае с поиском в двух коробках, $P(B)$ обозначает вероятность того, что проверка m коробок оказалася неудачной. По аналогии с предыдущим случаем, найдем выражение, определяющее $P(B)$:

$$P(B) = \sum_{k=1}^{N} P(B \backslash A_k) P(A_k) = \sum_{i\,:\,x_i \in M} x_i(1 - p_i) + \sum_{j\,:\,x_j \notin M} x_j;$$

А с его помощью, по аналогии с предыдущим пунктом, найдем выражения, определяющие $C_M(x) = (\xi_1, \ldots, \xi_N)$, воспользовавшись формулами Байеса:

$$\begin{aligned} \xi_l &= \frac{x_l(1 - p_l)}{\sum\limits_{i : x_i \in M} x_i(1 - p_i) + \sum\limits_{j : x_j \notin M} x_j}, \quad l : x_l \in M; \\ \xi_k &= \frac{x_k}{\sum\limits_{i : x_i \in M} x_i(1 - p_i) + \sum\limits_{j : x_j \notin M} x_j}, \quad k : x_k \notin M. \end{aligned} \tag{10}$$

Компоненты ξ_i, $i = 1, \ldots, N$ будут расчитываться по одной из этих двух формул в зависимости от того, производился поиск в i-й коробке на данном шаге или нет.

Перезапишем формулу (9), используя полученные выражения:

$$f_n^m(x) = \min_{M \subset N} \Big[mt - a \sum_{i\,:\,x_i \in M} p_i x_i + \Big(\sum_{i\,:\,x_i \in M} x_i(1 - p_i) + \sum_{j\,:\,x_j \notin M} x_j \Big) \cdot f_{n-1}\big(C_M(x)\big) \Big], \tag{9'}$$

где вектор $C_M(x) = (\xi_1, \ldots, \xi_N)$ определяется уравнениями (10):

2.3. Программная реализация множественного поиска

Разработка программного алгоритма на случай одновременного поиска объекта в m коробках не была доведена до конца по причине запутанной структуры взаимосвязей векторов $x = (x_1, \cdots, x_N)$ и $C_M(x) = (\xi_1, \ldots, \xi_N)$. Каждая компонента ξ_i, $i = 1, \ldots, N$, определяется набором компонент $M = (x_{i_1}, \ldots, x_{i_m})$ вектора x. Таким образом, получаем

$$C_N^m \cdot N, \text{ где } C_N^m \text{ — количество сочетаний,}$$

различных вариаций векторов $C_M(x) = (\xi_1, \ldots, \xi_N)$ на следующем шаге, каждый из которых связан со своим набором M.

Ввиду нехватки времени, было решено произвести разработку программного алгоритма для частного случая $m = 2$ и рассмотреть численные решения некоторых игр.

На вход программы подаются те же значения, что и в случае алгоритма простого поиска. Класс $Probability$, отвечающий за хранение векторов $C_{ij}(x) = (\xi_1, \ldots, \xi_N)$ претерпел незначительные изменения:

```
public class Probability
{
    public double Value { get; set; }
    public short Level { get; set; }
    public short Index { get; set; }
    public Probability Parent1 { get; set; }
    public Probability Parent2 { get; set; }
}
```

В нем появилась пара свойств $Parent1$, $Parent2$, отвечающих за хранение компонент x_i, x_j, которые являются определящими для вектора $C_{ij}(x) = (\xi_1, \ldots, \xi_N)$. Отсутствует также свойство $Result$ для того, чтобы избежать его возможной перезаписи по причине того, что количество возможных

значений функции (6′) на каждом шаге не меньше количества коробок:

$$C_N^2 \geq N.$$

Для хранения значений функции (6′) был описан класс *Results*:

```
public class Results
{
    public short Level { get; set; }
    public short Index1 { get; set; }
    public short Index2 { get; set; }
    public Probability Parent1 { get; set; }
    public Probability Parent2 { get; set; }
    public double Result { get; set; }
}
```

Свойства класса определяют следующие значения:

- *Level* — шаг k, на котором определяется значение функции затрат (6′);
- $Index1$ — значение индекса i, компоненты x_i, для которой рассчитывалось значение (6′);
- $Index2$ — соответственно, значение индекса j, компоненты x_j;
- $Parent1$ — первая выбранная коробка на предыдущем шаге;
- $Parent2$ — вторая выбранная коробка на предыдущем шаге;
- *Result* — значение функции (6′) при подстановке компонент x_i, x_j на текущем шаге.

Создание дерева элементов класса осуществляется по аналогии с простым поиском при помощи метода GenerateLevel:

```
public static void GenerateLevel(int level, int n)
{
```

```
    foreach (var first in probabilities.Where(x => x.Level == level).ToArray())
    {
        for (int j = 1; j < n - first.Index; j++)
        {
            var second = probabilities.IndexOf(first) + j;
            for (int i = 0; i < n; i++)
            {
                var newProbability = new Probability()
                {
                    Level = (short)(level + 1),
                    Index = (short)i,
                    Parent1 = first,
                    Parent2 = probabilities[second]
                };
                probabilities.Add(newProbability);
            }
        }
    }
}
```

Единственное отличие здесь в том, что "дочерние" элементы создаются для всех сочетаний C_N^2 элементов текущего шага задачи. Таким образом, каждому новому элементу ставится в соответствие пара "родительских" элементов. Например, в случае вектора $C_{ij}(x) = (\xi_1, \ldots, \xi_N)$, элементы x_i, x_j являются "родительскими" для каждой из компонент ξ_i, $i = 1, \cdots, N$.

После создания дерева элементов происходит заполнение параметра $Value$ каждого элемента. Для этого используется метод FillLevel:

```
public static void FillLevel()
{
    for (int i = 1; i <= k; i++)
    {
        var i1 = i;
        foreach (var item in probabilities.Where(x => x.Level == i1))
        {
```

```
            if (item.Index == item.Parent1.Index)
            {
                item.Value = CEquals1(item);
            }
            else if (item.Index == item.Parent2.Index)
            {
                item.Value = CEquals2(item);
            }
            else
            {
                item.Value = CNotEquals(item);
            }
        }
    }
}
```

Заполнение параметра $Value$ происходит аналогично алгоритму простого поиска в соответствии с формулами (8). Если значение индекса "дочернего" элемента совпадает с одним из "родительских", то заполнение происходит по соответствующей формуле из (8) с использованием метода $CEquals1$ или $CEquals2$. В противном случае, используется метод $CNotEquals$:

```
public static double CEquals1(Probability c)
{
    return c.Parent1.Value * (1.0 - p[c.Parent1.Index]) / Divider(c);
}

public static double CEquals2(Probability c)
{
    return c.Parent2.Value * (1.0 - p[c.Parent2.Index]) / Divider(c);
}

public static double CNotEquals(Probability c)
{
    var diff = c.Index - c.Parent1.Index;
```

```
    var position = probabilities.IndexOf(c.Parent1) + diff;
    return probabilities[position].Value / Divider(c);
}
```

Каждый из вышеперечисленных методов использует метод *Divider*, который рассчитывает соответствующий формулам (8) делитель. Подобного метода не было в случае одинарного поиска ввиду простоты делителя в формулах (2).

```
public static double Divider(Probability c)
{
    double summ=0;
    for (int i = 0; i < n; i++)
    {
        if (i != c.Parent1.Index && i != c.Parent2.Index)
        {
            var diff = i - c.Parent1.Index;
            var position = probabilities.IndexOf(c.Parent1) + diff;
            summ += probabilities[position].Value;
        }
    }
    summ += (1 - p[c.Parent1.Index]) * c.Parent1.Value;
    summ += (1 - p[c.Parent2.Index]) * c.Parent2.Value;
    return summ;
}
```

Заполнение параметра *Result* каждого элемента класса *Results* происходит в методе Main (см. Приложение 2). Для наибольшего значения параметра $Level = k$ вычисляются все возможные значения функции $(6')$ на первом шаге. После чего для значения параметра $Level = k - 1$ вычисляются значения данной функции на втором шаге, основываясь на минимальной комбинации "дочерних" к этому шагу элементов и т.д. Результатом данных операций будет являться значение минимальных ожидаемых затрат $(6')$, вычисленное на последнем шаге задачи. После получения этого значения, алгоритм еще раз проходит по дереву элементов и собирает

данные об оптимально-выбираемых коробках на каждом шаге.

Таким образом, на выходе программы имеются данные о последовательности оптимального выбора пар коробок $(i_1, j_1) \to (i_2, j_2) \to \ldots \to (i_k, j_k)$, и соответствующее ей значение минимальных ожидаемых затрат $f_k^2(x)$ игрока S.

Как и в случае с простым поиском, алгоритм реализации одновременного поиска в двух коробках является очень ресурсоемким. Для решения поставленной задачи необходимо запоминать и оперировать:

$$n = \sum_{i=1}^{k} \left[5N \cdot \left(C_N^2\right)^{i-1} + 6 \cdot \left(C_N^2\right)^{i}\right]$$

количеством переменных, что сильно ограничивает диапазон доступных решению задач. Первое слагаемое в сумме отвечает за хранение элементов класса *Probability*, второе — за хранение элементов класса *Results*.

2.4. Результаты программной реализации

Ниже приведены результаты работы алгоритма для различных входных данных, в том числе и для данных проверки работы программы.

Пример 1. Три коробки, два шага

Для проверки правильности работы алгоритма программы использовались следующие входные данные:

$$t = 5,\ a = 8,\ p = (0.5, 0.5, 0.5),\ x = (0.3, 0.5, 0.2).$$

Аналитическое решение уравнения $(6')$ для значений $N = 3,\ k = 2$, приближенно (с точностью до сотых) определяет следующие значения измененных векторов $C_{i,j}(x) = (\xi_1, \xi_2, \xi_3)$ при переходе с первого шага на второй:

$$C_{12}(x) = (0.25, 0.42, 0.33),$$

$$C_{13}(x) = (0.2, 0.67, 0.13),$$

$$C_{23}(x) = (0.46, 0.38, 0.16).$$

Ожидаемые минимальные затраты и соответствующая им последовательность выбора пар коробок равны соответственно:

$$f_2^2(x) = 11; \quad (2,\ 3) \to (1,\ 2).$$

Данные значения полностью совпадают с теми, что были получены в результате программной реализации, включая промежуточные значения, которые были опущены в данном рассуждении, например, значения функций $f_1^2(C_{ij}(x))$, где $ij = \{12,\ 13,\ 23\}$. Исходя из полного совпадения аналитического и численного решений можно сделать вывод, что алгоритм программы работает правильно.

Пример 2. Четыре коробки, пять шагов

Рассмотрим игру, где игрок S проводит поиски объекта среди четырех коробок за пять шагов со следующими параметрами:

$$t = 10,\ a = 5,\ p = (0.6, 0.65, 0.71, 0.71),\ x = (0.25, 0.28, 0.23, 0.24).$$

В результате работы программы получаем следующие значения минимальных затрат и оптимального пути поиска:

$$f_4^2(x) = 41.464; \quad (3,\ 4) \to (1,\ 3) \to (2,\ 4) \to (1,\ 3) \to (2,\ 4).$$

По полученным результатам видно, что в данной задаче, где вероятности обнаружения объекта в каждой коробке приблизительно равны, на каждом шаге игроку S выгодно просматривать отличную от предыдущего шага пару коробок, за исключением первых двух шагов.

Изменим условие задачи. Сместим вероятность расположения объекта в сторону 1-й коробки, т.е.:

$$x = (0.75, 0.15, 0.05, 0.05).$$

В результате получаем:

$$f_4^2(x) = 32.927; \quad (1,\,2) \to (1,\,4) \to (1,\,2) \to (1,\,3) \to (1,\,2).$$

Видно, что на каждом шаге игрок S просматривает 1-ую коробку, варьируя при этом выбор оставшейся, чаще отдавая предпочтение 2-ой коробке.

Хотелось бы иметь возможность решить данную задачу для большего числа шагов, что невозможно ввиду переполнения памяти. А следовательно, для задач с еще большим числом коробок можно будет находить и анализировать решения только для числа шагов, не превосходящего пяти, что сильно ограничивает класс доступных для решения задач.

Пример 3. Три коробки, k шагов

Вследствие выводов, полученных в предыдущем примере, уменьшим число просматриваемых коробок для увеличения числа доступных шагов. Рассмотрим задачу, где игрок S проводит поиски объекта среди трех коробок со следующими параметрами:

$$t = 3,\ a = 5,\ p = (0.55, 0.6, 0.65),\ x = (0.43, 0.36, 0.31).$$

Произведем рассчеты функции минимальных ожидаемых затрат для значений $k = \overline{1,9}$ и рассмотрим прирост функции затрат на каждом шаге. Как видно из таблицы 2, прирост функции оптимальных затрат монотонно убывает с возрастанием числа шагов. На основе этого можно сделать один из следующих выводов:

1. Если $f_k^2(x) - f_{k-1}^2(x) > 0, \quad \forall k = 1, 2, \ldots$, то в данной задаче последовательность $\{f_k^2(x)\}_{k=1}^{\inf}$ монотонно возрастает к существующему верхнему пределу;

2. Если $\exists\, k^* : f_{k^*}^2(x) - f_{k^*-1}^2(x) < 0$, то в данной задаче ожидаемые минимальные затраты начинают убывать при $k = k^*$.

Таблица 2: Значения $f_k(x)^2$ для различных $k,\ a = 5$.

k	$f_k^2(x)$	$f_k^2(x) - f_{k-1}^2(x)$
1	3.738	—
2	6.083	2.345
3	7.336	1.235
4	8.019	0.683
5	8.389	0.37
6	8.596	0.207
7	8.707	0.111
8	8.768	0.061
9	8.800	0.032

Наиболее вероятен первый исход по причине того, что последовательность:

$$\{f_k^2(x) - f_{k-1}^2(x)\}_{k=2}^{\inf},$$

убывает по модулю с ростом k.

Изменив условие данной задачи на более выигрышное для игрока S, т.е. $a = 20$, и проведя аналогичные расчеты, придем к подобным результатам, представленным в таблице 3.

Таблица 3: Значения $f_k(x)^2$ для различных k, $a = 20$.

k	$f_k^2(x)$	$f_k^2(x) - f_{k-1}^2(x)$
1	-3.05	—
2	-5.324	-2.274
3	-6.425	-1.101
4	-6.978	-0.553
5	-7.301	-0.323
6	-7.445	-0.144
7	-7.549	-0.104
8	-7.599	-0.050
9	-7.629	-0.030

Отличие здесь в том, что выгода игрока S от продолжения поисков ограничена, либо начинает убывать после какого-то номера $k = k^*$. Но в целом, игрок S в новой задаче ограничен в выгоде, в то время как в старой был ограничен в затратах.

Выводы

В результате анализа результатов численного решения ряда задач были сделаны следующие выводы:

- Игроку S не всегда выгодно отдавать предпочтение проверке той коробки, в которой объект наиболее вероятно находится, а порой выгодно и вовсе игнорировать эту коробку в процессе поисков;
- Существуют задачи, в которых игроку S выгодно искать объект как можно дольше, т.к. его выгода от этого только лишь увеличится. Численно установить наличие предела выгоды в подобных задачах не удалось ввиду недостатка ресурсоемкости компьютера;
- Существуют задачи с одновременным поиском в двух коробок, в которых прирост функции затрат (выгоды) игрока S с ростом шагов является убывающей по модулю последовательностью, т.е. затраты (выгода) ограничены. Справедливость этого утверждения для большого числа шагов в данных задачах установить не удалось.

Кроме того, в результате аналитического решения задачи поиска в двух коробках за два шага было установлено, что оптимальными стратегиями игрока H могут являться как тривиальные вектора $(1, 0)$, $(0, 1)$ или их смешанная стратегия $(0.5, 0.5)$, так и бесконечное множество векторов $(x_1, x_2), \quad x_1 + x_2 = 1$, в зависимости от коэффициентов рассматриваемой задачи.

Заключение

В результате проделанной работы была составлена теоретико-игровая модель многошаговой задачи поиска неподвижного пассивного объекта в дискретной среде с последующим нормативным анализом поведения игроков на случай двухшагового поиска объекта среди его двух возможных положений. Была реализована программная реализация данной модели на случай k-шагового поиска объекта среди N возможных положений. К сожалению, ресурсоемкость алгоритма программной реализации превысила ожидаемую, в связи с чем решение данной задачи для больших значений k и N невозможно, что сделало невозможным корректный численный анализ задач.

Данная теоретико-игровая модель была обобщена на случай одновременной проверки в m возможных положениях объекта за один шаг, после чего был разработан программный алгоритм, способный решать данную задачу для значения $m = 2$. В результате тестирования алгоритма на различных входных данных было обнаружено, что функция затрат игрока S в данном классе задач может быть ограничена как сверху (снизу). Данные выводы являются не совсем корректными ввиду отсутствия возможности проверить их при большом значении шага k, т.к. программный алгоритм оказался еще более ресурсоемким по сравнению с алгоритмом простого поиска.

В ходе дальнейшей работы планируется произвести оптимизацию работы существующих алгоритмов с целью уменьшить их ресурсоемкость, а также разработать программный алгоритм на случай одновременной проверки в m возможных положениях для дальнейшего численного анализа большего класса задач.

Список литературы

1. Абчук В. А., Суздаль В. Г. Поиск объектов. М.: Сов. радио, 1977. 336 с.

2. Беллман Р. Э. Динамическое программирование. Пер. с англ. / Под редакцией Н. Н. Воробьева М.: Изд-во иностранной литературы, 1960. 395 с.

3. Петросян Л. А, Гарнаев А. Ю. Игры поиска. СПб.: Изд-во СПбГУ, 1992. 216 с.

4. Хеллман О. Введение в теорию оптимального поиска. — Пер. с англ. / Под редакцией Н. Н. Моисеева М.:Наука. 1985. 248 с.

5. Gal S. A discrete search game // SIAM J. Appl. Math. 1974. Vol 27. No 4, P. 641-648.

6. Garnaev A. Search Games and Other Applications of Game Theory. Heidelberg, N-Y.: Springer, 2000. 145 c.

7. Koopman B. O. The theory of search. Part III // Oper. Res. 1957. Vol 5. P. 613-626.

8. Stone L. D. Theory of optimal search. N-Y.: INFORMS, 2007. 278 c.

Приложение

Приложение 1. Метод Main алгоритма простого поиска

```
public static double t = 5;  //затраты ресурсов
public static double a = 4;  //выигрыш
public static int n = 5;     //число коробок
public static int k = 4;     //число шагов k+1
public static List<Probability> probabilities;
public static double[] p;
static void Main(string[] args)
{
    double totalmin = 0;
    var minpath = double.MaxValue;
    p = new double[] { 0.62, 0.74, 0.75 ,0.63, 0.69};  //вероятность обнаружения p
    probabilities = new List<Probability>(20000);
    var path = new int[k + 1];                          //путь поиска
    probabilities.Add(new Probability() { Index = 0, Level = 0, Parent = null,
                                                        Value = 0.23 });
    probabilities.Add(new Probability() { Index = 1, Level = 0, Parent = null,
                                                        Value = 0.18 });
    probabilities.Add(new Probability() { Index = 2, Level = 0, Parent = null,
                                                        Value = 0.17 });
    probabilities.Add(new Probability() { Index = 3, Level = 0, Parent = null,
                                                        Value = 0.22 });
    probabilities.Add(new Probability() { Index = 4, Level = 0, Parent = null,
                                                        Value = 0.2 });
                                                    //объвление начальных x
    for (int i = 0; i < k; i++)        //создание дерева элементов
    {
        GenerateLevel(i, n);
    }
    FillLevel();                       //заполнение параметра Value
    for (int i = k; i >= 0; i--)       //заполнение параметра Result
    {                                  //методом F
        foreach (var prob in Probabilities.Where(c => c.Level == i))
        {
```

```
            var min2 = probabilities.Where(c => c.Parent == prob).OrderBy(c => c.Result);
            var min = min2.FirstOrDefault();
            if (min == null) prob.Result = F(prob.Value, p[prob.Index], 0);
            else prob.Result = F(prob.Value, p[prob.Index], min.Result);
        }
    }
    foreach (var c in probabilities.Where(c => c.Level == 0))  //поиск минимальных затрат
    {
        if (c.Result < minpath)
        {
            path[0] = c.Index;
            minpath = c.Result;
        }
    }
    totalmin = minpath;
    minpath = double.MaxValue;

    for (int i = 1; i <= k; i++)              //поиск оптимального пути
    {
        foreach (var c in probabilities.Where(c => (c.Level == i) && (c.Parent.Index == pat
        {
            if (c.Result < minpath)
            {
                path[i] = c.Index;
                minpath = c.Result;
            }
        }
        minpath = double.MaxValue;
    }
    for (int i = k; i >= 0; i--)               //вывод оптимального пути
    {
        Console.WriteLine(path[i] + 1);
    }
    Console.WriteLine(totalmin);               //вывод минимальных затрат
    Console.ReadLine();
}
```

```
public static double F(double x, double p, double min)   //метод F
{
   return t - a * x * p + (1 - p * x) * min;
}
```

Приложение 2. Метод Main алгоритма множественного поиска, $m = 2$

```
public static double t = 3;  //затраты ресурсов
public static double a = 20; //выигрыш
public static int n = 3;     //число коробок
public static int k = 8;     //число шагов k+1
public static List<Probability> probabilities;
public static List<Results> results;  //хранение результатов
public static double[] p;
static void Main(string[] args)
{
    double totalmin = 0;
    var minpath = double.MaxValue;
    p = new double[] { 0.55, 0.6, 0.65 };  //вероятность обнаружения p
    probabilities = new List<Probability>(20000);
    results = new List<Results>(20000);
    var path = new int[k+1,2];             //путь поиска
    probabilities.Add(new Probability() { Index = 0, Level = 0, Parent1 = null,
                                          Parent2 = null, Value = 0.43 });
    probabilities.Add(new Probability() { Index = 1, Level = 0, Parent1 = null,
                                          Parent2 = null, Value = 0.36 });
    probabilities.Add(new Probability() { Index = 2, Level = 0, Parent1 = null,
                                          Parent2 = null, Value = 0.31 });
                                                    //объявление начальных x
    for (int i = 0; i < k; i++)              //создание дерева элементов
    {                                        //класса Probability
        GenerateLevel(i, n);
    }
    FillLevel();                             //заполнение параметра Value
    for (int i = k; i >= 0; i--)             //заполнение дерева элементов
    {                                        //класса Results методом F
```

```
    foreach (var first in probabilities.Where(c => c.Level == i))
    {
        for (int j = 1; j < n - first.Index; j++)
        {
            var second = probabilities[probabilities.IndexOf(first) + j];
            var min2 = results.Where(c => (c.Parent1 == first)&&
                                    (c.Parent2 ==second)).OrderBy(c => c.Result);
            var min = min2.FirstOrDefault();
            if (min == null)
            {
                var newResult = new Results()
                {
                    Level = (short)i,
                    Parent1 = first.Parent1,
                    Parent2 = first.Parent2,
                    Index1 = first.Index,
                    Index2 = second.Index,
                    Result = F(first, second, 0)
                };  results.Add(newResult);
            }
            else
            {
                var newResult = new Results()
                {
                    Level = (short)i,
                    Parent1 = first.Parent1,
                    Parent2 = first.Parent2,
                    Index1 = first.Index,
                    Index2 = second.Index,
                    Result = F(first, second, min.Result)
                };  results.Add(newResult);
            }
        }
    }
}
```

```
    foreach (var c in results.Where(c => c.Level == 0))  //поиск минимальных затрат
    {
        if (c.Result < minpath)
        {
            path[0, 0] = c.Index1;
            path[0, 1] = c.Index2;
            minpath = c.Result;
        }
    }
    totalmin = minpath;
    minpath = double.MaxValue;
    for (int i = 1; i <= k; i++)     //поиск оптимального пути
    {
        foreach (var c in results.Where(c => (c.Level == i)&&
                           (c.Parent1.Index == path[i-1,0])&&
                           (c.Parent2.Index == path[i - 1, 1])))
        {
            if (c.Result < minpath)
            {
                path[i,0] = c.Index1;
                path[i, 1] = c.Index2;
                minpath = c.Result;
            }
        }
        minpath = double.MaxValue;
    }

    for (int i = k; i >= 0; i--)     //вывод оптимального пути
    {
        Console.WriteLine("{0}  {1}",path[i,0] + 1,path[i,1]+1);
    }
    Console.WriteLine(totalmin);     //вывод минимальных затрат
    Console.ReadLine();
}
```

```
public static double F(Probability x1,Probability x2, double min)  //метод F
{
    return 2*t - a * (x1.Value * p[x1.Index] + x2.Value * p[x2.Index]) + notsuccess(x1,x2)
}

public static double notsuccess(Probability c1,Probability c2)  //вероятность неудачного
{                                                               //поиска на текущем шаге
    double summ = 0;
    for (int i = 0; i < n; i++)
    {
        if (i != c1.Index && i != c2.Index)
        {
            var diff = i - c1.Index;
            var position = probabilities.IndexOf(c1) + diff;
            summ += probabilities[position].Value;
        }
    }
    summ += (1 - p[c1.Index]) * c1.Value;
    summ += (1 - p[c2.Index]) * c2.Value;
    return summ;
}
```