


МИНОБРНАУКИ РОССИИ
Санкт-Петербургский государственный
электротехнический университет
«ЛЭТИ» им. В.И. Ульянова (Ленина)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: «Поиск с возвратом»

Студент группы 1304

Преподаватель



Завражин Д.Г.

Шевелева А.М.

Санкт-Петербург
2023

Цель работы

Изучить и на практике освоить азы применения перебора с возвратом, метода ветвей и границ.

Задание

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до $N - 1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу – квадрат размера N . Он может получить ее, собрав из уже имеющихся обрезков(квадратов). Например, столешница размера 7×7 может быть построена из 9 обрезков. Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные. Размер столешницы – одно целое число N ($2 \leq N \leq 40$).

Выходные данные. Одно число K , задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N . Далее должны идти K строк, каждая из которых должна содержать три целых числа x, y и w , задающие координаты левого верхнего угла ($1 \leq x, y \leq N$) и длину стороны соответствующего обрезка(квадрата).

1 Подход к решению задачи

Можно заметить, что для некоторых N задачу можно свести к более простой. Для не являющегося простым числом N можно взять минимальный (простой) множитель s числа N и решить задачу для квадрата $s \times s$, а затем увеличить масштаб полученного разбиения, получив тем самым разбиение искомого квадрата $N \times N$.

Для случая $s = 2$ существует очевидное минимальное разбиение на четыре квадрата. Для остальных s можно показать, что в разбиении неизменно будет один квадрат со стороной $\left\lceil \frac{s}{2} \right\rceil$ и два со стороной $\left\lfloor \frac{s}{2} \right\rfloor$. При таком подходе незанятая площадь уменьшается в четыре раза.

После применения этих упрощений разработанная программа затем осуществляет перебора с возвратом, пытаясь замостить квадратами наибольшего размера оставшееся пустое пространства. В том случае, если уже рассматриваемая ветвь не может улучшить уже найденное решение, она далее не рассматривается, то есть в таких случаях применяется метод ветвей и границ.

2 Используемые функции и структуры данных

Программа написана на парадигме объектно-ориентированного программирования. Основная логика заключена в классе *Tabletop*, конструктор которого инициализирует все необходимые для поиска решения переменные, а сам поиск начинается посредством вызова метода *square()*.

В классе *Tabletop* имеется вспомогательный вложенный класс *Surface*, отвечающий за управление матрицей `std::vector<unsigned> cells`, которая используется для проверки пересечений квадратов и для хранения необходимых для воспроизведения после его нахождения конкретного решения содержащих длины сторон входящих в него квадратов пометок. У него имеются следующие методы:

- *unsigned cardinality() const*; – не имеет входных значений, возвращает количество уже использованных квадратных обрезков;
- *void cardinality(unsigned c)*; – принимает требуемое количество квадратов в инициализуемом решении, позволяет искусственно задать количество уже использованных квадратных обрезков (используется только в конструкторе класса *Tabletop* для инициализации худшего возможного решения);
- *const unsigned &at(unsigned x, unsigned y) const*; – принимает координаты ячейки, осуществляет доступ к ячейке по координатам, возвращает хранящееся там значение;
- *unsigned &at(unsigned x, unsigned y)*; – то же;
- *bool is_vacant(unsigned x, unsigned y) const*; – принимает координаты ячейки, проверяет, является ли ячейка свободной, возвращает 1 для свободной ячейки и 0 в противном случае;
- *bool is_meaningful(unsigned x, unsigned y) const*; – принимает координаты ячейки, проверяет, соответствует ли ячейка левому верхнему углу некоторого использованного обрезка, возвращает 1 если это так и 0 в противном случае;
- *bool add(unsigned x, unsigned y, unsigned side)*; – принимает координаты ячейки и желаемую длину стороны, условно добавляет новый обрезок, возвращает соответствующее тому, был ли квадрат добавлен, двоичное значение (1 – добавлен, 0 – не добавлен).

- `void empty(unsigned x, unsigned y, unsigned side);` – принимает координаты левого верхнего и длину стороны очищаемой области, очищает квадратную область (используется для отмены добавления обрезка).

Как видно, все они являются вспомогательными.

В свою очередь, класс *Tabletop* содержит следующие методы:

- `void fit(unsigned x, unsigned y);` – Принимает координаты ячейки, осуществляет перебор всех возможных обрезков, которые можно вставить в заданную позицию, и в тех случаях, когда это возможно, вызывает функцию *shift*;
- `void shift(unsigned x, unsigned y, unsigned shift = 1)` – Принимает координаты ячейки и желаемое смещение, находит место для размещения следующего квадрата по заданному горизонтальному смещению, а также обрубаёт бесперспективные потенциальные решения;
- `Tabletop &square();` – не имеет входных значений, задаёт начальные квадраты, обрабатывает случай $s = 2$, возвращает объект решения;

Его конструктор также сводит задачу к задаче с простой длиной стороны столешницы. Полный исходный код программы представлен в Листинге 2 в Приложении А.

3 Сложность по времени

Для измерения затрачиваемого на поиск решения времени функция *main()* из Листинга 2 была изменена следующим образом (см. Листинг 1):

Листинг 1 — Модифицированная функция *main()*

```
155 | #include <chrono>
156 | #define N_T 1000.0
157 |
158 | int main()
159 | {   unsigned side;
160 |     std::cin >> side;
161 |     auto begin = std::chrono::high_resolution_clock::now();
162 |     for(unsigned i = 0; i < N_T; ++i)
163 |         Tabletop(side).square();
164 |     auto end = std::chrono::high_resolution_clock::now();
165 |     std::cout << std::chrono::duration_cast<std::chrono::nanoseconds>(end-begin).count() << "ns";
166 |     return 0;
167 | }
168 |
```

Параметр *N_T* задаёт количество проводимых с целью усреднения измерений повторений. Данный эксперимент был проведён для 1 и простых чисел до 43; полученные данные приведены в Таблице 1.

Таблица 1 — Измерение времени

s	N_T	Время, нс
1	1000	584
2	1000	569
3	1000	904
5	1000	1878
7	1000	5608
11	1000	93173
13	1000	220669
17	1000	1684070
19	1000	5329709
23	100	22397301
29	20	190910888
31	20	474230990
37	5	2513853711
41	2	8946579318
43	2	16343088071

Как видно из Таблицы 1, сложность нахождения минимального разбиения является экспоненциальной.

Выводы

Были изучены и на практике освоены освоены азы применения перебора с возвратом, метода ветвей и границ, осуществляемый путём отбрасывания после нахождения какого-либо решения путей, которые не могут привести к его улучшению, что позволяет избежать полного перебора всех возможных комбинаций размещения квадратов.

Также было освоено осуществление оптимизации перебора с возвратом путём применения эвристик для сокращения количества рассматриваемых путей без исключения хотя бы одного верного искомого решения, в том числе путём задания позиций трёх наибольших квадратов, что позволяет втрое сократить замаскируемую площадь.

Также были отточены навыки измерения времени путём использования стандартной библиотеки *chrono* языка C++. Было выполнено усложнённое задание с $N = 40$ вместо тривиальных задач с $N = 20$ или $N = 30$.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Листинг 2 — Содержащийся в файле *main.cpp* исходный код

```
1  #include <iostream>
2  #include <vector>
3
4  class Tabletop
5  {
6  private:
7      // Surface is a helper class to manage already occupied cells
8      class Surface
9      {
10         // It also holds its side and a number of added squares
11     public:
12         const unsigned side;
13     private:
14         std::vector<unsigned> cells;
15         unsigned n_squares = 0;
16
17     public:
18         // The constructor initialises all cells to 0, meaning vacant
19         Surface(unsigned side)
20             : side{side}, cells(side * side) { }
21
22         Surface(const Surface &surface)
23             : side{surface.side}, cells{surface.cells}, n_squares{surface.n_squares} { }
24
25         Surface &operator=(const Surface &surface)
26         { if(this == &surface)
27             return *this;
28             this->~Surface();
29             return *new(this) Surface(surface);
30         }
31
32         // cardinality() is a getter for the number of added squares
33         unsigned cardinality() const
34         { return this->n_squares; }
35
36         // This is a setter in order to set cardinality of the dummy solution
37         // without wastingly appending side*side squares
38         void cardinality(unsigned c)
39         { this->n_squares = c; }
40
41         // at() is a getter for a cell value by its (x, y) coordinates
42         const unsigned &at(unsigned x, unsigned y) const
43         { return this->cells[x * this->side + y]; }
44
45         unsigned &at(unsigned x, unsigned y)
46         { return this->cells[x * this->side + y]; }
47
48         // is_vacant() tests for whether the cell is empty or not
49         bool is_vacant(unsigned x, unsigned y) const
50         { return this->at(x, y) == 0; }
51
52         // is_meaningful() tests for whether the cell holds a square side
53         // or not
54         bool is_meaningful(unsigned x, unsigned y) const
55         { return this->at(x, y) < -1; }
56
57         // add() adds a square conditionally
58         // Returns whether the square was added
59         bool add(unsigned x, unsigned y, unsigned side)
60         { for(int i = x; i < x + side; i++)
61             for (int j = y; j < y + side; j++)
62                 if(not this->is_vacant(i, j))
```

```

63         return false;
64     for(int i = x; i < x + side; ++i)
65         for(int j = y; j < y + side; ++j)
66             this->at(i, j) = -1;
67     this->at(x, y) = side; // a "meaningful" cell
68     return ++this->n_squares;
69 }
70
71 // empty() empties cells of a square-shaped area
72 void empty(unsigned x, unsigned y, unsigned side)
73 { for(int i = x; i < x + side; i++)
74     for(int j = y; j < y + side; j++)
75     { if(this->is_meaningful(i, j))
76         --this->n_squares;
77         this->at(i, j) = 0;
78     } }
79 };
80
81 Surface surface, solution;
82 unsigned factor;
83
84 // Recursively fits a piece using backtracking
85 void fit(unsigned x, unsigned y)
86 { if(not surface.is_vacant(x, y))
87     return this->shift(x, y);
88     unsigned side = std::min(this->surface.side - x, this->surface.side - y);
89     for(; side > 0; --side)
90     { if(this->surface.side < x + side or this->surface.side < y + side)
91         continue;
92         if(surface.add(x, y, side))
93         { this->shift(x, y, side);
94             this->surface.empty(x, y, side);
95         } } }
96
97 // Conditionally selects the next square placement
98 // The surface is traversed line-by-line
99 // Branches are cut if they do not improve the already established solution
100 void shift(unsigned x, unsigned y, unsigned shift = 1)
101 { if(this->solution.cardinality() <= this->surface.cardinality())
102     return;
103     if(y + shift < this->surface.side)
104         return this->fit(x, y + shift);
105     if(x + 1 < this->surface.side)
106         return this->fit(x + 1, this->surface.side / 2);
107     this->solution = this->surface;
108 }
109
110 public:
111     // The constructor:
112     // (1) Determines whether the square is simply a scaled up version of a
113     // smaller one;
114     // (2) Initialises the surface and the dummy solution
115     Tabletop(unsigned side)
116     : surface{0}, solution{0} // these are temporary
117     { this->factor = 1;
118         // It's not worth it to iterate over primes instead
119         for(unsigned factor = 2; factor <= side / 2; ++factor)
120         { if(side % factor == 0)
121             { this->factor = side / factor;
122                 side = factor;
123                 break;
124             } }
125         this->surface = this->solution = Surface(side);
126         solution.cardinality(side * side); // It is pointless to initialise fully
127     }
128
129     Tabletop &square()

```

```

130     { unsigned side = this->surface.side / 2 + this->surface.side % 2;
131       surface.add(0, 0, side);
132       surface.add(side, 0, this->surface.side - side);
133       surface.add(0, side, this->surface.side - side);
134       if(this->surface.side == 2)
135         surface.add(side, side, this->surface.side - side);
136       else
137       { this->fit(side, side);
138         this->surface = this->solution;
139       }
140       return *this;
141     }
142
143     friend std::ostream &operator<<(std::ostream &, const Tabletop &);
144   };
145
146   // Outputs a solution
147   std::ostream &operator<<(std::ostream &out, const Tabletop &tabletop)
148   { out << tabletop.surface.cardinality() << std::endl;
149     for(int i = 0; i < tabletop.surface.side; i++)
150       for(int j = 0; j < tabletop.surface.side; j++)
151         if(tabletop.surface.is_meaningful(j, i))
152         { out << i * tabletop.factor + 1 << ' ';
153           out << j * tabletop.factor + 1 << ' ';
154           out << tabletop.surface.at(j, i) * tabletop.factor << std::endl;
155         }
156     return out;
157   }
158
159   int main()
160   { unsigned side;
161     std::cin >> side;
162     std::cout << Tabletop(side).square();
163     return 0;
164   }
165

```