ПЛОВДИВСКИ УНИВЕРСИТЕТ "ПАИСИЙ ХИЛЕНДАРСКИ", БЪЛГАРИЯ
НАУЧНИ ТРУДОВЕ, ТОМ 35, КН. 3, 2007 – МАТЕМАТИКА
PLOVDIV UNIVERSITY "PAISSII HILENDARSKI", BULGARIA
SCIENTIFIC WORKS, VOL. 35, BOOK 3, 2007 – MATHEMATICS

CLIQUES, PACKINGS OF SEGMENTS AND BINARY MATRICES

Dimcho S. Dimov, Dobromir P. Kralchev, Alexander P. Penev

Abstract. Due to its great variety of applications, the packing problem attracted our attention. The problem is reduced via a graph-theoretic interpretation (searching for cliques in a graph) to searching for a submatrix of a special kind. An algorithm is constructed by means of self-monitoring, which is suitable for real-world problems.

Key words: heuristics, reflexive (self-monitoring) algorithms, packings, graphs, cliques, binary matrices
Mathematics Subject Classification 2000: Primary 68T20; Secondary 68R10, 05C69, 05C50

1. Introduction

In programming one may sometimes come upon a problem that has an obvious, though possibly ineffective solution. Either an effective algorithm does not exist at all, or it is unknown to the programmer, who has little time to make a deep analysis. It is highly desirable that one should have a simple and reliable technique to manage such cases. Self-monitoring is often useful in this situation.

For instance, many problems are easily formulated in terms of finding a suitable packing of segments. Processor time sharing is a classical example. Timetable generating, which contains a packing subproblem, does not have a general solution yet.

The general packing problem is continuous, but it can be reduced to the discrete variant [1]; it is the last one that is discussed below. The relations between the segments of the discrete packing problem can be encoded in a graph thus reducing the problem to finding a clique in a graph.

There exist different approaches for finding cliques. A solution can be deduced from a previous investigation of ours on the assignment problem. This solution turns out to be most appropriate for the current discussion. A reflexive algorithm is constructed on the basis of it.

2. The packing problem

Given k segments of lengths λ_1, λ_2, ..., λ_k, is it possible to arrange them upon another segment of a length λ_0 without overlapping (external touch is allowed) if there are additional constraints on the possible position of each segment?

If $\sum_{i=1}^{k} \lambda_i > \lambda_0$, then no solution exists. That is why, further on we assume that $\sum_{i=1}^{k} \lambda_i \leq \lambda_0$. When the equality holds, the problem is *canonical*.

Let the big segment be $[0; \lambda_0]$. Any possible position of the i-th segment is of the kind $[x_i\,; x_i + \lambda_i]$. Assume that the additional constraints are independent, that is, they have the form $x_i \in M_i$, $\forall i \in \{1, 2, \dots, k\}$, where M's do not depend on x's. This is equivalent to $x_i \in \lambda M_i$, $\forall i \in \{1, 2, \dots, k\}$, where $\lambda M_i = M_i \cap [0; \lambda_0 - \lambda_i]$. We say that the packing problem is *discrete*, when λ_0, λ_1, λ_2, ..., $\lambda_k \in \mathbb{Z}$ and M_1, M_2, ..., $M_k \subseteq \mathbb{Z}$.

3. Solution

The input of the discrete packing problem can be encoded as a binary matrix $A = (a_{j\,i})$ consisting of $\lambda_0 + 1$ rows and k columns ($j = 0, 1, 2, \dots, \lambda_0$ from bottom to top; $i = 1, 2, \dots, k$ from left to right): if $j \in M_i$, then $a_{j\,i} = 1$, else $a_{j\,i} = 0$. (Since the lengths of the segments are positive, the top λ_i cells of the i-th column can be converted to zeros, and the λ_0-th row of A can be safely omitted.) Then a packing corresponds to a k-tuple of units from different columns, such that

$$x_{i_1} - x_{i_2} \notin \left(-\lambda_{i_1}; \lambda_{i_2}\right), \forall i_1 \forall i_2 \in \{1, 2, \dots, k\}, i_1 \neq i_2; \tag{1}$$

$$x_i \in [0; \lambda_0 - \lambda_i], \forall i \in \{1, 2, \dots, k\}; \tag{2}$$

where x_i is the index of the row of the unit, taken from the i-th column. (If the conversion of the top cells has taken place, then (2) can be omitted.)

3.1. Graph-theoretic interpretation. Consider an unoriented graph G, whose vertices correspond to the units of the matrix A (after conversion of the top cells); two vertices are connected if and only if their units can take part in a packing, that is, they are from different columns (say, i_1 and i_2) and $x_{i_1} - x_{i_2} \notin \left(-\lambda_{i_1}\,;\lambda_{i_2}\right)$, where x_{i_1} and x_{i_2} are the indices of their rows. Then a packing (of some m segments) corresponds to an m-clique in G and a solution (a packing of all the k segments) corresponds to a k-clique. Obviously, G is a k-partite graph and $cl(G) \leq k$. We need an algorithm which checks whether $cl(G) = k$ and if so, finds a k-clique (if $cl(G) < k$, the exact value of $cl(G)$ is of no interest).

Finding cliques is NP-complete (cf. [2]). Fortunately, there still exist algorithms whose average running-time (not the worst-case one!) is good enough for the most cases met in practice.

3.2. Cliques and binary matrices. Let $G = (V, E)$ be an unoriented graph with n vertices, $V = \{v_1, v_2, \ldots, v_n\}$, $B = (b_{j\,i})$ be an n x n binary matrix defined as follows: if $i \neq j$ and $\{v_i\,; v_j\} \notin E$, then $b_{j\,i} = 1$, else $b_{j\,i} = 0$. A submatrix of B is said to be *symmetrically positioned* in B if the sets of indices of its rows and its columns coincide (so it must be a square matrix). Their common value defines a subset of V; moreover, the correspondence "a subset of $V \leftrightarrow$ a symmetrically positioned submatrix of B" is one-to-one.

A symmetrically positioned submatrix whose elements are all zeros will be called a *clique square*. If a subset of V is a clique, then its corresponding submatrix of B is a clique square and vice versa. Therefore finding cliques in the graph G is equivalent to finding clique squares in the matrix B.

3.3. Searching for clique squares. The problem now sounds like this: given a symmetric binary n x n matrix B and an integer k, $1 \leq k \leq n$, find in B a clique square of a size k.

In our investigation [3] we designed an algorithm for solving a similar problem: given a binary n x n matrix and an integer g, $1 \leq g \leq 2n$, find a submatrix of zeros, such that the sum of its width and height is equal to g. The average running-time of the algorithm is less than half a second even for a 1000 x 1000 matrix.

We would like to apply this algorithm to our new problem. The only difference is that now the submatrix must be symmetrically positioned.

3.3.1. *An algorithm for clique squares.* We use a Pascal-like pseudocode. The symbols '{' and '}' embrace a set, and '//' start a C-style comment. A symmetrically positioned submatrix is encoded as a set of integers — the indices of its rows. When an empty set is returned, there is no clique square.

```
function FindCliqueSquareInMatrix(
  n: integer; // the size of the matrix B
  B: matrix; // a symmetric binary matrix, tr(B) = 0
  k: integer // the size of the clique square searched for
): set of integers; // the clique square
begin
  Result := FindCliqueSquareInSubmatrix(n, B, {1,2,...,n}, k, true);
end;

function FindCliqueSquareInSubmatrix(
  n: integer; // the size of the matrix B
  B: matrix; // a symmetric binary matrix, tr(B) = 0
  M: set of integers; // a symmetrically positioned submatrix
  k: integer; // the size of the clique square searched for
  bad0to1: boolean // a flag for "bad" zeros
): set of integers; // the clique square
var i, j: integer;
begin
  Result := {};
  for i∈M do begin
    Result := CliqueSquarePos(n,B,M,k,i);
    if Result <> {} then break
    else if bad0to1 then for j∈M do begin
      B[i, j] := 1; B[j, i] := 1;
    end;
  end;
end;
```

The FindCliqueSquareInMatrix function merely calls a more general one, which has additional parameters. This is necessary, because the algorithm is recursive.

The FindCliqueSquareInSubmatrix routine explores each zero in the main diagonal of a submatrix and tries to find a clique square containing it. On failure it replaces the “bad” zero and all the elements in its row and column with units, if it has been told so (this happens only at the top level of the recursion).

It should be noticed that our pseudocode describes the algorithm, not its implementation. Some minor details may be changed in the implementation. For example, if you use dynamic data structures, you may delete the row and the column of a “bad” zero instead of filling them with units.

```
function CliqueSquarePos (
  n: integer; // the size of the matrix B
  B: matrix; // a symmetric binary matrix, tr(B) = 0
  M: set of integers; // a symmetrically positioned submatrix
  k: integer; // the size of the clique square searched for
  p: integer // a position that must belong to the clique square
): set of integers; // the clique square
var MaxSquare, MinSquare: set of integers;
begin
  MaxSquare := GetMaxSquare(n, B, M, p);
  if |MaxSquare| < k then Result := {}
  else begin
    MinSquare := GetMinSquare(n, B, MaxSquare);
    if |MinSquare| ≥ k then // found
      Result := {the first k elements of MinSquare}
    else begin // recursive searching
      Result := FindCliqueSquareInSubmatrix(
                  n, B, MaxSquare\MinSquare,
                  k - |MinSquare|, false
                );
      if Result <> {} then Result := MinSquare ∪ Result;
    end;
  end;
end;
```

The CliqueSquarePos routine searches a submatrix (defined by the set M) of the matrix B for a clique square of a size k containing the p-th element in the main diagonal of B. The function does so by calculating the maximal and minimal squares. The *maximal square* of a position p is the submatrix consisting of those rows and columns of B, whose p-th elements are zeros. The *minimal square* of a position p is the submatrix of its maximal square consisting of those rows and columns of its that contain only zeros. Maximal and minimal squares are symmetrically positioned.

```
function GetMaxSquare(
  n: integer; // the size of the matrix B
  B: matrix; // a symmetric binary matrix, tr(B) = 0
  M: set of integers; // a symmetrically positioned submatrix
  p: integer // a position that must belong to the clique square
): set of integers; // the maximal square
```

```
var j: integer;
begin
  Result := {};
  for j∈M do if B[j, p] = 0 then Result := Result ∪ {j};
end;

function GetMinSquare(
  n: integer; // the size of the matrix B
  B: matrix; // a symmetric binary matrix, tr(B) = 0
  M: set of integers // the maximal square
): set of integers; // the minimal square
var
  i: integer; // a column index
  j: integer; // a row index
  flag: boolean; // a row of zeros
begin
  Result := {};
  for j∈M do begin
    flag := true;
    for i∈M do
      if B[j, i] = 1 then begin flag := false; break end;
    if flag then Result := Result ∪ {j};
  end;
end;
```

Obviously, any clique square containing the p-th element in the main diagonal of B is a part of the maximal square of this element (which itself is not necessarily a clique square). On the other hand, the minimal square is a clique square. If its size $|\text{MinSquare}|$ is greater than or equal to k, then a suitable clique square has been found. Otherwise, the clique square (if it exists) must have at least $r = k - |\text{MinSquare}|$ of the remaining rows and columns of the maximal square. And vice versa, such a clique square of a size r can be extended (using the rows and columns of the minimal square) to a clique square of a size k. This explains the recursive call in CliqueSquarePos.

Converting "bad" zeros to units is not necessary for the correctness of the algorithm, but it is a very good optimization. Indeed, if a zero is proved not to participate in a clique square, then converting this zero to a unit will not destroy any clique square and will prevent us from exploring again the same zero (in subsequent recursive calls). Converting one "bad" zero usually causes more conversions, thus simplifying the matrix.

3.3.2. *Experiments with the algorithm for clique squares.* The algorithm was implemented and tested on Celeron 366 MHz. For $n = 1000$ the maximal and the average times are about 200 ms, resp. 100 ms if there is a clique square; otherwise, the average time is 400 ms and the maximal one is several minutes!

A traditional approach is to make a deeper analysis of the original problem in the hope of finding some special features that one can use to construct a faster algorithm. For example, an important fact is that the graph is k-partite.

However, a developer rarely has much time for such an analysis. Therefore our goal is to present a simple, yet powerful idea, applicable in a lot of similar situations. That is why, we shall not use the special features of the problem, even if they can help us speed up the algorithm.

The difficulty can be easily overcome if heuristics are allowed (this is often the case in AI). A higher, monitoring level must be added to watch for the running-time. If the lower level (the proposed algorithm) has been running for a long time, it must be interrupted and a negative answer ('no clique square found') must be returned. The modified algorithm consisting of two levels united by a common goal is a *reflexive*, or *self-monitoring* algorithm.

Experiments were made on the last variant. A lot of matrices were generated at random for various sizes n, for each $k = 0.1n, 0.2n, \dots, 0.9n, n,$ and for each density from 0% to 100% at intervals of 10%, where k is the size of the clique square. (*Density* is the ratio of the count of the units to the count of all the elements of a matrix). The running-time limit was set to 1000 ms.

n	**Percentages of results**			**Maximal and average running-time (ms)**					
	found	**not found**	**stopped**	**found**		**not found**		**total**	
				max	**avg**	**max**	**avg**	**max**	**avg**
500	9.1%	65.3%	25.6%	60	3	990	256	2090	445
1000	9.1%	60.4%	30.5%	170	91	990	370	1710	558
1500	9.1%	52.4%	38.5%	280	181	990	751	1640	819
2000	9.1%	13.8%	77.1%	440	320	990	986	1920	984

Table 1. Running-time of the reflexive algorithm for clique squares.

There are three possible results (return values): 'found', 'not found' and 'stopped' (= interrupted). Apparently, the average time of the 'found' answers is much smaller than the average time of the 'not found' answers.

The running-time limit of 1000 ms was properly chosen: the maximal time of the 'found' cases is much smaller than the limit. In fact, the maximal time cannot be fully trusted: it may increase if more tests are run.

The total maximal time is greater than the running-time limit, because the limit is implemented through a timer and the WM_TIMER message is of low priority. Its handler is executed in a separate thread and it is not started immediately. This increases the running-time of the 'stopped' answers.

The average running-time can be trusted much more than the maximal one. It supports the drawn conclusion: most matrices that have a clique square are processed quickly. Their percentage (9.1%) is stable and relatively small. The percentage of the 'stopped' cases tends to grow, which is natural, because the running-time limit is fixed.

3.4. Back to packings of segments. Now that we have an algorithm for finding clique squares (or cliques, equivalently), we can use it to solve the original problem. (Of course, we can use other algorithms for cliques, too.)

3.4.1. *An algorithm for finding packings of segments.* The reduction of the problem has already been described. More formally:

```
procedure PackingsToCliques(
  k: integer; // the count of the small segments
  $\lambda_0$: integer; // the length of the big segment
  $(\lambda_i)_{i=1}^{k}$: sequence of integers; // the lengths of the small segments
  A: matrix; // a binary matrix $\lambda_0$ x k
  var n: integer; // the size of the matrix B
  var B: matrix; // the binary matrix n x n
  var $(r_m)_{m=1}^{n}$: sequence of integers; // the rows of the 1s of A
  var $(c_m)_{m=1}^{n}$: sequence of integers // the columns of the 1s of A
);
var
  m', m'': integer; // indices
  elem: boolean; // the current element of B
begin
  n := the count of the units of A;
  $(r_m)_{m=1}^{n}$ := the row indices of the units of A;
  $(c_m)_{m=1}^{n}$ := the column indices of the units of A;
  for m' := 1 to n do for m'' := 1 to n do begin
    elem := not Can1sBeTogether$\left(c_{m'}, c_{m''}, r_{m'}, r_{m''}, \lambda_{m'}, \lambda_{m''}\right)$;
    // 0 = false; 1 = true
    B[m', m''] := elem;
    B[m'', m'] := elem;
  end;
end;
```

```
function Can1sBeTogether(
  $i_1$, $i_2$: integer; // column indices
  $j_1$, $j_2$: integer; // row indices
  $\lambda_1$, $\lambda_2$: integer; // lengths of segments
): boolean;
begin
  if $i_1$ = $i_2$
  then Result := ($j_1$ = $j_2$)
  else Result := ($j_1$ - $j_2$) $\notin$ (-$\lambda_1$; $\lambda_2$)
end;
```

The PackingsToCliques procedure reformulates the packing problem in terms of clique squares. The Can1sBeTogether function checks if some two units of A can take part in the same packing; this is possible, when they coincide or are from different columns whose small segments do not overlap.

```
function CliqueSquareToPacking(
  ClSq: set of integers; // the clique square
  $(r_m)_{m=1}^{n}$: sequence of integers; // the rows of the 1s of A
  $(c_m)_{m=1}^{n}$: sequence of integers // the columns of the 1s of A
): sequence of integers; // the packing
var
  $(m_i)_{i=1}^{k}$: sequence of integers;
begin
  if ClSq = {} then
    Result := empty sequence
  else begin
    $(m_i)_{i=1}^{k}$ := the sequence of the elements of ClSq ordered
               in such a way that $c_{m_i}$ = i, $\forall$i$\in${1,2,...,k};
    Result := $\left(r_{m_i}\right)_{i=1}^{k}$;
  end;
end;
```

The CliqueSquareToPacking function translates solution back to the original formulation. Obtaining the sequence $(m_i)_{i=1}^{k}$ requires some kind of sorting, which can be avoided through carefully numbering the units of A and implementing sets.

```
function FindPackingOfSegments(
  k: integer; // the count of the small segments
  $\lambda_0$: integer; // the length of the big segment
  $(\lambda_i)_{i=1}^{k}$: sequence of integers; // the lengths of the small segments
  A: matrix // a binary matrix $\lambda_0$ x k
): sequence of integers; // the packing
var
  n: integer;
  $(r_m)_{m=1}^{n}$: sequence of integers; // the rows of the 1s of A
  $(c_m)_{m=1}^{n}$: sequence of integers // the columns of the 1s of A
  B: matrix; // a binary matrix n x n
  ClSq: set of integers; // a clique square
begin
  PackingsToCliques$\left(k,\ \lambda_0\ ,\ (\lambda_i)_{i=1}^{k}\ ,\ A,\ n,\ B,\ (r_m)_{m=1}^{n}\ ,\ (c_m)_{m=1}^{n}\right)$;
  ClSq := FindCliqueSquareInMatrix(n, B, k);
  Result := CliqueSquareToPacking$\left(ClSq\ ,\ (r_m)_{m=1}^{n}\ ,\ (c_m)_{m=1}^{n}\right)$;
end;
```

The FindPackingOfSegments routine accepts the input data (a sequence $\lambda_1, \lambda_2, \ldots, \lambda_k$ and a matrix A with λ_0 rows and k columns), then constructs the matrix B as described above and searches B for a clique square of a size k. It returns either the sequence of rows of units of A that form a packing or an empty sequence (if there is no packing).

3.4.2. *Experiments with the packing algorithm.* The described algorithm was implemented and tested. The parameter which affects the running-time most is n — the count of the units of the matrix A.

A series of matrices were generated at random for each n: for each density α from 10% to 90% at intervals of 10% a few dozens of matrices with $N = n/\alpha$ elements were sampled (several matrices were processed for each pair $<k, \lambda_0>$, $\lambda_0 = N/k$, where $k_{max} = \sqrt{N}$ and $k = 0.1\,k_{max}\ , 0.2\,k_{max}\ , \ldots , k_{max}$). For each matrix the numbers $\lambda_1, \lambda_2, \ldots, \lambda_k$ were chosen at random in such a way that $\sum_{i=1}^{k} \lambda_i \leq \lambda_0$. The running-time limit was set to 10 s. The results are summarized in table 2.

n	**Percentages of results**			**Maximal and average running-time (ms)**					
	found	**not found**	**stopped**	**found**		**not found**		**total**	
				max	**avg**	**max**	**avg**	**max**	**avg**
500	68.7%	5.6%	25.8%	7800	785	9990	9990	10060	3681
600	65.8%	5.1%	29.1%	6100	1089	9990	9990	10060	4148
700	65.3%	3.6%	31.1%	6920	1519	9990	9990	10060	4469
800	68.2%	4.9%	26.9%	9290	2058	9990	9990	10060	4591
900	66.9%	5.1%	28.0%	9230	2414	9990	9990	10060	4935
1000	65.8%	6.2%	28.0%	8080	3033	9990	9990	10060	5428

Table 2. Running-time of the reflexive packing algorithm.

Most remarks about the algorithm for clique squares are valid here, too. The main difference is that the new algorithm is somewhat slower due to the additional processing of data. The percentage of 'found' cases here is much greater (and slowly decreases), because the matrices B which are passed to FindCliqueSquareInMatrix are not uniformly distributed.

It is important to know the reliabilty of the algorithm, that is, the probability of a correct answer.

n	**Slow 'found' from all (sfa)**	**Slow 'not found' from all (snfa)**	**'Found' from slow (ffs)**	**Positive 'stopped' from all (psfa)**	**Reliability**
500	0.0%	5.6%	0.0%	0.0%	100.0%
600	0.0%	5.1%	0.0%	0.0%	100.0%
700	0.0%	3.6%	0.0%	0.0%	100.0%
800	0.4%	4.9%	8.3%	2.2%	97.8%
900	0.2%	5.1%	4.2%	1.2%	98.8%
1000	0.0%	6.2%	0.0%	0.0%	100.0%

Table 3. Reliability of the reflexive packing algorithm.

A *slow* test case is one whose running-time is over 90% of the limit.

In the table above:

- *sfa* is the percentage of the slow 'found' answers from all the tests.
- *snfa* is the percentage of the slow 'not found' answers from all the tests.
- *ffs* is the percentage of the slow 'found' answers from all the slow answers,

$$ffs = \frac{sfa}{sfa + snfa} \cdot$$

Assume that *ffs* is approximately equal to the percentage of the tests that have a packing from all the 'stopped' cases.

• *psfa* is the percentage of the 'stopped' cases that have a packing from all the tests,

$$psfa \approx ffs\,.\,stopped,$$

where the value of *stopped* is taken from table 2. These are the wrong answers.
• All the other answers are correct, that is,

$$reliabilty = 1 - psfa.$$

Reliabilty has never fallen below 97%.

4. Conclusion

Classical searching techniques combined with self-monitoring enabled us to construct a relatively fast and reliable algorithm for finding packings of segments. Its average time for big matrices is about 5 s and its reliabilty is above 97%, which is enough for most cases met in practice.

References

[1] Dobromir Kralchev, Dimcho Dimov, Alexander Penev, Packings of segments, *Scientific Conference "Mathematics, Informatics & Computer science", devoted to 20 years mathematics and informatics in "St. Kiril and Methodii" University*, Veliko Tarnovo, Bulgaria, 2006, May 12–13, pp. 69–74 (in Bulgarian).

[2] M. R. Garey, D. S. Johnson, Computers and Intractibility, a guide to the theory of NP-completeness, W. H. Freeman and Co., SanFrancisco, 1979.

[3] Dimcho Dimov, Dobromir Kralchev, Alexander Penev, Stanimir Stanchev, Existence of solutions to the assignment problem, *International Conference on Automatics and Informatics*, Sofia, May 30 – June 2, 2001, pp. I-81 – I-83.

Dimcho S. Dimov, Alexander P. Penev
"Paissii Hilendarski" University
Dept. of Mathematics and Informatics
236 Bulgaria Blvd.
4000 Plovdiv, Bulgaria
e-mail: apenev@pu.acad.bg

Dobromir P. Kralchev
University of Food Technologies
Dept. of Informatics and Statistics
26 Maritsa Blvd.
4000 Plovdiv, Bulgaria
e-mail: dobromir_kralchev@abv.bg

Received 14 December 2006

КЛИКИ, ПАКЕТИРАНЕ НА ОТСЕЧКИ И ДВОИЧНИ МАТРИЦИ

Димчо Димов, Добромир Кралчев, Александър Пенев

Резюме. Поради голямото разнообразие от приложения задачата за пакетиране привлече вниманието ни. Чрез интерпретация в термините на теорията на графите (търсене на клики в граф) задачата се свежда до търсене на подматрица от определен вид. Конструиран е самонаблюдаващ се алгоритъм — идея, удобна за задачи, възникващи в практиката.