European Journal of Operational Research 260 (2017) 494–506

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

Markov Chain methods for the Bipartite Boolean Quadratic Programming Problem

Daniel Karapetyan [a,b,c,*], Abraham P. Punnen [c], Andrew J. Parkes [b]

[a] Institute for Analytics and Data Science, University of Essex, Colchester CO4 3SQ, UK
[b] ASAP Research Group, School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK
[c] Department of Mathematics, Simon Fraser University Surrey, Central City, 250-13450 102nd AV, Surrey, British Columbia V3T 0A3, Canada

ARTICLE INFO

Article history:
Received 27 April 2016
Accepted 2 January 2017
Available online 6 January 2017

Keywords:
Artificial intelligence
Bipartite Boolean quadratic programming
Automated heuristic configuration
Benchmark

ABSTRACT

We study the Bipartite Boolean Quadratic Programming Problem (BBQP) which is an extension of the well known Boolean Quadratic Programming Problem (BQP). Applications of the BBQP include mining discrete patterns from binary data, approximating matrices by rank-one binary matrices, computing the cut-norm of a matrix, and solving optimisation problems such as maximum weight biclique, bipartite maximum weight cut, maximum weight induced sub-graph of a bipartite graph, etc. For the BBQP, we first present several algorithmic components, specifically, hill climbers and mutations, and then show how to combine them in a high-performance metaheuristic. Instead of hand-tuning a standard metaheuristic to test the efficiency of the hybrid of the components, we chose to use an automated generation of a multi-component metaheuristic to save human time, and also improve objectivity in the analysis and comparisons of components. For this we designed a new metaheuristic schema which we call Conditional Markov Chain Search (CMCS). We show that CMCS is flexible enough to model several standard metaheuristics; this flexibility is controlled by multiple numeric parameters, and so is convenient for automated generation. We study the configurations revealed by our approach and show that the best of them outperforms the previous state-of-the-art BBQP algorithm by several orders of magnitude. In our experiments we use benchmark instances introduced in the preliminary version of this paper and described here, which have already become the de facto standard in the BBQP literature.

1. Introduction

The (Unconstrained) Boolean Quadratic Programming Problem (BQP) is to

$$\begin{aligned} &\text{maximise} && f(x) = x^T Q' x + c' x + c'_0 \\ &\text{subject to} && x \in \{0,1\}^n, \end{aligned}$$

where Q' is an $n \times n$ real matrix, c' is a row vector in $\mathbb{R}^n$, and c'_0 is a constant. The BQP is a well-studied problem in the operational research literature (Billionnet, 2004). The focus of this paper is on a problem closely related to BQP, called the *Bipartite (Unconstrained) Boolean Quadratic Programming Problem* (BBQP) (Punnen, Sripratak, & Karapetyan, 2015b). BBQP can be defined as follows:

$$\begin{aligned} &\text{maximise} && f(x,y) = x^T Q y + cx + dy + c_0 \\ &\text{subject to} && x \in \{0,1\}^m, \quad y \in \{0,1\}^n, \end{aligned}$$

where $Q = (q_{ij})$ is an $m \times n$ real matrix, $c = (c_1, c_2, \ldots, c_m)$ is a row vector in $\mathbb{R}^m$, $d = (d_1, d_2, \ldots, d_n)$ is a row vector in $\mathbb{R}^n$, and c_0 is a constant. Without loss of generality, we assume that $c_0 = 0$, and $m \le n$ (which can be achieved by simply interchanging the rows and columns if needed). In what follows, we denote a BBQP instance built on matrix Q, row vectors c and d and $c_0 = 0$ as BBQP(Q, c, d), and (x, y) is a feasible solution of the BBQP if $x \in \{0, 1\}^m$ and $y \in \{0, 1\}^n$. Also x_i stands for the ith component of the vector x and y_j stands for the jth component of the vector y.

A graph theoretic interpretation of the BBQP can be given as follows (Punnen et al., 2015b). Let $I = \{1, 2, \ldots, m\}$ and $J = \{1, 2, \ldots, n\}$. Consider a bipartite graph $G = (I, J, E)$. For each node $i \in I$ and $j \in J$, respective costs c_i and d_j are prescribed. Furthermore, for each $(i, j) \in E$, a cost q_{ij} is given. Then the *Maximum Weight Induced Subgraph Problem* on G is to find a subgraph $G' = (I', J', E')$ such that $\sum_{i \in I'} c_i + \sum_{j \in J'} d_j + \sum_{(i,j) \in E'} q_{ij}$ is maximised, where $I' \subseteq I$, $J' \subseteq J$ and G' is induced by $I' \cup J'$. The Maximum Weight Induced Subgraph Problem on G is precisely the BBQP, where $q_{ij} = 0$ if $(i, j) \notin E$.

* Corresponding author at: Institute for Analytics and Data Science, University of Essex, Colchester CO4 3SQ, UK.
E-mail addresses: daniel.karapetyan@gmail.com (D. Karapetyan), apunnen@sfu.ca (A.P. Punnen), andrew.parkes@nottingham.ac.uk (A.J. Parkes).

http://dx.doi.org/10.1016/j.ejor.2017.01.001

There are some other well known combinatorial optimisation problems that can be modelled as a BBQP. Consider the bipartite graph $G = (I, J, E)$ with w_{ij} being the weight of the edge $(i, j) \in E$. Then the *Maximum Weight Biclique Problem* (MWBP) (Ambühl, Mastrolilli, & Svensson, 2011; Tan, 2008) is to find a biclique in G of maximum total edge-weight. Define

$$q_{ij} = \begin{cases} w_{ij} & \text{if } (i, j) \in E, \\ -M & \text{otherwise,} \end{cases}$$

where M is a large positive constant. Set c and d as zero vectors. Then BBQP(Q, c, d) solves the MWBP (Punnen et al., 2015b). This immediately shows that the BBQP is NP-hard and one can also establish some approximation hardness results with appropriate assumptions (Ambühl et al., 2011; Tan, 2008). Note that the MWBP has applications in data mining, clustering and bioinformatics (Chang, Vakati, Krause, & Eulenstein, 2012; Tanay, Sharan, & Shamir, 2002) which in turn become applications of BBQP.

Another application of BBQP arises in approximating a matrix by a rank-one binary matrix (Gillis & Glineur, 2011; Koyutürk, Grama, & Ramakrishnan, 2005; 2006; Lu, Vaidya, Atluri, Shin, & Jiang, 2011; Shen, Ji, & Ye, 2009). For example, let $H = (h_{ij})$ be a given $m \times n$ matrix and we want to find an $m \times n$ matrix $A = (a_{ij})$, where $a_{ij} = u_i v_j$ and $u_i, v_j \in \{0, 1\}$, such that $\sum_{i=1}^{m} \sum_{j=1}^{n} (h_{ij} - u_i v_j)^2$ is minimised. The matrix A is called a rank one approximation of H and can be identified by solving the BBQP with $q_{ij} = 1 - 2h_{ij}$, $c_i = 0$ and $d_j = 0$ for all $i \in I$ and $j \in J$. Binary matrix factorisation is an important topic in mining discrete patterns in binary data (Lu et al., 2011; Shen et al., 2009). If u_i and v_j are required to be in $\{-1, 1\}$ then also the resulting factorisation problem can be formulated as a BBQP.

The Maximum Cut Problem on a bipartite graph (MaxCut) can be formulated as BBQP (Punnen et al., 2015b) and this gives yet another application of the model. BBQP can also be used to find approximations to the cut-norm of a matrix (Alon & Naor, 2006).

For theoretical analysis of approximation algorithms for BBQP, we refer to Punnen, Sripratak, and Karapetyan (2015a).

A preliminary version of this paper was made available to the research community in 2012 (Karapetyan & Punnen, 2012). Subsequently Glover, Ye, Punnen, and Kochenberger (2015) and Duarte, Laguna, Martí, and Sánchez-Oro (2014) studied heuristic algorithms for the problem. The testbed presented in our preliminary report (Karapetyan & Punnen, 2012) continues to be the source of benchmark instances for the BBQP. In this paper, in addition to providing a detailed description of the benchmark instances, we refine the algorithms reported in Karapetyan and Punnen (2012), introduce a new class of algorithms and give a methodology for automated generation of a multi-component metaheuristic. By (algorithmic) component we mean a black box algorithm that modifies the given solution. All the algorithmic components can be roughly split into two categories: hill climbers, i.e. components that guarantee that the solution not be worsened, and mutations, i.e. components that usually worsen the solution. Our main goals are to verify that the proposed components are sufficient to build a high-performance heuristic for BBQP and also investigate the most promising combinations. By this computational study, we also further support the ideas in the areas of automated parameter tuning and algorithm configuration (e.g. see Adenso-Díaz & Laguna, 2006; Bezerra, López-Ibáñez, & Stützle, 2015; Hutter, Hoos, Leyton-Brown, & Stützle, 2009; Hutter, Hoos, & Stützle, 2007). Thus we rely entirely on automated configuration. During configuration, we use smaller instances compared to those in our benchmark. This way we ensure that we do not over-train our metaheuristics to the benchmark instances – an issue that is often quite hard to avoid with manual design and configuration. We apply the resulting multi-component metaheuristic to our benchmark instances demonstrating that a combination of several simple components can yield powerful metaheuristics clearly outperforming the state-of-the-art BBQP methods.

The main contributions of the paper include:

- In Section 2, we describe several BBQP algorithmic components, one of which is completely new.
- In Section 3 we take the Markov Chain idea, such as in the Markov Chain Hyper-heuristic (McClymont & Keedwell, 2011), but restrict it to use static weights (hence having no online learning, and so, arguably, not best labelled as a 'hyper-heuristic'), but instead adding a powerful extension to it, giving what we call 'Conditional Markov Chain Search (CMCS)'.
- In Section 4 we describe five classes of instances corresponding to various applications of BBQP. Based on these classes, a set of benchmark instances is developed. These test instances were first introduced in the preliminary version of this paper (Karapetyan & Punnen, 2012) and since then used in a number of papers (Duarte et al., 2014; Glover et al., 2015) becoming de facto standard testbed for the BBQP.
- In Section 5 we use automated configuration of CMCS to demonstrate the performance of individual components and their combinations, and give details sufficient to reproduce all of the generated metaheuristics. We also show that a special case of CMCS that we proposed significantly outperforms several standard metaheuristics, on this problem.
- In Section 6 we show that our best machine-generated metaheuristic is, by several orders of magnitude, faster than the previous state-of-the-art BBQP method.

2. Algorithmic components

In this section we introduce several algorithmic components for BBQP. Except for 'Repair' and 'Mutation-X/Y', these components were introduced in Karapetyan and Punnen (2012). A summary of the components discussed below is provided in Table 1. The components are selected to cover a reasonable mix of fast and slow hill climbing operators for intensification, along with mutation operators that can be expected to increase diversification, and with Repair that does a bit of both. Note that a hill climbing component can potentially implement either a simple improvement move or a repetitive local search procedure with iterated operators that terminates only when a local maximum is reached. However in this project we opted for single moves leaving the control to the metaheuristic framework.

2.1. Components: Optimise-X/Optimise-Y

Observe that, given a fixed vector x, we can efficiently compute an optimal $y = y_{\text{opt}}(x)$:

$$y_{\text{opt}}(x)_j = \begin{cases} 1 & \text{if } \sum_{i \in I} q_{ij} x_i + d_j > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

This suggests a hill climber operator Optimise-Y (OptY) that fixes x and replaces y with $y_{\text{opt}}(x)$. Eq. (1) was first introduced in Punnen et al. (2015b) and then used as a neighbourhood search operator in Karapetyan and Punnen (2012), Duarte et al. (2014) and Glover et al. (2015).

OptY implements a hill climber operator in the neighbourhood $N_{\text{OptY}}(x, y) = \{(x, y') : y' \in \{0, 1\}^n\}$, where (x, y) is the original solution. Observe that the running time of OptY is polynomial and the size of the neighbourhood $|N_{\text{OptY}}(x, y)| = 2^n$ is exponential; hence OptY corresponds to an operator that could be used in a very large-scale neighbourhood search (VLNS), a method that is often considered as a powerful approach to hard combinatorial optimisation problems Ahuja, Ergun, Orlin, Punnen, (2002).

Observe that OptY finds a local maximum after the first application because $N(x, y) = N(x, y_{\text{opt}}(y))$ (that is, it is an 'idempotent

Table 1
List of the algorithmic components used in this paper, and described in Section 2.

Name	Description
— Hill climbing operators: that is, components guaranteeing that the solution will not be worsened	
OptX	Optimise-X, Section 2.1. Fixes vector y while optimising x
OptY	As OptX, but reversing roles of x and y
FlpX	Flip-X, Section 2.2. Checks if flipping x_i for some $i \in I$ and subsequently optimising y improves the solution
FlpY	As FlpX, but reversing roles of x and y
— Mutations: that is, components that may worsen the solution	
Repair	Repair, Section 2.3. Finds a single term of the objective function that can be improved and 'repairs' it
MutX4	Mutation-X(4), Section 2.4. Flips x_i for four randomly picked $i \in I$
MutY4	As MutX4, but reversing roles of x and y
MutX16	As MutX4, but for 16 randomly picked x_i
MutY16	As MutY4, but for 16 randomly picked y_i

operator'); hence, there is no gain from applying OptY again immediately after it was applied. Though, for example, iterating and alternating between OptX and OptY would give a VLNS.

Note that $y_{\text{opt}}(x)_j$ can take any value if $\sum_{i\in I} q_{ij}x_i + d_j = 0$, without affecting the objective value of the solution. Thus, one can implement various 'tie breaking' strategies including randomised decision whether to assign 0 or 1 to $y_{\text{opt}}(x)_j$, however in that case OptY would become non-deterministic. In our implementation of OptY we preserve the previous value by setting $y_{\text{opt}}(x)_j = y_j$ for every j such that $\sum_{i\in I} q_{ij}x_i + d_j = 0$. As will be explained in Section 5.1, changing a value y_j is a relatively expensive operation and thus, whenever not necessary, we prefer to avoid such a change.

By interchanging the roles of rows and columns, we also define

$$x_{\text{opt}}(y)_i = \begin{cases} 1 & \text{if} \quad \sum_{j\in J} q_{ij}y_j + c_i > 0, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

and a hill climber operator Optimise-X (OptX) with properties similar to those of OptY.

2.2. Components: Flip-X / Flip-Y

This class of components is a generalisation of the previous one. In Flip-X (FlpX), we try to flip x_i for every $i \in I$, each time re-optimising y. More formally, for $i = 1, 2, \ldots, m$, we compute $x' = (x_1, \ldots, x_{i-1}, 1 - x_i, x_{i+1}, \ldots, x_m)$ and then verify if solution $(x', y_{\text{opt}}(x'))$ is an improvement over (x, y). Each improvement is immediately accepted, but the search carries on for the remaining values of i. In fact, one could consider a generalisation of Flip-X that flips x_i for several i at a time. However, exploration of such a neighbourhood would be significantly slower, and so we have not studied such a generalisation in this paper.

By row/column interchange, we also introduce the Flip-Y (FlpY) hill climbing operator. Clearly, FlpX and FlpY are also VLNS operators, though unlike OptX and OptY they are not idempotent and so could be used consecutively.

FlpX and FlpY were first proposed in Punnen et al. (2015b) and then used in Glover et al. (2015).

2.3. Components: Repair

While all the above methods were handling entire rows or columns, Repair is designed to work on the level of a single element of matrix Q. Repair is a new component inspired by the WalkSAT heuristic for SAT problem (Papadimitriou, 1991; Selman, Kautz, & Cohen, 1995) in that it is a version of 'iterative repair' (Zweben, Davis, Daun, & Deale, 1993) that tries to repair some significant 'flaw' (deficiency of the solution) even if this results in creation of other flaws, in a hope that the newly created flaws could be repaired later. This behaviour, of forcing the repair of randomly selected flaws, gives some stochasticity to the search that is also intended to help in escaping from local optima.

Recall that the objective value of BBQP includes terms $q_{ij}x_iy_j$. For a pair (i, j), there are two possible kinds of flaws: either q_{ij} is negative but is included in the objective value (i.e. $x_iy_j = 1$), or it is positive and not included in the objective value (i.e. $x_iy_j = 0$). The Repair method looks for such flaws, especially for those with large $|q_{ij}|$. For this, it uses the tournament principle; it randomly samples pairs (i, j) and picks the one that maximises $(1 - 2x_iy_j)q_{ij}$. Once an appropriate pair (i, j) is selected, it 'repairs' the flaw; if q_{ij} is positive then it sets $x_i = y_j = 1$; if q_{ij} is negative then it sets either $x_i = 0$ or $y_j = 0$ (among these two options it picks the one that maximises the overall objective value). Our implementation of Repair terminates after the earliest of two: (i) finding 10 flaws and repairing the biggest of them, or (ii) sampling 100 pairs (i, j).

Note that one could separate the two kinds of flaws, and so have two different methods: Repair-Positive, that looks for and repairs only positive 'missing' terms of the objective function, and Repair-Negative, that looks for and repairs only negative included terms of the objective function. However, we leave these options to future research.

2.4. Components: Mutation-X/Mutation-Y

In our empirical study, we will use some pure mutation operators of various strengths to escape local maxima. For this, we use the $N_{\text{OptX}}(x, y)$ neighbourhood. Our Mutation-X(k) operator picks k distinct x variables at random and then flips their values, keeping y unchanged. Similarly we introduce Mutation-Y(k). In this paper we use $k \in \{4, 16\}$, and so have components which we call MutX4, MutX16, MutY4 and MutY16.

An operator similar to Mutation-X/Y was used in Duarte et al. (2014).

3. The Markov chain methods

The algorithmic components described in Section 2 are designed to work within a metaheuristic; analysis of each component on its own would not be sufficient to conclude on its usefulness within the context of a multi-component system. To avoid bias due to picking one or another metaheuristic, and to save human time on hand-tuning it, we chose to use a generic schema coupled with automated configuration of it.

3.1. Conditional Markov Chain Search (CMCS)

The existing framework that was closest to our needs was the Markov Chain Hyper-Heuristic (MCHH) (McClymont & Keedwell, 2011). MCHH is a relatively simple algorithm that applies components in a sequence. This sequence is a Markov chain; the 'state' in the Markov chain is just the operator that is to be applied, and so the Markov nature means that the transition to a new state

(component/operator) only depends on the currently-applied component and transition probabilities. Transition probabilities, organised in a transition matrix, are obtained in MCHH dynamically, by learning most successful sequences.

While MCHH is a successful approach capable of effectively utilising several algorithmic components, it does not necessarily provide the required convenience of interpretation of performance of individual components and their combinations because the transition probabilities in MCHH change dynamically. To address this issue, we chose to fix the transition matrix and learn it offline. We can then perform the analysis by studying the learnt transition probabilities.

The drawback of learning the probabilities offline is that MCHH with static transition matrix receives no feedback from the search process and, thus, has no ability to respond to the instance and solution properties. To enable such a feedback, we propose to extend the state of the Markov chain with the information about the outcome of the last component execution; this extension is simple but will prove to be effective. In particular, we suggest to distinguish executions that improved the solution quality, and executions that worsened, or did not change, the solution quality.

We call our new approach *Conditional Markov Chain Search* (CMCS). It is parameterised with two transition matrices: M^{succ} for transitions if the last component execution was successful (improved the solution), and M^{fail} for transitions if the last component execution failed (has not improved the solution).[1]

Algorithm 1: Conditional Monte-Carlo search.

input : Ordered set of components $\mathcal{H}$;
input : Matrices M^{succ} and M^{fail} of size $|\mathcal{H}| \times |\mathcal{H}|$;
input : Objective function $f(S)$ to be maximised;
input : Initial solution S;
input : Termination time *terminate-at*;
$S^* \leftarrow S$;
$h \leftarrow 1$;
while *now* $<$ *terminate-at* **do**
  $f_{\text{old}} \leftarrow f(S)$;
  $S \leftarrow \mathcal{H}_h(S)$;
  $f_{\text{new}} \leftarrow f(S)$;
  if $f_{new} > f_{old}$ **then**
    $h \leftarrow RouletteWheel(M^{\text{succ}}_{h,1}, M^{\text{succ}}_{h,2}, \ldots, M^{\text{succ}}_{h,|\mathcal{H}|})$;
    if $f(S) > f(S^*)$ **then**
      $S^* \leftarrow S$;
  else
    $h \leftarrow RouletteWheel(M^{\text{fail}}_{h,1}, M^{\text{fail}}_{h,2}, \ldots, M^{\text{fail}}_{h,|\mathcal{H}|})$;
return S^*;

CMCS does not in itself employ any learning during the search process, but is configured by means of offline learning, and so the behaviour of any specific instance of CMCS is defined by two matrices M^{succ} and M^{fail} of size $|\mathcal{H}| \times |\mathcal{H}|$ each. Thus, we refer to the general idea of CMCS as *schema*, and to a concrete instance of CMCS, i.e. specific values of matrices M^{succ} and M^{fail}, as *configuration*.

For the termination criterion, we use a predefined time after which CMCS terminates. This is most appropriate, as well as convenient, when we need to compare metaheuristics and in which different components run at different speeds so that simple counting of steps would not be a meaningful termination criterion.

CMCS requires an initial solution; this could have been supplied from one of the several construction heuristics developed for BBQP (Duarte et al., 2014; Karapetyan & Punnen, 2012), however, to reduce potential bias, we initialise the search with a randomly generated solution with probability of each of $x_i = 1$ and $y_j = 1$ being 50%.

3.2. CMCS properties

Below we list some of the properties of CMCS that make it a good choice in our study. We also believe that it will be useful in future studies in a similar way.

- CMCS is able to combine several algorithmic components in one search process, and with each component taken as a black box.
- CMCS has parameters for inclusion or exclusion of individual components as we do not know in advance if any of our components have poor performance. This is particularly true when considering that performance of a component might well depend on which others are available – some synergistic combinations might be much more powerful than the individuals would suggest.
- CMCS has parameters that permit some components to be used more often than others as some of our hill climbing operators are significantly faster than others; this also eliminates the necessity to decide in advance on the frequency of usage of each of the components. Appropriate choices of the parameters should allow the imbalance of component runtimes to be exploited.
- CMCS is capable of exploiting some (recent) history of the choices made by the metaheuristic, as there might be efficient sequences of components that should be exploitable.
- As we will show later, CMCS is powerful enough to model some standard metaheuristics and, thus, allows easy comparison with standard approaches.
- The performance of CMCS does not depend on the absolute values of the objective function; it is rank-based in that it only uses the objective function to find out if a new solution is better than the previous solution. This property helps CMCS perform well across different families of instances. In contrast, methods such as Simulated Annealing, depend on the absolute values of the objective function and thus often need to be tuned for each family of instances, or else need some mechanism to account for changes to the scale of the objective function.
- The transition matrices of a tuned CMCS configuration allow us conveniently interpret the results of automated generation.

3.3. Special cases of CMCS

Several standard metaheuristics are special cases of CMCS. If $\mathcal{H} = \{\text{HC}, \text{Mut}\}$ includes a hill climbing operator 'HC' and a mutation 'Mut' then

$$M^{\text{succ}} = \begin{pmatrix} & \text{HC} & \text{Mut} \\ \hline \text{HC} & 1 & 0 \\ \text{Mut} & 1 & 0 \end{pmatrix} \quad \text{and}$$

$$M^{\text{fail}} = \begin{pmatrix} & \text{HC} & \text{Mut} \\ \hline \text{HC} & 0 & 1 \\ \text{Mut} & 1 & 0 \end{pmatrix}$$

[1] Note that executions that do not change the solution quality at all are also considered as a failure. This allows us to model a hill climber that is applied repeatedly until it becomes trapped in a local maximum.
Let $\mathcal{H}$ be the pool of algorithmic components. CMCS is a single-point metaheuristic that applies one component $h \in \mathcal{H}$ at a time, accepting both improving and worsening moves. The next component $h' \in \mathcal{H}$ to be executed is determined by a function *next* : $\mathcal{H} \to \mathcal{H}$. In particular, h' is chosen using roulette wheel with probabilities $p_{hh'}$ of transition from h to h' defined by matrix M^{succ} if the last execution of h was successful and M^{fail} otherwise. All the moves are always accepted in CMCS. Pseudo-code of the CMCS schema is given in Algorithm 1.

implements Iterated Local Search (Lourenço, Martin, and Stützle, 2010); the algorithm repeatedly applies HC until it fails, then applies Mut, and then returns to HC disregarding the success or failure of Mut.

If $M^{\text{succ}}_{h,h'} = M^{\text{fail}}_{h,h'} = 1/|\mathcal{H}|$ for all $h, h' \in \mathcal{H}$ then CMCS implements a simple uniform random choice of component (Cowling, Kendall, & Soubeiga, 2001).

A generalisation of the uniform random choice is to allow non-uniform probabilities of component selection. We call this special case *Operator Probabilities* (Op. Prob.) and model it by setting $M^{\text{succ}}_{h,h'} = M^{\text{fail}}_{h,h'} = p_{h'}$ for some vector p of probabilities. Note that Operator Probabilities is a static version of a Selection Hyper-heuristic (Cowling et al., 2001).

Obviously, if $M^{\text{succ}} = M^{\text{fail}}$ then CMCS implements a static version of MCHH.

By allowing $M^{\text{succ}} \neq M^{\text{fail}}$, it is possible to implement a Variable Neighbourhood Search (VNS) using the CMCS schema. For example, if

$$M^{\text{succ}} = \begin{array}{c|cccc} & \text{HC1} & \text{HC2} & \text{HC3} & \text{Mut} \\ \hline \text{HC1} & 1 & 0 & 0 & 0 \\ \text{HC2} & 1 & 0 & 0 & 0 \\ \text{HC3} & 1 & 0 & 0 & 0 \\ \text{Mut} & 1 & 0 & 0 & 0 \end{array}$$

and

$$M^{\text{fail}} = \begin{array}{c|cccc} & \text{HC1} & \text{HC2} & \text{HC3} & \text{Mut} \\ \hline \text{HC1} & 0 & 1 & 0 & 0 \\ \text{HC2} & 0 & 0 & 1 & 0 \\ \text{HC3} & 0 & 0 & 0 & 1 \\ \text{Mut} & 1 & 0 & 0 & 0 \end{array}$$

then CMCS implements a VNS that applies HC1 until it fails, then applies HC2. If HC2 improves the solution then the search gets back to HC1; otherwise HC3 is executed. Similarly, if HC3 improves the solution then the search gets back to HC1; otherwise current solution is a local maximum with respect to the neighbourhoods explored by HC1, HC2 and HC3 (assuming they are deterministic) and mutation Mut is applied to diversify the search.

However, even though the previous examples are well-known metaheuristics, they are rather special cases from the perspective of CMCS, which allows much more sophisticated strategies. For example, we can implement a two-loop heuristic, which alternates hill climbing operator HC1 and mutation Mut1 until HC1 fails to improve the solution. Then the control is passed to the second loop, alternating HC2 and Mut2. Again, if HC2 fails, the control is passed to the first loop.

To describe such more sophisticated strategies, it is convenient to represent CMCS configurations with automata as in Fig. 1. Blue and red lines correspond to transitions in case of successful and unsuccessful execution of the components, respectively. Probabilities of each transition are shown with line widths (in Fig. 1 all the shown probabilities are 100%). The advantage of automata representation is that it visualises the probabilities of transition and sequences in which components are executed (and so complements, not supplants, the formal description via the pseudo-code and the explicit transition matrices), as common when describing transition systems.

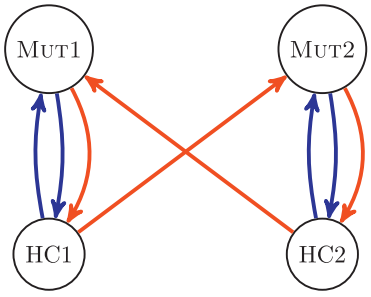


Fig. 1. Implementation of a two-loop heuristic within the CMCS framework. Blue lines show transitions in case of success, and red lines show transitions in case of failure of the component. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

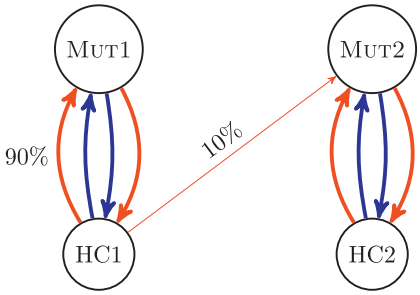


Fig. 2. Implementation of a two-phase heuristic with probabilistic transition from the first phase to the second phase. All the probabilities are 100% unless otherwise labelled.

The transitions in the above example are deterministic, however, this is not an inherent limitation; for example, one could implement a two phase search with the transition being probabilistic, see Fig. 2. We also note here that CMCS can be significantly enriched by having several copies of each component in $\mathcal{H}$ and/or employing dummy components for describing more sophisticated behaviours; but we leave these possibilities to future work.

These are just some of the options available with CMCS, showing that it is potentially a powerful tool. However, this flexibility does come with the associated challenge – of configuring the matrices to generate effective metaheuristics. For example, if $|\mathcal{H}| = 10$ then CMCS has $2|\mathcal{H}|^2 = 200$ continuous parameters.

By simple reasoning we can fix the values of a few of these parameters:

- If component h is a deterministic hill climbing operator then $M^{\text{fail}}_{h,h} = 0$, as when it fails then the solution remains unchanged and so immediate repetition is pointless.
- If component h is an idempotent operator (e.g. OptX or OptY) then $M^{\text{succ}}_{h,h} = M^{\text{fail}}_{h,h} = 0$; again there is no use in applying h several times in a row.

Nevertheless, the significant number of remaining parameters of CMCS makes it hard to configure. For this reason we propose, and exploit a special case of the CMCS schema, with much fewer parameters but that still provides much of the power of the framework of the full CMCS. Specifically, we allow at most k non-zero elements in each row of M^{succ} and M^{fail}, calling the resulting metaheuristic 'CMCS[k-row]'. Clearly, CMCS[$|\mathcal{H}|$-row] is identical to the full version of CMCS. In practice, however, we expect one to use only smaller values of k; either $k = 1$ or $k = 2$.

When $k = 1$, the corresponding automata has at most one outgoing 'success' arc, and one outgoing 'failure' arc for each component. Hence CMCS turns into a deterministic control mechanism. Note that iterated local search and VNS are in fact special cases of CMCS[1-row].

When $k = 2$, the corresponding automata has at most two outgoing 'success' arcs from each component, and their total probability of transition is 100%. Hence, the 'success' transition is defined by a pair of components and the split of probabilities between them. 'Failure' transition is defined in the same way.

In Section 5, we show that CMCS[2-row] is sufficiently powerful to implement complex component combinations but is much easier to configure and analyse than full CMCS.

4. Benchmark instances

The testbed which is currently de facto standard for BBQP was first introduced in our unpublished work (Karapetyan & Punnen, 2012). Our testbed consists of five instance types that correspond to some of the real life applications of BBQP. Here we provide the description of it, and also make it openly available for download.[2] We keep record of the best known solutions for each of the test instances which will also be placed on the download page.

In order to generate some of the instances, we need random bipartite graphs. To generate a random bipartite graph $G = (V, U, E)$, we define seven parameters, namely $m = |V|$, $n = |U|$, $\underline{d}_1$, $\bar{d}_1$, $\underline{d}_2$, $\bar{d}_2$ and μ such that $0 \le \underline{d}_1 \le \bar{d}_1 \le n$, $0 \le \underline{d}_2 \le \bar{d}_2 \le m$, $m\underline{d}_1 \le n\bar{d}_2$ and $m\bar{d}_1 \ge n\underline{d}_2$.

The bipartite graph generator proceeds as follows.

1. For each node $v \in V$, select d_v uniformly at random from the range $[\underline{d}_1, \bar{d}_1]$.
2. For each node $u \in U$, select d_u uniformly at random from the range $[\underline{d}_2, \bar{d}_2]$.
3. While $\sum_{v\in V} d_v \ne \sum_{u\in U} d_u$, alternatively select a node in V or U and re-generate its degree as described above.[3]
4. Create a bipartite graph $G = (V, U, E)$, where $E = \emptyset$.
5. Randomly select a node $v \in V$ such that $d_v > \deg v$ (if no such node exists, go to the next step). Let $U' = \{u \in U : \deg u < d_u \text{ and } (v, u) \notin E\}$. If $U' \ne \emptyset$, select a node $u \in U'$ randomly. Otherwise randomly select a node $u \in U$ such that $(v, u) \notin E$ and $d_u > 0$; randomly select a node $v' \in V$ adjacent to u and delete the edge (v', u). Add an edge (v, u). Repeat this step.
6. For each edge $(v, u) \in E$ select the weight w_{vu} as a normally distributed integer with standard deviation $\sigma = 100$ and given mean μ.

The following are the instance types used in our computational experiments:

1. The *Random* instances are as follows: q_{ij}, c_i and d_j are integers selected at random with normal distribution (mean $\mu = 0$ and standard deviation $\sigma = 100$).
2. The *Max Biclique* instances model the problem of finding a biclique of maximum weight in a bipartite graph. Let $G = (I, J, E)$ be a random bipartite graph with $\underline{d}_1 = n/5$, $\bar{d}_1 = n$, $\underline{d}_2 = m/5$, $\bar{d}_2 = m$ and $\mu = 100$. (Note that setting μ to 0 would make the weight of any large biclique likely to be around 0, which would make the problem much easier.) If w_{ij} is the weight of an edge $(i, j) \in E$, set $q_{ij} = w_{ij}$ for every $i \in I$ and $j \in J$ if $(i, j) \in E$ and $q_{ij} = -M$ otherwise, where M is large number. Set c and d as zero vectors.
3. The *Max Induced Subgraph* instances model the problem of finding a subset of nodes in a bipartite graph that maximises the total weight of the induced subgraph. The Max Induced Subgraph instances are similar to the Max Biclique instances except that $q_{ij} = 0$ if $(i, j) \notin E$ and $\mu = 0$. (Note that if $\mu > 0$ then the optimal solution would likely include all or almost all the nodes and, thus, the problem would be relatively easy.)
4. The *MaxCut* instances model the MaxCut problem as follows. First, we generate a random bipartite graph as for the Max Induced Subgraph instances. Then, we set $q_{ij} = -2w_{ij}$ if $(i, j) \in E$ and $q_{ij} = 0$ if $(i, j) \notin E$. Finally, we set $c_i = \frac{1}{2}\sum_{j\in J} q_{ij}$ and $d_j = \frac{1}{2}\sum_{i\in I} q_{ij}$. For an explanation, see Punnen et al. (2015b).
5. The *Matrix Factorisation* instances model the problem of producing a rank one approximation of a binary matrix. The original matrix $H = (h_{ij})$ (see Section 1) is generated randomly with probability 0.5 of $h_{ij} = 1$. The values of q_{ij} are then calculated as $q_{ij} = 1 - 2h_{ij}$, and c and d are zero vectors.

Our benchmark consists of two sets of instances: Medium and Large. Each of the sets includes one instance of each type (Random, Max Biclique, Max Induced Subgraph, MaxCut and Matrix Factorisation) of each of the following sizes:

Medium : 200×1000, 400×1000, 600×1000, 800×1000, 1000×1000;

Large : 1000×5000, 2000×5000, 3000×5000, 4000×5000, 5000×5000.

Thus, in total, the benchmark includes 25 medium and 25 large instances.

5. Metaheuristic design

In this section we describe configuration of metaheuristics as discussed in Section 3 and using the BBQP components given in Section 2. In Sections 5.1 and 5.2 we give some details about our experiments, then in Section 5.3 describe the employed automated configuration technique, in Section 5.4 we provide details of the configured metaheuristics, and in Section 5.5 analyse the results.

Our test machine is based on two Intel Xeon CPU E5-2630 v2 (2.6 gigahertz) and has 32 gigabytes RAM installed. Hyperthreading is enabled, but we never run more than one experiment per physical CPU core concurrently, and concurrency is not exploited in any of the tested solution methods.

5.1. Solution representation

We use the most natural solution representation for BBQP, i.e. simply storing vectors x and y. However, additionally storing some auxiliary information with the solution can dramatically improve the performance of algorithms. We use a strategy similar to the one employed in Glover et al. (2015). In particular, along with vectors x and y, we always maintain values $c_i + \sum_j y_j q_{ij}$ for each i, and $d_j + \sum_i x_i q_{ij}$ for each j. Maintenance of this auxiliary information slows down any updates of the solution but significantly speeds up the evaluation of potential moves, which is what usually takes most of time during the search.

5.2. Solution polishing

As in many single-point metaheuristics, the changes between diversifying and intensifying steps of CMCS mean that the best found solution needs to be stored, and also that it is not necessarily a local maximum with respect to all the available hill climbing operators. Hence, we apply a polishing procedure to every CMCS configuration produced in this study, including special cases of VNS, Op. Prob. and MCHH. Our polishing procedure is executed after the CMCS finishes its work, and it is aimed at improving the best solution found during the run of CMCS. It sequentially executes OptX, OptY, FlpX and FlpY components, restarting this sequence every time an improvement is found. When none of these algorithms can improve the solution, that is, the solution is a local maximum with respect to all of our hill climbing operators, the procedure terminates.

While taking very little time, this polishing procedure has notably improved our results. We note that this polishing stage is a Variable Neighbourhood Descent, and thus a special case of CMCS;

[2] http://csee.essex.ac.uk/staff/dkarap/?page=publications&key=CMCS-BBQP .

[3] In practice, if $m(\underline{d}_1 + \bar{d}_1) \approx n(\underline{d}_2 + \bar{d}_2)$, this algorithm converges very quickly. However, in theory it may not terminate in finite time and, formally speaking, there needs to be a mechanism to guarantee convergence. Such a mechanism could be turned on after a certain (finite) number of unsuccessful attempts, and then it would force the changes of degrees d_v that reduce $|\sum_{v\in V} d_v - \sum_{u\in U} d_u|$.

hence, the final polishing could be represented as a second phase of CMCS. We also note that the Tabu Search algorithm, against which we compare our best CMCS configuration in Section 6.1, uses an equivalent polishing procedure applied to each solution and thus the comparison is fair.

5.3. Approach to configuration of the metaheuristics

Our ultimate goal in this experiment is to apply automated configuration (e.g. in the case of CMCS, to configure M^{succ} and M^{fail} matrices), which would compete with the state-of-the-art methods on the benchmark instances (which have sizes 200 × 1000 to 5000 × 5000) and with running times in the order of several seconds to several minutes. As explained in Section 3, instead of hand designing a metaheuristic we chose to use automated generation based on the CMCS schema. Automated generation required a set of training instances. Although straightforward, directly training on benchmark instances would result in over-training (a practice generally considered unfair because an over-trained heuristic might perform well only on a very small set of instances on which it is tuned and then tested) and also would take considerable computational effort. Thus, for training we use instances of size 200 × 500. We also reduced the running times to 100 milliseconds per run of each candidate configuration, that is, matrices when configuring CMCS or MCHH, probability vector for Op. Prob., and component sequence for VNS.

Let T be the set of instances used for training. Then our objective function for configuration is

$$f(h,T) = \frac{1}{|T|}\sum_{t\in T}\frac{f_{\text{best}}(t) - h(t)}{f_{\text{best}}(t)} \cdot 100\%, \tag{3}$$

where h is the evaluated heuristic, $h(t)$ is the objective value of solution obtained by h for instance t, and $f_{\text{best}}(t)$ is the best known solution for instance t. For the training set, we used instances of all of the types. In particular, we use one instance of each of the five types (see Section 4), all of size 200 × 500, and each of these training instances is included in T 10 times, thus $|T| = 50$ (we observed that without including each instance several times the noise level significantly obfuscated results). Further, when testing the top ten candidates, we include each of the five instances 100 times in T, thus having $|T| = 500$.

We consider four types of metaheuristics: VNS, Op. Prob., MCHH and CMCS[2-row], all of which are also special cases of CMCS. All the components discussed in Section 2, and also briefly described in Table 1, are considered for inclusion in all the metaheuristics. Additionally, since Repair is a totally new component, we want to confirm its usefulness. For this we also study a special case of CMCS[2-row] which we call '*CMCS[2-row reduced]*'. In CMCS[2-row reduced], the pool of potential components includes all the components in Table 1 except Repair.

To configure VNS and Op. Prob., we use brute force search as we can reasonably restrict the search to a relatively small number of options. In particular, when configuring Op. Prob., the number of components $|\mathcal{H}|$ (recall that $\mathcal{H}$ is the set of components employed by the metaheuristic) is restricted to at most four, and weights of individual components are selected from {0.1, 0.2, 0.5, 0.8, 1} (these weights are then rescaled to obtain probabilities). We also require that there has to be at least one hill climbing operator in $\mathcal{H}$ as otherwise there would be no pressure to improve the solution, and one mutation operator as otherwise the search would quickly become trapped in a local maximum. Note that we count Repair as a mutation as, although designed to explicitly fix flaws, it is quite likely to worsen the solution (even if in the long run this will be beneficial). When configuring VNS, $\mathcal{H}$ includes one or several hill climbing operators and one mutation and the configuration process has to also select the order in which they are applied.

	OptX	OptY	FlpX	MutY4	MutX16
OptX	–	78.2%	5.4%	12.9%	3.5%
OptY	86.9%	–	0.0%	13.1%	0.0%
FlpX	16.2%	30.1%	19.4%	5.0%	29.3%
MutY4	35.6%	24.7%	22.9%	4.1%	12.8%
MutX16	1.4%	84.6%	0.0%	14.0%	0.0%

Fig. 3. Transition matrix of MCHH. Dashes show prohibited transitions, i.e. the transitions that are guaranteed to be useless and so are constrained to zero, as opposed to being set to zero by the tuning generation process. In this table, and subsequent ones, the row specifies the previously executed component, and the column specifies the next executed component.

To configure CMCS and static MCHH, we use a simple evolutionary algorithm, with the solution describing matrices M^{succ} and M^{fail} (accordingly restricted), and fitness function (3). Implementation of a specialised tuning algorithm has an advantage over the general-purpose automated algorithm configuration packages, as a specialised system can exploit the knowledge of the parameter space (such as entanglement of certain parameters). In this project, our evolutionary algorithm employs specific neighbourhood operators that intuitively make sense for this particular application. For example, when tuning 2-row, we employ, among others, a mutation operator that swaps the two non-zero weights in a row of a weight matrix. Such a move is likely to be useful for 'exploitation'; however it is unlikely to be discovered by a general purpose parameter tuning algorithm.

We compared the tuning results of our CMCS-specific algorithm to ParamILS (Hutter et al., 2009), one of the leading general purpose automated parameter tuning/algorithm configuration software. We found out that, while ParamILS performs well, our specialised algorithm clearly outperforms it, producing much better configurations. It should be noted that there can be multiple approaches to encode matrices M^{succ} and M^{fail} for ParamILS. We tried two most natural approaches and both attempts were relatively unsuccessful; however, it is possible that future research will reveal more efficient ways to represent the key parameters of CMCS. We also point out that CMCS can be a new interesting benchmark for algorithm configuration or parameter tuning software.

5.4. Configured metaheuristics

In this section we describe the configurations of each type (VNS, Op. Prob., MCHH, CMCS[2-row reduced] and CMCS[2-row]) generated as described in Section 5.3. From now on we refer to the obtained configurations by the name of their types. Note that the structures described in this section are all machine-generated, and thus when we say that 'a metaheuristic chose to do something', we mean that such a decision emerged from the generation process; the decision was not a human choice.

VNS chose three hill climbing operators, OptY, FlpY and OptX, and a mutation MutX16, and using the order as written. It is interesting to observe that this choice and sequence can be easily explained. Effectively, the search optimises y given a fixed x (OptY), then tries small changes to x with some lookahead (FlpY), and if this fails then optimises x globally but without lookahead (OptX). If the search is in a local maximum with respect to all three neighbourhoods then the solution is perturbed by a strong mutation MutX16. Observe that the sequence of hill climbing operators does not obey the generally accepted rule of thumb to place smaller neighbourhoods first; the third hill climbing operator OptX has clearly smaller neighbourhood than FlpY. However, this sequence has an interesting internal logic. Whenever FlpY succeeds in improving the solution, the resultant solution is a local minimum with respect to OptX. Accordingly, VNS jumps back to OptY when FlpY succeeds. However, if FlpY fails then the solution might not

	OptX	OptY	FlpX	MutX4	MutY4	MutY16
OptX	–	100%	.	.	.	.
OptY	8%	–	.	.	92%	.
FlpX	38%	62%	.	.	.	.
MutX4	.	.	.	100%	.	.
MutY4	45%	.	.	.	.	55%
MutY16	.	.	.	54%	.	46%

(a) M^{succ}

	OptX	OptY	FlpX	MutX4	MutY4	MutY16
OptX	–	.	.	.	100%	.
OptY	.	–	80%	20%	.	.
FlpX	.	.	–	.	.	100%
MutX4	28%	.	72%	.	.	.
MutY4	68%	.	.	.	.	32%
MutY16	51%	.	.	.	.	49%

(b) M^{fail}

Fig. 4. Transition matrices of CMCS[2-row reduced]. Dashes show prohibited transitions, see Section 3.3. CMCS[2-row reduced] transition frequencies are shown in Fig. 7a.

	OptX	OptY	FlpX	Repair	MutY4	MutY16
OptX	–	66%	.	.	.	34%
OptY	41%	–	.	.	59%	.
FlpX	.	29%	71%	.	.	.
Repair	41%	.	.	.	.	59%
MutY4	.	.	40%	60%	.	.
MutY16	.	.	.	55%	45%	.

(a) M^{succ}

	OptX	OptY	FlpX	Repair	MutY4	MutY16
OptX	–	.	.	.	100%	.
OptY	25%	–	.	75%	.	.
FlpX	45%	55%	–	.	.	.
Repair	2%	.	.	.	.	98%
MutY4	87%	.	.	13%	.	.
MutY16	.	.	.	.	82%	18%

(b) M^{fail}

Fig. 5. Transition matrices of CMCS[2-row], our best performing metaheuristic. Dashes show prohibited transitions. CMCS[2-row] transition frequencies are shown in Fig. 7b.

be a local minimum with respect to OptX, and then OptX is executed. This shows that the automated configuration is capable of generating meaningful configurations which are relatively easy to explain but might not be so easy to come up with.

The Op. Prob. chose four components: OptX (probability of picking is 40%), FlpX (20%), Repair (20%) and MutX16 (20%). Note that the actual runtime frequency of OptX is only about 30% because the framework will never execute OptX twice in a row.

Out of 9 components, MCHH chose five: OptX, OptY, FlpX, MutY4 and MutX16. The generated transition matrix (showing the probabilities of transitions) is given in Fig. 3.

CMCS[2-row reduced] chose to use only OptX, OptY, FlpX, MutX4, MutY4 and MutY16 from the pool of 8 components it was initially permitted (recall that CMCS[2-row reduced] was not allowed to use Repair), and transition matrices as given in Fig. 4 and visually illustrated in Fig. 7a. The line width in Fig. 7a indicates the frequency of the transition when we tested the configuration on the tuning instance set. Although these frequencies may slightly vary depending on the particular instance, showing frequencies preserves all the advantages of showing probabilities but additionally allows one to see: (i) how often a component is executed (defined by the total width of all incoming/outgoing arrows), (ii) the probability of success of a component (defined by the total width of blue outgoing arrows compared to the total width of the red outgoing arrows), and (iii) most common sequences of component executions (defined by thickest arrows).

CMCS[2-row] decided to use only OptX, OptY, FlpX, Repair, MutY4 and MutY16 from the set of 9 moves it was initially permitted, and transition matrices as shown in Fig. 5.

5.5. Analysis of components and metaheuristics

Table 2 gives the tuning objective function (3) and the average number of component executions per run (i.e. in 100 milliseconds when solving a 200 × 500 instance) for each metaheuristic. CMCS, even if restricted to CMCS[2-row] and even if the pool of components is reduced, outperforms all standard metaheuristics (VNS, Op. Prob. and MCHH), even though Op. Prob. and VNS benefit from higher quality configuration (recall that VNS and Op. Prob. are configured using complete brute-force search). An interesting observation is that the best performing metaheuristics mostly employ fast components thus being able to run many more iterations than, say, VNS or Op. Prob.

Table 2
Performance of the emergent metaheuristics on the training instance set. Rows are ordered by performance of metaheuristics, from worst to best.

Metaheuristic	Objective value (3)	Comp. exec.
VNS	0.598%	384
Op. Prob.	0.448%	520
MCHH	0.395%	2008
CMCS[2-row reduced]	0.256%	5259
CMCS[2-row]	0.242%	5157

Fig. 6 gives the relative frequency of usage of each component by each metaheuristic. Most of the components appear to be useful within at least one of the considered metaheuristic schemas; only MutX4 is almost unused. It is however not surprising to observe some imbalance between the Mutation-X and Mutation-Y components because the number of rows is about half of the number of columns in the training instances. The selection of components is hard to predict as it significantly depends on the metaheuristic schema; indeed, different types of metaheuristics may be able to efficiently exploit different features of the components. Thus components should not be permanently discarded or selected based only on expert intuition and/or a limited number of experiments. We believe that the approach to component usage analysis proposed and used in this paper (and also in works such as Hutter et al., 2009; Bezerra et al., 2015, and others) is in many circumstances more comprehensive than manual analysis.

While frequencies of usage of the components vary between all the metaheuristics, Op. Prob. is clearly an outlier in this respect. We believe that this reflects the fact that Op. Prob. is the only metaheuristic among the considered ones that does not have any form of memory and thus does not control the order of components. Thus it prefers strong (possibly slow) components whereas other metaheuristics have some tendency to form composite components from fast ones, with the latter (history-based) approach apparently being superior.

More information about the performance of CMCS[2-row reduced] and CMCS[2-row] configurations can be collected from Fig. 7 detailing the runtime frequencies of transitions in each of them. Edge width here is proportional to square root of the

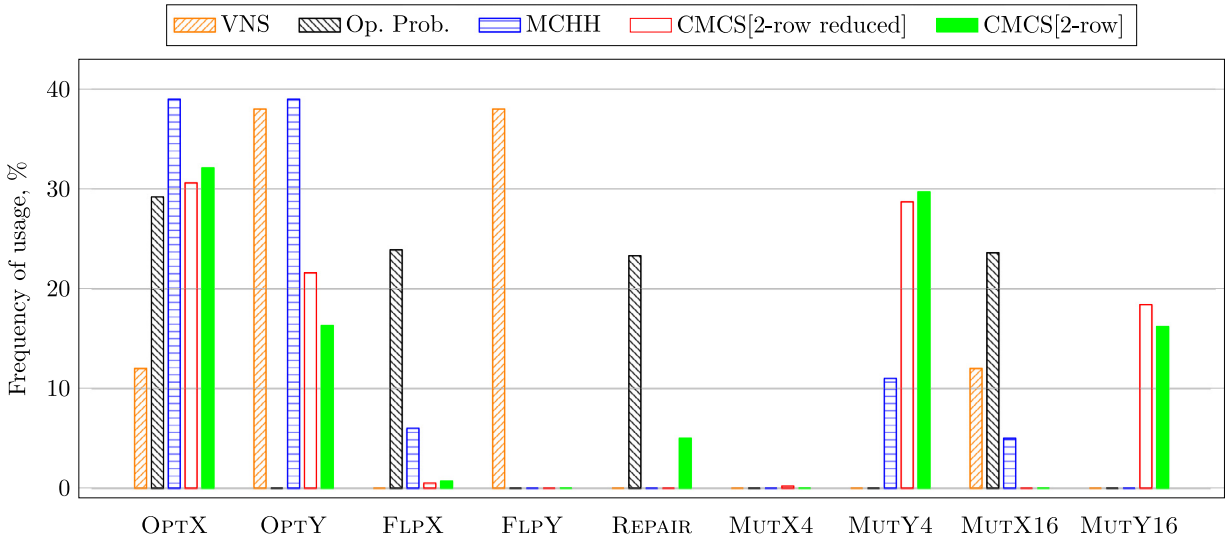
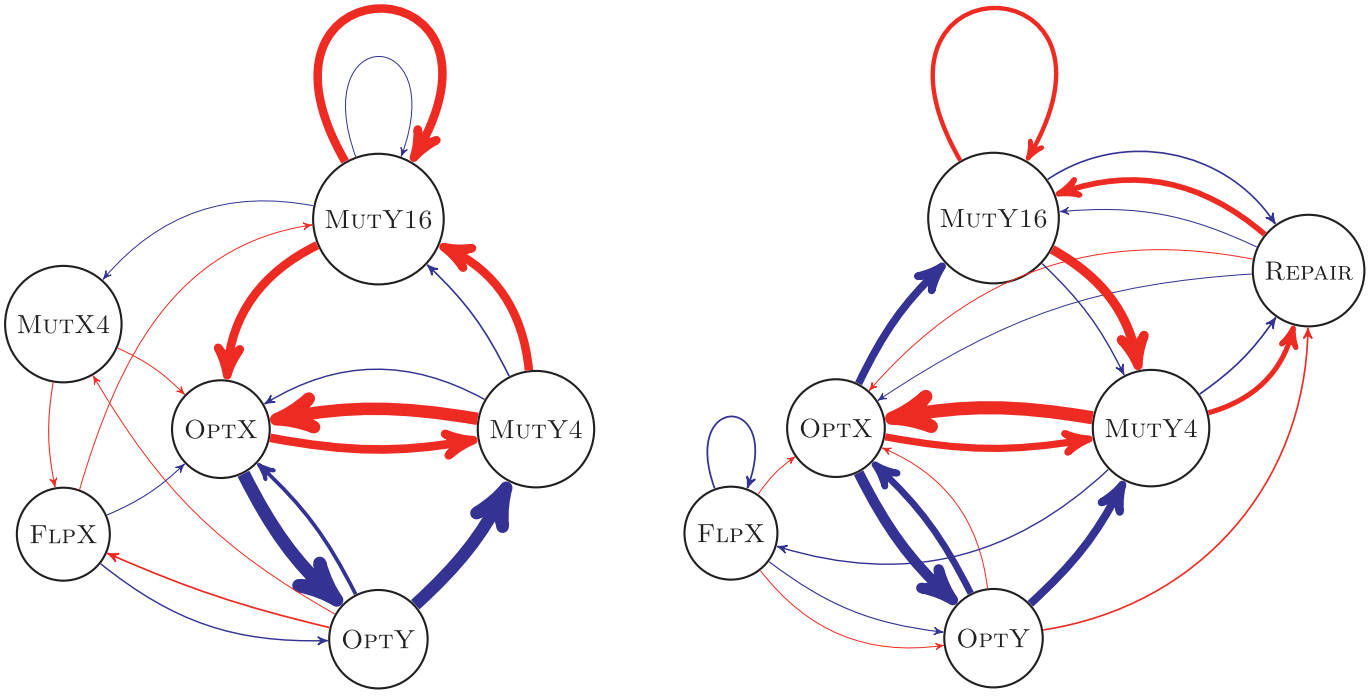


Fig. 6. Runtime frequency of usage of the components in tuned metaheuristics.



(a) Emergent CMCS[2-row reduced], i.e. a metaheuristic which was allowed to use any components except Repair.

(b) Emergent CMCS[2-row], i.e. our best performing metaheuristic which was allowed to use any components.

Fig. 7. Runtime frequencies of CMCS[2-row reduced] and CMCS[2-row] tested on the training instance set. The names and brief descriptions of each component are given in Table 1.

runtime frequency of the corresponding transition occurring in several test runs; thus it allows to see not only the probabilities of transitions from any individual component, but also how frequently that component was executed and how often it was successful, compared to other components.

Firstly, we observe that the two metaheuristics employ similar sets of components; the only difference is that CMCS[2-row] does not use MutX4 but adds Repair (recall that Repair was purposely removed from the pool of components of CMCS[2-row reduced]). Furthermore, the core components (OptX, OptY, MutY4 and MutY16) are exactly the same, and most of interconnections between them are similar. However, the direction of transitions to and from MutY16 is different. One may also notice that both metaheuristics have 'mutation' blocks; that is, mutations that are often executed in sequences. It is then not surprising that CMCS[2-row] connects Repair to the other mutation components.

Both metaheuristics include some natural patterns such as alternation of OptX and OptY, or iterated local search OptX–MutY4, which we could also expect in a hand-designed metaheuristic. It is also easy to suggest an explanation for the loop at MutY16 as it allows the component to be repeated a couple of times intensifying the mutation. However, the overall structure of the metaheuristics is complex and hard to explain. Our point here is that, although the observed chains of components make sense, it is unlikely that a human expert would come up with a heuristic of such a level of detail.

Table 3
Evaluation of metaheuristics on Medium Instances, 10 seconds per run. Reported are the gaps, as percentages, to the best known solutions. Best value in a row is bold, and where heuristic finds the best known (objective value) solution, the gap is underlined. (Note that due to rounding, a gap value of 0.00 is not automatically the same as having found the best known.)

Instance	VNS	Op. Prob.	MCHH	CMCS[2-row reduced]	CMCS[2-row]
Rand 200×1000	0.01	**0.00**	**0.00**	**0.00**	**0.00**
Rand 400×1000	0.05	**0.00**	**0.00**	**0.00**	**0.00**
Rand 600×1000	**0.00**	**0.00**	0.00	**0.00**	**0.00**
Rand 800×1000	0.08	**0.00**	0.01	0.00	0.00
Rand 1000×1000	0.07	0.03	0.20	**0.00**	0.04
Biclique 200×1000	0.88	**0.00**	**0.00**	**0.00**	**0.00**
Biclique 400×1000	**0.00**	**0.00**	0.14	0.09	0.09
Biclique 600×1000	**0.09**	0.54	0.95	0.55	1.48
Biclique 800×1000	**0.00**	0.53	0.34	0.24	0.56
Biclique 1000×1000	**0.00**	0.14	0.13	0.16	0.35
MaxInduced 200×1000	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
MaxInduced 400×1000	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
MaxInduced 600×1000	0.18	**0.00**	**0.00**	**0.00**	**0.00**
MaxInduced 800×1000	0.30	0.08	0.08	0.09	**0.00**
MaxInduced 1000×1000	0.16	0.04	0.04	0.04	**0.03**
BMaxCut 200×1000	1.76	0.14	0.09	0.43	**0.06**
BMaxCut 400×1000	2.25	0.67	1.25	0.89	**0.40**
BMaxCut 600×1000	2.46	1.18	3.19	1.16	**0.53**
BMaxCut 800×1000	4.35	2.19	2.75	1.49	**1.05**
BMaxCut 1000×1000	4.51	2.65	2.39	**0.39**	0.46
MatrixFactor 200×1000	**0.00**	0.27	0.05	0.03	**0.00**
MatrixFactor 400×1000	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
MatrixFactor 600×1000	**0.00**	**0.00**	0.12	**0.00**	**0.00**
MatrixFactor 800×1000	0.43	0.01	**0.00**	**0.00**	**0.00**
MatrixFactor 1000×1000	0.09	**0.00**	0.10	**0.00**	0.03
Average	0.71	0.34	0.47	0.22	**0.20**
Max	4.51	2.65	3.19	1.49	**1.48**

Table 4
Evaluation of metaheuristics on Large Instances, 100 seconds per run. The format of the table is identical to that of Table 3.

Instance	VNS	Op. Prob.	MCHH	CMCS[2-row reduced]	CMCS[2-row]
Rand 1000×5000	0.07	**0.00**	0.08	0.04	0.04
Rand 2000×5000	0.38	0.17	0.15	0.13	**0.07**
Rand 3000×5000	0.50	0.19	0.22	0.24	**0.12**
Rand 4000×5000	0.29	0.13	0.19	0.08	**0.07**
Rand 5000×5000	0.38	0.31	0.31	0.23	**0.11**
Biclique 1000×5000	0.92	**0.06**	0.23	0.22	0.08
Biclique 2000×5000	**0.05**	0.37	0.53	0.57	0.52
Biclique 3000×5000	**0.00**	0.11	0.13	0.07	0.43
Biclique 4000×5000	**0.00**	**0.00**	0.26	0.27	0.38
Biclique 5000×5000	**0.00**	0.16	**0.00**	0.03	**0.00**
MaxInduced 1000×5000	0.21	0.01	0.01	0.05	**0.01**
MaxInduced 2000×5000	0.36	0.08	0.19	0.01	**0.01**
MaxInduced 3000×5000	0.53	0.11	0.21	0.20	**0.08**
MaxInduced 4000×5000	0.52	0.30	0.28	**0.14**	0.20
MaxInduced 5000×5000	0.52	0.32	0.42	0.23	**0.14**
BMaxCut 1000×5000	2.57	**0.71**	1.39	2.90	2.69
BMaxCut 2000×5000	5.61	**2.63**	3.41	3.99	3.75
BMaxCut 3000×5000	6.00	2.86	4.11	3.35	**2.69**
BMaxCut 4000×5000	6.09	4.33	4.07	3.41	**3.34**
BMaxCut 5000×5000	5.28	3.76	4.34	2.65	**2.49**
MatrixFactor 1000×5000	0.09	0.35	0.10	**0.04**	0.07
MatrixFactor 2000×5000	0.41	0.12	**0.11**	0.13	0.16
MatrixFactor 3000×5000	0.55	0.17	0.43	0.24	**0.16**
MatrixFactor 4000×5000	0.45	0.34	0.43	0.28	**0.13**
MatrixFactor 5000×5000	0.42	0.38	0.40	0.38	**0.16**
Average	1.29	0.72	0.88	0.80	**0.72**
Max	6.09	4.33	4.34	3.99	**3.75**

6. Evaluation of metaheuristics

So far we have only been testing the performance of the metaheuristics on the training instance set. In Tables 3 and 4 we report their performance on benchmark instances, giving 10 seconds per Medium instance and 100 seconds per Large instance. For each instance and metaheuristic, we report the percentage gap, between the solution obtained by that metaheuristic and the best known objective value for that instance. The best known objective values are obtained by recording the best solutions produced in all our experiments, not necessarily only the experiments reported in this paper. The best known solutions will be available for download, and their objective values are reported in Tables 5 and 6.

The results of the experiments on benchmark instances generally positively correlate with the configuration objective function (3) reported in Table 2, except that Op. Prob. shows performance better than MCHH, and is competing with CMCS[2-row reduced] on Large instances. This shows a common problem that the

Table 5
Empirical comparison of the CMCS[2-row] and Tabu Search (Glover et al., 2015) (which performs on average similarly to the method of Duarte et al., 2014) on the Medium instances. Reported are the gaps to the best known solution, in per cent. As in Tables 3 and 4, where the heuristic finds the best known (objective value) solution, the value (0.00) is underlined. Where CMCS[2-row] finds a solution at least as good as the one found by Tabu Search, the gap is shown in bold. Similarly, where Tabu Search (1000 seconds) finds a solution at least as good as the one found by CMCS[2-row] (1000 seconds), the gap is shown in bold.

		CMCS[2-row]				Tabu Search
Instance	Best known	1 seconds	10 seconds	100 seconds	1000 seconds	1000 seconds
Rand 200×1000	612,947	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Rand 400×1000	951,950	0.05	**0.00**	**0.00**	**0.00**	**0.00**
Rand 600×1000	1,345,748	**0.00**	**0.00**	**0.00**	**0.00**	0.00
Rand 800×1000	1,604,925	0.09	**0.00**	**0.00**	**0.00**	0.01
Rand 1000×1000	1,830,236	**0.04**	**0.04**	**0.02**	**0.00**	0.07
Biclique 200×1000	2,150,201	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Biclique 400×1000	4,051,884	0.27	0.09	**0.00**	**0.00**	**0.00**
Biclique 600×1000	5,501,111	**0.59**	1.48	**0.47**	**0.47**	0.65
Biclique 800×1000	6,703,926	**0.68**	**0.56**	**0.04**	**0.04**	0.79
Biclique 1000×1000	8,680,142	**0.10**	**0.35**	**0.35**	**0.11**	0.91
MaxInduced 200×1000	513,081	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
MaxInduced 400×1000	777,028	0.01	**0.00**	**0.00**	**0.00**	**0.00**
MaxInduced 600×1000	973,711	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
MaxInduced 800×1000	1,205,533	**0.01**	**0.00**	**0.00**	**0.00**	0.07
MaxInduced 1000×1000	1,415,622	**0.03**	**0.03**	**0.03**	**0.01**	0.06
BMaxCut 200×1000	617,700	1.59	**0.06**	**0.00**	**0.00**	0.14
BMaxCut 400×1000	951,726	1.34	**0.40**	**0.00**	**0.00**	1.13
BMaxCut 600×1000	1,239,982	**1.83**	**0.53**	**0.53**	**0.37**	2.00
BMaxCut 800×1000	1,545,820	1.74	**1.05**	**0.08**	**0.08**	1.66
BMaxCut 1000×1000	1,816,688	**1.83**	**0.46**	**0.23**	**0.23**	2.47
MatrixFactor 200×1000	6283	0.18	**0.00**	**0.00**	**0.00**	**0.00**
MatrixFactor 400×1000	9862	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
MatrixFactor 600×1000	12,902	0.05	**0.00**	**0.00**	**0.00**	0.03
MatrixFactor 800×1000	15,466	0.49	**0.00**	**0.00**	**0.00**	0.19
MatrixFactor 1000×1000	18,813	**0.08**	**0.03**	**0.00**	**0.00**	0.11
Average		0.44	**0.20**	**0.07**	**0.05**	0.41
Max		**1.83**	**1.48**	**0.53**	**0.47**	2.47

evaluation by short runs on small instances, as used for training, may not always perfectly correlate with the performance of the heuristic on real (or benchmark) instances Hutter et al. (2007). However, in our case, the main conclusions are unaffected by this. In particular, we still observe that CMCS[2-row] outperforms other metaheuristics, including CMCS[2-row reduced], hence proving usefulness of the Repair component. Also CMCS[2-row] clearly outperforms MCHH demonstrating that even a restricted version of the CMCS schema is more robust than the MCHH schema; recall that CMCS is an extension of MCHH with conditional transitions.

We made the source code of CMCS[2-row] publicly available.[4] The code is in C# and was tested on Windows and Linux machines. We note here that CMCS is relevant to the Programming by Optimisation (PbO) concept Hoos (2012). We made sure that our code complies with the 'PbO Level 3' standard, i.e. 'the software-development process is structured and carried out in a way that seeks to provide design choices and alternatives in many performance-relevant components of a project'. Hoos (2012). Our code is not compliant with 'PbO Level 4' because some of the choices made (specifically, the internal parameters of individual components) were not designed to be tuned along with the CMCS matrices; for details of PbO see Hoos (2012).

6.1. Comparison to the state-of-the-art

There have been two published high-performance metaheuristics for BBQP: *Iterated Local Search* by Duarte et al. (2014) and *Tabu Search* by Glover et al. (2015). Both papers agree that their approaches perform similarly; in fact, following a sign test, Duarte et al. conclude that 'there are not significant differences between both procedures'. At first, we compare CMCS[2-row] to Tabu Search for which we have detailed experimental results Glover et al. (2015). Then we also compare CMCS[2-row] to ILS using approach adopted in Duarte et al. (2014).

Tabu Search has two phases: (i) a classic tabu search based on a relatively small neighbourhood, which runs until it fails to improve the solution, and (ii) a polishing procedure, similar to ours, which repeats a sequence of hill climbing operators OptY, FlpX, OptX and FlpY until a local maximum is reached.[5] The whole procedure is repeated as many times as the time allows.

The experiments in Glover et al. (2015) were conducted on the same benchmark instances, first introduced in Karapetyan and Punnen (2012) and now described in Section 4 of this paper. Each run of Tabu Search was given 1000 seconds for Medium instances ($n = 1000$) and 10,000 seconds for Large instances ($n = 5000$). In Table 5 we report the performance results of CMCS[2-row], our best performing metaheuristic, on Medium instances with 1, 10, 100 and 1000 seconds time limits, and in Table 6 on Large instances with 10, 100, 1000 and 10,000 seconds time limits, and explicitly compare those results to the performance of Tabu Search and so implicitly compare to the results of Duarte et al. Duarte et al. (2014) that were not significantly different from Tabu.

Given the same time, CMCS[2-row] produces same (for 10 instances) or better (for 20 instances) solutions. The worst gap between best known and obtained solution (reported in the Max row at the bottom of each table) is also much larger for Tabu Search than for CMCS[2-row]. CMCS[2-row] clearly outperforms Tabu Search even if given a factor of 100 less time, and competes with it even if given a factor of 1000 less time. Thus we conclude that CMCS[2-row] is faster than Tabu Search by two to three orders of magnitude. Further, we observe that CMCS[2-row] does not converge prematurely, that is, it continues to improve the solution when given more time.

[4] http://csee.essex.ac.uk/staff/dkarap/?page=publications&key=CMCS-BBQP .

[5] In Glover et al. (2015), a composite of OptY and FlpX is called Flip-*x*-Float-*y*, and a composite of OptX and FlpY is called Flip-*y*-Float-*x*.

Table 6
Empirical comparison of CMCS[2-row] with Tabu Search (Glover et al., 2015) (which performs on average similarly to the method of Duarte et al., 2014) on the Large instances. The format of the table is identical to that of Table 5.

		CMCS[2-row]				Tabu Search
Instance	Best known	10 seconds	100 seconds	1000 seconds	10,000 seconds	10,000 seconds
Rand 1000×5000	7,183,221	0.04	0.04	0.01	0.01	**0.01**
Rand 2000×5000	11,098,093	0.18	**0.07**	**0.07**	**0.02**	0.09
Rand 3000×5000	14,435,941	**0.16**	**0.12**	**0.11**	**0.07**	0.22
Rand 4000×5000	18,069,396	**0.14**	**0.07**	**0.01**	**0.01**	0.19
Rand 5000×5000	20,999,474	0.26	**0.11**	**0.08**	**0.07**	0.25
Biclique 1000×5000	38,495,688	0.22	0.08	**0.02**	**0.00**	0.02
Biclique 2000×5000	64,731,072	1.67	**0.52**	**0.19**	**0.28**	0.94
Biclique 3000×5000	98,204,538	1.68	**0.43**	**0.01**	**0.04**	1.50
Biclique 4000×5000	128,500,727	**0.38**	**0.38**	**0.22**	**0.00**	2.19
Biclique 5000×5000	163,628,686	**0.38**	**0.00**	**0.00**	**0.00**	1.01
MaxInduced 1000×5000	5,465,051	**0.01**	**0.01**	**0.00**	**0.00**	0.02
MaxInduced 2000×5000	8,266,136	**0.10**	**0.01**	**0.01**	**0.00**	0.12
MaxInduced 3000×5000	11,090,573	**0.15**	**0.08**	**0.04**	**0.03**	0.18
MaxInduced 4000×5000	13,496,469	**0.29**	**0.20**	**0.06**	**0.05**	0.36
MaxInduced 5000×5000	16,021,337	**0.19**	**0.14**	**0.08**	**0.08**	0.29
BMaxCut 1000×5000	6,644,232	2.98	2.69	2.17	**1.20**	1.70
BMaxCut 2000×5000	10,352,878	5.39	3.75	3.39	**1.80**	2.58
BMaxCut 3000×5000	13,988,920	3.49	**2.69**	**1.99**	**1.81**	3.45
BMaxCut 4000×5000	17,090,794	4.36	**3.34**	**3.31**	**2.31**	4.28
BMaxCut 5000×5000	20,134,370	**3.15**	**2.49**	**2.34**	**1.79**	3.90
MatrixFactor 1000×5000	71,485	0.11	0.07	**0.00**	**0.00**	0.02
MatrixFactor 2000×5000	108,039	0.19	0.16	**0.06**	**0.04**	0.09
MatrixFactor 3000×5000	144,255	**0.17**	**0.16**	**0.14**	**0.11**	0.26
MatrixFactor 4000×5000	179,493	**0.26**	**0.13**	**0.10**	**0.10**	0.29
MatrixFactor 5000×5000	211,088	**0.21**	**0.16**	**0.13**	**0.04**	0.33
Average		1.05	**0.72**	**0.58**	**0.39**	0.97
Max		5.39	**3.75**	**3.39**	**2.31**	4.28

As pointed out above, it is known from the literature that ILS (Duarte et al., 2014) performs similarly to Tabu Search, and hence the conclusions of the comparison between CMCS[2-row] and Tabu Search can be extended to ILS as well. However, to verify this, we reproduced the experiment from Duarte et al. (2014). In that experiment, Duarte et al. solved each of the medium and large instances, giving ILS 1000 seconds per run, and then reported the average objective value. We tested CMCS[2-row] is exactly the same way, except that we allowed only 10 seconds per run. Despite a much lower time budget, our result of 14523968.32 is superior to the result of 14455832.30 reported in (Duarte et al., 2014, Table 8). This direct experiment confirms that CMCS[2-row] significantly outperforms ILS.

We note here that this result is achieved in spite of CMCS[2-row] consisting of simple components combined in an entirely automated way; without any human intelligence put into the detailed metaheuristic design. Instead, only a modest computational power (a few hours of CPU time) was required to obtain it. (Note that this computational power should not be compared to the running time of the algorithm itself; it is a replacement of expensive time of a human expert working on manual design of a high-performance solution method.) We believe that these results strongly support the idea of automated metaheuristic in general and CMCS schema in particular.

7. Conclusions

In this work, we considered an important combinatorial optimisation problem called Bipartite Boolean Quadratic Programming Problem (BBQP). We defined several algorithmic components for BBQP, primarily aiming at components for metaheuristics. To test and analyse the performance of the components, and to combine them in a powerful metaheuristic, we designed a flexible metaheuristic schema, which we call Conditional Markov Chain Search (CMCS), the behaviour of which is entirely defined by an explicit set of parameters and thus which is convenient for automated configuration. CMCS is a powerful schema with special cases covering several standard metaheuristics. Hence, to evaluate the performance of a metaheuristic on a specific problem class, we can configure the CMCS restricted to that metaheuristic, obtaining a nearly best possible metaheuristic of that particular type for that specific problem class. The key advantages of this approach include avoidance of human/expert bias in analysis of the components and metaheuristics, and complete automation of the typically time-consuming process of metaheuristic design.

Of the methods we consider, the CMCS schema is potentially the most powerful as it includes the others as special cases, however, it has a lot of parameters, and this complicates the selection of the matrices. To combat this, we proposed a special case of CMCS, CMCS[k-row], which is significantly easier to configure, but that still preserves much of the flexibility of the approach.

By configuring several special cases of CMCS on a set of small instances and then testing them on benchmark instances, we learnt several lessons. In particular, we found out that CMCS schema, even if restricted to the CMCS[2-row] schema, is significantly more powerful than VNS, Op. Prob. and even MCHH (with a static transition matrix). We also verified that the new BBQP component, Repair, is useful, as its inclusion in the pool of components improved the performance of CMCS[2-row]. Finally, we showed that the best found strategies are often much more sophisticated than the strategies implemented in standard approaches.

Our best performing metaheuristic, CMCS[2-row], clearly outperforms the previous state-of-the-art BBQP methods. Following a series of computational experiments, we estimated that CMCS[2-row] is faster than those methods by roughly two to three orders of magnitude.

7.1. Future work

A few other BBQP algorithmic components could be studied and exploited using the CMCS schema. Variations of the Repair heuristic, as discussed in Section 2.3, should be considered more

thoroughly. Another possibility for creating a new class of powerful components is to reduce the entire problem by adding constraints of the form $x_i = x_{i'}$, $x_i \neq x_{i'}$ or $x_i = 1$, or even more sophisticated such as $x_i = x_{i'} \vee x_{i''}$. Note that such constraints effectively reduce the original problem to a smaller BBQP; then this smaller BBQP can be solved exactly or heuristically. Also note that if such constraints are generated to be consistent with the current solution then this approach can be used as a hill climbing operator.

It is interesting to note that the reduced size subproblem could itself be solved using a version of CMCS configured to be effective for brief intense runs. This gives the intriguing possibility of an upper-level CMCS in which one of the components uses a different CMCS – though we expect that tuning such a system could be a significant, but interesting, challenge.

The CMCS schema should be developed in several directions. First of all, it should be tested on other domains. Then a few extensions can be studied, e.g. one could add a 'termination' component that would stop the search – to allow variable running times. It is possible to add some form of memory and/or backtracking functionality, for example to implement a tabu-like mechanism. Another direction of research is population-based extensions of CMCS. Of interest are efficient configuration procedures that would allow to include more components. Finally, of course, one can study methods for online learning, that is adaptation of the transition probabilities during the search process itself; in which case it would be most natural to call the method 'Conditional Markov Chain Hyper-heuristic'.

Acknowledgement

This research work was partially supported by an NSERC Discovery accelerator supplement awarded to Abraham P. Punnen, EPSRC Grants EP/H000968/1 and EP/F033214/1 ('The LANCS Initiative'), and also LANCS Initiative International Scientific Outreach Fund that supported the visit of Daniel Karapetyan to the Simon Fraser University.

References

Adenso-Díaz, B., & Laguna, M. (2006). Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research, 54*(1), 99–114.

Ahuja, R. K., Ergun, Ö., Orlin, J. B., & Punnen, A. P. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics, 123*, 75–102.

Alon, N., & Naor, A. (2006). Approximating the cut-norm via Grothendieck's inequality. *SIAM Journal on Computing, 35*(4), 787–803.

Ambühl, C., Mastrolilli, M., & Svensson, O. (2011). Inapproximability results for maximum edge biclique, minimum linear arrangement, and sparsest cut. *SIAM Journal on Computing, 40*(2), 567–596.

Bezerra, L. C. T., López-Ibáñez, M., & Stützle, T. (2015). Comparing decomposition-based and automatically component-wise designed multi-objective evolutionary algorithms. In *Evolutionary multi-criterion optimization: 8th international conference, EMO 2015, Guimarães, Portugal, March 29–April 1, 2015. Proceedings, part I* (pp. 396–410)). Cham: Springer International Publishing. doi:10.1007/978-3-319-15934-8_27.

Billionnet, A. (2004). Quadratic 0-1 bibliography. URL: http://cedric.cnam.fr/fichiers/RC611.pdf.

Chang, W.-C., Vakati, S., Krause, R., & Eulenstein, O. (2012). Exploring biological interaction networks with tailored weighted quasi-bicliques. *BMC Bioinformatics, 13*(Suppl. 1), S16.

Cowling, P., Kendall, G., & Soubeiga, E. (2001). A hyperheuristic approach to scheduling a sales summit. In E. Burke, & W. Erben (Eds.), *Selected papers from the 3rd international conference on the practice and theory of automated timetabling (PATAT 2001)*. In *Lecture notes in computer science: Vol. 2079* (pp. 176–190). Springer. doi:10.1007/3-540-44629-X_11.

Duarte, A., Laguna, M., Martí, R., & Sánchez-Oro, J. (2014). Optimization procedures for the bipartite unconstrained 0-1 quadratic programming problem. *Computers & Operations Research, 51*, 123–129.

Gillis, N., & Glineur, F. (2011). Low-rank matrix approximation with weights or missing data is NP-hard. *SIAM Journal on Matrix Analysis and Applications, 32*(4), 1149–1165.

Glover, F., Ye, T., Punnen, A., & Kochenberger, G. (2015). Integrating tabu search and VLSN search to develop enhanced algorithms: A case study using bipartite Boolean quadratic programs. *European Journal of Operational Research, 241*, 697–707.

Hoos, H. H. (2012). Programming by optimization. *Communications of the ACM, 55*(2), 70–80.

Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Research, 36*(1), 267–306. doi:10.1613/jair.2861.

Hutter, F., Hoos, H. H., & Stützle, T. (2007). Automatic algorithm configuration based on local search. In *Proceedings of the 22nd national conference on artificial intelligence, AAAI'07: Vol. 2* (pp. 1152–1157). AAAI Press.

Karapetyan, D., & Punnen, A. P. (2012). Heuristic algorithms for the bipartite unconstrained 0-1 quadratic programming problem. URL: http://arxiv.org/abs/1210.3684.

Koyutürk, M., Grama, A., & Ramakrishnan, N. (2005). Compression, clustering, and pattern discovery in very high-dimensional discrete-attribute data sets. *IEEE Transactions on Knowledge and Data Engineering, 17*(4), 447–461.

Koyutürk, M., Grama, A., & Ramakrishnan, N. (2006). Nonorthogonal decomposition of binary matrices for bounded-error data compression and analysis. *ACM Transactions on Mathematical Software, 32*(1), 33–69.

Lourenço, H. R., Martin, O. C., & Stützle, T. (2010). Iterated local search: Framework and applications handbook of metaheuristics. In *International series in operations research & management science: Vol. 146* (pp. 363–397). Boston, MA: Springer US. doi:10.1007/978-1-4419-1665-5_12.

Lu, H., Vaidya, J., Atluri, V., Shin, H., & Jiang, L. (2011). Weighted rank-one binary matrix factorization. In *Proceedings of the eleventh SIAM international conference on data mining* (pp. 283–294). SIAM/Omnipress.

McClymont, K., & Keedwell, E. C. (2011). Markov chain hyper-heuristic (MCHH): An online selective hyper-heuristic for multi-objective continuous problems. In *Proceedings of the 13th annual conference on genetic and evolutionary computation* (pp. 2003–2010). New York, NY, USA: ACM. doi:10.1145/2001576.2001845.

Papadimitriou, C. H. (1991). On selecting a satisfying truth assignment. In *Proceedings of the 32nd annual symposium on Foundations of computer science* (pp. 163–169). doi:10.1109/SFCS.1991.185365.

Punnen, A. P., Sripratak, P., & Karapetyan, D. (2015a). Average value of solutions for the bipartite Boolean quadratic programs and rounding algorithms. *Theoretical Computer Science, 565*, 77–89.

Punnen, A. P., Sripratak, P., & Karapetyan, D. (2015b). The bipartite unconstrained 0-1 quadratic programming problem: Polynomially solvable cases. *Discrete Applied Mathematics, 193*, 1–10.

Selman, B., Kautz, H., & Cohen, B. (1996). Local search strategies for satisfiability testing. In *DIMACS series in discrete mathematics and theoretical computer science: Vol. 26* (pp. 521–532). American Mathematical Society.

Shen, B.-h., Ji, S., & Ye, J. (2009). Mining discrete patterns via binary matrix factorization. In *Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 757–766). ACM Press.

Tan, J. (2008). Inapproximability of maximum weighted edge biclique and its applications. In *Proceedings of the 5th international conference on theory and applications of models of computation* (pp. 282–293). Springer-Verlag.

Tanay, A., Sharan, R., & Shamir, R. (2002). Discovering statistically significant biclusters in gene expression data. *Bioinformatics, 18*(Suppl. 1), S136–S144.

Zweben, M., Davis, E., Daun, B., & Deale, M. J. (1993). Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man, and Cybernetics, 23*(6), 1588–1596. doi:10.1109/21.257756.