10

Other Combinatorial Problems

The problems covered in the previous chapters are only some of many combinatorial problems to which stochastic local search algorithms can be applied successfully. In this chapter, we present and discuss SLS applications to a selection of other combinatorial problems, which have been selected partly because of their fundamental nature, partly because of their relevance for certain application areas. In each of the main sections, we will introduce one combinatorial problem, discuss its applications and commonly used benchmark instances, and present one or more SLS approaches for solving this problem. The problems we cover are: Graph Colouring, Quadratic Assignment, Set Covering, Combinatorial Auctions Winner Determination, and DNA Code Design. All of these problems have interesting and important applications in various application areas. While the first three of these problems have been extensively studied in the literature for many years, the latter two have only relatively recently gained their current prominence. The algorithms presented in this chapter are primarily intended to illustrate the application of SLS methods to the respective problems; pointers to other SLS algorithms and more detailed information on the problems covered here are provided in the "Further Readings and Related Work" section of this chapter.

10.1 Graph Colouring

The Graph Colouring Problem (GCP) plays a central role in graph theory and is at the core of many application relevant problems such as timetabling [Leighton, 1979; de Werra, 1985; Schaerf, 1999] and frequency assignment [Gamst, 1986]. It can be defined as follows: A k-colouring of an undirected graph $G = (V, E)$, where V is the set of $|V| = n$ vertices and $E \subseteq V \times V$ is the set of edges, is a mapping $\Psi : V \mapsto \{1, 2, \ldots, k\}$ that assigns a positive integer from $\{1, 2, \ldots, k\}$ (representing the colours) to each vertex such that the endpoints of every edge in E have assigned a different colour, *i.e.*, $\forall (u, v) \in E : \Psi(u) \neq \Psi(v)$. The decision version of the Graph Colouring Problem asks whether for a given graph G a k-colouring can be found. In the optimisation version, the objective is to find the minimum number k such that a k-colouring exists; this minimum number k is also known as the chromatic number χ_G of G.

Example 10.1: Simple GCP instance

In Figure 10.1 we give a simple GCP instance with six vertices $v_1, \ldots, v_6$; each vertex is assigned one of the three colours $\{1, 2, 3\}$ that are here represented by the colours red, blue and green. In this example we have $\Psi(v_1) = \Psi(v_6) =$ red, $\Psi(v_2) = \Psi(v_4) =$ blue, $\Psi(v_3) = \Psi(v_5) =$ green. χ_G for this graph is three. In fact, the left and the right triple of vertices form a clique (a clique of a graph is a fully connected sub-graph) and it is clear that the number of colours required is at least as large as the number of vertices in a maximum clique, *i.e.*, the size of the maximum clique gives a lower bound for the chromatic number.

Alternatively to this formulation as an assignment problem, the GCP can also be represented as a partitioning problem, in which a k-colouring corresponds to a partition of the set of vertices into k sets $V_1, \ldots, V_k$, such that for no edge (u, v) in E the incident vertices u and v belong to the same set V_i. This equivalent definition of the GCP has the advantage that it avoids symmetric solutions, which are obtained in the assignment formulation, because any permutation of the colours yields an isomorphic (*i.e.*, equivalent) solution, since the numbering of the colours is not essential. For

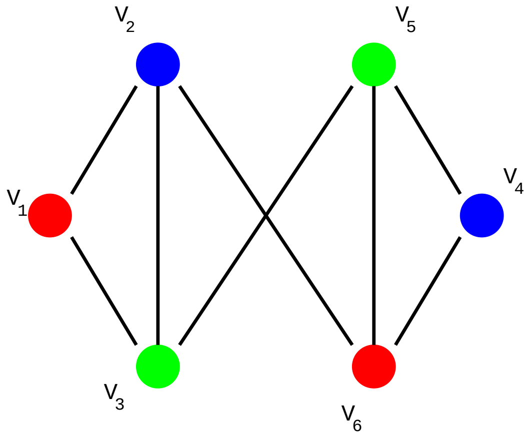

Figure 10.1: Simple GCP instance with six vertices; assignments are represented by colours.

example, the (unique) partition representation of Example 10.1 is given by $V_1 = \{v_1, v_6\}, V_2 = \{v_2, v_4\}, V_3 = \{v_3, v_5\}$, while there exist six equivalent solutions in the assignment case, which are obtained by changing the colours assigned to vertices.

The decision version of the GCP is an $\mathcal{NP}$-complete problem that can be seen as a special case of the Constrained Satisfaction Problem (CSP), as discussed in Chapter 6.5, Section 6.5. In the CSP formulation, all constraints are binary inequality constraints; this special structure can be exploited to derive algorithms that are more efficient that standard CSP solvers. Considering this as well as the central role of the GCP in graph theory and its numerous applications, Graph Colouring Problems are typically treated independently of general CSPs.

It is well known that the optimisation version of the GCP is $\mathcal{NP}$-hard [Garey and Johnson, 1979]. Furthermore, the GCP is not approximable within a fixed ratio of the chromatic number, since it is known that achieving a k-colouring within a ratio of $|V|^{1/7-\epsilon}$, for any $\epsilon > 0$, of the optimum solution is $\mathcal{NP}$-hard [Bellare *et al.*, 1998]. Therefore, in some sense, GCP is among the hardest combinatorial optimisation problems. In fact, the best ploynomial-time approximation algorithm is only guaranteed to achieve an approximation ratio of $\mathcal{O}(|V|\frac{(\log\log|V|)^2}{(\log|V|)^3})$ [Halldórsson, 1993].

Applications and Benchmark Instances

The GCP arises in many application relevant problems. Probably the most intuitive application of the GCP is that of colouring maps, where each region or country corresponds to a vertex of the graph and a connection between two regions exists if they have a common border. (For an example of a map-colouring problem recall Figure 6.6 on page 241). Other applications of the GCP include the determination of lower bounds on the number of time slots in timetabling problems [Leighton, 1979; de Werra, 1985; Carter, 1986], special cases of frequency assignment problems [Gamst, 1986], the register allocation problem, in which variables are to be assigned to a limited number of registers in a CPU [Briggs *et al.*, 1994; Chaitin *et al.*, 1981; Chaitin, 1982; Chow and Hennessy, 1990], the estimation of sparse Jacobian matrices [Coleman and Moré, 1983; Hossain and Steihaug, 2002], or the testing for unintended short circuits on printed circuit boards [Garey *et al.*, 1976]. Recently, the Quasigroup Completion Problem (QCP), which is a special case of the GCP, has received significant attention. Given an $n \times n$ quadratic grid and n colours, the objective it to assign a colour to each grid cell in such a way that every row and column contains all n colours. Often, in the QCP also some vertices are pre-coloured and their colour cannot be modified by the algorithm.

Random GCP instances are often used as a benchmark for graph colouring algorithms and are generated according to a number of different models. One common class are $G_{n,p}$ graphs, where n is the number of vertices, and each among the $n(n-1)/2$ edges is included with probability p into the graph. These graphs have been a popular class of benchmark instances for GCP algorithms. Another class of widely used GCP benchmark instances is based on $U_{n,d}$ graphs; these are generated by first placing n vertices at random positions in a two dimensional unit square, where the x and the y coordinate are chosen according to a uniform distribution in $[0, 1]$ and including an edge between vertices u and v, if the Euclidean distance between the two vertices is smaller than some value d. While in $G_{n,p}$ instances any pair of vertices can have an edge independent of other pairs of vertices, this is not the case for the geometric random $U_{n,d}$ graphs. Empirical results suggest that the latter class of graphs is easier for exact algorithms than the $G_{n,p}$ graphs [Mehrotra and Trick, 1996].

Other widely used classes of random graphs are generated with a known predetermined chromatic number. One such class are Leighton graphs that are constructed by introducing cliques of size between two and the chromatic number.

A common way of generating guaranteed k-colourable graphs is to partition the vertices in a graph into k sets and then to introduce edges only between vertices that are not in a same parition; often edges are included with a probability of p, independent of other edges. These graphs are generally referred to as $G_{n,p,k}$ graphs. However, there are a large number of ways of generating such instances, *e.g.* depending on whether the partitions are generated as nearly as possible of a same size or whether restrictions on the vertex degrees are introduced, a large number of different ways of generating k-colourable graphs can be defined; we refer to [Culberson *et al.*, 1995] for more details. One popular class of such $G_{n,p,k}$ graphs are the flat graphs, where an additional constraint on the degree of vertices is used to generated instances with a small variation of vertex degrees; this restriction was found to be useful when the goal is to generate hard GCP instances [Culberson and Luo, 1996].

Currently, the largest collection of GCP instances that stem in part from GCP applications is available via the web-page of COLOR02/03 at http://mat.gsia.cmu.edu/COLOR02/. These include instances derived from register allocation problems, course scheduling problems, quasi-group completion problems (also known as Latin squares, with or without pre-coloured squares), several other combinatorial problems, as well as randomly generated graphs. Additionally, several instances for extensions of the basic GCP model are included.

Simulated Annealing for Graph Colouring

Among the first SLS algorithms applied to GCP is Simulated Annealing (SA) [Chams *et al.*, 1987; Johnson *et al.*, 1991]. A systematic experimental analysis of SA algorithms was presented by Johnson, Aragon, McGeoch, and Schevon [Johnson *et al.*, 1991], who compared three different variants of SA for the GCP. All the three variants are based on a standard SA algorithm that uses a geometric cooling schedule and that performs at each temperature a number of moves that is proportional to the neighbourhood size (see Example **??** on page **??** for a very similar SA application to the TSP). The search is terminated, if the ratio of accepted moves over a number of successive temperatures is below a given threshold.

The first two variants exploit the partition-based formulation of GCP. In the *Penalty Function SA Algorithm*, each candidate solution is a partition of V into k non-empty colour classes $V_1, \ldots, V_k$, which need not correspond to a (feasible) k-colouring. The initial solution is a random partitioning of the vertices into a given number of colour classes. A neighbouring solution is generated by randomly picking first a non-empty colour class V_i, then a vertex $v \in V_i$, and finally a new colour class $V_j \in \{V_1, \ldots, V_{k+1}\}$; all random selections are performed according to a uniform distribution over the respective sets. (If a previously empty colour class $k+1$ is chosen, v becomes the only element of this class.) Penalty Function SA uses a particular evaluation function that on the one side favours large colour classes and on the other side discourages edge constraint violations, *i.e.*, edges whose two endpoints are in a same colour class; this evaluation function is defined as

$$g(s) = -\sum_{i=1}^{k} |V_i|^2 + \sum_{i=1}^{k} 2 \cdot |V_i| \cdot |E_i|, \tag{10.1}$$

where E_i is the set of all edges with both endpoints in V_i. It is easy to show that all local minima of this objective function correspond to legal colourings. Although this evaluation function function does not explicitely count k, the first term gives a bias towards minimising the number of colour classes, while the second will punish constraint violations, that are incurred when reducing the number of colours too much; hence, it is hoped that k is minimised as a "side-effect" of minimising this evaluation function [Johnson *et al.*, 1991]. Additionally, the first term of $g(s)$ introduces a bias towards unbalanced colourings rather than equal sized colour classes, due to the use of the term $|V_i|^2$.

In the *Kempe Chain SA Algorithm*, candidate solutions are restricted to legal colourings, *i.e.*, at each local search step no edge has its endpoints in a same colour class. In this case, the evaluation function is reduced to

$$g(s) = -\sum_{i=1}^{k} |V_i|^2.$$

An additional, major difference to the previously described Penalty function SA Algorithm is the use of a *Kempe chain neighbourhood*, which at each step modifies the colour of various vertices in a subgraph of G. A move in the Kempe chain neighbourhood is generated as follows.

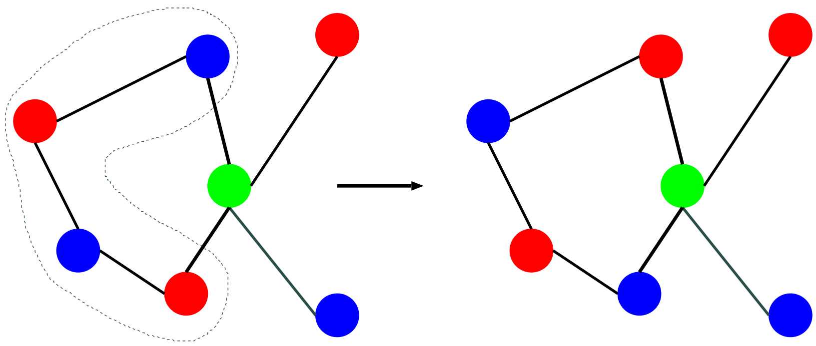

Figure 10.2: Illustration of the Kempe chain neighbourhood for GCP. The marked set of blue and red coloured vertices are connected by a path, forming a Kempe chain and exchange their colours in one local search step.

- Randomly choose a non-empty colour class V_i, a vertex $v \in V_i$, and a non-empty colour class V_j, such that one can build a Kempe Chain that contains vertex v;
- let KV_{ij} be the maximally connected subgraph that contains vertex v and only vertices of colour classes V_i and V_j; KC_{ij} is also called a *Kempe chain*;
- swap the colours i and j for all vertices in KC_{ij}.

Figure 10.2 illustrates a Kempe chain move. Note that full Kempe chains, *i.e.* chains comprising all vertices of V_i and V_j, are avoided, because they would lead to a pure renaming of vertices.

The initial solution in the Kempe Chain SA is generated by the sequential algorithm, a constructive heuristic that assigns colours to vertices in some given order and at each step a vertex is assigned to the colour class with the lowest possible index.

Finally, the *Fixed-k SA Algorithm* solves a series of GCP instances with a fixed number of colours, k, in a way analogous to how SLS algorithms are often applied to MAX-CSP (see Section 7.3 for an overview of SLS algorithms to MAX-CSP.). If for a specific k, a k-colouring is found, k is reduced by one and the algorithms searches for a $k-1$-colouring of the graph. In this approach the evaluation function only measures the number of edges with endpoints having the same colour, *i.e.*, the number of violated edge constraints. A move is generated by first selecting a vertex involved in a constraint violation uniformly at random and then assigning it to a uniformly at randomly chosen, different colour.

Experimental results with these three SA algorithms on a variety of GCP instances based on random $G_{n,p}$, $U_{n,d}$, and flat graphs have shown that neither of them completely dominates the performance of the others. However, it was observed that Fixed-k SA tended to achieve the best performance on sparse graphs and the Kempe Chain SA Algorithm was better for denser graphs. Interestingly, variants of truncated exhaustive search algorithms where also shown to perform extremely well on some of the tested instances.

A Hybrid Evolutionary Algorithm for Graph Colouring

Among the currently most successful SLS algorithms for GCP we find the Hybrid Evolutionary Algorithm (HEA). Like Fixed-k SA, HEA searches k-colourings of a graph for fixed k; the optimisation variant of GCP can be solved by repeatedly using this type of algorithm while iteratively decreasing k. Candidate solutions in HEA are evaluated according to the number of edge conflicts, *i.e.*, the number of edges whose endpoints are assigned the same colour.

The population in HEA is initialised by a greedy algorithm that constructs (infeasible) k-colourings based on a modification of the DSATUR algorithm [Brélaz, 1979]. This construction heuristic is initialised with k empty colour classes. In each construction step, a vertex is chosen that has the minimum number of allowed colour classes and the vertex is assigned to the colour with the lowest possible index. Typically, this heuristic cannot assign all vertices to colours such that a k-colouring results. Each unassigned vertex is then assigned to a random colour class. Once a candidate solution is completed, it is improved by a Tabu Search algorithm, the same that is also applied to solutions returned by a recombination and the mutation operator.

In each iteration of HEA, one single candidate solution (offspring) is generated through a recombination operator which combines two parents that are chosen randomly according to a uniform distribution; the resulting candidate solution is improved by an efficient Tabu Search algorithm and

procedure $HEA - GCP(\pi', k)$
  input *graph* $G = (V, E)$, *integer* k
  output *solution* $\hat{s} \in S(\pi')$ **or** $\emptyset$
  $sp := init(\pi')$;
  for $i := 1$ **to** $|sp|$ **do**
    $s_i := tabuSearch(\pi, s_i)$
  end
  $\hat{s} := best(\pi', sp)$
  while (**not** $terminate(\pi', sp)$) **do**
    $s_1, s_2 := selectParents(\pi', sp)$
    $s := GCXrecomb(\pi', s_1, s_2)$
    $s' := tabuSearch(\pi', s)$
    if $g(s') < g(\hat{s})$
      $\hat{s} = s'$;
    end
    $sp := updatePop(\pi, sp, s)$
  end
  return $\hat{s}$
end $HEA - GCP$

Figure 10.3: Algorithm outline of HEA for GCP; $best(\pi', sp)$ denotes the individual from a population sp with the best evaluation function value. (For details, see text.)

replaces the worse of the two parents. Different from typical Evolutionary algorithms, HEA does not make use of a mutation operator. An outline of the algorithm is given in Figure 10.3.

The main innovation by HEA is a particular recombination operator called Greedy Partition Crossover (GPX). This crossover exploits the partition representation of the GCP (and the candidate solutions) as a partitioning of V into sets $\{V_1, \ldots, V_k\}$, and tries to greedily transfer colour classes of maximal size alternatingly from the two parents to the offspring. GPX takes as its input two candidate solutions $s_1 = \{V_1^1, \ldots, V_k^1\}$ and $s_2 = \{V_1^2, \ldots, V_k^2\}$ and builds a new partition alternatingly selecting times colour (sub)classes of each parent. At step i, $i = 1, \ldots, k$, it chooses in parent s_1 (if i is odd) or in parent s_2 (if i is even) a colour class with the maximum number of vertices to become colour class V_i of the offspring; after each step the vertices in the set V_i are removed from both parents. The vertices that remaining unassigned are then added to a randomly chosen partition.

The intuition behind the partition-based recombination is that the important information to be transmitted to the offspring is which sets of vertices belong to a same colour class in the parents, while the particular colour assigned to a vertex is not important. Intuitively, this is obvious, because a k-colouring is unaffected by a permutation of the colours. The following example illustrates how GPX works.

Example 10.2: Greedy Partition Crossover

Let us assume we have three colour classes and ten vertices. Parent one has a partition

$$s_1 = \{\{1,2,3,4\},\{5,6,7\},\{8,9,10\}\}$$

and parent two a partition

$$s_2 = \{\{4,6,7,8\},\{1,2,10\},\{3,5,9\}\}.$$

At the first step of GPX, the partition $\{\{1,2,3,4\}$ of s_1 is copied to the offspring candidate solution s_o and the vertices of this partition are deleted. This results in candidate solutions

$$s_1' = \{\{5,6,7\},\{8,9,10\}\}, s_2' = \{\{6,7,8\},\{10\},\{5,9\}\}, s_o = \{\{1,2,3,4\}\}.$$

Next, the largest partition of s_2', $\{6,7,8\}$, is chosen and added to s_o and again the corresponding vertices are deleted in both parents, resulting in

$$s_1'' = \{\{5\},\{9,10\}\}, s_2'' = \{\{10\},\{5,9\}\}, s_o = \{\{1,2,3,4\},\{6,7,8\}\}.$$

The next step leads, after copying colour class $\{9,10\}$ to s_o to

$$s_1''' = \{\{5\}\}, s_2''' = \{\{5\}\}, s_o = \{\{1,2,3,4\},\{6,7,8\},\{9,10\}\}.$$

At this point, the only remaining vertex 5 is assigned to a random colour class in s_o.

The candidate solutions obtained from GPX are improved by a Tabu Search algorithm which uses a 1-exchange neighbourhood consisting of all those candidate solutions that are obtained by moving a single vertex from one colour class to a different one (*i.e.*, by assigning one vertex a different colour). If the colour class of vertex v changes from V_i to V_j, it is forbidden to assign colour i to vertex v in the next tl iterations except such a move would lead to an improvement over the best candidate solution seen up to this point (aspiration criterion). Analogously to the Tabu Search algorithms for the more general Constraint Satisfaction Problem (see Section 6.6), the neighbourhood is restricted to vertices that are involved in some conflict, *i.e.*, that have the same colour as some adjacent vertex. Then, at each iteration, a best possible non-tabu neigbhouring candidate solution is selected. It should be noted that this Tabu Search algorithm is, when run as a stand-alone algorithm on a time equalised basis, among the best available local search algorithms for the GCP.

HEA was tested on large $G_{n,p}$, Leighton, and flat graphs and it was compared to long runs of the same Tabu Search algorithm used within HEA. Empirical results indicate that HEA achieves excellent performance, especially on large $G_{n,p}$ graphs. For all instances (and number of colours) tested it was (with only one exception) either much faster in identifying the same k-colouring as the underlying Tabu Search algorithm or it found better k-colourings. Furthermore, for the more structured flat graphs, HEA was shown to perform very well. The excellent performance of HEA is confirmed by the fact that for four 1,000 vertex graphs, three randomly generated $G_{n,p}$ graphs and one flat graph, it was able to improve on the best previously known colourings. In general, HEA appears to be currently one of the best performing SLS algorithms for the GCP.

10.2 The Quadratic Assignment Problem

The QAP has been the subject of an enormous amount of research efforts and, besides the Travelling Salesman Problem, it is one of the most studied combinatorial optimisation problems [Çela, 1998]. It is of particular interest in the context of SLS algorithms, which outperform all other types of QAP algorithms by a very large margin. The QAP can best be described as the problem of assigning a set of objects to a set of locations with given distances between the locations and given flows between the objects, where the may, *e.g.*, correspond to the amount of material to be exchanged among machines in a production environment. The goal then is to place the objects on locations in such a way that the sum of the product between flows and distances is minimal.

Formally, a QAP instance is specified by n objects and n locations, where both the objects and locations are represented by integers from the set $I = \{1, \ldots, n\}$, and two $n \times n$ matrices $A = [a_{ij}]$ and $B = [b_{rs}]$, where a_{ij} is called the *distance* between locations i and j and b_{rs} is called the *flow* between objects r and s. (A and B are called the *distance* and *flow matrix*, respectively.) The goal in the QAP is to find an optimal *assignment*, *i.e.*, a mapping $\psi : I \mapsto I$, from objects to locations such that (i) every object is assigned to some location, (ii) every location is assigned at most one object and (iii) the function

$$f(\psi) = \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij} a_{\psi(i)\psi(j)} \tag{10.2}$$

is minimised. Here $\psi(i)$ denotes the location of object i under assignment ψ and the term $b_{ij} a_{\psi(i)\psi(j)}$ intuitively represents the cost contribution of simultaneously assigning object i to location $\psi(i)$ and object j to location $\psi(j)$. In other words, given n, A and B, the objective of the QAP is to find $\psi^* \in argmin\{f(\psi) \mid \psi \in \Psi(I)\}$, where $\Psi(I)$ is the set of all assignments of objects represented by index set I to locations represented by the same index set. In fact, given the problem constraints of the QAP, an assignment ψ corresponds to a permutation of the objects.

Example 10.3: A Real-Life Quadratic Assignment Problem

As a real-world example of the QAP, consider the problem of optimising the layout layout of a typewriter or computer keyboard. In this case, the objects correspond to the letters of the alphabet and the locations to the keys of a keyboard. In this problem are given the empirical frequencies of letter pairs (*e.g.*, b_{we} gives the frequency that an "e" follows the letter "w"), the empirically measured time required for pressing a pair of keys (*e.g.*, a_{ij} is the time needed to press key "j" after having pressed key "i"), and the optimisation objective measures the typing efficiency that corresponds to the total time required to type a typical text. Note that the flow and the distance matrix are typically asymmetric matrices in this case. Intuitively, in this case the objective function of the QAP measures the speed of typing an arbitrary text.

This problem is called the *Quadratic* Assignment Problem because it can be seen as an integer optimisation problem with a quadratic objective function. Let x_{ij} be a binary variable which takes value 1 if object i is assigned to location j and 0 otherwise. Then, the QAP can be formulated as:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{l=1}^{n} \sum_{k=1}^{n} a_{ij} b_{kl} x_{ik} x_{jl} \tag{10.3}$$

subject to

$$\sum_{i=1}^{n} x_{ij} = 1 \quad j = 1, \ldots, n \tag{10.4}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i = 1, \ldots, n \tag{10.5}$$

$$x_{ij} \in \{0, 1\} \quad i, j = 1, \ldots, n \tag{10.6}$$

The constraints 10.4 and 10.5 enforce that every object is assigned to some location and that every location is assigned exactly one object, respectively; constraints 10.6 are the integrality constraints for the variables x_{ij}.

This formulation of the QAP corresponds to the so-called Koopman–Beckman formulation [Koopmans and Beckmann, 1957]. Note that, as done in the original paper by Koopman and Beckman, it is straightforward to include also a fixed assignment cost of objects to locations, which amounts to adding a cost term $\sum_{i=1}^{n} c^{i}_{\psi_i}$ to Equation 10.8. Here, we use the formulation without fixed assignment costs, because in the literature the variant defined by Equations 10.8 and 10.3 to 10.6 is the most studied one. A more general formulation of the QAP was introduced by Lawler [Lawler, 1963], where a four-dimensional array of coefficients $C = [c_{ijkl}]$ is given and the objective function becomes $f(\psi) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij\psi(i)\psi(j)}$. Clearly, a QAP in the Koopman–Beckman formulation can be cast into the form proposed by Lawler by setting $c_{ijkl} = d_{ij} \cdot f_{kl}$.

The QAP is an $\mathcal{NP}$-hard optimisation problem [Sahni and Gonzalez, 1976]; even achieving approximation ratios of $1 + \epsilon$ is $\mathcal{NP}$-hard. Furthermore, it is considered as one of the hardest optimisation problems in practice, since the size of QAP instances for which provably optimal solutions can be found using complete algorithms is limited to around $n = 30$ [Anstreicher *et al.*, 2002; Hahn *et al.*, 2001; Hahn and Krarup, 2000]. At the time of this writing, the largest non-trivial QAP instance solved to optimality, instance `ste36a`, has only 36 locations [Brixius and Anstreicher, 2001; Nyström, 1999]; this has to be seen in relation to the state of the art in complete algorithms for the TSP, where the largest instance that has been solved to optimality comprises 15,112 cities. Despite the small size of the instance, the computation times required are extremely high, taking approximately 180 hours of CPU time on a 800 MHz Pentium III PC [Brixius and Anstreicher, 2001]. In contrast, the best performing SLS algorithms for QAP typically require only a few seconds on a comparable machine to find the optimal solution for this instance. Hence, in practice the only feasible way to solve large QAP instances is to apply efficient SLS algorithms.

Applications and Benchmark instances

The QAP can be seen as an abstract model of a variety of practical layout and location problems. Examples of problems that were formulated as a QAP include backboard wiring [Steinberg, 1961], where computer components have to be placed such that the total amount of wiring required to connect them is minimised, hospital layout [Elshafei, 1977; Krarup and Pruzan, 1978], where the goal is to place facilities of a hospital to buildings such that the total amount of communication times distance is minimised, a typewriter keyboard design [Burkard and Offermann, 1977], the problem described in Example 10.3 or the printing of grey of a specific density [Taillard, 1995] and many others [Burkard *et al.*, 1998; Çela, 1998].

In the context of the numerous research efforts on the QAP and QAP algorithms, a large number of benchmark instances have been used. Many of these instances are available through QAPLIB, an online resource for the QAP that comprises benchmark instances, several QAP solver implementations, and further information on the QAP. QAPLIB is accessible at www.seas.upenn.edu/qaplib/. The QAPLIB benchmarks stem from various classes of instances with strongly varying characteristics. It is well known that the particular instance class and their corresponding characteristics have a considerable influence on the performance of SLS methods [Taillard, 1995; Gambardella *et al.*, 1999; Stützle and Hoos, 1999]. Generally, the QAPLIB instances can be classified into the following four categories [Taillard, 1995]:

Class 1 **Unstructured, randomly generated instances.** In these instances, the distance and flow matrix entries are generated randomly according to a uniform distribution over a given range of values. These instances are among the hardest for complete algorithms. Nevertheless, most SLS algorithms find solutions within $1 - 2\%$ from the best known solution quality relatively fast.

Class 2 **Instances with grid-based distance matrices.** In this class of instances, the distance matrix stems from a $n_1 \times n_2$ grid and the distances are defined as the Manhattan distance between grid points. These instances have multiple global optima (at least 4 in case $n_1 \neq n_2$ and at least 8 in case $n_1 = n_2$) due to the symmetries of the distance matrices. The flow matrices of these instances are generated according to various distributions, but not necessarily uniform ones.

Class 3 **"Real-life" instances.** Instances from this class are "real-life" instances from practical applications of the QAP like those described above. The matrix entries of real-life QAP instances exhibit a clear structure; in particular, the flow matrices have many zero entries and the remaining entries are not uniformly distributed.

Class 4 **Random "real-life like" instances.** Most of the real-life instances are of relatively small size. Therefore, Taillard proposed a new type of randomly generated instances, where the matrix entries are generated in such a way that they resemble distributions found in real-life instances [Taillard, 1995].

In addition, QAPLIB contains a number of additional instances, *e.g.*, instances from an instance generator with known optimal assignments [Li and Pardalos, 1992] or instances that stem from encodings of weighted tree problems [Christofides and Benavent, 1989].

Unfortunately, many of the QAPLIB instances are of relatively small size ($n \leq 50$) and do not appear to pose serious challenges to high performing SLS algorithms. In addition, QAPLIB does not provide sets of instances with systematically varied characteristics, and hence does not support comprehensive studies of algorithm behaviour in dependence of instance features.

To overcome these limitations, recent studies have proposed and used a large number of additional instances. The largest instance collection has been compiled by Stützle in the context of the research performed in the Metaheuristics Network (see also www.metaheuristics.org). It comprises instances of size between 50 and 500 with systematically varied characteristics of the underlying distance and flow matrices. Large, randomly generated real-life like instances with up to 768 objects were proposed by Taillard and are available via ina.eivd.ch/collaborateurs/etd/.

Several measures have been used to characterise QAP instances. One of these is the flow dominance (*fd*), which is defined as the coefficient of variation of entries of the flow matrix B multiplied by 100, *i.e.*, $\mathit{fd}(B) = 100 \cdot \frac{\sigma_B}{\mu_B}$, where $\mu_B = \frac{1}{n^2} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij}$ and $\sigma_B = \sqrt{\frac{1}{n^2-1} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} (b_{ij} - \mu_B)}$. A high flow dominance indicates that a large part of the overall flow is exchanged among relatively few items. Hence, randomly generated problems according to a uniform distribution will have a rather low flow dominance, whereas real-life problems, in general, have much higher flow dominance values. The distance dominance can be defined analogously. Another feature that has been used to characterise QAP instances is the sparsity of the flow or distance matrix, defined as $sp = n_0/n^2$, where n_0 is the number of zero-entries in the given flow or distance matrix.

Analyses of the search spaces of given QAP instances have shown that the different problem characteristics also translate into different search space characteristics. In general, real-life and real-like instances typically show a much larger fitness-distance correlation than class 1 instances [Stützle and Hoos, 2000]; in fact, FDC values for instances from class 1 are close to zero. Similarily, systematic variations in autocorrelation lengths have been reported between instances from the different classes [Merz and Freisleben, 2000a].

Reactive Tabu Search for QAP

Most 'simple' SLS algorithms for the QAP are based on a 2–exchange neighbourhood that contains all assignments that can be obtained by swapping the locations of two objects, *i.e.*, $N(\psi) = \{\psi' \mid (\exists r, s : r \neq s \wedge \psi'(r) = \psi(s) \wedge \psi'(s) = \psi(r)) \wedge (\forall i \notin \{r, s\} : \psi'(i) = \psi(i))\}$. This neighbourhood provides the basis for the well-known Reactive Tabu Search algorithm for the QAP (RTS-QAP).

The initial solution of RTS-QAP is a random assignment generated according to a uniform distribution. RTS-QAP can be seen as an extension of a simple best-improvement search procedure that uses the objective function of the QAP for evaluating candidate solutions. (It may be noted that by using various speed-up techniques, including caching of the Δ-evaluation of earlier iterations, each best-improvement search step can be performed in time $O(n^2)$ after the search has been initialised in time $O(n^3)$.) The tabu status is associated with atomic assignments of individual objects to locations. A search step is tabu if both objects r and s involved in the respective exchange would become assigned to a location that they occupied in the most recent tl iterations. At any time, a search step that leads to an improvement in the incumbent candidate solution can be performed regardless of its tabu status (aspiration criterion).

During the search process, RTS-QAP dynamically adjusts the most critical parameter of this basic tabu search procedure, the tabu list length *tl*. Furthermore, an escape mechanism is triggered when severe search stagnation is detected. The mechanism used for dynamically adjusting *tl* is based on the search history. More specifically, RTS-QAP stores all the candidate solutions encountered in the search trajectory since the last escape phase (or search initialisation) together with some additional information, such as the iteration number or how often a given candidate solution has been encountered (stored in variable ψ_{rep}) in a hash table (see, *e.g.*, Chapter 11 in [Cormen *et al.*, 2001]) to check whether the search process is cycling, *i.e.*, whether candidate solutions are revisited.

For the reaction mechanism, RTS-QAP uses the variables ma, the moving average of recurrences of candidate solutions in the search trajectory; $stlc$, the number of iterations since the last change of the tabu tenure; and nr, which counts the number of frequently repeated candidate solutions. These variables are initialised to zero at the start of the algorithm and after

```
procedure RTS-QAP(n, A, B)
    input matrices A, B
    output permutation ψ
    ψ := init; ψ̂ := ψ; tl := 1
    while (not terminate(ψ, maxIter)) do
        ψ' := step − TS(ψ, tl)
        if (checkRepetitions(ψ, ψ', tl)) then
            ψ := ψ'
            if f(ψ) < f(ψ̂) then ψ̂ = ψ
        else
            ψ := Escape(ψ)
            tl := 1end
    end
    return ψ̂
end RTS-QAP
```

Figure 10.4: Outline of the RTS-QAP algorithm; for details, see text.

each application of the escape mechanism.

Figure 10.4 shows a high-level outline of the RTS-QAP algorithm. The essential function for the reaction mechanism for parameter tl is $checkRepetitions$, which is given in Figure 10.5. Here, first the new candidate solution ψ' is searched in the hash table. If this search is successful, the fact that with ψ' a previously visited candidate solution is encountered indicates that the search process may be trapped. There are two possible ways to react to such a situation. As an immediate solution, one may increase the tabu tenure tl to avoid revisiting candidate solutions. This is done by increasing tl by a factor *Increase*; multiple such reactions lead to a geometric increase of tl and, hence, help to avoid revisiting candidate solutions. However, it still may happen that a number of candidate solutions, which are stored in a set of frequently revisited candidate solutions (FRS), are encountered many times; this is taken as a sign that the search trajectory is trapped in a limited area of the search space and the escape mechanism is triggered. In particular, the escape mechanism is triggered if more than *NR* candidate solutions are frequently revisited, where *NR* is a parameter of the algorithm set to three in RTS-QAP. Technically, a candidate solution is considered "frequently visited" if it was encountered more than *REP* times since the last escape (or search initialisation).

However, if tl is only increased, this may lead to a too strongly confined search trajectory. To counteract this problem, occasionally the tabu tenure is reduced by a factor *Decrease*; this is done, if more than ma iterations no change of tl occurred. Additionally, the tabu tenure is decreased whenever a situation is encountered in which all possible search steps are tabu.

The escape mechanism first clears the hash table and the tabu status of all tabu attributes, sets $tl = 1$, and then applies a large random modification to the current candidate solution. In particular, it applies $1 + (1 + r) \cdot ma/2$ random search steps, where r is a random number drawn according to a uniform distribution in the interval $[0, 1]$. In each of the steps, the pair of objects to be exchanged is selected randomly according to a uniform distribution ignoring the tabu status.

Experiments with RTS-QAP showed that it performs better than a strict tabu search algorithm and the Robust Tabu Search [Taillard, 1991] for the unstructured, randomly generated instances of Class 1. Further experiments studied the role of the additional RTS parameters that are required to steer the reaction mechanism; however, the essential point is that these parameters are typically much less sensitive to particular instances than the original parameter that is adapted. In fact, experiments with various parameter settings showed that (i) the escape mechanism is important for achieving optimal performance and that (ii) the parameter settings of *Decrease* and *Increase* have only a minor influence on the performance of the algorithm, if these were chosen within reasonable limits (Battiti and Tecchiolli recommend values of 0.9 and 1.1, respectively). In general, RTS-QAP appears to be particularly well-suited for solving unstructured QAP instances, where it achieves state-of-the-art performance. However, on more structured instances, it is often surpassed by hybrid SLS algorithms like the one presented in the following.

Population-based Iterated Local Search

One of the best performing Iterated Local Search algorithms for the QAP is a particular population-based ILS extension. In the following, we will first describe the underlying ILS algorithm (ILS-QAP) and then explain the population-based extension.

ILS-QAP starts from a random assignment and applies a first-improvement

```
function checkRepetitions(ψ, ψ', tl)
    input candidate solution ψ, ψ', tabu tenure tl
    output true or false
    stlc := stlc + 1
    revisited := searchHashTable(ψ')
    if (revisited) do
        cl = computeRevisitLength(ψ')
        ψ'_rep := ψ'_rep + 1    % Increase No. repetitions of ψ'
        if cl < Max_RevisitLength then
            adjustMovingAverage(ma, cl)
            tl := tl · Increase
            stlc := 0
        end
        if ψ'_rep > REP then
            FRS := FRS ∪ ψ'
            nr := nr + 1
            if nr > NR then
                nr := 0
                return true    % Execute an escape in this case!
            end
    else
        introduceHashTable(ψ')
    end
    if (stlc > ma or N(ψ) = ∅)
        tl := tl · Decrease
        stlc := 0
    return false
end checkRepetitions
```

Figure 10.5: Outline of the reaction mechanism in RTS-QAP. *NR*, *REP* and *Revisit_Max* are constants that are set to 3, 3, and 50, respectively in RTS-QAP.

2-opt local search procedure based on the same neighbourhood that is used in RTS-QAP. One particularity of this local search procedure is that the neighbourhood is scanned in a random but fixed order. Hence, even when initialised with the same assignment, it may return different locally optimal candidate solutions; in the context of the ILS algorithm, this has the advantage that even after applying a relatively weak perturbation to a locally optimal assignment ϕ', a subsequent local search is rather unlikely to return to the same ϕ'.

In order to achieve a performance improvement of this first-improvement 2-opt local search procedure compared to a best-improvement 2-opt procedure that uses the same speed-up techniques as the best-improvement search underlying RTS-QAP, it is essential to use don't look bits that are associated with each item (see also Section 8.2 on page 318). If during the neighborhood scan for an item no improving move is found, its don't look bit is turned on (set to 1) and the item is not considered as a starting item for a neighborhood scan in the next iteration. Whenever an item is involved in a 2-exchange move and changes its location, the don't look bit is turned off again.

Each perturbation phase consists of a random k-exchange move. As outlined in Section 318, the don't look bit technique is integrated into the perturbation by resetting to zero only the don't look bits of those items that change their location in the k-exchange move. In fact, this resetting strategy of the don't look bits results in an additional significant speed up and, if k is not too large, it allows to increase strongly the number of local searches that can be applied in a fixed computation time when compared to the strategy of resetting all don't look bits to zero. As in basic Variable Neighbourhood Search (VNS) [Hansen and Mladenović, 1999; Hansen and Mladenovic, 2001], the value of k is dynamically modified during the search. In particular, we vary k between two values k_{min} and k_{max} starting at k_{min}: If after the perturbation and the subsequent local search no better candidate solution is found, k is increased by one; otherwise it is set to k_{min}. Whenever k reaches the value k_{max}, it is reset to k_{min}. An exception is made when starting the ILS: We start at k_{max} and decrease k by one after each iteration until we reach k_{min} (backward VNS); only then we change to the "forward" VNS. This mechanism for dynamically modifying k makes the algorithm's performance somwhat more robust compared to using a fixed value of k; this is particularly important, because there appears to be no single fixed value for k that leads to very good performance across a diverse set of instances.

The acceptance criterion selects the new assignment if and only if its quality is equal to or better as as the previous locally optimal assignment; hence, the ILS algorithm performs an iterated descent in the space of 2-exchange local optima.

The population-based ILS extension of this algorithm, PBILS-QAP, follows, at least in principle, the same ideas as earlier population-based ILS algorithm for the TSP (see Section **??**, page **??***ff.*). In the QAP case, PBILS applies to each assignment of the population one iteration of the ILS algorithm described above. Then, the new population is determined by a variant of the $(\mu + \lambda)$-selection that is frequently used in Evolution Strategies [Schwefel, 1981]. In the original $(\mu + \lambda)$-selection, μ is the population size and λ the number of offspring (in PBILS-QAP, we have $\mu = \lambda$); then the new population is comprised of the μ best out of $\mu + \lambda$ candidate solutions. This selection mechanism strongly favours the best candidate solutions and can hence easily lead to premature convergence of the population and subsequent search stagnation. In order to avoid this, we apply two additional diversification techniques. Firstly, when selecting the surviving assignments that will comprise the next generation, the $\mu + \lambda$ given assignments (current population + offspring) are considered in the order of their solution quality. Before inserting an assignment ψ into the population, first the distance to each of the assignments that are already included in the population is measured. Here, we define the distance between two assignments ψ and ψ' to be the number of objects that are placed on distinct locations in ψ and ψ', i.e., $d(\psi, \psi') = |\{i \mid \psi(i) \neq \psi'(i)\}|$. This is a direct extension of the well-known Hamming distance for bit strings. An assignment is inserted into the population if the minimum distance to any of the assignments already included in the population is larger than some threshold distance d_θ. In fact, we vary d_θ dynamically during the run of the algorithm by subsequently lowering the actual value of d_θ during a run up to some lower, minimum value d_{min}. This ensures that a high degree of diversification is achieved in early stages of the search process. Secondly, PBILS-QAP uses a restart operator that applies a strong perturbation to all the candidate solutions of the current population; this restart is invoked if no improved assignment is found for a number of iterations or the average distance between the elements in the population is below some threshold value.

PBILS-QAP shows better performance than several of the population-based ILS variants described in Section **??**, ILS-QAP, and several other ILS variants that use only a single candidate solution **??**. Experimental results with PBILS-QAP have shown that it is a state-of-the-art algorithm for real-life and real-life like QAP instances [Stützle, 1999]. In fact, an extensive experimental evaluation of several metaheuristics for the QAP in the context of the metaheuristics network (see www.metaheuristics.org) has shown that PBILS-QAP performed best across a large set of QAP instances.

Generalisations and Related Problems

Several variants and generalisations of the QAP can be found in the literature. For example, the *Bottleneck QAP* is a variant of the standard QAP in which the goal is to minimise the objective function

$$f_b(\psi) = \max\{b_{ij} a_{\psi(i)\psi(j)} \mid 1 \leq i, j \leq n\}. \tag{10.7}$$

That is, in the Bottleneck QAP one tries to minimise the maximum cost instead of the sum of the costs. The Bottleneck QAP first arose in the backboard wiring application of Steinberg [Steinberg, 1961] in cases where the goal is to minimize the maximum length of wires.

In the *Quadratic Semi-assignment Problem (QSAP)*, $m < n$ locations are given and the goal is to assign all objects to locations such that (i) every location is assigned at least one object and (ii) such that the sum of the flows between the objects times the distances between the objects' locations is minimized. The QSAP is $\mathcal{NP}$-hard and apparently difficult to solve [**?**]; a Tabu Search approach to the QSAP is proposed in [W. Domschke, 1992].

The *Biquadratic Assignment Problem (BiQAP)* is a generalisation of the QAP to a quartic assignment problem, where two four-dimensional matrices $n \times n \times n \times n$ matrices $A = (a_{ijkl})$ and $B = (b_{rsuv})$ are given and, under the straightforward extension of the assignment constraints the function

$$f(\psi) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{l=1}^{n} b_{ijkl} a_{\psi(i)\psi(j)\psi(l)\psi(k)} \tag{10.8}$$

has to be minimised. The BiQAP arises in the design of Very Large Scale Integrated (VLSI) sequential circuits. Burkard et al. [Burkard *et al.*, 1994] give a detailed description of the VLSI design problem that leads to the BiQAP. In addition, they investigated lower bounds for the BiQAP but found that these still left a large gap to the optimal solution and concluded that complete algorithms are likely to perform extremely poorly on the BiQAP [Burkard *et al.*, 1994]. Therefore, several SLS algorithms were proposed for the BiQAP, including iterative improvement algorithms, Simulated Annealing, Tabu Search (all these were implemented by Burkard and Çela [Burkard and Çela, 1995]) and GRASP [Mavridou *et al.*, 1998] and tested on randomly generated instances with known optimal solutions based on a generator described in [Burkard *et al.*, 1994]. Of the SLS algorithms tested, the GRASP appears to return the best quality solutions, however, at the cost of substantial computation times.

10.3 Set Covering

The Set Covering Problem (SCP) is a well-known combinatorial optimisation problem with a large number of applications. Given a finite set $A = \{a_1, \ldots, a_m\}$ and a family $\mathbf{F} = \{F_1, \ldots, F_n\}$ of A of subsets of A that covers A, *i.e.*, every element of A appears in at least one set in $\mathbf{F}$. In the *minimum SCP*, the goal is to find a minimum size subset $\mathbf{C}^* \subseteq \mathbf{F}$ that covers A, *i.e.*, to find $\mathbf{C}^* \in argmin\{\#\mathbf{C} \mid \mathbf{C} \subseteq \mathbf{F} \wedge \bigcup \mathbf{C} = A\}$. In the *weighted SCP*, additionally a weight function $w : \mathbf{F} \mapsto \mathbb{R}^+$ is given that assigns a positive weight (or cost) to each element of $\mathbf{F}$, and the objective is to find a set cover C with total minimal weight, *i.e.*, $C^* \in argmin\{\sum_{C \in \mathbf{C}} w(C) \mid \mathbf{C} \subseteq \mathbf{F} \wedge \bigcup \mathbf{C} = A\}$. Clearly, the minimum SCP can be seen as a special case of the weighted SCP: Every minimum SCP instance is equivalent to a weighted SCP instance in which all elements of $\mathbf{F}$ have the same weight. For that reason, the minimum SCP is also know as the *unicost SCP*. In the remainder of this section we will mainly focus on the more general weighted SCP, which in the following we simply refer to as the SCP.

Frequently, the SCP is formalised as an integer programming (IP) problem. In this case, a variable x_i is associated with each subset F_i and $x_i = 1$ indicates that subset F_i is chosen to be in the current candidate cover $\mathbf{C}$, while $x_i = 0$ indicates $F_i \notin \mathbf{C}$. Each set F_i $(i \in \{1, \ldots, n\})$ is represented by a column in a $m \times n$ matrix $B = (b_{ji})$, with $b_{ji} = 1$ if $a_j \in F_i$ and $b_{ji} = 0$ otherwise. Intuitively, each row of B corresponds to an element of

A and we say that a column i covers a row j if element i is contained in subset F_i, *i.e.*, $b_{ji} = 1$. Furthermore, let $c_i = w(F_i)$, the weight of F_i, be the cost associated to having a column in a solution. The SCP can then be represented by the following integer programming problem:

$$\min f(x) = \sum_{i=1}^{n} c_i \cdot x_i \tag{10.9}$$

subject to

$$\sum_{i=1}^{n} b_{ji} \cdot x_i \geq 1 \quad j = 1, \ldots, m \tag{10.10}$$

$$x_i \in \{0, 1\} \quad i = 1, \ldots, n \tag{10.11}$$

The constraints 10.10 enforce that each element of A (*i.e.*, each row of B) is covered by at least one element of **C** (*i.e.*, by a column i of B for which $x_i = 1$), and the integrality constraints 10.11 specify the domains of the x_i.

Most of the literature on SLS algorithms for the SCP uses the IP formulation; hence, in the following, we will follow the same convention and refer to the elements of A as rows (of the matrix B), to the subsets F_i as columns, and to the weights $w(F_i)$ as column costs. Consequently, we say a given column i covers a row j if F_i contains element a_j, *i.e.*, if $b_{ji} = 1$.

Example 10.4: Set Covering Problem

Consider the following example of a minimum set covering problem, with $A = \{a, b, c, d, e, f, g\}$ and $\mathbf{F} = \{F_1, \ldots, F_6\}$ defined as follows:

$$\begin{aligned} F_1 &= \{a, b, f, g\} \\ F_2 &= \{a, b, g\} \\ F_3 &= \{a, b, c\} \\ F_4 &= \{e, f, g\} \\ F_5 &= \{f, g\} \\ F_6 &= \{d, f\} \\ F_7 &= \{d\} \end{aligned}$$

Notice that F_2 and F_5 are a proper subset of F_1, which is also the largest of the sets F_i. However, the only possible solution that includes F_1, which is

$s_1 = \{F_1, F_3, F_4, F_6\}$, has a cardinality of four, while an optimal solution requires only three subsets.

In the integer programming formulation, this SCP instance is represented by the matrix

$$B = \begin{pmatrix} 1 & 1 & \mathbf{1} & \mathbf{0} & 0 & \mathbf{0} & 0 \\ 1 & 1 & \mathbf{1} & \mathbf{0} & 0 & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 0 & \mathbf{1} & 1 \\ 0 & 0 & \mathbf{0} & \mathbf{1} & 0 & \mathbf{0} & 0 \\ 1 & 0 & \mathbf{0} & \mathbf{1} & 1 & \mathbf{1} & 0 \\ 1 & 1 & \mathbf{0} & \mathbf{1} & 1 & \mathbf{0} & 0 \end{pmatrix}$$

where the columns corresponding to the F_i contained in a particular optimal cover are shown in boldface. Note that the first column of B contains entries $b_{ji} = 1$ for the four elements of F_1, a, b, f and g. This leads to the following integer programme (the indicator variables x_i that have value 1 in an optimal solution are printed in boldface):

$$\min f(x) = \sum_{i=1}^{7} x_i$$

subject to

$$\begin{array}{ccccccccccccccl} x_1 & + & x_2 & + & \mathbf{x_3} & & & & & & & & & \geq 1 \\ x_1 & + & x_2 & + & \mathbf{x_3} & & & & & & & & & \geq 1 \\ & & & & \mathbf{x_3} & & & & & & & & & \geq 1 \\ & & & & & & & & & & \mathbf{x_6} & + & x_7 & \geq 1 \\ & & & & & & \mathbf{x_4} & & & & & & & \geq 1 \\ x_1 & + & & & & & \mathbf{x_4} & + & x_5 & + & \mathbf{x_6} & & & \geq 1 \\ x_1 & + & x_2 & + & & & \mathbf{x_4} & + & x_5 & & & & & \geq 1 \end{array} \qquad (10.12)$$

$$x_i \in \{0, 1\}, \quad i = 1, \ldots, 7$$

It may be noted that this SCP instance has two optimal covers; although one of these covers each element of A exactly once and the other covers one element twice, both have the same objective function value (number of subsets $F_i \in \mathbf{F}$) and are hence equally optimal.

The SCP is an $\mathcal{NP}$-hard combinatorial optimisation problem. In the case of the minimum SCP, an approximation algorithm exists that is guaranteed to return a solution that is at most by a factor of $1 + \ln m$ worse than the minimum number of sets need to cover A [Johnson, 1974b]; the same bound also holds for the weighted case [Chvátal, 1979]. However, even for the minimum SCP, it can be shown that approximating it within a logarithmic factor is $\mathcal{NP}$-hard[Raz and Safra, 1997], *i.e.*, there does not exist an algorithm that returns a solution that is only worse by some constant factor than the optimal solution.

Applications and Benchmark instances

The SCP has many important real-world applications ranging from airline crew scheduling [Housos and Elmoth, 1997], driver scheduling in public transportation [Lourenço *et al.*, 2001], and scheduling and production planning in several industries [Vasko and Wolf, 1988], which can all be modelled as SCP instances. How these application problems are mapped onto the SCP is illustrated in the following example.

Example 10.5: SCP in Crew Scheduling Applications

In this example, we outline how a particular problem that arises in the context of scheduling crews in the airline industry can be formulated as a weighted SCP. The problem is, given a timetable of flight legs, to assign a crew to each flight leg so that the overall cost of a solution is as low as possible. Each crew has a home base, and a schedule for a crew comprises a series of flight legs that start and end at its home base. A possible schedule has to obey legal and contractual rules such as prescribed rest times or maximum working times.

To solve the problem of finding optimal schedules that cover every flight leg, first a large number of possible schedules are generated for each crew and their costs are computed. Such a schedule is also called a *pairing*, *i.e.*, a pairing is a sequence of flight legs that can be performed by a single crew.

Second, a subset of the generated schedules is selected such that every flight leg is included in at least one pairing and the total cost is minimised. Obviously, every flight leg needs to be covered, since there must be at least one crew for each flight leg; however, it is possible that a flight leg is covered several times and one crew is simply travelling on a flight as passengers. In real applications, instances with up to a few thousand flight legs and hundred thousands of possible schedules may be obtained [Wedelin, 1995b].

As can be expected, in this application, the cardinality of each of the sets F_i, *i.e.*, the number of non-zero entries in each column, is typically rather small. In the IP formulation, this leads to matrices B with very low density, *i.e.*, matrices that contain a very small fraction of non-zero entries.

The application relevance of the SCP is witnessed by the fact that many commonly used benchmark instances stem from real-world SCP applications, including the benchmark set introduced by Balas and Carrera [Balas and Carrera, 1996], which is derived from applications in the airline industry (`aa*`) or bus companies (`bus*`). The instances range in size from 105 to 681 rows (items in A) and from 2241 to 9524 columns (given subsets $F_i \in \mathbf{F}$). As typical in most "real-world" SCP applications, the density of the matrices is low, ranging between 0.51% and 4.11%, which reflects the fact that the sets F_i are small compared to A.

Another prominent set of benchmark instances stems from the FASTER (Ferrovie Airo Set covering TendER) competition, which was organized by the Italian railway company; these instances are derived from a real-world set covering problem arising in railway crew scheduling applications. The smallest of the seven instances has 507 rows and 63,009 columns, while the largest comprises 4872 rows and 968,672 columns. These instances also show some particularities as all column costs are either one or two and a column covers at most 12 rows.

Several SCP instances from crew-scheduling applications in the airline companies were proposed by Wedelin [Wedelin, 1995a]. The original instances were reduced by a preprocessing stage that eliminates columns and / or rows based on some simple criteria. The resulting instances range from as few as 29 rows and 157 columns to 1,585 rows and 105,804 columns. The matrix density drops from 8.2% for the smallest instance to 0.3% for the largest one.

Algorithms for the SCP are also often tested on randomly generated instances. The most widely used set of such instances is available from ORLIB at http://mscmga.ms.ic.ac.uk/info.html; it comprises instances from 200 rows and 1,000 columns up to 1,000 rows and 10,000 columns. These instances are randomly generated in such a way that every column covers at least one row, *i.e.* none of the F_i is empty, and every row is covered by at least two columns. The column costs are randomly generated integers from the interval $[1, 100]$. The densities of these instances vary between 2% and 20%. In addition, one set of relatively small unicost instances with 50 rows and 500 columns is available. Interestingly, results from a recent study indicate that these randomly generated instances differ from instances derived from real-world SCP applications in terms of their search space characteristics [Finger *et al.*, 2002].

Iterated Greedy (IG) Algorithms

The first heuristic approaches to the SCP were greedy construction heuristics [Chvátal, 1979; Balas and Ho, 1980], which iteratively add columns (*i.e.*, subsets) to a partial candidate solution until all rows (*i.e.*, elements of A) are covered. Greedy construction heuristics are also underlying some of the currently best performing SLS algorithms for the SCP that can be seen as *Iterated Greedy* heuristics [Pranzo and Stützle, 2003].

The main idea behind IG is to alternate construction search phases with *destruction phases*, during which some solution components are removed from a complete candidate solution. At the end of each construction phase, a complete candidate solution $s^{'}$ is obtained, and — analogously to Iterated Local Search (ILS) — an acceptance criterion is used to decide whether the search constinues from this new complete candidate solution or from the one that served as the starting point of the most recent destruction phase. (An outline of IG is shown in Figure 10.7.)

IG has strong analogies to Iterated Local Search (ILS): the construction and destruction phases in IG correspond to the local search and perturbation phases in ILS, respectively. Obviously, this general IG scheme can be improved and extended in various ways, *e.g.*, by adding an additional local search phase. Furthermore, the performance of any algorithm based on this scheme strongly depends on the heuristics and selection methods used in the

procedure $IteratedGreedy(\pi')$
  input *problem instance* $\pi' \in \Pi'$, *objective function* $f(\pi)$
  output *solution* $\hat{s} \in S(\pi')$ **or** $\emptyset$
  $s := init(\pi')$
  $\hat{s} := s$
  while not $terminate(\pi', s)$ **do**
    $s'_p = destructionPhase(\pi', s)$
    $s' = constructionPhase(\pi', s'_p)$
    if $(f(s') < f(\hat{s}))$
      $\hat{s} := s''$
    end
    $s = accept(\pi', s, s')$
  end
  if $\hat{s} \in S'$ **then**
    return $\hat{s}$
  else
    return $\emptyset$
  end
end $IteratedGreedy$

Figure 10.6: Algorithmic outline of Iterated Greedy (IG). (For details, see text.)

construction and destruction phases as well as on the acceptance criterion.

Jacobs and Brusco proposed the application of an IG heuristic to the SCP (IG-JB) [Jacobs and Brusco, 1995]. (The authors actually refer to the algorithm as a Simulated Annealing heuristic, however, a description of the approach as an IG algorithms appears to be more appropriate.) In the case of the Weigthed Minimum SCP, the candidate solutions are covers of the given set A. As previously explained, these covers as well as the partial covers that represent partial candidate solutions can be equivalently represented as sets of columns of the matrix B from the IP formulation of the given SCP instance. In the following we will use this representation of partial covers. The core of the IG-JB algorithm follows the general IG outline as shown in Figure 10.7; however, after each construction phase an additional procedure $RemoveRedundantColumns$ is applied to eliminate redundant columns from the cover $\mathbf{C}'$. A column i becomes redundant if all the rows covered by it are also covered by one or more columns (*i.e.* elements of $\mathbf{C}'$ that are added later during the construction process. In particular, $RemoveRedundantColumns$ examines the columns of the solution in nonincreasing order of their cost and it is checked, whether a column can be removed without resulting in an unfeasible solution.

The initial solution in IG-JB is generated using a greedy construction heuristic based on an algorithm by Balas and Ho [Balas and Ho, 1980]. This greedy construction heuristic assumes that all columns are ordered according to their cost values in nondecreasing order. It then iterates through the following two steps until all rows are covered:

1. Randomly select a currently uncovered row i according to a uniform distribution

2. Add the lowest cost column j that covers row i.

Once a cover is constructed, redundant columns are eliminated using procedure $RemoveRedundantColumns$.

The procedure *DestructionPhase* iteratively removes a fixed number of $k_1 \cdot |\mathbf{C}|$ columns from the current cover, where $k_1, \quad 0 < k_1 < 1$ is a parameter and $|\mathbf{C}|$ is the number of columns in a cover C. During this process, the columns to be removed are chosen uniformly at randomly from the remaining elements of $\mathbf{C}$.

Next, in procedure *ConstructionPhase* a complete candidate solution is regained as follows: First, a candidate set is built that consists of all columns with cost less than $k_2 \cdot f(\mathbf{C})$, where $k_2, k_2 > 0$ is a parameter and $f(\mathbf{C})$ is the objective function value of cover $\mathbf{C}$ before invoking the destruction phase. The parameter k_2 has a direct influence of the size of the candidate set. Note that if the candidate set is too small, there is a risk to never find an optimal cover. Second, for each of the columns in the candidate set the *cover value* $\gamma_i = c_i/b_i$ is computed, where b_i is the number of rows covered when adding column i to the current partial cover. In other words, the cover value gives the unit cost of covering one additional row. Last, a column with minimum cover value γ_{min} is added to the partial solution; if there is more than one column with minimum cover value, one of them is chosen uniformly at random. These steps are iterated until a complete candidate solution, *i.e.* a cover of A is obtained.

Finally, the acceptance criterion in IG-JB is taken from Simulated Annealing: If $f(s^{'}) < f(s)$, $s^{'}$ becomes the new incumbent solutions, otherwise $s^{'}$ replaces s with a probability of $\exp((f(s^{'}) - f(s))/T)$, where T is the temperature parameter. For the acceptance criterion a standard geometric cooling schedule is applied.

Computational results show that IG-JB outperforms an earlier proposed Lagrangian heuristic for the SCP by Beasley [Beasley, 1990], but it is now outperformed by more recent SLS algorithms for the SCP [Brusco *et al.*, 1999; Caprara *et al.*, 1999; M. Yagiura and Ibaraki, 2001], including the IG-MS algorithm presented in the following.

The IG Algorithm by Marchiori and Steenbeck (IG-MS)

Another variation on an Iterated Greedy heuristic was proposed by Marchiori and Steenbeck [Marchiori and Steenbeek, 2000a] (IG-MS). In addition to the standard procedures of IG, IG-MS uses (i) a procedure *RecomputeCore* that occasionally computes a new core problem, *i.e.*, a smaller SCP instance containing only a subset of the columns that are most likely to appear in optimal solutions, and (ii) an additional local optimisation procedure that tries to improve upon the solution returned by procedure *ConstructionPhase*.

Before explaining the components of the IG-MS algorithm in detail, we need to introduce the heuristic values that control the the construction and destruction phases of IG-MS are defined. Let $\mathbf{C}$ be a current (partial) cover and $cov(\mathbf{C})$ be the set of rows that are covered by $\mathbf{C}$; $cov(i, \mathbf{C})$ is the set of rows that are covered by column i, but are not covered by any columns in $\mathbf{C}\backslash\{i\}$ (recall that $\mathbf{C}$ can be seen as a subset the columns in B). We denote with $c_{\mathit{min}}\,(j)$ the minimum cost of a column that covers row j. Then, for IG-MS the cover utility $cv(i, \mathbf{C})$ that evaluates the usefulness of column i with respect to a cover $\mathbf{C}$ is defined as

$$cv(i, \mathbf{C}) = \sum_{j \in cov(i,\mathbf{C})} c_{\mathit{min}}\,(j)$$

Note that if $cv(i, \mathbf{C}) = 0$, then column i is redundant with respect to $\mathbf{C}$.

procedure *Greedy Construction IG-MS*
  input *partial (or empty) cover* $\mathbf{C}$
  output *complete cover* $\mathbf{C}$
  while ($\mathbf{C}$ is not a complete cover) **do**
    $j := SelectAdd(\mathbf{C})$
    $\mathbf{C} := \mathbf{C} \cup j$
    while $(RemoveColumns)$ **do**
      $j := SelectRemove(\mathbf{C})$
      $\mathbf{C} := \mathbf{C} \backslash \{j\}$
    end
  end
return $\mathbf{C}$ **end** *Greedy Construction IG-MS*

Figure 10.7: Algorithmic outline of the greedy construction heuristic of IG-MS. (For details, see text.)

$cv(i, \mathbf{C})$ is used to define a selection utility $sv(i, \mathbf{C})$ that is defined as

$$sv(i, \mathbf{C}) = \begin{cases} \infty, & \text{if } i \text{ is redundant w.r.t. } \mathbf{C}; \\ \\ c_i / cv(i, \mathbf{C}), & \text{otherwise}; \end{cases} \tag{10.13}$$

The greedy construction heuristic of IG-MS iteratively adds columns that are not in a partial cover $\mathbf{C}$ with maximum selection value. Additionally, columns may be removed from $\mathbf{C}$ if the predicate *RemoveColumns* is true; this is the case if $\mathbf{C}$ contains at least one redundant column; otherwise, with a probability w_r, which is set to 0.3 in IG-MS [Marchiori and Steenbeek, 2000a], *RemoveColumns* is true and false in the remaining cases (and if $\mathbf{C} = \emptyset$). The procedure *SelectRemove* selects a column of $\mathbf{C}$ with maximum selection value.

Once the greedy construction finishes with a cover $\mathbf{C}$ that does not contain any redundant column, $\mathbf{C}$ is locally optimised. The local optimisation tentatively adds a column i to $\mathbf{C}$ trying to make at least two of the columns in $\mathbf{C}$ different from j redundant in such a way that the sum of the costs of the columns that can be removed after adding i is larger than c_i. If such a column i can be found, the cost of a cover can be reduced; such a column i is also called a *superior column*. The local optimisation procedure first generates a list of superior columns that are ordered according to nonincreasing gains and, second, tests these columns for possible improvements (after adding the first superior column, the remaining ones need not anymore be superior).

The destruction phase is implemented as a selection procedure from a set E of *elite columns*; E comprises all columns that are in the incumbent cover $\hat{\mathbf{C}}$ (*i.e.*, in the best cover seen up to a given point in the search process) and for which $cv(i, \mathbf{C}) > c_i$. For each of the columns in E the number of times they occured in an incumbent cover $\hat{\mathbf{C}}$ is counted; the destruction procedure selects from E the columns that occured rarely in an incumbent cover, while the other columns are selected independently with a probability that is set to a value chosen randomly in the interval $[0.1, 0.9]$. In general, this desctruction phase returns a partial cover that contains a subset of the columns of the best cover seen so far; it therefore corresponds to an acceptance criterion that only accepts improved solutions. Finally, a procedure *RecomputeCore* recomputes the SCP core problem, *i.e.*, a candidate list of columns that are considered for inclusion in a candidate solution. The core problem is built by first including columns from the set E with a very high probability, second adding columns i for which a row j exists that is covered by i and we have $k_0 \cdot c_i < c_{min}$, and third adding a column i that covers a row j that is covered by less than k_1 columns. The core problem is re-computed every 100 iterations of the algorithm.

IG-MS was compared to several SLS algorithms including the CFT heuristic [Caprara *et al.*, 1999], one of the currently best performing algorithms for the SCP. IG-MS found the best solutions obtained by CFT on almost all instances tested including Wedelin's and Balas-Carrera's real-world instances and the randomly generated instances available from ORLIB; the only differences were that IG-MS found on two instances slightly better best solutions and on one slightly worse ones. The computation time required by IG-MS for obtaining these solution qualities appears to be very competitive to CFT; however, because of differences in the experimental protocols used for evaluating the two algorithms by their respective authors, further experimental research is required to establish their relative performance more precisely.

Related Problems

There are a number of subset problems related to Set Covering. The Minimum k-set Covering Problem is an SCP with the additional restriction that the cardinality of all sets in subsets A_i of A is bounded from above by a constant k; this variant is approximable within $\sum_{i=1}^{k} \frac{1}{i} - 1/2$ [Duh and Furer, 1997].

The Set Partitioning Problem arises if each element of A must appear in exactly one subset A_i. In this case, for all $A_i, A_j \in s$ we have that $A_i \cup A_j = \emptyset$ and $\bigcup_{i=1}^{n} A_i = A$. In the integer programming formulation this results in replacing the "$\geq$" in Equation 10.10 on page 385 by an equality. Similarily to the SCP, the Set Partitioning Problem arises in a wide variety of applications like crew scheduling and vehicle routing [Balas and Padberg, 1976]. Differently from the SCP case, only few SLS algorithms for the Set Partitioning Problem were proposed in the literature [Chu and Beasley, 1998; Maniezzo and Milandri, 2002].

In the Set Packing Problem we are given a set $A = \{1, \ldots, m\}$ of m elements and n subsets $A_i, i = 1, \ldots n$ as in the SCP case, but the goal now is to find the maximum number of sets such that all chosen subsets A_i are mutually disjoint, *i.e.*, for all $A_i, A_j \in s$ we have that $A_i \cup A_j = \emptyset$. Again the problem can be extended to the weighted case by assigning to each set A_i a positive real weight c_i and in the weighted Set Packing Problem the goal becomes to maximise the total weight of the sets in the set packing. Note that the integer programming formulation of the problem is analogous to the SCP formulation except that we have to replace the "$\geq$" in Equation 10.10 by a "$\leq$". The Set Packing Problem often arises in applications where as much demand as possible is to be satisfied, however, without creating conflicts.

10.4 Combinatorial Auctions

Auctions play an important role in economics as well as in multi-agent systems, where auction mechanisms are used for resource allocation and task distribution. The items that are auctioned range from household goods to radio frequencies, network bandwidth, and pollution rights. In a combinatorial auction, bids can be placed on bundles of items. In situations where a complete bundle of goods is required for a certain purpose or task, the ability to bid directly for a bundle instead of bidding individually on all respective items allows bidders to minimise their risk of getting stuck with incomplete bundles. At the same time, overlaps between such bundle bids makes it difficult to determine an assignment of items to bids that maximises the revenue of the auctioneer. SLS algorithms are amongst the best-performing methods for finding optimal or very high-quality solutions to this Combinatorial Auctions Winner Determination Problem.

In this section we introduce the Combinatorial Auctions Winner Determination Problem (CAWDP) and present two high-performance SLS algorithms for solving this $\mathcal{NP}$-hard combinatorial optimisation problem, the Casanova and the Exponentiated Subgradient algorithms, both of which are inspired by high-performance SLS algorithms for SAT. We also discuss some generalisations of CAWDP as well as related problems.

Winner Determination in Combinatorial Auctions

In a combinatorial auction, a seller has a *set of items* $I = \{i_1, \ldots, i_m\}$ to be auctioned. Potential buyers value different subsets or *bundles* of items, $S \subseteq M$, and submit *bids* of the form $b = (S, p)$, where p, the *price of* b, is a positive real number that represents the amount the buyer is willing to pay for bundle S. An instance of a combinatorial auction is then given by a set of items I and a set of bids B.

An *allocation* is a set of bids $A \subseteq B$ that are considered winning. For a given allocation A, all bids $b \in A$ are called *satisfied* or *winning*, and all $b \notin A$ are called *unsatisfied*. An allocation $A \subseteq B$ is called *feasible* if it does not contain any pair of bids that require the same item, *i.e.*, if $\nexists i \in I : \#\{(S,p) \in A \mid i \in S\} > 1$; the set of all feasible allocations for given sets of items and bids is denoted $\mathit{FAlloc}(I, B)$. The value of a feasible allocation A, denoted $\mathit{val}(I, B, A)$, is defined as the sum of the prices of the bids satisfied under A.

The Combinatorial Auctions Winner Determination Problem (CAWDP) is defined as follows: Given a set of items $I = \{i_1, \ldots, i_m\}$ and a set of bids $B = \{b_1, \ldots, b_n\}$, find a feasible allocation A^* with maximal value, *i.e.*, find $A^* \in \mathit{argmax}\{\mathit{val}(I, B, A) \mid A \in \mathit{FAlloc}(I, B)\} = \mathit{argmin}\{-\mathit{val}(I, B, A) \mid A \in \mathit{FAlloc}(I, B)\}$. (Although it may be more intuitive to think of CAWDP as a maximisation problem, following our general convention, we present it here as a minimisation problem.) Typically, the output desired from a winner determination algorithm is a set of winning bids, *i.e.*, a feasible allocation, as well as an assignment of items to winning bids; the latter is easily and efficiently computed from the former. Here and in the following, it is assumed that any items that are not assigned to a winning bid do not cause any direct cost to the auctioneer; if this so-called *free disposal* assumption is not met, the winner-determination problem can become substantially harder to solve (see, *e.g.*, [Sandholm *et al.*, 2002]).

Example 10.6: Simple CAWDP Instance

Consider a combinatorial auction with items $I = \{a, b, c, d, e\}$ and the following set if bids, B:

$$\begin{array}{rcl} b_1 & = & (\{a, c\}, 5.5) \\ b_2 & = & (\{a, c, d\}, 15) \\ b_3 & = & (\{b\}, 1) \\ b_4 & = & (\{b, d\}, 12) \\ b_5 & = & (\{d\}, 8) \\ b_6 & = & (\{d, e\}, 10) \end{array}$$

Note that the combined value of the two bids for the individual items b and d is lower than the value of the bundle bid for both (b_4), which reflects the complementarity of these items.

Both, $A_1 = \{b_2, b_4\}$ and $A_2 = \{b_1, b_4\}$ are allocations, but clearly, while A_2 is feasible, A_1 is infeasible because b_2 and b_4 both require item d. The value of A_2, $val(I, B, A_2)$ is 17.5, which is the maximal value over all possible feasible allocations for this problem instance. Under the optimal assignment A_2, bids b_1 and b_4 win, with items a and c assigned to b_1 and b, d assigned to b_4. Note that e remains unassigned under this assignment; there is a feasible assignment that assigns all items to bids ($A_3 = \{b_1, b_3, b_6\}$), but its value is lower than 17.5.

CAWDP can be represented as a Boolean Linear Program (BLP). In this formulation, allocations are represented using a binary variable x_j for each bid $b_j \in B$ that indicates whether b_j is part of the current allocation ($x_j = 1$)

or not ($x_j = 0$). Optimal allocations correspond to solutions of

$$\mathbf{x}^* \in argmin\{\ -\sum_{j=1}^{n} x_j \cdot p_j \quad | \quad (x_1, \ldots, x_n) \in \{0,1\}^n \text{ and } \forall i \in I : \sum_{j=1}^{n} c_{ij} \cdot x_j \leq 1\ \}$$

where $c_{ij} = 1$ if $i \in S_j$ for some $(S_j, p_j) \in B$, and 0 otherwise. Using this BLP formulation, standard solvers for BLPs or more general Integer Programs (IPs) can be applied to the Combinatorial Auction Winner Determination Problem.

Winner determination in combinatorial auctions is an $\mathcal{NP}$-hard problem. It has been shown that no polynomial-time algorithm can achieve approximation ratios better than or equal to $n^{1-\epsilon}$, where n is the number of bids, for any $\epsilon > 0$ (unless $\mathcal{NP} = \mathcal{ZPP}$; $\mathcal{ZPP}$ is the class of problems that can be solved in expected polynomial time by a probabilistic algorithm with zero error probability) [Sandholm, 2002; 1999]. The best known approximation algorithm for CAWDP achieves a worst case approximation ratio of $O(n/log(n))$ [Halldórsson, 2000].

Benchmark Instances for CAWDP

Along with the development of CAWDP algorithms, various classes of random instance distributions have been proposed and used for empirical performance evaluations. These instance distributions are based on generative probabilistic models of varying complexity. The simplest instance generators are based on simple distributions for bundle sizes and compositions as well as prices. The design of complex generators, on the other hand, is based on features of various types of real-world application domains of combinatorial auctions.

Given a number of items and bids, simple random instance distributions are obtained by using a combination of probability distributions for determining for each bid $b = (S, p)$ independently the bid length (*i.e.*, the size of S), the items in S, and the price p. In one of the simplest cases, the so-called "uniform" (or constant) model, the same fixed number of items are used in each bid and the bid price is chosen uniformly at random from the interval $[0, 1]$ [Sandholm, 1999]. Slightly more complex models use uniform or parameterised normal, binomial, or exponential bid-length distributions in combination with various price distributions, including uniform distributions over an interval whose size is linear in the length of the respective bid, and normal distributions [Sandholm, 1999; Fujisima *et al.*, 1999; Boutilier *et al.*, 1999; Andersson *et al.*, 2000]. In all of these cases, the items in S are drawn uniformly at random from the set of all items (without replacement).

The CATS (Combinatorial Auctions Test Suite) benchmark collection contains a number of instance distributions that are modelled based on (potential) real-world applications of combinatorial auctions as well as generators for the artificial random distributions mentioned above [Leyton-Brown *et al.*, 2000c]. The CATS distributions model problems involving paths in space, such as combinatorial auctions for truck routes or network bandwidth; spatial proximity problems that corrspond to situations arising in real estate or drilling rights auctions; temporal matching problems, where corresponding time-slices must be secured on multiple resources, *e.g.*, machines; and temporal scheduling problems, which essentially correspond to distributed job-shop scheduling problems with one resource (see also Chapter **??**). These parameterised distributions are defined based on random generation mechanisms for bundles and bid prices that are based on simplified models of realistic combinatorial auctions applications.

Since their introduction, CATS benchmark instances have been used for a number of empirical studies on CAWDP algorithms. There is some evidence, however, that most instances from the "real-world" CATS distributions (particularly those from the matching and scheduling models) tend to be relatively easy compared to similarly sized instances obtained from some of the artificial random distributions, including the uniform model mentioned above; at the same time, other artificial distributions have been shown to produce mostly easy instances [Leyton-Brown *et al.*, 2000b]. Current research uses regression learning techniques to investigate features that render CAWDP instances computationally hard for state-of-the-art algorithms [Leyton-Brown *et al.*, 2000b].

The Casanova Algorithm

There are many ways in which SLS methods can be applied to the Combinatorial Auctions Winner Determination Problem. One approach is to search the space of feasible allocations in such a way that in each step an unsatisfied bid $b \notin A$ is selected and added to the current allocation A, after all satisfied bids in A that overlap with b have been removed from A. Formally, this approach uses the neighbourhood relation defined by $N(A, A') \Leftrightarrow \exists (S', p') \notin A : A' = A \cup \{(S', p'\}\backslash\{(S, p) \in A \mid S \cap S' \neq \emptyset\}$. It may be noted that this neighbourhood relation is not symmetric: in general, if $N(A, A')$ does not imply $N(A', A)$. Consequently, the search steps in SLS algorithms based on this neighbourhood relation are typically not reversible.

The class of SLS algorithms for CAWDP that are based on this neighbourhood relation is known as CASLS. Casanova is currently one of the best performing CASLS algorithms [Hoos and Boutilier, 2000]; it is conceptually very similar to Novelty$^+$, one of the best-performing SLS algorithms for SAT. Like Novelty$^+$, Casanova is based on a randomised best-improvement method with a limited form of memory; the evaluation function used in Casanova measures the sum of the revenue per item over all bids in the currently satisfied bids, *i.e.*, $g(I, B, A) = \sum \langle p/|S| \mid (S, p) \in A \rangle$; **[NOTE: $\langle x \mid P(x) \rangle$ denotes a multiset of elements x satisfying a given condition P(x); this notation will be motivated and introduced early in the book and used consistently throughout.]** the value $p/|S|$ is also called the *score* of bid (S, p).

Casanova works as follows: The search process starts from an empty allocation. Then, in each step, with probability wp (walk probability), a currently unsatisfied bid is selected uniformly at random (random walk step); with probability $1 - wp$ a bid is selected "greedily" based on the score and age of each bid, where the *age* of a bid b, $age(b)$, is defined as the number of search steps since b last became satisfied, or, if b has never been satisfied since the last search initialisation, the number of steps since the search was last initialised. In a greedy selection step, first all bids are ranked according to their score. Then, either the highest ranked bid b_1 or the second-highest b_2 is selected as follows: if $age(b_1) \geq age(b_2)$, select b_1; otherwise select b_2 with probability np (novelty probability) and b_1 with probability $1 - np$. After $maxSteps$ steps, the search is reinitialised with an empty allocation. After a total of $maxTries$ such independent tries, the search is terminated and the best allocation, in terms of total revenue, found throughout the entire search process is returned as the search result.

Empirical studies have shown that compared to CASS [Fujisima *et al.*, 1999], a state-of-the-art systematic search algorithm for CAWDP at the time when Casanova was developed, Casanova achieves superiour performance

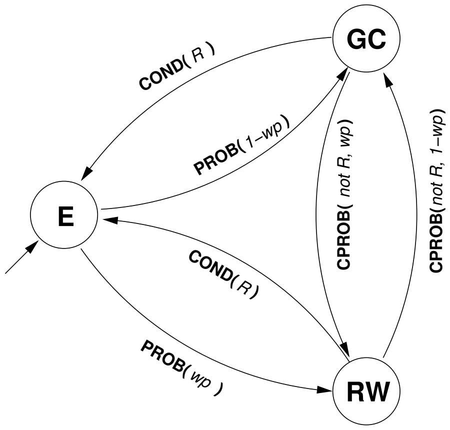

Figure 10.8: GLSM model for Casanova; the restart predicate R is equal to countm(*maxSteps*), GSLM state E initialises the search at the empty allocation, GC performs a greedy Casanova step, and RW performs a random walk step (see text for details).

on a broad range of benchmark instances, both in terms of the CPU time required for obtaining optimal solutions (without proving optimality) as well as in terms of the solution qualities obtained for fixed run-time. Casanova's performance advantage is particularly pronounced for certain types of structured randomised instances that are obtained from encoding generalised types of bids, such as so-called CNF and k-*of* bids that can directly express substitutabilities between bundles of items, into standard CAWDP instances [Hoos and Boutilier, 2000] (see also page **??**). There is also limited empirical evidence that Casanova outperforms CPLEX, a state-of-the-art commercial integer programming package, on various types of CAWDP instances from the CATS benchmark suite [Schuurmans *et al.*, 2001]. While the relative performance of Casanova compared to the latest systematic search algorithms for CAWDP (*e.g.*, Sandholm *et al.*'s CABOB algorithm [Sandholm *et al.*, 2001]) is currently unclear, it is reasonable to assume that there is considerable room for improving Casanova's performance by optimising its implementation as well as the underlying SLS technique.

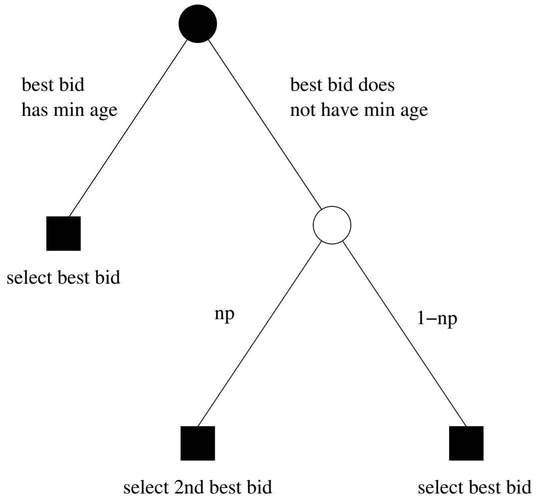

Figure 10.9: Decision tree representation of a greedy Casanova step. Deterministic and probabilistic choices are represented by black and white circles, respectively; edges are labelled with the respective conditions and probabilities. Black boxes indicate bid selection actions.

The Exponentiated Subgradient Algorithm for CAWDP

Like Casanova, the Exponentiated Subgradient Algorithm for the Combinatorial Auction Winner Determination Problem (ESG-CAWDP) is closely related to a high-performance SAT algorithm, ESG-SAT (see Chapter 6, page **??**). ESG-CAWDP is a Dynamic Local Search Algorithm that associates a penalty weight $itp(i)$ with each given item i and performs an iterative best improvement search in the space of all allocations; it uses a neighbourhood relation under which two allocations A and A' are neighbours if they differ in exactly one bid, *i.e.*, $N(A, A') :\Leftrightarrow \exists b \in B : (A' = A \cup \{b\} \text{ or } A = A' \cup \{b\})$, and a modified evaluation function of the form $g'(I, B, A) = \sum \langle p \mid (S, p) \in A \rangle - \sum \langle itp(i) \cdot ovd(i, A) \mid i \in I \rangle$, where $ovd(i, A) = \#\{(S, p) \in A \mid i \in S\} - 1$ is the "overdemand" for item i under allocation A, and $itp(i)$ is the penalty for item i, which is adjusted during the search. Any item i with $ovd(i, A) > 0$ for a given allocation A constitutes a *conflict* in A; feasible allocations are those that have no conflicts.

ESG-CAWDP works as follows. The search is initialised by selecting an allocation A uniformly at random such that for any given bid the probability that is contained in A is 1/2; furthermore, all item penalties are set to one. Then, a series of best improvement steps is performed; in each of these steps, one of the allocations A' that are direct neighbours of the current allocation A and that have a maximal evaluation function value $g'(I, B, A')$ amongst all the neighbouring assignments of A, is selected uniformly at random. When this best improvement search reaches a local maximum w.r.t. g', with probability η, the search is continued by selecting a neighbour of A uniformly at random; otherwise, the local search phase is terminated.

After each local search phase, the item penalties are updated in a two-stage process: First, each item penalty $itp(i)$ is multiplied by a factor $\alpha^{\theta(i)}$, where $\theta(i) = -1/2$ if i is not involved in a conflict in the current assignment A, and $\theta(i) = ovd(i, A) - 1/2$ otherwise (scaling stage); α is a parameter of the algorithm. Then, all item penalties are smoothed using the formula $itp(i) \leftarrow itp(i) \cdot \rho + (1 - \rho) \cdot \overline{itp}$, where $\overline{itp}$ is the average over all item penalties after scaling, and ρ is a parameter between zero and one. The algorithm terminates after a given number of iterations of local search phases and subsequent penalty updates have been peformed or a specified solution quality has been reached.

The ESG-CAWDP algorithm as described here can be easily generalised to arbitrary Boolean Linear Programming problems; an efficient implementation of this more general version, ESG-BLP, has been used for a performance evaluation on various sets of CAWDP instances [Schuurmans *et al.*, 2001]. Empirical results indicate that on a range of CAWDP instances from the CATS benchmark suite, ESG-BLP does not quite reach the performance of Casanova, in terms of the CPU time or number of search steps required for finding optimal solutions. ESG-BLP typically also falls short of the performance of CPLEX, when comparing the CPU time required for finding optimal quality solutions (without proving optimality). However, in many cases, its performance appears to be reasonably close to that of CPLEX and Casanova. Interestingly, ESG-CAWDP has been shown to outperform both Casanova and CPLEX on sets of CAWDP-encoded SAT problems, which suggests that for certain types of structured CAWDP instances, ESG-CAWDP is the best-performing algorithm currently known. It is unclear, however, whether similar performance advantages can be obtained for more practically relevant CAWDP instances.

Generalisations and Related Problems

Combinatorial Auctions can be generalised in various ways. One of the most straightforward generalisations allows multiple units of the same item to be offered and bid on (see, *e.g.*, [Leyton-Brown *et al.*, 2000a]). In principle, the corresponding *Multi-unit Combinatorial Auctions Winner Determination Problem* can be easily cast as a (single unit) CAWDP instance according to our definition, by listing each unit of each item separately (after disambiguating their representation), making CAWDP algorithms applicable. A natural and more compact representation can be obtained by associating quantities with all items, both in the set of available items, as well as for the items requested within any bid. This representation can be easily transformed into a BLP formulation very similar to the one for the single-unit case, and the resulting problem instances can in principle be solved using ESG-BLP. Generally, one would expect algorithms that work directly on the compact representation of multi-unit combinatorial auctions to solve these problems more efficiently than conventional CAWDP algorithm applied to the respective CAWDP-encoded instances; however, the extent to which this is the case is presently unclear.

Other variants and generalisations of combinatorial auction problems are obtained by considering different *bidding languages*, which allow the bidder to express various types of preferences and valuations over bundles of items. One example for such a generalised bidding language allows the bidder to submit sets of bids that are connected by an XOR-constraint; feasible allocations can then include no more than one bid from any such set [Sandholm, 1999; Nisan, 2000]. Such XOR-bids allow bidders to express valuations under which the value of a bundle is less than the sum of thhe values of its components or subsets (substitutability). Combinatorial auctions with XOR-bids can be easily and efficiently encoded into standard combinatorial auctions using so-called *dummy goods* that are included in each set of XOR-bids [Sandholm, 1999; 2002; Fujisima *et al.*, 1999].

Another way of allowing bidders to express substitutabilities and other forms of complex valuations is to consider bids in the form of propositional formulae, whose atoms corresponds to items [Hoos and Boutilier, 2000; Boutilier and Hoos, 2001]. In such *logical bidding languages*, prices can be attached to the formulae comprising the bids as well as to subformulae, and the value of a bid is determined based on the satisfaction status of its respective formula and subformulae, where each item is satisfied if and only if it is assigned to the respective bid (for details, see [Boutilier and Hoos, 2001]). Standard combinatorial auctions correspond to a case where each bid is a conjunction of items, and prices are only attached to entire bids. A slightly generalised logical bidding language allows so-called CNF bids, *i.e.*, conjunctions of disjunctions of items, which provide another mechanism for expressing certain substitutabilities [Hoos and Boutilier, 2000]. Logical bidding languages allow certain types of complex valuations to be expressed substantiallly more concisely than standard combinatorial auctions or combinatorial auctions with XOR-bids [Boutilier and Hoos, 2001].

Finally, there are various problems that are closely related to combinatorial auctions. In *combinatorial reverse auctions*, a buyer wants to obtain certain items at the lowest possible cost, and various sellers submit offers (also called *asks*) for bundles of these items; the objective is to find a cost-minimal set of offers that covers the need of the buyer [Sandholm *et al.*, 2002]. Reverse combinatorial auctions have important applications, *e.g.*, in procurement.

Combinatorial exchanges are a market mechanism that allow multiple users (who may each buy, sell, or both) to submit bids and asks for bundles of items; here, the objective is to label the bids and asks as winning and losing such that the supply does not exceed the demand, and the surplus, *i.e.*, the difference between the total value of the winning bids and the total value of the winning offers, is maximised [Sandholm *et al.*, 2002]. Combinatorial exchanges can be seen as a direct generalisation of both, combinatorial auctions and reverse auctions; hence, the same hardness and inapproximability results as for the CAWDP problem apply to the problem of determination winners in combinatorial exchanges.

10.5 DNA Code Design

DNA is one of the most important classes of biomolecules, as the genetic information of all organisms is stored in the form of long strands of DNA. At an abstract level, a DNA strand can be represented as a string over a four-letter alphabet $\{A, C, T, G\}$; hence, DNA strands can in principle encode any kind of information, which can then be processed using established laboratory techniques. This provides the basis for various approaches to biomolecular computation, nanostructure design, and molecular tagging. DNA molecules also play a crucial role in many biochemical techniques, including PCR (Polymerase Chain Reaction) and DNA Microarray technology, both of which have countless applications in the biological and biomedical sciences. Many of these applications require sets of short DNA strands that satisfy certain combinatorial constraints. The problem of designing such sets in many ways resembles well-known problems from coding theory, particularly problems in designing error-correcting codes, and is therefore also known as DNA Code Design Problem.

In this section, we first introduce and discuss the problem of designing DNA codes for various combinations of combinatorial constraints. Next, we briefly outline some prominent applications of DNA codes and describe widely used benchmark instances. Then, we introduce a recent, state-of-the-art SLS algorithm that has been successfully used for improving the best known results for various code design problems. Finally, we give a brief overview of some related code design problems.

DNA Code Design Problems

A DNA (deoxyribonucleic acid) strand is a linear biopolymer that consist of nucleotide subunits each of which is formed by a section of a sugar-phosphate backbone and a nitrogenous base. In DNA, four different nucleotides occur, labelled A, C, G, and T according to the bases they contain (Adenine, Cytosine, Guanine, and Thymine). Each DNA strand has two chemically distinct ends, called the 5'- and the 3'-end. In the context of this section, DNA strands can be represented as strings over the four-letter alphabet $\{A, C, G, T\}$, where the left end of such a string corresponds to the 5'-end and the right end to the 3'-end of the strand. We call such DNA strands *DNA words*.

Hydrogen bonds can form between A and T as well as between C and G; the base pairs A, T and C, G (as well as T, A and C, G) are called complementary. The complementarity extends to entire strands: two DNA strands are complementary if one can be obtained from the other by replacing every base by its complement and reversing the orientation of the strand. Formally, this is captured in the following definition:

Definition 10.1 (DNA words and complementarity)

A *DNA word* is a string over the four letter alphabet $\{A, C, G, T\}$. The length of a DNA word w is denoted $|w|$. The *complement* of a DNA word $w = b_1 b_2 \cdots b_{k-1} b_k$ is defined as $compl(w) = \overline{b_k}\ \overline{b_{k-1}}\ \cdots\ \overline{b_2}\ \overline{b_1}$, where $\overline{A} = T$, $\overline{T} = A$, $\overline{C} = G$, and $\overline{G} = C$. □

For example, for $w = AATCGCAC$, $|w| = 8$ and $compl(w) = GTGCGATT$. Note that the complement of a complement of a word w is always w itself, *i.e.*, $compl(compl(w)) = w$; furthermore, complementarity is symmetric, *i.e.*, $w = compl(w')$ if and only if $w' = compl(w)$.

Two complementary DNA strands can bond to each other; this process is called hybridisation and leads to the formation of the well-known double-helix structure. The complementarity between bases or entire DNA strands also underly many important biological and technological processes involving DNA, such as protein biosynthesis, DNA replication, and genetic recombination. Hybridisation can also occur between strands that are not perfect complements of each other, *i.e.*, in cases where one of the two strands differs in some positions from the complement of the other.

A DNA code is a set of (typically short) DNA words that satisfies certain constraints. The constraints reflect properties of the DNA strands that minimise the potential for undesired hybridisation interactions and maximise the efficiency of desired interactions. In many applications of DNA codes, one set of code words is used for representing information, while the set of the complements of these words is used for performing certain operations on the information-carrying strands, such as marking occurrences of a certain piece of information i by means of hybridisation between all strands containing i, and strands containing the complement of the representation of i. The following constraints have been considered in a number of DNA code design applications, particularly in DNA computing [Frutos *et al.*, 1997], where DNA words are used for representing the data being processed.

The Hamming-Distance Constraint, HD(d)

Any two words w_1, w_2 from the set have Hamming distance at least d, *i.e.*, disagree in at least d positions. This constraint intuitively ensures that any two words are sufficiently different to not be “confused” in any hybridisation-based operation on specific words.

The Complement Hamming Distance Constraint, CHD(d)
For any two words w_1, w_2 from the set (where w_2 can be identical to w_1), the Hamming distance between w_1 and $compl(w_2)$ is at least d. This constraint intuitively ensures that hybridisations between different code words or between multiple copies of a code word do not occur; such hybridisations can, for example, greatly decrease the efficiency of a DNA-based computation.

The GC Content Constraint, GCC(p)
Any word w in the set contains G or C in exactly $\lceil p \cdot |w| \rceil$ positions, where p, $0 \leq p \leq 1$, is a parameter. Since base pairings involving G and C are stronger (*i.e.*, thermodynamically more stable) than those involving A and T, fixing the GC content of all code words ensures that the desired hybridisation between any word and its complement is of approximately equal strength.

The problem of designing a DNA code for a given combination of constraints and word length can now be defined as a combinatorial optimisation problem as follows:

Definition 10.2 (The DNA Code Design Problem (DNA-CDP))

An instance of the DNA Code Design Problem (DNA-CDP) for a given set of constraints C on sets of code words is given by a word length n; the objective is to find a maximum size set S of DNA words of length n that satisfies all constraints in C. Note that by using an objective function $f(S) = -|S|$, this can be formulated as a minimisation problem. □

Note that in DNA-CDP, the candidate solutions are *sets of DNA words*, and these sets are feasible if they satisfy *all* given constraints. Many applications of DNA-CDP, particularly in DNA computing, require DNA codes of a certain size – this obviously corresponds to the decision variants associated with the DNA-CDP problem as defined here.

Example 10.7: A Simple DNA Code Design Problem

Consider the problem of designing a maximum size DNA code with word length $n = 8$ given the constraints HD(6), CHD(6), and GCC(0.5). The set S_1 comprising the DNA words

AGCTCTGT GACGTTTG TTACGCGT GGTAGGAT
TGTCATCG GCTTCCTA CCCAAAAG

is a solution to this problem instance. Note that the CHD constraint ensures that every word w from this set and its complement, $compl(w)$ have Hamming distance at least 6.

If the first word, w_1 = AGCTCTGT, is replaced by w_1' = AGCTCTGA, a DNA code is obtained that still satisfies the given GCC constraint, but there are conflicts between the words w_1' and w_6 = GCTTCCTA (the Hamming distance between those words is 5) and between w_1' and w_3 = TTACGCGT (the Hamming distance between w_1' and $compl(w_3)$ is 5).

In fact, the 7-word set given above is the best solution (*i.e.*, the largest DNA code) for this problem instance currently known. When increasing the word length to $n = 10$, codes of size up to 41 are known. When relaxing the HD(d) and CHD(d) constraints to $d = 4$, solutions with up to 112 words can be obtained. Similarly, for word length $n = 8$ when only using the constraints HD(6) and CHD(6), DNA codes of size 10 are known.

Although code design problems that are related to DNA code design have been studied extensively in coding theory, not much is known regarding the theoretical complexity of constructing or approximating maximum size codes for the combinatorial constraints considered here. In our formulation, the space of candidate solutions for a DNA Code Design problem is doubly exponential in the given word length, n. While codes up to certain sizes can be obtained efficiently using construction methods such as the Gilbert-Varshamov algorithm [**?**], empirical evidence indicates that finding optimal or close to optimal solutions to DNA code design problems requires at least exponential time in the length of the code words [Tulpan *et al.*, 2002].

Applications and Benchmark Instances

Currently, there are three main application areas in which DNA Code Design problems arise: DNA computing, DNA nanostructure design, and DNA tagging in chemical libraries.

In DNA computing, DNA words are used for encoding information in the form of DNA strands. There are two main approaches for using DNA-based computation for solving suitably encoded combinatorial problems, such as the Hamiltonian Path Problem or SAT: In the solution-based approach, most of the crucial steps of a computation happen in solution, *i.e.*, in a situation where all DNA strands can move relatively freely in a liquid. In the surface-based approach, some of the information-carrying DNA strands are fixed to a surface and crucial computation steps involve exposing this surface and the affixed DNA strands to solutions containing other DNA strands or reactive agents, in particular specific enzymes that replicate or degrade DNA strands under certain conditions. In both approaches, established biomolecular methods such as sequence-specific enzymatic digestion (*i.e.*, strand cleavage), PCR (for strand replication), gel electrophoresis (for length-specific separation of DNA strands), and DNA microarray assays, are used to perform the steps of these computations.

In DNA nanostructure design, DNA codes are used for creating DNA molecules that assemble into larger structures with well–defined properties, such as 2d-crystals, cubes, cages, and simple nanomechanical devices. Such DNA nanostructures can be constructed from building blocks consisting of multibranched complexes formed by partial hybridisation between several strands of DNA; these building blocks attach to each other via “sticky ends”, *i.e.*, free ends of single stranded DNA. In this context, DNA codes are used for designing DNA strands that will assemble correctly and efficiently into the multibranched complexes required for building larger structures with specific properties, such as a specific 2d-crystal [Seeman, 1990].

Specific DNA nanostructures can be used for performing self-assembly computations. The underlying idea is to create “DNA tiles”, *i.e.*, multi-strand DNA complexes with sticky ends, for encoding data as well as computation rules. The tiles are designed in such a way that specific tiles can assemble into larger structures via hybridisation between their respective sticky ends. It is known that the self-assembly of such tiles into larger structures can simulate the computation of a Turing machine and hence perform arbitrary universal computations. Similar as in nanostructure design, DNA codes are used for designing tiles that combine in such a way that the correctness of the self-assembly computation is ensured [Winfree *et al.*, 1998]

Sequence specific hybridisations between complementary strands of DNA can also be used in the context of DNA tagging, a technique that uses DNA strands as "molecular barcodes" for identifying and accessing specific elements from chemical libraries that can encompass thousands of distinct chemical compounds, such as protein sequences. Such encoded combinatorial chemical libraries can be used, *e.g.*, for identifying proteins that show specific interactions with a target molecule – a key step in many approaches to drug design. Combinatorial chemistry methods can be used to create resin beads to which multiple copies of a candidate protein as well as of a unique DNA tag identifying that protein are bonded. In this context, DNA codes are used for designing DNA tags that can be effectively detected and isolated based on correct hybridisation with their respective complementary strands [Brenner and Lerner, 1992].

Finally, DNA codes can be applied in the design of DNA tags used in universal DNA microarrays. Here, DNA tags are used in specifically designed "adaptors" that allow the reliable and massively parallel detection of arbitrary genomic sequences using the same universal DNA microarray. Designing and synthesising these adaptors is substantially easier and cheaper than creating a customised DNA microarray, and through the use of carefully designed DNA codes, the universal microarray system can be designed in such a way that the potential for errors due to mishybridisations is typically much smaller than when using customly designed DNA microarrays [Gerry *et al.*, 1999].

So far, the sets of DNA words used for these "real-world" applications have been mostly constructed manually using ad-hoc methods that in some cases are based on results from coding theory.

The SLS-THC Algorithm

The recent DNA Word Design algorithm by Tulpan, Hoos, and Condon is based on a randomised iterative improvement search method similar to WalkSAT, one of the best known algorithms for the propositional satisfiability problem [Selman *et al.*, 1994b; McAllester *et al.*, 1997] (see Chapter 6, page **??**) and Casanova, one of the best-performing algorithms for the winner determination problem in combinatorial auctions [Hoos and Boutilier, 2000] (see Section 10.4, page **??**). The search space used in this approach consists of the DNA word sets with the target number of words. Two candidate solutions, *i.e.*, DNA word sets, are direct neighbours, if and only if one can be obtained from the other by changing exactly one position in one word – this is called the *1-mutation neighbourhood*.

The SLS-THC algorithm is based on an evaluation function g that counts the number of word pairs in a given candidate solution S that violate the given binary constraints; here, this can be any combination of the HD and CHD constraints (a variant of the algorithm that allows supports the GCC constraint will be discussed later). The main idea behind the algorithm is to iteratively pick a pair of words that are in conflict w.r.t. a given constraint and to modify one of them such that the number of conflicts (*i.e.*, the number of word pairs that violate at least one constraint) is maximally reduced.

An outline of the SLS-THC algorithm is shown in Figure 10.10. The search process is initialised at a set of words that is determined by a simple randomised process that generates any DNA word of length n with equal probability. Note that the initial word set may contain multiple copies of the same word.

In each step of the search process (*i.e.*, one execution of the inner for-loop from Figure 10.10), first, a pair of words violating one of the Hamming distance constraints is selected uniformly at random. Then, for each of these words, all possible single-base modifications are considered. As an example of single-base modifications take the code word $ACTT$ of length 4. A new code word $GCTT$ can be obtained by replacing letter A from the first code word with letter G. For a pair of words of length n this yields $6n$ new words, some of which might be identical.

With a fixed probability θ, one of these modifications is accepted uniformly at random, regardless of the number of constraint violations that will result from it. In the remaining cases, a modification that leads to a maximal decrease in the number of constraint violations is accepted. (If there are multiple such modifications, one of them is chosen uniformly at random.) Note that using this scheme, in each step of the algorithm, exactly one base in one word is modified. When counting the number of constraint violations, the degree of violation is not considered, *i.e.*, when considering, *e.g.*, the HDD(4) constraint, one violation is counted for any two words with Hamming distance $k < 4$, regardless of their actual Hamming distance k is

```
procedure SLS-THC
    input: k ∈ ℕ: number of code words, n ∈ ℕ: word length, C: set of constraints
            maxTries ∈ ℕ, maxSteps ∈ ℕ, θ ∈ [0,1]
    output: Set S of m words that fully or partially satisfies C
    for i := 1 to maxTries do
        S := initial set of words
        Ŝ := S
        for j := 1 to maxSteps do
            if g(S) = 0 then
                return S
            end if
            Randomly select words w_1, w_2 ∈ S that violate one of the
            Hamming distance constraints
            M_1 := all words obtained from w_1 by substituting one base
            M_2 := all words obtained from w_2 by substituting one base
            with probability θ do
                select word w' from M_1 ∪ M_2 uniformly at random
            otherwise
                select word w' from M_1 ∪ M_2 such that the number of constraint
                violations is maximally decreased
            end with probability
            if w' ∈ M_1 then
                replace w_1 by w' in S
            else
                replace w_2 by w' in S
            end if
            if g(S) < g(Ŝ) then
                Ŝ := S;
            end if
        end for
    end for
    return Ŝ
end SLS-THC
```

Figure 10.10: Outline of the SLS-THC algorithm for DNA word design; $g(S)$ denotes the number of constraint violations in word set S (see text for details).

3 or 0. The noise parameter θ controls the greediness of the search process: for high values of θ, constraint violations are not resolved efficiently, while for low values of θ, the search has more difficulties to escape from local optima of the underlying search space.

Throughout the run of the algorithm, the best candidate solution encountered so far, *i.e.*, the DNA word set with the fewest constraint violations, is memorised. Note that even if the algorithm terminates without finding a valid set of size k, a valid subset can always be obtained by iteratively selecting pairs of words that violate a Hamming distance constraint and removing one of the two words involved in that conflict from the set. Hence, a word set of size k with t constraint violations can always be reduced to a valid set of at least size $k - t$.

To obtain good performance it is crucial to implement this algorithm in such a way that Hamming distances between words and/or their reverse complements are not recomputed in each iteration of the algorithm; instead, these are computed once after generating the initial set, and updated after each search step. This can be done very efficiently, since any modification of a single word can only affect the Hamming distances between this word and the $k - 1$ remaining words in the set.

This basic SLS algorithm can be easily extended to also accomodate the GCC constraint, by restricting the candidate solutions to sets of words that all satisfy a given GCC constraint [Tulpan *et al.*, 2002]. Furthermore, a variant of the SLS-THC algorithm that initialises some of the DNA words to elements of a previously determined word set, while the remaining words are generated randomly as described above, can be used for extending known DNA codes.

Empirical analyses have shown that run-time distributions that characterise the run-time behaviour of SLS-THC on hard word design problems often suffer from search stagnation that severely compromises its performance. This can be overcome by extending the algorithm with a mechanism for diversifying the search by occasional random replacement of a small fraction of the current set of DNA words [Tulpan *et al.*, 2002]. It has been empirically demonstrated that SLS-THC in many cases reaches or exceeds the best known construction with theoretical guarantees; particularly impressive results have been reported for word lengths ranging from 4 to 12 under combinations of HD and CHD constraints, where for 56 out of 57 problem instances previously best theoretical results could be empirically matched or improved [Tulpan *et al.*, 2002; Marathe *et al.*, 2001] and practically relevant problem instances with combinations of HD, CHD, and GCC constraints [Tulpan *et al.*, 2002; Frutos *et al.*, 1997]. Recent empirical results indicate that compared to current implementations of the Gilbert-Varshamov algorithm, SLS-THC consistently finds larger word sets when run sufficiently long [**?**].

There is some indication that by using different neighbourhood relations that allow more than one base to be modified in each search step, the performance of the SLS-THC algorithm can be further improved, not only in terms of the number of search steps, but also in terms of the CPU time required for solving given instances of the DNA-CDP problem [**?**]. Furthermore, a hybrid SLS algorithm consisting of a GV construction phase followed by a SLS-THC local search phase that starts from the word set obtained from the construction phase appears to perform better than SLS-THC with random initialisation [**?**].

Generalisations and Related Problems

There are many variants of the DNA Code Design Problem that use additional or alternative constraints to the HD, CHD, and GCC constraints discussed here. As an example, consider constraints on Hamming distance between misaligned strands and constraints on overlapped pairings involving three or more strands (see Figure 10.11); both types of constraints play an important role in applications where information is encoded into DNA by concatenating individual code words, similar to the character-by-character encoding of text strings into bit vectors. The SLS algorithm presented earlier in this section can be extended to accomodate these additional constraints.

As previously motivated, the Hamming distance constraints considered here are designed to optimise desired hybridisation pairings between code words while reducing the occurrence of erroneous pairings as much as possible. In reality, however, the number of complementary base pairs between two strands is only a very crude measure for their probability of bonding and the strength of the pairing. It is therefore often useful to consider more accurate models of DNA hybridisation that, for instance, reflect the different strength of the bonds underlying GC and AT pairings, or the distribution of mismatched base pairs over an imperfectly hybridised pair of DNA strands

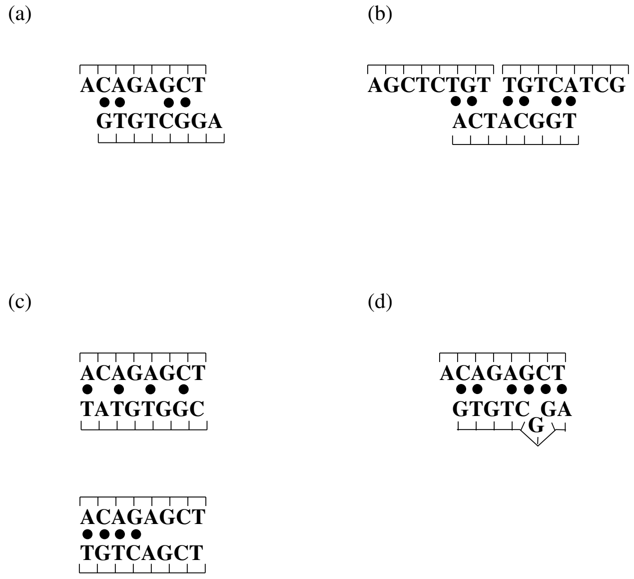

Figure 10.11: Illustration of strand interactions that give rise to additional constraints for DNA code design: (a) Slide misalignment leads to 5 complementary base pairings compared to only 2 pairs for the same two strands under perfect alignment; (b) overlap between three strands can lead to additional correct pairings; (c) a given number of base pairs can be distribute differently, leading stability of the duplex; (c) DNA secondary structure formation (here a single buldge loop) can increase thermodynamic stability. (The black circles indicate pairings between complementary bases.)

(see Figure 10.11). Based on thermodynamic models of DNA hybridisation, Gibbs free energy values can be calculated for pairs of DNA strands, which can then be used for predicting the temperature at which a given hybridised pair of strands tends to separate into single strands under specific reaction conditions (melting temperature of the duplex). Thermodynamic constraints, *i.e.*, constraints on the free energies or melting temperatures of pairs of DNA strands, can be used for designing DNA codes with more accurately controlled physical properties. While results from coding theory can be exploited to some extent for DNA code design using Hamming distance constraints, the use of thermodynamic constraints requires differenr approaches. Interestingly, the SLS algorithm presented above can be extended to deal with thermodynamic constraints in a rather natural way.

Additional complications in the design and application of DNA codes arise from the fact that DNA strands can pair in ways that do not correspond to a double helix structure where each base in one strand as juxtaposed to exactly one base in the other strand, to which it may or may not be complementary. Instead, strands may pair in ways that give rise to asymmetric bulges or loop structures (see Figure 10.11). Furthermore, under certain conditions, a single strand can partially hybridise with itself. In general, the structures that arise from such types of base pairings are known as DNA secondary structures. (Primary structure refers to the base sequence of a DNA strand, and tertiary structure to its three-dimensional conformation.) Constraints that reflect certain aspects of DNA secondary structure can be used in the design of DNA codes for applications where the formation of secondary structure, *e.g.*, of DNA strands corresponding to individual code words or concatenations of multiple code words, interferes with desired hybridisation pairings. Such constraints are typically based on thermodynamic models of DNA secondary structure.

Finally, RNA (ribonucleic acid) molecules share many of the salient properties of DNA and play a similarly crucial role in many biological processes. Most of the considerations and techniques for the design of DNA codes discussed in this section apply analogously to closely related RNA code design.

10.6 Further Readings and Related Work

The Graph Colouring Problem has been widely studied and a large number of heuristic GCP algorithms have been proposed in the literature. Constructive methods include (i) greedy constructive methods, where, given some predefined order of the vertices, these are assigned colours according to some heuristic, (ii) the DSATUR algorithm [Brélaz, 1979], which at each step chooses the next vertex to colour as one that is adjacent to the largest number of distinctly coloured vertices (which is called the saturation degree of a vertex, hence, the name of the algorithm) breaking ties by the maximal degree of a vertex in the uncoloured subgraph, and (iii) the Recursive Largest First (RLF) algorithm [Leighton, 1979], which builds colour classes successively according to some heuristic searching for large independent sets that leave the number of edges in the uncoloured subgraph minimal. Good results are reported by a randomised extension of RLF, XRLF, that uses limited enumeration once a subgraph becomes small enough [Johnson *et al.*, 1991].

A wide range of SLS methods have been applied to the GCP. These include several Tabu Search algorithms [**?**; Fleurent and Ferland, 1996a; Dorne and Hao, 1999; Galinier and Hao, 1999], the most effective being a straightforward extension of the successful TSGH algorithm by Hao and Galinier for binary CSPs to graph colouring (see Section 6.6), GRASP [Laguna and Martí, 2001]; Iterated Local Search [Chiarandini and Stützle, 2002]; distributed local search [Morgenstern, 1996]; various pure and hybrid Evolutionary Algorithms [Fleurent and Ferland, 1996a; Davis, 1991; Eiben *et al.*, 1998; **?**]; Iterated Greedy algorithms [Culberson and Luo, 1996]; and ACO algorithms [**?**; Costa and Hertz, 1997; **?**]. Although the performance of these and other SLS algorithms for the GCP strongly varies across different classes of graphs, one common trend appears to be that hybrid algorithms such as the one presented in Section 10.1 are among the top performers [Galinier and Hao, 1999]. Search space analysis methods yielded some insights into the factors underlying the performance of SLS algorithms for the GCP [Hamiez and Hao, 2001]; however, further significant research efforts appear to be necessary to gain a better understanding of the behaviour of these algorithms.

The Quadratic Assignment Problem has been the subject of a large number of studies and continues to be of significant interest to many researchers. For general overviews with a special emphasis on more mathematical background, we refer to two overview papers [Pardalos *et al.*, 1994; Burkard *et al.*, 1998] and the book by Çela [Çela, 1998]; these also provide more details on extensions of the QAP, special cases, and asymptotic results on the behaviour of specific classes of QAP instances.

The literature on SLS algorithms for the QAP is vast, primarily because the QAP is extremely difficult to solve for complete algorithms, and large instances can only be solved by SLS methods and because, similar to the TSP, the QAP plays a central role as a benchmark problem for computational approaches to $\mathcal{NP}$-hard optimisation problems. While pure construction heuristics do not play a very important role for the QAP, most of the research on SLS algorithms is concentrated on "simple" SLS algorithms and hybrid SLS techniques. These include applications of Simulated Annealing [Burkard and Rendl, 1984; Connolly, 1990], Threshold Accepting [Nissen and Paul, 1995], Tabu Search [Battiti and Tecchiolli, 1994; Skorin-Kapov, 1990; Taillard, 1991; Misevicius, 2002], Memetic Algorithms [Fleurent and Ferland, 1994; Merz and Freisleben, 2000a; Vazquez and Whitley, 2000], Evolution Strategies [Nissen, 1994], GRASP [Li *et al.*, 1994], ACO Algorithms [Gambardella *et al.*, 1999; Maniezzo, 1999; Maniezzo *et al.*, 1994b; Stützle, 1997; Stützle and Hoos, 2000], and Scatter Search [Cung *et al.*, 1997]. As in the case of the GCP, the performance of these SLS algorithms depend strongly on characteristics of the given problem instance. While for unstructured, randomly generated instances algorithms based on Tabu Search are the currently best performing ones, state-of-the-art performance on structured, real-life QAP instances and large randomly generated real-life like instances is achieved by hybrid SLS algorithms like Memetic Algorithms or Ant Algorithms. Because the Set Covering Problem is of enormous practical relevance, a large number of solution approaches, including complete and incomplete algorithms, have been proposed for this problem. Complete algorithms can solve instances with up to a few hundred rows and few thousand columns; a comparison of exact algorithms can be found in [Caprara *et al.*, 1998]. It is worth noting that among the complete SCP algorithms, general-purpose integer programming software like CPLEX performs extremely well, outperforming several other complete algorithms that were specifically developed for the SCP. However, complete algorithms are restricted to rather small sized SCP instances and SLS algorithms are important for generating good solutions to large instances.

Many SLS algorithms for the SCP are based on greedy construction heuristics. Currently, several of the best performing heuristic algorithms are based on Lagrangian relaxation with subgradient optimisation. The essential idea is to compute surrogate costs (Lagrangian costs) that give an indication of the utility of selecting a specific column. Several such heuristics were proposed [Balas and Ho, 1980; Beasley, 1990; Balas and Carrera, 1996; Ceria *et al.*, 1998; Caprara *et al.*, 1999]. In recent years, an increasing number of general-purpose SLS techniques has been applied to the SCP. These include Genetic Algorithms [Beasley and Chu, 1996; Eremeev, 1999] and especially SLS algorithms that are based on the principles of the Iterated Greedy Heuristic [Brusco *et al.*, 1999; Jacobs and Brusco, 1995;

Marchiori and Steenbeek, 2000a; M. Yagiura and Ibaraki, 2001]. In general, the best performance appears to be reached by the algorithm of Marchiori and Steenbeck [Marchiori and Steenbeek, 2000a] (presented earlier in this chapter), and the SLS algorithm by Yagiura, Kishida and Ibaraki that exploits a novel 3-exchange neighbourhood for the SCP [M. Yagiura and Ibaraki, 2001]. These latter two algorithms and the CFT heuristic by Caprara, Fischetti and Toth [Caprara *et al.*, 1999] are currently the best performing SLS algorithms for the SCP.

Although the Combinatorial Auctions Winner Determination Problem has not received as much attention as the GCP, QAP, or SCP, there is a substantial (and steadily growing) body of literature on the CAWDP and related problems. A recent article by Sandholm [Sandholm, 2002] provides a good and detailed introduction to combinatorial auctions and the winner determination problem, complexity results, and various earlier approaches for solving the CAWDP. Recent high-performance systematic search algorithms include CASS [Fujisima *et al.*, 1999] and its multi-unit generalisation CAMUS [Leyton-Brown *et al.*, 2000a], as well as CABOB **??**. Currently, the two algorithms described in this chapter are the only SLS algorithms for CAWDP. However, CAWDP is equivalent to the Maximum Weighted Set Packing Problem, a well-studied problem that is conceptually closely related to the SCP, and winner determination in multi-unit combinatorial auctions is equivalent to the multi-dimensional knapsack problem; for both problems, various SLS algorithms and construction heuristics have been proposed and studied (see, *e.g.*, [DeVries and Vohra, to appear; Holte, 2001; Arkin and Hassin, 1997; Chandra and Halldórsson, 2000], but there is currently no indication that these algorithms reach or exceed the performance of Casanova or ESG-CAWDP. Finally, for an overview of generalisations of the CAWDP as well as related problems we refer the interested reader to a recent article by Sandholm *et al.* [Sandholm *et al.*, 2002].

The design of DNA codes subject to various combinatorial constraints is a relatively young area subject of study, yet already a number of researchers have worked on this problem. A recent paper by Brennemann and Condon [Brenneman and Condon, to appear] provides a good introduction to DNA code design problems and their applications. Besides the SLS-THC algorithm described here, several Evolutionary Algorithms have been used for solving DNA-CDP instances with various constraints in the context of particular applications in DNA computing [Deaton *et al.*, 1996; 1999;

Zhang and Shin, 1998]; however, these algorithms are not described in sufficient detail to assess their performance compared to other DNA-CDP algorithms. Generally, DNA code design is strongly related to the problem of designing other types of codes, particularly constant weighted binary codes, which has been studied extensively in coding theory. A good introduction to code design problems and local search algorithms for these problems can be found in a book chapter by Honkala and Östergård [Honkala and Östergård, 1997]. In fact, various SLS methods, including Simulated Annealing and Evolutionary Algorithms, have been successfully applied to binary code design problems [Gamal *et al.*, 1987; Comellas and Roca, 1993].

10.7 Summary

In this chapter we gave an overview of SLS applications to five combinatorial problems. While these problems may not be the most representative for the types of problems that have been solved successfully using SLS algorithms, they illustrate a range of computational problems for which SLS methods achieve state-of-the-art performance. We introduced the Graph Colouring Problem, the Quadratic Assignment Problem, the Set Covering Problem, the Combinatorial Auctions Winner Determination Problem, and the DNA Code Design Problem; we briefly discussed their respective computational complexity, applications, and commonly used benchmark instances; we presented selected SLS algorithms for each of the problems; and we gave a brief overview of generalisations and related problems.

The Graph Colouring Problem (GCP) is a well-studied combinatorial problem in graph theory that involves assigning colours to the vertices of a given graph such that there is no edge between any two vertices of the same colour. This problem can be seen a special case of the Constraint Satisfaction Problem (see Chapter **??**), but is typically solved with specialised algorithms. We first presented three variants a Simulated Annealing algorithms that uses a standard geometric cooling schedule. They mainly differ in the evaluation function used and the neighbourhood definition. The second algorithm we described is the Hybrid Evolutionary Algorithm (HEA) of Galinier and Hao that combines a genetic algorithm using a partition-based crossover with an effective tabu search algorithm to improve solutions. HEA is currently one of the best performing SLS algorithms for the GCP.

In the Quadratic Assignment Problem (QAP), the objective is to assign a number of objects to locations such that the product of the distances between the locations and the flow between the corresponding objects is minimised. The QAP serves as an abstract model for a number of practical layout and location problems; as such, it is one of the most widely studied combinatorial optimisation problems. We presented a Reactive Tabu Search algorithm that is particularly successful for randomly generated, unstructured QAP instances. The central goal of the reaction mechanism used in that algorithm is adapt the tabu list length, a critical parameter in tabu search algorithm, at computation time. The second algorithm we discussed is a population-based extension of a simple iterated local search (PBILS) that tries to achieve a balance between exploration and exploitation by enforcing a minimum distance among pairs of candidate solutions in the population which is varied at computation time and the use of restart operators. PBILS was shown to perform particularly well on real-world and large, randomly generated real-world like instances.

The Minimum Weighted Set Covering Problem (SCP) is another hard combinatorial optimisation problem with many real-world applications. Here, given a set A the objective is to select a number of subsets from a given family of sets such that every element in A is covered by that selection and that the sum of the weights associated with the selected subsets is minimal. For the SCP we described two Iterated Greedy (IG) algorithms. The first, IG-JB, is a rather straightforward adaptation of the IG principles to the SCP, which achieved good results at the time when the algorithm was first presented; however, nowadays it is outperformed by a number of other algorithms among which we have the IG algorithm of Marchiori and Steenbeck (IG-MS), which achieves state-of-the-art performance. The most notable features of IG-MS are that it uses a more complex construction phase, where also occasional removals of solution components are considered, it adds an iterative improvement algorithm to improve solutions returned from the constructive phase, and it iteratively modifies the core problem that defines the sub-set of solution components that is considered for constructing solutions.

The Combinatorial Auctions Winner Determination Problem (CAWDP) is a very application-relevant problem which recently has received a steadily increasing amount of attention in the AI and OR communities. This problem models an auction in which bids are submitted for bundles of items, and the objective is to determine a feasible, *i.e.*, non-overlapping set of winning bids such that the revenue of the auctioneer is maximised. We presented the first SLS algorithm devised for this problem, Casanova. This algorithm, whose performance is still considered state-of-the-art, is based on a relatively simple randomised best-improvement method with a limited form of memory similar to the Novelty$^+$ algorithm for SAT (see Chapter **??**). We also covered ESG-CAWDP, another recent SLS algorithm for the CAWDP based on Dynamic Local Search. Like Casanova, CAWDP was inspired by and bears close resemblance to a high-performance SLS algorithm for SAT, ESG-SAT. Overall, very little research has been done on SLS algorithms for CAWDP, which suggests that compared to GCP, QAP, or SCP, there is considerable more potential for the development of improved algorithms.

The final problem, the DNA Code Design Problem (DNA-CDP), arises from applications in biomolecular computing and biotechnology. In this hard optimisation problem, the goal is to find maximum size sets of DNA sequences (words) that satisfy given combinatorial constraints, *e.g.*, on the Hamming distance between pairs of words. Although, different from all the other problems covered in this chapter, DNA-CDP is not proven to be $\mathcal{NP}$-hard, it appears to be very hard to solve in practice. We described SLS-THC, a recent SLS algorithm for this problem that was successfully used to obtain improvements over the best known solutions to a number of prominent DNA Code Design Problems. Similar to the Casanova algorithm for the Combinatorial Auctions Winner Determination Problem, SLS-THC is a relatively simple randomised best-improvement method motivated by well known SLS algorithms such as the WalkSAT algorithm family for SAT and the Min Conflicts Heuristic for CSP.

10.8 Exercises

Exercise 10.1 (Medium) Prove that in the Penalty Function SA Algorithm for the Graph Colouring Problem, described in Section 10.1, all local minima correspond to legal colourings of the given graph G.

Exercise 10.2 (Medium) Sometimes, benchmark instances for several of the problems treated here are obtained by encodings of other problems.

Common examples include

- Encodings of the Travelling Salesman Problem and the Graph Partitioning Problem as Quadratic Assignment Problems.
- Encodings of SAT as Combinatorial Auctions Winner Determination Problem
- ...

Give for each of these examples the corresponding encoding. Do you expect any advantages from using these encodings?

Exercise 10.3 (Hard) Derive the formula for the Δ-evaluation in the QAP for an instance, where both matrices can be asymmetric and then simplify this formula for the symmetric case with a zero diagonal. What is the computational complexity of the Δ-evaluation?

Now, let ψ' be the solution which is obtained by exchanging objects r and s in solution ψ. Show that for exchanging objects u and v, with $(\{u, v\} \cap \{r, s\} = \emptyset)$ the move can be evaluated in constant time.

Exercise 10.4 (easy) Consider Example 10.4. Formulate and solve it as a Set Partitioning Problem and as a Set Packing Problem.

Exercise 10.5 (Easy) Compare the Casanova algorithm for the Combinatorial Auctions Winner Determination Problem (CAWD) with the Novelty$^+$ algorithm for SAT. Point out common aspects and differences of the underlying search techniques and relate these to the common features and differences of the CAWD and SAT.

Exercise 10.6 (Medium) Specify an extension of the SLS-THC algorithm for DNA Code Design that finds DNA codes satisfying the HD(c) constraint and an additional SHD(c') constraint that ensures a minimal Hamming distance of c' between any two DNA words under any possible slide misalignment (see Figure **??**). Discuss implementation and algorithmic efficiency issues of your extended algorithm.

Exercise 10.7 (Implementation) Implement a simple Iterated Local Search algorithm for the QAP along the lines of the one underlying the population-based ILS presented in Section 10.2. Only accept better or at least equally good solutions in the acceptance criterion. Study the run-time behaviour of such an ILS algorithm, especially taking into account a possible stagnation behaviour, along the lines discussed in Section 4.4 of Chapter 4.

Does your run-time analysis give an indication, why a population-based ILS extension may be a viable way to improve performance?

Study also other possibilities of improving performance by (i) allowing larger perturbations or (ii) accepting also worse solutions in the acceptance criterion. Which of the two possibilities would you prefer?