UNIVERSITÀ DEGLI STUDI DI TRENTO

DOTTORATO DI RICERCA IN MATEMATICA

XI CICLO

Mauro Brunato

CHANNEL ASSIGNMENT ALGORITHMS IN CELLULAR NETWORKS

Relatore
Prof. Alan A. Bertossi

Contents

Acknowledgments

To my wife Anna and to our baby,
whose hidden features
no local search heuristic
could disclose yet.

I would like to thank my advisor prof. Alan Bertossi, for his precious advice and his support to my work. Many thanks to Dr. Roberto Battiti for his continuous help: without his expertise in testing my ideas, and his willingness to share his own, none of the results shown in this work would have ever been achieved.

I would also like to thank my colleagues, friends and room mates Gianluca and Davide, Marco (il *Gentile Scienziato*), my wife Anna and most of the people cited in the acknowledgment section of Gianluca's thesis [Occ99].

Last, out of all people who made personal telecommunications affordable let me mention the great actress Hedy Lamarr (whose death, on january 18, 2000, is mourned by every fan of old Hollywood movies), who patented the first frequency hopping device in 1942.

Chapter 1

Introduction

1.1 Cellular Networks

Among the several hobbies people practice in our era, the most widely spread is the game of foreseeing the future. Science fiction writers, scientists, press writers, futurologists, wizards and "common people" have written so many different predictions about the future that most of our technological reaches have been predicted by some of them, sometimes with remarkable detail: portable computers, space vehicles, magnetic levitation trains, anti-adherent cooking tools...

What about communications? Well, even in this field most technological advances have been widely predicted, from color TV to fiber-optics and laser communications. But if you asked anyone living in the times of Jules Verne or even in the fifties about personal radio communications and the chance that in the last quarter of the century people would have been able to communicate throughout the world by means of portable radio devices smaller than a small packet of cigarettes... they all would have stated the absolute impossibility of the fact.

Indeed, point to point communications, where a radio equipment communicates with another radio equipment placed somewhere else in the world have strong limitations:

- In order to communicate through far distances, they must develop a very strong signal, which in turn leads to some uneasy consequences:
  - a strong electromagnetic signal requires a powerful power supply: rechargeable electric accumulators have shown remarkable progress from the times of Volta to nowadays, but they are still many orders of magnitude too weak to allow long-distance durable communications;
  - they also require a conveniently sized antenna...
  - ...not to mention the large amount of EM power crossing the brain and the body of the user;
- if many people are to use the communication equipment at the same time, they must use different communications channels. Of course, FM bands can be reduced, they can be multiplexed in time and using some orthogonal codes, but to travel to long distances the signal must not have too high a frequency, otherwise it would be stopped by any obstacle, and this limits the amount of usable bands.

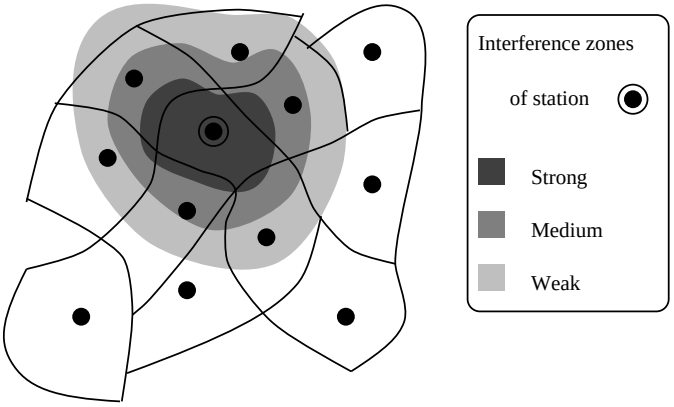

Figure 1.1: A simple geographic cellular network and the interference area of a server

1.1.1 What does make them feasible?

To circumvent these limitations, widespread communications systems are based on the *cellular* concept. All portable equipment have a limited power; portable phones do not communicate directly, but there is a distributed system of base radio stations to support communications.

Consider a geographic area in which a number of multi-channel transceiver server stations are placed at a suitable mutual distance. Stations are connected to each other by means of a wired network, which we are not concerned about in the present work. Each station acts as a network entry point for all the mobile radio hosts that can communicate with it at the highest signal-to-noise ratio among all stations. As a result, the geographic area is divided into *cells* whose borders are not well defined, as they depend on the ever-changing radio signal levels, but could be roughly sketched as in Fig. 1.1, which still depicts a *good* situation where every cell is a simply connected domain: reflections, interference and terrain configurations, as well as handoff policies, might substantially alter it. However, a more regular setting such as the common hexagonal pattern will be considered in experiments, since it is still the most common test ground.

Modeling real-world instances is in fact a very hard and delicate work, and the situation is made worse by the policies of many cellular service providers who keep their data reserved, or provide them in a strictly confidential way, so that the scientific community cannot share them.

1.1.2 Communication Channels

We often see the Channel Assignment problem referred to as *Frequency* Assignment. While the interference issues arise in particular from the use of similar frequencies, this is not always the case. In fact, given a certain frequency bandwidth inside which communications equipment are required to operate, the communications resource can be shared in at least three ways.

- The communication bandwidth is divided in small frequency bands, and each equipment operating in a certain area is required to employ a different band. This is called *Frequency Division Multiplexing*, and is widely used in common

radio broadcasts, where every broadcast station uses a particular frequency, upon which the radio receiver must be tuned in order to receive the transmission.

- Use of the communication bandwidth is granted to a piece of equipment only at given times: every transmitter uses the whole band, but the use of it must be restricted only at periodic time interval. Usually, equipment requiring to communicate are placed in a round-robin waiting queue. This *Time Division Multiplexing* is often employed by CB amateurs who establish a discussion group on a single channel, where everybody in turn either speaks or passes to the next person. A non-radio application of this technique is the IBM Token Ring wire network.

- Every communications equipment uses all of the bandwidth for all the time. Transmission is coded into different (orthogonal) spectrum shapes, so that different transmissions can be told from each other by a spectrum analysis. This type of encoding is called *Spread Spectrum* encoding, and it is mostly used by military, due to its resilience to external interference (usually carried on a narrow band) and to interception (the receiver must know the spectrum code).

Cellular communications systems mostly employ Frequency Division Multiplexing; transmissions are usually FM-encoded, so they need a $25 \div 30$kHz channel bandwidth. The GSM system employs a mixture of time and frequency division: the whole frequency band is divided into 200kHz slices, and each mobile equipment can employ one frequency band for one eighth of the time; since larger bandwidths imply faster data transmission, the overall data speed of a connection over each time shared channel is similar to the speed of a 25kHz exclusive channel.

Today, radio LANs in the $2 \div 4$GHz band employ spread-spectrum technology.

1.1.3 The Channel Assignment Problem

Usually a server station can be received by server stations in other cells. In this case, mutually interfering stations must employ different communication channels (i. e. frequency bands, time slices or codes from an orthogonal set), in order to avoid *co-channel interference* (interference caused by transmissions on the *same* channel). In its simplest (and most unrealistic) form, the channel assignment problem is equivalent to the Euclidean graph coloring problem, hence it is NP-hard. This problem can be treated with simple greedy heuristics [ES84] [BB95] [BBB99]. Because a server station must communicate with several mobile hosts at once, however, we must assign more than one channel to each server. When this problem is handled with graph-coloring heuristics, we need to substitute every node with a clique of cardinality equal to the required number of channels, in order to give the appropriate number of channels to each transceiver while respecting the interference constraints. This approach causes, however, the quadratic explosion of the problem.

Moreover, radio interference is *additive*, and simple adjacency restrictions (like those in the graph or list coloring problems) are not sufficient to catch the complexity of the real-world issues. If the interference phenomena are strong enough, even stations that use different channels may interfere, provided that they operate on adjacent frequency bands or on subsequent time slices (propagation delays may cause a time slice to partially invade another one). This problem becomes significant when the overall frequency spectrum has to be minimized; indeed, the strong request of radio bands for several purposes makes the reserved bandwidth for cellular communications

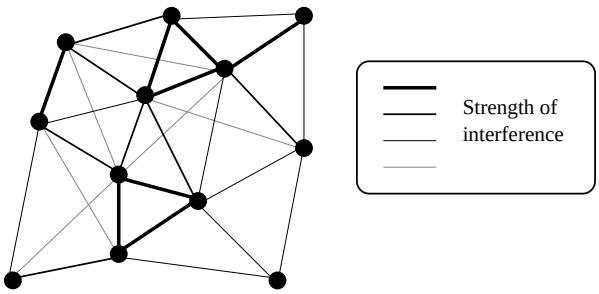

Figure 1.2: Interference graph corresponding to system in figure 1.1

rather small ($\Delta f \approx 60$MHz in the 900MHz band for the GSM system [BF96]), and hardware techniques can't do all the job by themselves. Minimum Shift Keying and Gaussian Minimum Shift Keying can be used to eliminate spectrum bumps around the channel, while frequency hopping schemes can eliminate constant interferences by continuously jumping from one channel to the other.

1.2 Mathematical Model

In the channel assignment problem the system can be modeled as a complete graph (a clique) having vertices $v_1, \ldots, v_{n_{\mathrm{CE}}}$, one for each base station, any edge (v_i, v_j) having an associated real number (a weight) W_{ij} proportional to the interference strength between the corresponding base stations i and j. Of course, in practical situations many edges have a null weight, so that completeness becomes just a local feature. For example, the "real-world" system represented in figure 1.1 can be modeled with the graph in figure 1.2, where the various interference strengths are represented with different line widths and grey levels.

To take into account the different communication needs of every base station (the number of communication channels it requires to fulfill the requests by the mobile transceivers), we associate a nonnegative integer T_i to every node v_i.

Every communication request is fulfilled by assigning a channel to the mobile host issuing it. The set of channels is discrete, and it can be modeled with a set of nonnegative integers $\mathrm{CH} \subset \mathbb{N}$. If properly calculated, the map between the set of channels and the set of integers can associate near channels (that have the highest mutual interference) with consecutive integers, so that the difference of associated integers gives a measure of mutual interference between channels.

We shall assume that, to avoid interference, two channels $a \in \mathrm{CH}$ and $b \in \mathrm{CH}$ used by two different cells i and j must differ at least by the weight of the connecting edge:

$$|a - b| \leq W_{ij}. \tag{1.1}$$

A global number W is provided to set the minimum difference between two channels in the same base station to avoid interference. If channels $a \in \mathrm{CH}$ and $b \in \mathrm{CH}$ are used in the same cell, then we must have

$$|a - b| \leq W. \tag{1.2}$$

The channel assignment problem has many non-equivalent formulations.

Problem formulation 1. *Given the number of cells n_{CE}, the requested traffic at every cell $(T_i)_{i=1,\dots,n_{\mathrm{CE}}}$, the interference graph with weights $(W_{ij})_{i,j=1,\dots,n_{\mathrm{CE}}}$ and the local interference constraint W, assign channels to cells so that*

1. *cell i has T_i channels, denoted by $C_{i1},\dots,C_{iT_i}\in\mathbb{N}$,*
2. *frequency constraints are fulfilled, and*
3. *the quantity*

$$n_{\mathrm{CH}}=\max CH=\max\{C_{ij},i=1,\dots n_{\mathrm{CE}},j=1,\dots,T_i\}$$

is as small as possible.

Problem formulation 2. *Given the number of cells n_{CE}, the requested traffic at every cell $(T_i)_{i=1,\dots,n_{\mathrm{CE}}}$, the interference graph with weights $(W_{ij})_{i,j=1,\dots,n_{\mathrm{CE}}}$, the local interference constraint W, and the set CH of available channels, assign channels to cells so that*

1. *cell i is assigned n_i channels, denoted by $C_{i1},\dots,C_{in_i}\in\mathbb{N}$,*
2. *frequency constraints are fulfilled, and*
3. *the quantity*

$$\sum_{i=1}^{n_{\mathrm{CE}}}\max\{T_i-n_i,0\},$$

is as small as possible.

Problem formulation 3. *Given the number of cells n_{CE}, the requested traffic at every cell $(T_i)_{i=1,\dots,n_{\mathrm{CE}}}$, the interference graph with weights $(W_{ij})_{i,j=1,\dots,n_{\mathrm{CE}}}$, the local interference constraint W, and the set CH of available channels, assign channels to cells so that*

1. *cell i is assigned T_i channels, denoted by $C_{i1},\dots,C_{iT_i}\in\mathbb{N}$,*
2. *the cardinality of the set*

$$\left\{(i,j,i',j')\middle|i,i'=1,\dots n_{\mathrm{CE}},j=1,\dots,T_i,j'=1,\dots,T_{i'},|C_{ij}-C_{i'j'}|<W_{ii'}\right\},$$

where for every i we set $W_{ii}=W$, is as small as possible.

In formulation 1 we try to minimize the global bandwidth of the system while respecting all constraints. The problem is well posed, because we can add channels one by one by selecting integers large enough to satisfy all constraints at every step, until we have fulfilled all traffic requirements. Note that we are not looking for a minimum number of channel (although the problem would equally be well posed), but we want to minimize the highest channel that we must use.

Formulation 2 allows us to model a real-world problem: the global bandwidth is limited by the equipment design and by regulations. So we have to sacrifice another constraint: the number of communications requests. If a request exceeds the number of available channels at a cell, so that in no way an interference-free assignment could be accomplished, the request is rejected.

Formulation 3 tries to fulfill all communications requests, while allowing some degree of interference, i.e. allowing a lower *quality of service*. In fact, we minimize the count of couples of interfering channels at every couple of cells. In some situations, in fact, the ability to communicate is more important than the quality of communication. This is the case of some spread-spectrum techniques, where the number of orthogonal codes is virtually infinite, but where every code is received as noise by all other apparata.

Note that when

$$\forall i, j \in 1, \ldots, n_{\mathrm{CE}} \quad W_{ij} \in \{0, 1\} \quad \text{and} \quad T_i = 1$$

all of the above is reduced to different formulations of the graph coloring problem. For binary interference and arbitrary traffic requirements we have the list coloring problem, where more than one color must be assigned to a node.

The problem can be made more realistic by minimizing cumulated interference, for example the sum of all interference constraint violations. Given the interference constraints (equations 1.1 and 1.2), a measure of their violation is the difference between their distance and the minimum allowed distance $W_{ij} - |a - b|$:

$$\sum_{i=1}^{n_{\mathrm{CE}}} \sum_{i'=i}^{n_{\mathrm{CE}}} \sum_{j=1}^{T_i} \sum_{j'=1}^{T_{i'}} \max\bigl\{0, W_{ii'} - |C_{ij} - C_{i'j'}|\bigr\}.$$

1.3 Techniques for Assigning Channels

Let us briefly summarize the main channel assignment algorithms (for more details see [KN96] [JS96]).

In section 4.3, page 29, we shall compare these algorithms with the technique we shall propose in the next chapters, based on objective function minimization.

Fixed Allocation In the FCA (Fixed Channel Allocation) algorithm, each cell is assigned a fixed pool of frequencies, so that no near cells can use the same channel. No communication is needed between cells; when all channels assigned to a cell are in use, subsequent requests issued within that cell shall be rejected.

Channel Borrowing Each cell has an assigned pool of frequencies, but a channel can be borrowed from a neighbor, provided that its use does not cause interference.

Dynamic Assignment By DCA (Dynamic Channel Assignment) techniques, every cell can have access to every channel, as long as it does not cause interference.

1.4 Scope of this work

In this chapter we have described the Channel Assignment problem, giving a mathematical model for it. Then we have given a fast overview of the channel assignment problem and the algorithms that can be found in the problem literature. A more detailed review can be found in chapter 2.

In chapter 3 we present some applications of local search techniques to the Channel Assignment problem; the application of these heuristics to the Channel Assignment

Problem is new, and results prove their superiority in the considered cases. The topic and partial results have been discussed in the seminar *Heuristic Algorithms for Wireless Networks* at the conference *Computer Science and Operations Research: Recent Advances in the Interface*, held in Monterey (USA) in 1998.

In chapter 4 we describe in greater detail the objective-function approach which is central to our work. We show how the minimization problem can be solved in polynomial time. The work discussed in this chapter has been presented at the *Workshop sui Sistemi Distribuiti: Architetture, Algoritmi, Linguaggi* (*WSDAAL97*), held in Bertinoro (Italy) and at the First Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (*DIAL M for Mobility*) held in Budapest (Hungary), and it appeared on its proceedings book [BBB97].

In chapter 5 we introduce some modifications to the algorithm in order to distribute it among the cell. To do so, we need a communication scheme to broadcast all status changes to a small local cluster and a local synchronization method to make sure that a cell allocates a channel only when it has enough knowledge of the status of its neighbors. Partial covering of the arguments of this chapter can be found on our paper [BBB00], accepted for publication on *Mobile Networks*.

In chapter 6 we shall draw some conclusions suggested by the experimental analysis and point out some possible future developments in the research in this field.

Finally, appendix A will report some of the code we used to implement the simulations, while appendix B we describe an interesting and useful side product of our research: a tutorial web site with a few explanations about the Channel Assignment problem and some Java demonstration applets emulating our algorithms.

Chapter 2

Review of Channel Assignment Algorithms

In this chapter we briefly describe the various algorithms considered in the literature as partial solutions of the channel assignment problem. A more comprehensive survey can be found in [KN96], [JS96].

We shall restrict ourselves to variants of formulation 2, in which the set of frequencies is determined in advance, interferences must be avoided and the largest number of requests must be fulfilled. In most papers the problem is again simplified by assuming that to avoid interferences two channels just need to be different ($W_{ij} \in \{0, 1\}$), but many of the techniques we describe can be applied to a more general case. In particular, the techniques we propose in the following chapters are not restricted to this simplified interference model.

2.1 Fixed Allocation

In the Fixed Allocation strategy every cell is permanently assigned a set of nominal channels according to interference and traffic constraints. The assignment policy is required to decide which channels should be assigned to which cells before activating the system.

This policy will try to solve some variant of the problem formulation 2 from page 5. While we can safely suppose that interference constraints are available in advance (by accurate simulation or by field measurements), the traffic requirements (T_i) cannot be accurately foreseen, if not by statistical means.

In its simplest form, an FCA algorithm will allocate the same number of channels to every cell. To do this, the channel set is partitioned into a number of subsets of equal cardinality and these sets are assigned to cells according to some possibly regular scheme. Consider, for example, the common hexagonal tiling. If the set of available channels is partitioned into three subsets, numbered 1, 2 and 3, then the regular pattern at left of figure 2.1 shows a possible assignment for a reuse distance equal to two. Note that the grey cluster is repeated over and over to build up an assignment where no adjacent cells are assigned the same set of channels. When a reuse distance of two hops is needed, the whole channel set must be partitioned into *seven* subsets, numbered from 1 to 7, and their assignment (together with the basic cluster) is shown on the right side of figure 2.1.

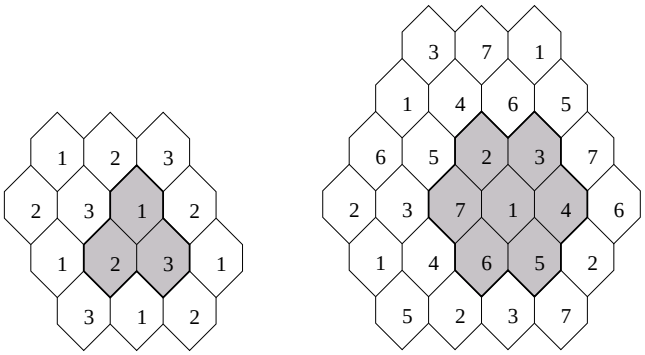

Figure 2.1: Reuse schemes for interference distance equal to one and two hops

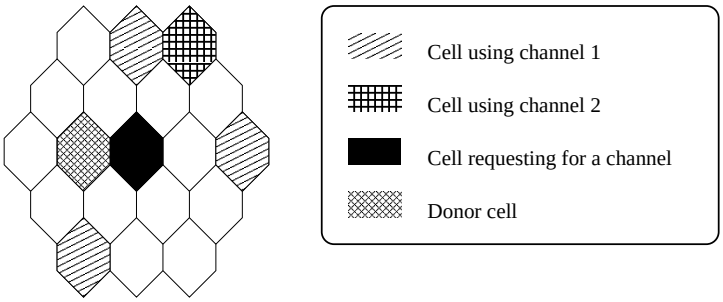

Figure 2.2: Taking channel locking into consideration

However, when coming to the real world, a fixed pool of channels is hardly enough, even at low traffic rates, because of random fluctuations of the number of requests: this is the so-called *hot spot* problem.

2.1.1 Channel Borrowing

To overcome the hot spot and other problems, after assigning a set of safe nominal channels to every cell we can allow a cell to *borrow* channels from neighbors, provided that the borrowed channel does not cause interference with channels in use in other cells.

Of course, while the channel is borrowed it cannot be used by the owner cell, until it is released by the borrowing cell, so the probability of the donor rejecting a call is increased. To avoid this, the strategy can be refined to borrow a channel from the neighbor having the largest number of free nominal channels, thus realizing the SBR (Simple Borrow from the Richest) algorithm.

Moreover, a borrowed channel is surely going to prevent the use of the same channel on a cell where it is nominal. Consider figure 2.2, where the black cell at the center needs to borrow a channel from its cross-hatched left neighbor. Suppose that the reuse distance is one hop (i.e., only prime neighbors need to use different channels). The donor cell offers channels 1 and 2 to the central cell. Which channel should be accepted? Whichever channel is employed by the central cell, it shall be blocked on

its neighbors, in other words its six neighbors will be prevented from using the same channel until the central cell releases it. If we take into consideration the second ring of neighbors, however, we see that channel 1 is used by three cells and, due to their position, it is already blocked on four out of six neighbors of the central cell. On the other hand, channel 2 is in use on one cell, blocking only one neighbor of the central cell. So selecting channel 1 should be a better choice, because only two more cells would be affected by its use (in the other four cells it was already blocked), while selecting channel two would extend prohibition to five more neighbors, since only one was previously affected.

Taking into account this problem leads us to the BDCL (Borrowing with Directional Channel Locking) strategy, where a channel is more likely to be chosen if it is already blocked in the potentially interfering cells, thus giving rise to the smallest blocking probability. It is one of the best known FCA strategies.

2.2 Dynamic Allocation

On the other hand, a whole set of strategies does not require the initial assignment of nominal channels: every cell is free to choose among the full set of channels, provided that it does not cause interference. Usually the channel assignment algorithm on a base station sounds like the following:

> When asked to issue a connection to a mobile, select one channel out of those that are not in use within the reuse distance.

If there is more than one eligible channel, which one should be selected? In fact, the functionality of an algorithm often lies in its tie-breaking technique. There are some evidently bad policies, such as always choosing the lowest frequency non-blocked channel: when a channel is chosen too often, it will also be blocked on a large number of cells; a better strategy is to choose among eligible channels at random. Of course, considerations about which channel is going to cause the smallest blocking probability in the system are welcome; for instance, the algorithms we shall consider in the experimental section will be inspired to the previously discussed BDCL algorithm: the cell chooses the channel which is most 'blocked' (due to the interference constraints) in the neighboring cells, so that it gives rise to the least blocking probability.

Another possible meaning of the word "dynamic" is the following. Some cellular systems (for example GSM) do not force a communication to use the same channel from the beginning to the end of the session. If required, an ongoing call can switch to a different frequency. If an incoming call is blocked due to a bad channel assignment (no algorithm is perfect, and for every assignment policy it is possible to devise a sequence of requests that make it behave badly), a channel may be unblocked by reorganizing the assignments in nearby cells. Chapter 3 deals with such a system: every incoming call starts a new organization of the whole network. On the contrary, in chapters 4 and 5 we consider situations where reorganizations are possible (on a local basis) but are not well accepted: they usually require a large amount of communication to take place.

2.2.1 Centralized Schemes

Sometimes, a good assignment policy requires a central authority to take care of all requests, dispatching its decisions to cells but maintaining a central knowledge base. In this case, the status of the system is globally known and every decision can take

advantage of the whole configuration, of statistics about future requirements (for instance, the central authority might know that a small subset of channels should be kept unblocked in a certain area for future use), and so it is more likely to maintain a good configuration of the system.

On the other hand, having a central authority is more error prone: what if it breaks down? Moreover, such system must rely on a good communication network: the time elapsed from the instant in which the mobile host requires a channel to a base station to the instant in which the allotted channel is actually used must not be very long, while many cellular networks are spread throughout entire countries.

Chapters 3 and 4 both consider situations in which decisions are made by a single central authority. In chapter 4, in particular, we do not state a specific criterion for the choice of the non-blocked channel to assign, but we use a dynamic programming approach: we devise an objective function on the status of the cells, so that its minimum correspond to a "good" solution, then we minimize it.

2.2.2 Distributed Schemes

A more realistic approach distributes knowledge and responsibility among all nodes, or at least a well distributed subset. For example, a *dominating set* could be identified, and each dominating node could coordinate its own basin. Knowledge about the status of the whole system is not strictly necessary, good strategies can be devised requiring only information about surrounding cells.

Chapter 5 shows how cells can coordinate their computation in order to obtain performance results comparable to those of centralized schemes.

2.3 Common Criteria for Channel Choice

As we pointed out before, many radio communications systems do not allow the channel assignment to be recomputed at will. So the need arises to maintain the system in a "good" state, in some not well defined sense. We usually measure the "goodness" of a state by seeing how it develops and what blocking frequency it induces on subsequent channel requests. On criterion used to minimize the future blocking frequency is the main point of the BDCL algorithm: choosing the channel that is already blocked in most neighboring cells, in order to increase as little as possible the number of channel locks. Other criteria are based on more global considerations. We shall consider the *regular pattern* scheme and the *Compact pattern* scheme.

2.3.1 The Regular Pattern Scheme

Based on the relatively good performance of regular assignment schemes, for example Fixed Channel Assignment, The regular pattern requirement states that dynamic assignment policies should promote channel distributions with some pattern regularity in them, for example by partitioning the cells into a number of congruent regular patterns (see figure 2.3) and suggesting that a cell should use channels that are already in use in other cells of the pattern where it belongs. Of course, this requirement must not be mandatory, otherwise we would just have an FCA algorithm with a little more freedom on channel choice.

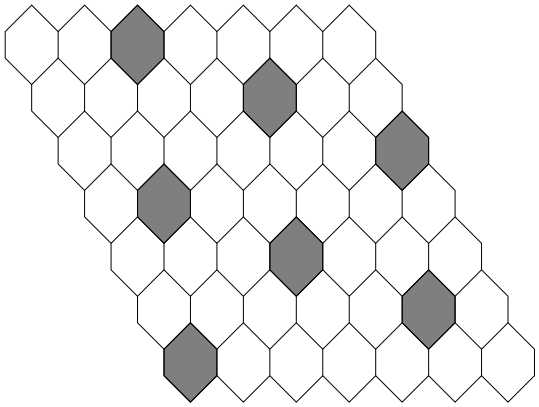

Figure 2.3: A regular pattern

2.3.2 The Compact Pattern Scheme

Some papers discuss the usefulness of the regular pattern criterion. An extension to it can be formulated, for example by requiring that each pattern be as crowded as possible. For instance, [YY94] defines a *compact pattern* as the pattern with minimum average distance between cochannel cells (the pattern shown in figure 2.3 is regular for reuse distance equal to 2).

While the authors of that paper obtain good results by enclosing the compact-pattern requirement in their algorithm, the use of the same requirement in our simulations has proven totally ininfluential. For example, figure 5.1 at page 32 shows a comparison of our BBB algorithm (which shall be discussed in chapter 4) and of a version of it without the compact-pattern requirement (which will be referred to as "reuse pattern" in that chapter).

The most important problem introduced by a regular-pattern criterion, be it compact or not, is its globality: decisions on channel allocation must be taken by considering the whole pattern, i.e. arbitrarily distant cells. This prevents the use of such criterion on distributed systems, where every cell must decide which channel allocate for a certain request by only considering a limited neighborhood, since channel usage information travels through messages from cell to cell, so it is neither fast nor accurate (the situation may vary while previous status messages have not been acknowledged yet). Thus, the observation that the compact-pattern criterion is ininfluential on our algorithm will be the key factor allowing us to introduce, in chapter 5, the distributed BBB algorithm (dBBB).

Nonetheless, compact pattern criteria prove useful in many centralized algorithms. In particular, some generalizations of the concept, such as making it non-uniform [ZY91], gives good results when compared with combinatorial algorithms such as BDCL, in particular when nonuniform call distributions aggravate the problem of distributing channels prior to system operations (as in all FCA strategies).

2.4 Comparison among strategies

No strategy is better than any other in all conditions; some assignment policies cannot even be compared because they make different assumptions about the network (for example its reconfigurability, or the nature of interference). The main evaluation parameter, when talking about channel assignment techniques, is the *call refusal rate*, i.e. the ratio between refused calls and the total number of communication requests. For a given technique, the refusal rate depends upon the total number of ongoing calls per unit of time. This is a well known quantity in telephone systems, and it can be measured in *erlang* (1 erlang = 1 ongoing call per second). We shall refer to it as the *traffic rate* of the system. Of course, the traffic rate needs not be uniform over the whole network: there are some places with a higher number of calls than others.

When the traffic rate of the system is low, FCA performs fairly well, even though traffic fluctuations may temporarily exhaust the channels assigned to a cell: even if the number of expected calls at a certain time is low, from time to time the number of calls issued in a cell will overcome the number of available channels. Borrowing algorithms have been introduced to solve this problem, thus they are more efficient than FCA with low traffic rates, but their performance deteriorates when the traffic rate increases, achieving the same performance of FCA at higher computational and communication costs.

Dynamic techniques often outperform FCA at low traffic rates, but they are much worse at high traffic rates, because many cells might find no available channels at all for a long time, due to non-optimal decisions at previous instants. On the contrary, FCA ensures that, at any traffic rate, a minimum number of calls will always be accepted by every cell.

Chapter 3

Heuristics for Local Search

For many general classes of functions arising from our setting the problem of assigning channels to communication requests in a cellular network becomes **NP**-hard. For example, consider the case in which we allow a global reconfiguration of the system. In this case the only requirements to fulfill are traffic and interference: we don't need to keep the system in a suitable state for future evolution, because it can be reconfigured at will. So the problem is a generalization of a graph coloring problem.

Many approaches are possible when looking for a satisfying solution of a minimization problem:

- We can search the whole solution space of the $n_{\mathrm{CE}} \times n_{\mathrm{CH}}$ binary matrices (A_{ij}) for a solution minimizing a proper linear combination of the number of traffic and interference violations.
- Otherwise we can limit the search in a space of *admissible* solutions; we always fulfill a few of the given criteria, for example the traffic, and try to minimize the violations of the other requirement, e.g. interference.

The second method has been used in conjunction with a powerful search technique, the Reactive Search, to find a good assignment policy given the traffic in every cell. In this case we tried to reduce two factors: the overall bandwidth of the system, i.e. the total number of different channels used by the cells in the system, and reduction of interference, considered as an additive constraint, and not (as in most of the literature) a binary restriction.

3.1 An Overview

Local search is a method of exploring the solution space in search of a minimum by walking through the local neighborhood of a given point. Clearly, the walk shall not be blind, but driven by the evaluation of the objective function at some points. Many criteria can be used to choose from time to time which direction further steps should take.

3.1.1 Steepest Descent and Some Refinements

Suppose that our search space is the set of all binary strings of given length N: many problems can be reduced to a binary representation, in particular the channel assign-

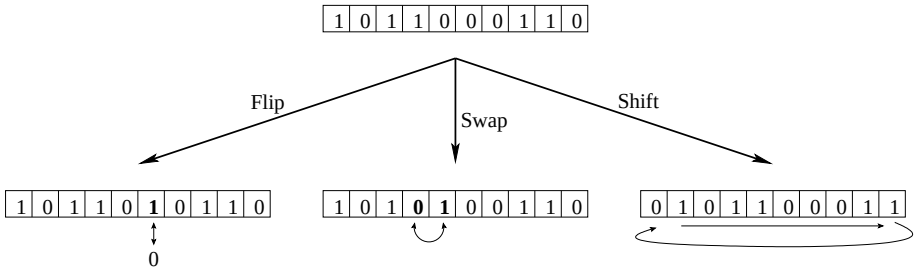

Figure 3.1: Some neighborhood rules

ment problem. The objective function will be calculated over this string. Suppose we want to find its minimum. Then we can start from a random string: every step of the algorithm will be a slight change of the string. The allowed changes on a string determine its neighborhood. Figure 3.1 shows some examples of allowed changes, that usually depend on the nature of the problem:

- flipping (i.e. inverting) a single bit of the string;
- exchanging values of two adjacent bits;
- shifting the string left or right.

Once defined the neighborhood of a string, the algorithm must decide which neighbor is the most promising. If we choose to employ a simple **Steepest Descent** algorithm, we shall choose the neighbor in which the function value is as small as possible.

If all neighbors give higher function values than the current configuration, then the system has reached a local minimum. How can the search be continued? For example by storing the current string (if it proves better than other previously found local minima) and generating a new random string from which the search can restart.

Sometimes, however, the structure of the problem suggests to insist, once a local minimum is found, to search for other local minima near it. So a random restart may not be the best technique, and a way to continue walking through neighborhood steps, even towards higher function values, should be devised.

Many local search techniques try to mimic some aspect of physical behaviors in solving their problems. For example, the idea of *simulated annealing* is to let moves towards higher function values be executed with a probability which decreases with the function increment and with time. The technique emulates, to a certain extent, thermal random motion in a heated system which is slowly cooling down. So the probability to go from a local minimum to another one is not zero, even though it may be very, very small and require a lot of iterations to accomplish.

3.2 History-Sensitive Heuristics

As we said before, many problems have a complex structure, and this makes sometimes desirable to explore the surroundings of a local minimum instead of restarting from a new point. More generally, all information gathered during the execution of the algorithm might be worth for some decisions. Techniques that base their decisions on the past history of the execution are called *memory-based* or *history-sensitive*, while

techniques such as steepest descent and simulated annealing, that choose the next step by considering only the current status, are called *memoryless*, or *Markovian*. Of course, lots of theorems are known about Markovian processes, and so the theoretical analysis of memoryless techniques is easier. Nonetheless, these results are usually asymptotic: good results are ensured only after many iterations. On the other hand, techniques based on the past history of the system, so that every step depends on an arbitrary function of past configurations, are much more difficult to analyze, and theoretical convergence results are seldom stated. However, experimental results often support the claim of their superiority, at least in some applications.

A few examples of memory-based techniques may help to understand the general framework.

- The *Reduction Technique*, proposed in [Lin65], is applied to the Traveling Salesman Problem. Its base principle is that edges that are common to many good solutions should be used more often than the others; to do this, the algorithm must *remember* some data from the solutions it has explored.
- The *Denial Strategy*, shown in [SW68], is applied to the same problem as the previous technique (TSP), but in contrast it proposes to forbid edges that are chosen too often, in order to promote differentiation. Note that differentiation is a key idea to escape the local minimum.
- The *Variable-Depth Search*, proposed by Kernighan and Lin in 1970, is applied to Graph partitioning and it forbids moves once they are done. It is one of the first examples of move prohibition.
- The *Tabu Search* [Glo89][Glo90][Glo94][HJ90] considers temporary prohibition of some moves in order to avoid cycles. It shall be explained in section 3.3
- The *Reactive Search* technique [BT94a][BT94b][Bat96] proposes dynamic adjustment of prohibition time and will be discussed in section 3.3.1.

3.3 Tabu Search

Once we have decided to explore the surroundings of a local minimum, we have the problem of escaping it for a time long enough to avoid falling back into it. The problem is illustrated in figure 3.2. The algorithm must find a way to avoid falling back on its previous steps: indeed, once the local minimum is left, switching back to a steepest descent scheme would make the just abandoned minimum very appetizing.

The Tabu Search algorithm [Glo89][Glo90] implements the steepest descent algorithm with a simple variation: once a move is made, it cannot be undone for a fixed number of steps. In other words, not all neighbor configurations are eligible for the next step.

If the system is described by a binary string and the legal moves are made by flipping a single bit, then after flipping a bit the Tabu algorithm will prevent the same bit to flip again to the previous state for a given number of steps T. In other words, the bit is "frozen" for T steps. Clearly, this prohibition prevents the system from returning to a previously visited configuration for at least $2T$ steps. In fact, once a given bit is flipped it may be restored only after at least T steps (the time after which the prohibition decays); during this time $T - 1$ other bits have been flipped, and they will be restored

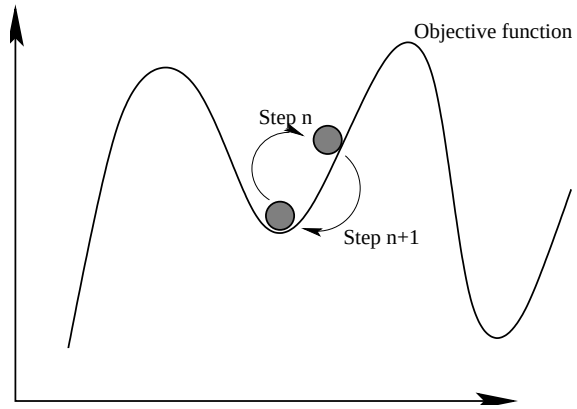

Figure 3.2: The need to avoid the minimum

during the next T steps. This means that small values of T could not prevent cycles from occurring.

Note also that the prohibition time cannot be too large, and in particular we must have $T < N$, where N is the length of the string, otherwise after N steps all the possible N moves would be prohibited and the system could not proceed.

In other words, it is clear that the determination of the right prohibition time is crucial for the algorithm and it should be carefully tuned by former analysis or by trial and error.

3.3.1 Reacting on Parameters

We have seen that the purpose of prohibition is to force the system to search in the proximity of local minima without having to jump to random solutions and without coming back too soon.

The basic idea of Reactive Search [BT94b] is to modify the prohibition time in accordance to the behavior of the search. For example, if a configuration is visited too often, the prohibition time should be increased, in order to encourage further exploration. If no configuration is repeated, the the prohibition time is large enough, but it might prevent the approach to other minima, so it might be lowered. The right balance between increasing and decreasing the prohibition time is the main problem of this heuristic.

3.4 Experimental settings

We used the library presented in appendix A to work on the following Channel Assignment problem.

We are given an adjacency graph with $n_{\mathrm{CE}} = 150$ nodes, with random edges (30% density). The system has n_{CH} available channels and every cell issues an equal number (traf) of communication request.

We shall refer to figure 3.3. The Tabu Search algorithm moves inside the set of solutions that fulfill the traffic requirements, trying to minimize the number of interfering assignments. To do this, it is given a square *adjacency matrix* $(\mathrm{interf}_{ij})_{i,j=1,\ldots,n_{\mathrm{CE}}}$,

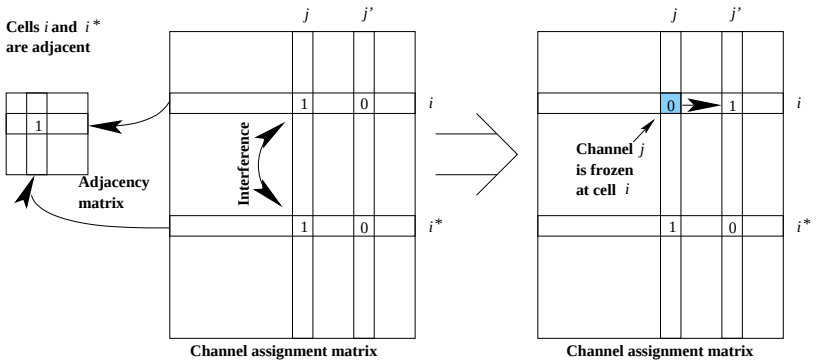

Figure 3.3: Tabu Search applied to channel assignment

with as many rows and columns as there are cells in the cellular system (in our case 150), whose entry interf_{ij} is 1 if and only if cells i and j must use different channels. The matrix is, of course, symmetric and in our experiment it is filled at random with a 30% probability of placing a "1" at a given position.

The binary string which describes the status of the system has length $n_{\text{CE}} \times n_{\text{CH}}$ and is organized as an $n_{\text{CE}} \times n_{\text{CH}}$ binary matrix where the element at row i ($i = 1, \ldots, n_{\text{CE}}$) and column j ($j = 1, \ldots, n_{\text{CH}}$) is 1 if and only if cell i is assigned channel j.

Of course, a solution is admissible if and only if it assigns exactly traf channels to every cell, that is if and only if every row of the binary status matrix contains exactly traf 1's and $n_{\text{CH}} -$ traf 0's. So our space of admissible solutions is the following:

$$\mathcal{S} = \left\{ M \in \mathcal{M}_{n_{\text{CE}} \times n_{\text{CH}}}(\{0,1\}) \quad \middle| \quad \forall i \in \{1, \ldots, n_{\text{CE}}\} \quad \sum_{j=1}^{n_{\text{CH}}} m_{ij} = \text{traf} \right\}$$

If we choose to move inside the space of feasible solutions, then we cannot employ simple bit flipping as our basic move, as this would necessarily change the number of 1's in the string. Our basic move is then moving a 1 to a 0 entry of the same row, thus changing one assigned channel of one cell, maintaining the number of assigned channels unchanged.

The function to minimize is clearly related to interference. It just counts the number of interference constraint violations: for every couple of adjacent (i.e. interfering) cells it counts the number of equal channels in use:

$$\begin{aligned} f &: \mathcal{S} \to \mathbb{R} \\ M &\mapsto \sum_{i=1}^{n_{\text{CE}}} \sum_{i'=i+1}^{n_{\text{CE}}} \left(\text{interf}_{ii'} \sum_{j=1}^{n_{\text{CH}}} m_{ij} m_{i'j} \right) \end{aligned}$$

At every step, we choose the move with the highest decrease of interference (or the lowest increase, because we accept to worsen the situation), breaking ties at random. Not all moves between admissible configurations are allowed, because some of them are *prohibited*: some bits are locked to their 0 state for some number of steps.

Suppose, for example, that cells i and i^* interfere (there is a 1 in the interference matrix) at the channel j, and that we decide to drop channel j replacing it with another

one. Then we shall move the 1 from position j of row i to the non-prohibited position that guarantees the lowest interference count. Once done, the channel that has been dropped remains prohibited (frozen in its 0 state) for cell i for a certain number of steps.

Of course the system needs to remember the moves made during the last steps, in order to avoid undoing them. Moreover, while the simple Tabu scheme just requires an array of forbidden moves, our Reactive Search needs a criterion to decide when prohibition time should be increased and when it should be lowered. The criterion we chose is the following: we count the number of times a configuration is visited, and we mark it as "frequently visited" if it has been visited more than a given number of times (say 3, as in the experiment). When too many configurations are "frequently visited" (three configurations, for example), then it is time to increase by a given factor (for example 10%) the prohibition time, unless it is already too high. If no local minima are found for too long, then the prohibition time is lowered by a given factor.

These criteria require the storage of all visited configurations. Since their number and length are prohibitive for a complete storage, we just store a 64-bit fingerprint of the visited configuration in a dictionary, associated with an integer counter which stores the number of visits already made to that configuration. The 64-bit fingerprint is a hash value obtained by large integer arithmetic operations on the binary string, so that the probability of two visited configurations having the same key is low.

Other mechanisms that we employ to differentiate the search are:

- an *escape* mechanism: when the surroundings of our solution space are probably exhausted (too many configurations have been visited too often), then the algorithm restarts at random;
- the *aspiration* criterion: if a move is prohibited, but applying it would lead to the lowest objective function value ever found, then it is applied notwithstanding the prohibition.

3.5 Results

In figures 3.4 and 3.5 we show a comparison among different heuristics, namely:

- the Steepest Descent (SD) algorithm, restarting every time a local minimum is found;
- a modified version of the Tabu Search (TS) algorithm where at any time the prohibition time is proportional to the number of candidates for the next move;
- the Reactive Search (RS) technique with the mechanisms discussed in section 3.4.

Figure 3.4 shows a comparison among the three mentioned heuristics when the requested traffic per cell is traf $= 1$ and $n_{\mathrm{CH}} = 12$ channels are available. So the problem is just a graph coloring problem on a random graph with $n_{\mathrm{CE}} = 150$ nodes, 30% edge probability and $n_{\mathrm{CH}} = 12$ colors. Figure 3.5 refers to a more complex setting: traf $= 3$ and $n_{\mathrm{CH}} = 35$.

In the upper graph of each figure the minimum number of interferences found by each of the algorithms is plotted against the number of steps. Each graph is the average of ten runs. The lower graph of each figure compares the same algorithms by giving their best result versus execution time. Number of steps is in fact a deceptive measure

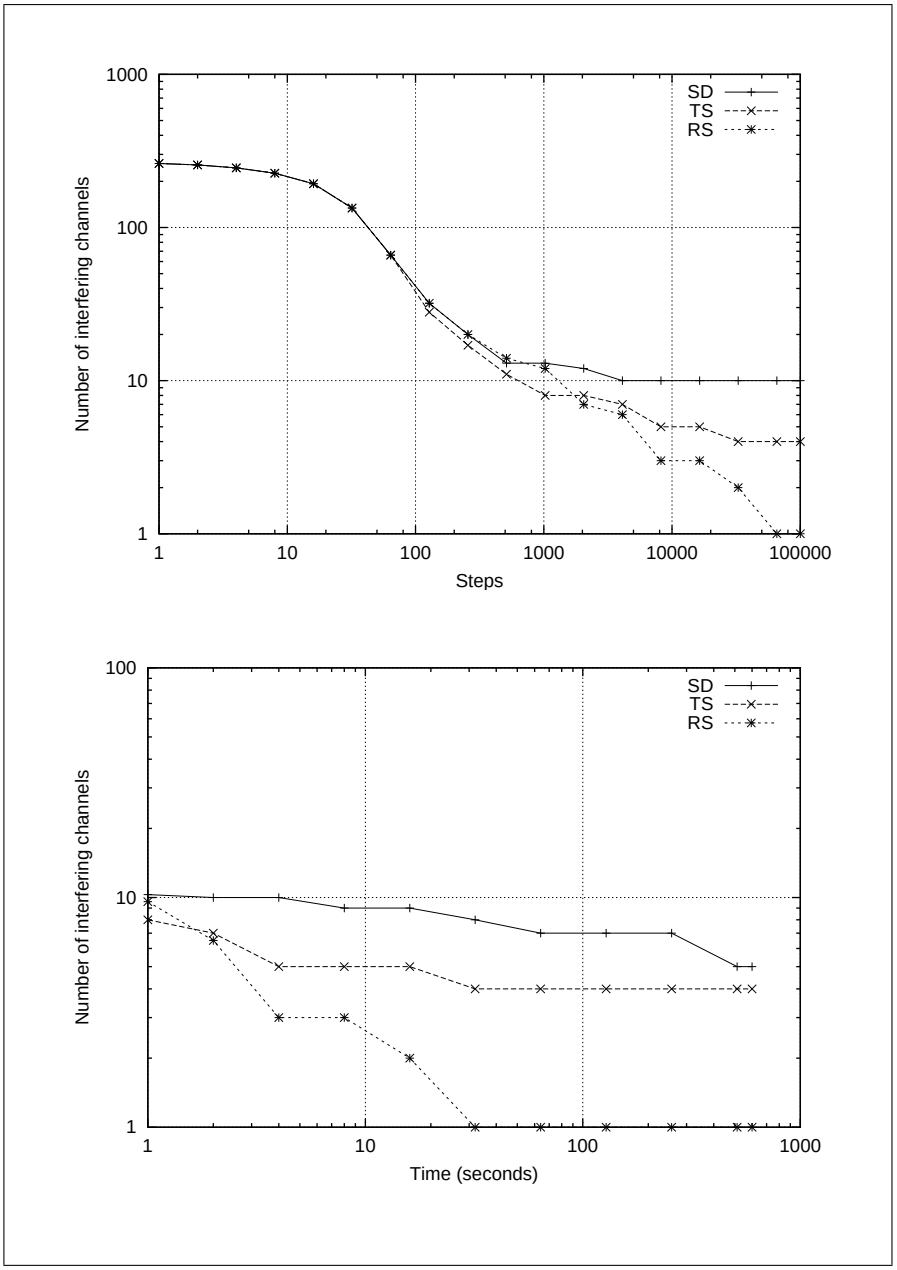

Figure 3.4: Comparison among heuristics, traf $= 1, n_{\mathrm{CH}} = 12$

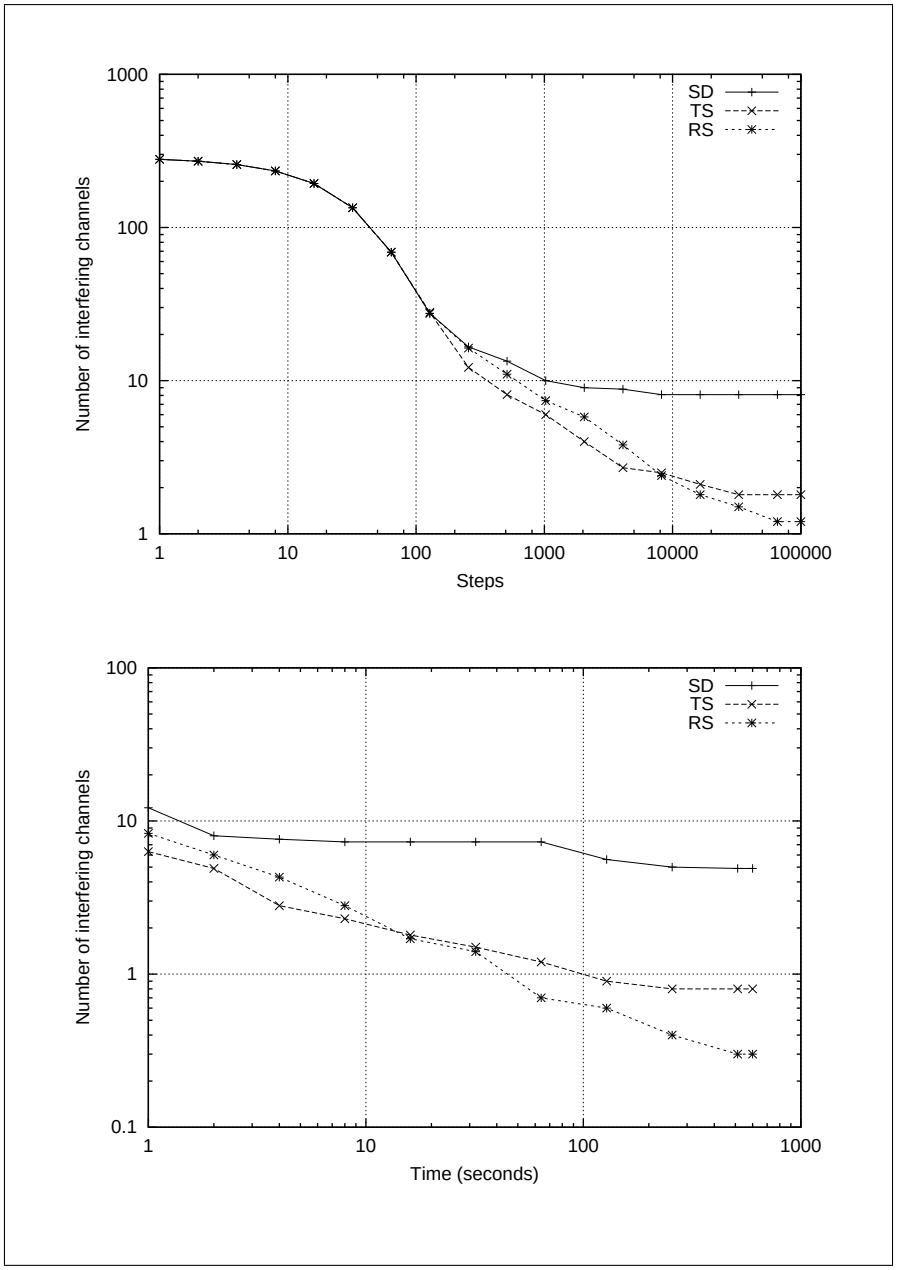

Figure 3.5: Comparison among heuristics, traf $= 3, n_{\mathrm{CH}} = 35$

of time, since complex techniques such as Reactive Search require more processor time per step with respect to simple techniques such as Steepest Descent.

The results, however, show that Reactive Search performs quite better than the other heuristics we have considered. Not only does it find better configurations than other heuristics, but it finds them much faster. We had to represent the results in a logarithmic scale to appreciate the results of all heuristics.

Chapter 4

A New Centralized Channel Assignment Algorithm

Many penalty-function heuristics have been applied to the channel assignment problem. Some of them require the rearrangement of the whole cellular system when a new channel is requested [SB96] [DAKR93]; unless the traffic is very low and service communications among cells are cheap and fast, this approach is of little practical use. Other penalty-function heuristics, such as the one we shall consider next, just rearrange the channel assignment inside the cell where the new communication request is issued, by choosing those channels that minimize the probability of a future channel refusal, trying to keep the system in a suitable status for future requests (this effort of maintaining a good configuration is clearly unnecessary in the former case).

In both cases, two heuristic steps must be taken. First, we must heuristically determine a good penalty function, where "good" means that its minimum should correspond to a suitable configuration. Then, when this function is determined, we must actually locate its global minimum, or at least find some good local minimum; to do so, we need to apply a minimum-search heuristic technique to the penalty function.

The topic discussed in this chapter is the main argument of a work we presented at various conferences and workshops [BBB97][BBB00].

4.1 A penalty function heuristic

Following [DFR96], let n_{CE} be the number of cells and n_{CH} the total number of channels. Every cell i, $i = 1, \ldots, n_{\mathrm{CE}}$, has a traffic demand traf_i which changes with time. Let us denote with $d_{ii'}$ the Euclidean distance between the centers of cells i and i', and let $\mathrm{interf}_{ii'}$ be a $\{0, 1\}$-valued function which states if the two cells interfere or not; the notation can nonetheless be extended to the case of various degrees of interference.

When a connection or termination request is issued in cell i^*, the frequency allocation in this cell must be optimized. The status of channel allocation is given by a $\{0, 1\}$-valued matrix A_{ij} whose entry (i, j) is 1 if and only if channel j is currently in use in cell i. The new channel allocation for cell i^* is stored in vector V_j, $j = 1, \ldots, n_{\mathrm{CH}}$.

An objective function is built whose minimum is likely to be a good solution of the new allocation for cell i^*. First, a term to privilege those solutions without interference

(all terms depend on V, our unknown solution) is introduced:

$$a(V) = \sum_{j=1}^{n_{\mathrm{CH}}} \sum_{\substack{i=1 \\ i \neq i^*}}^{n_{\mathrm{CE}}} V_j A_{ij} \,\mathrm{interf}_{ii^*} \,. \tag{4.1}$$

This term adds 1 for each cell interfering with i^* which uses a channel in use in i^*. Second, the requests of the cell i^* should be respected as much as possible:

$$b(V) = \left(\mathrm{traf}_{i^*} - \sum_{j=1}^{n_{\mathrm{CH}}} V_j \right)^2 . \tag{4.2}$$

The only reason to make this term quadratic is that it must be nonnegative: (an absolute value would also work). Third, a *packing condition* is added: a channel should be reused as near as possible (outside the interference zone), to restrict the blocking probability in other cells.

$$c(V) = -\sum_{j=1}^{n_{\mathrm{CH}}} \sum_{\substack{i=1 \\ i \neq i^*}}^{n_{\mathrm{CE}}} V_j A_{ij} \frac{1 - \mathrm{interf}_{ii^*}}{d_{ii^*}}. \tag{4.3}$$

This subtracts a positive term for each cell outside the interference zone which reuses a channel employed in cell i^*; the larger the distance, the smaller the subtracted term. Next, changes in the present allocation of the cell i^* should be minimized:

$$d(V) = -\sum_{j=1}^{n_{\mathrm{CH}}} V_j A_{i^* j}. \tag{4.4}$$

This subtracts 1 every time a channel currently used by cell i^* is chosen for the next configuration (this means that a mobile host needs to change its channel as rarely as possible). If some frequency hopping technique is used, however, this requirement does not make much sense, as the mobile host is equipped for frequent configuration changes. Last, experimental evidence shows that to achieve a good performance the channel reuse should follow a regular scheme (for example, a compact pattern [YY94]). This is achieved by introducing the $\{0, 1\}$-valued matrix $\mathrm{res}_{ii'}$ whose entry (i, i') is 1 if and only if cells i and i' belong to the same reuse scheme (i.e. should use the same channels if possible). Common reuse schemes follow some sort of "knight" move (for instance, the one shown in Fig. 4.1).

$$e(V) = \sum_{j=1}^{n_{\mathrm{CH}}} \sum_{\substack{i=1 \\ i \neq i^*}}^{n_{\mathrm{CE}}} V_j A_{ij} (1 - \mathrm{res}_{ii^*}). \tag{4.5}$$

Note that all terms are arranged to go towards a lower value when the constraints are satisfied. Let us combine them in a single objective function to minimize:

$$J(V) = A \cdot a(V) + B \cdot b(V) + C \cdot c(V) + D \cdot d(V) + E \cdot e(V), \tag{4.6}$$

where A, B, C, D and E give different importance to the various constraints.

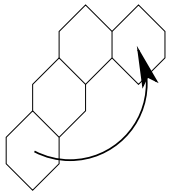

Figure 4.1: Building a reuse scheme: the basic move

4.2 The polynomial algorithm BBB

So far a possible penalty function $J(V)$ has been heuristically determined. To minimize $J(V)$, [DFR96] employs Hopfield neural networks, but actually the minimization of this function is straightforward and does not require any heuristic search technique. In fact, we can rewrite $J(V)$ as a quadratic function in which the quadratic term *depends only on the number of channels*, and not on the single channels used. Let us rewrite equation 4.1:

$$a(V) = \sum_{j=1}^{n_{\mathrm{CH}}} V_j a_j, \qquad \text{where} \qquad a_j = \sum_{\substack{i=1 \\ i \neq i^*}}^{n_{\mathrm{CE}}} A_{ij}\, \mathrm{interf}_{ii^*};$$

the term a_j simply counts the number of cells in the interference zone of i^* which use the channel j. The traffic term $b(V)$ found in equation 4.2 can be rewritten as

$$b(V) = \left(\sum_{j=1}^{n_{\mathrm{CH}}} V_j \right)^2 - 2\,\mathrm{traf}_{i^*} \sum_{j=1}^{n_{\mathrm{CH}}} V_j.$$

The $\mathrm{traf}_{i^*}^2$ term is constant and can be omitted, while the quadratic term is the square of the number of channels reserved for the cell i^* (the number of 1's in vector V). Let us rewrite it in a way similar to the other ones:

$$b(V) = \left(\sum_{j=1}^{n_{\mathrm{CH}}} V_j \right)^2 + \sum_{j=1}^{n_{\mathrm{CH}}} V_j b, \qquad \text{where} \qquad b = -2\,\mathrm{traf}_{i^*}\,.$$

Clearly, b does not depend on j. The other terms can be rewritten as follows:

$$c(V) = \sum_{j=1}^{n_{\mathrm{CH}}} V_j c_j, \quad d(V) = \sum_{j=1}^{n_{\mathrm{CH}}} V_j d_j, \quad e(V) = \sum_{j=1}^{n_{\mathrm{CH}}} V_j e_j$$

where

$$c_j = -\sum_{\substack{i=1 \\ i \neq i^*}}^{n_{\mathrm{CE}}} A_{ij} \frac{1 - \mathrm{interf}_{ii^*}}{d_{ii^*}},$$

$$d_j = -A_{i^* j}, \qquad e_j = \sum_{\substack{i=1 \\ i \neq i^*}}^{n_{\mathrm{CE}}} A_{ij}(1 - \mathrm{res}_{ii^*}).$$

The term c_j evaluates the packing condition for channel j; the term d_j rewards the choice of channel j if it was already in use; the term e_j penalizes the use of a channel outside the reuse scheme.

We can collect the single coefficients into global ones:

$$w_j = A \cdot a_j + B \cdot b + C \cdot c_j + D \cdot d_j + E \cdot e_j; \tag{4.7}$$

the global objective function (4.6) is then

$$J(V) = \left(\sum_{j=1}^{n_{\mathrm{CH}}} V_j \right)^2 + \sum_{j=1}^{n_{\mathrm{CH}}} w_j V_j, \tag{4.8}$$

where, as we have already pointed out, the square term is just the square of the number of assigned channels.

To minimize $J(V)$ we calculate the weights w_j for each channel; each calculation requires at most n_{CE} steps to test interferences, reuses and packing. Globally, the calculation of the weights w_j requires time $O(n_{\mathrm{CE}} n_{\mathrm{CH}})$. If we had fixed the number n of channels that we want to assign, the minimization would be achieved by taking the channels j whose w_j are the least (the quadratic term is constant among the solutions with the same number of channels). To take advantage of this, we calculate a permutation σ_j, $j = 1, \ldots, n_{\mathrm{CH}}$, such that the vector $(w_{\sigma_j})_{j=1,\ldots,n_{\mathrm{CH}}}$ is sorted in increasing order. The sort requires time $O(n_{\mathrm{CH}} \log n_{\mathrm{CH}})$. At last, let us call J_n the minimum of the objective function restricted to n-channel solutions. Its value is

$$J_n = n^2 + \sum_{j=1}^{n} w_{\sigma_j}, \qquad n = 0, \ldots, n_{\mathrm{CH}},$$

and the difference between the minima for successive values of n is

$$J_n - J_{n-1} = 2n - 1 + w_{\sigma_n}, \qquad n = 1, \ldots, n_{\mathrm{CH}}.$$

So, a simple scan of the vector w_{σ_j} is enough to find the minimum for all n, that is the global minimum, in time $O(n_{\mathrm{CH}})$.

Hence, the global minimum of the objective function,

$$\min_{V \in \{0,1\}^{n_{\mathrm{CH}}}} J(V) = \min_{n=0,\ldots,n_{\mathrm{CH}}} J_n,$$

can be found in total time $O\big(n_{\mathrm{CH}}(n_{\mathrm{CE}} + \log n_{\mathrm{CH}})\big)$. The procedure returns also the number of channels in the optimal solution, say n^*, therefore the channels to be assigned to cell i^* are

$$n_{\sigma_1}, n_{\sigma_2}, \ldots, n_{\sigma_{n^*}}.$$

Let us call this penalty-function minimization heuristic the BBB algorithm. The first advantage of using BBB is that, of course, we find the true global minimum of $J(V)$. In addition, consider that Hopfield networks, like many other techniques, require our function coefficients to vary only in a certain range in order to ensure stability and convergence of the search; in other words, coefficients are critical not only in weighting the various constraints (which is precisely what they are introduced for), but also in making the minimum-search procedure succeed. Algorithm BBB does not use any heuristic search algorithm, so it is not restricted to those coefficient values that ensure convergence, and we may let them vary over all the nonnegative real range, thus having more freedom in tuning them.

Table 4.1: Coefficients for function $J(V)$

Coefficient	Value
A	7000
B	45
C	1.2625
D	0.01
E	4.17625

4.3 Experimental settings

We consider a 7×7 hexagonal grid, as the one shown in Fig. 5.8, like in most of the literature. A fixed server station is placed at the center of each cell, while a number of mobile hosts is free to move across the whole land. The total number of available channels is 70 and the co-channel interference is extended to the second ring of neighbors. The grid does not wrap like a torus.

A C++ program has been written to run a comparative simulation of five algorithms (FCA, SBR, DCA with local optimization, BDCL and BBB).

For FCA, the reuse scheme (a reuse distance of two cells has been considered) consists in seven partitions of ten channels each. The Euclidean distance between the centers of two neighbors is 1, and the reuse scheme given by the function $\text{res}_{ii'}$ of Section 4.1 is built by iterating the basic "knight" move of Fig. 4.1, which gives the same pattern as the token placement in Fig. 5.8. The same scheme has been used to distribute the seven channel groups among the cells for the FCA algorithm.

Coefficients for the function $J(V)$ are shown in table 4.1. Their values are the same that have been used in [DFR96].

4.4 Results

Each algorithm (FCA SBR, DCA, BDCL, BBB) has been simulated for connection rates of 160, 170, 180, 190 and 200 calls per hour with Poissonian distribution (and hence exponential inter-arrival time), corresponding to a traffic of 8, 8.5, 9, 9.5 and 10 erlang. The mean duration of a connection is exponential with an average of 180sec. Each simulation consisted of 50 runs of 100000 seconds of simulated time each.

Fig. 4.2 shows the results of this simulation. Every error bar represents the 95% confidence interval calculated over 50 independent runs. It is clear that algorithm BBB outperforms all these algorithms.

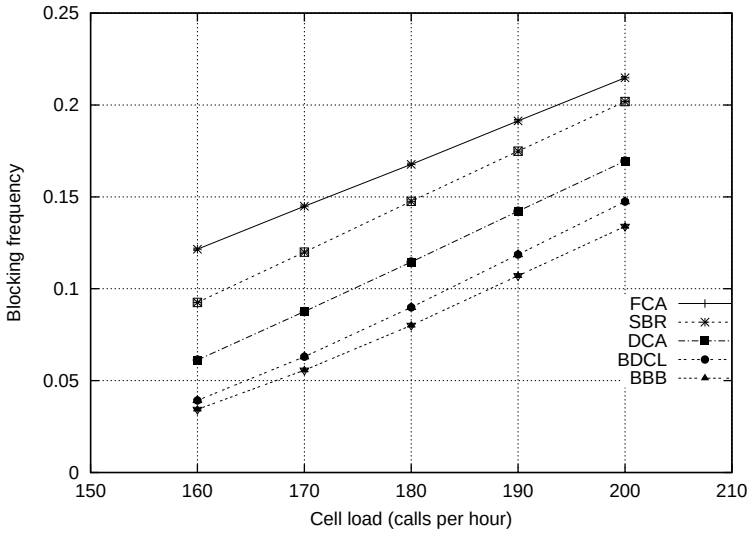

Figure 4.2: Comparison among algorithms

Chapter 5

A Distributed Version

When devising a distributed algorithm we usually have to cope with a number of problems that never arise in sequential settings. In particular, the strict sequential execution of instructions when we work with a single Von Neumann processor is no more ensured.

Nonetheless, some operations need either to be ordered in time or to be sure that the conditions of the system are not varied until completion. So the problems of *synchronization*, *mutual exclusion* and *message passing* arise.

Earlier parts of this work, including the broadcast and token passing algorithms, but not the theoretical analysis, have been included in [BBB00].

5.1 The distributed algorithm dBBB

We first note that algorithm BBB can be improved by storing at each cell a permanently sorted array of weights to be updated at each change of state in the nearby cells. The sorting time can thus be cut down to a simple update of the sorted array at each call. Moreover, by considering only local interference the calculation of the weights does not depend on the total number n_{CE} of cells, but it can be performed in $O(1)$ time.

We need, however, to simplify our objective function by eliminating non-locality. There are only two global terms in the function:

- The "reuse scheme" given by the array $\mathrm{res}_{ii'}$; some tests (see figure 5.1 at page 32) show that it is not influential on the overall system performance.
- The "packing condition", whose weight decreases at increasing distances; the same preliminary tests cited above prove that we can restrict the "packing condition" to the $2r$-th ring of neighboring cells (where r is the interference radius).

So we can rewrite the packing term

$$c'(V) = \sum_{j=1}^{n_{\mathrm{CH}}} V_j c_j$$

where

$$c'_j = - \sum_{\substack{i=1 \\ 0<d_{ii^*}\leq 2r}}^{n_{\mathrm{CE}}} A_{ij} \frac{1 - \mathrm{interf}_{ii^*}}{d_{ii^*}}.$$

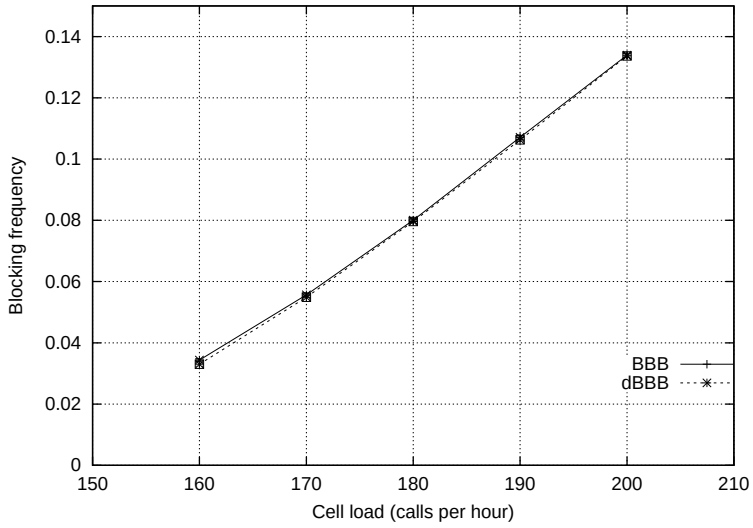

Figure 5.1: Global vs. local optimization

and we can exclude the reuse term $e(V)$, thus obtaining the new weighting system

$$w'_j = A \cdot a_j + B \cdot b + C \cdot c'_j + D \cdot d_j$$

which gives the new local objective function

$$J'(V) = \left(\sum_{j=1}^{n_{\mathrm{CH}}} V_j \right)^2 + \sum_{j=1}^{n_{\mathrm{CH}}} w'_j V_j .$$

To minimize this function a cell only needs information about its neighborhood. We shall see in the following sections (5.2 and 5.3) how this information can be spread in an optimal way.

5.1.1 Experimental results

The elimination of the reuse scheme and the localization of the channel packing condition as in Section 5.1 finally lead us to the data in Fig. 5.1, where we compare the original BBB algorithm with its distributed version dBBB. As it is apparent, all differences lie within the confidence intervals. This enables us to transform the algorithm into a local one with no performance loss.

5.2 A broadcasting scheme

Once the objective function is localized, we need a good communication strategy to replace the central authority which took all the decisions in the sequential algorithm.

When a cell initiates or terminates a call, it must broadcast its new status to its neighbors, up to a certain distance; to do so, we need a simple broadcasting scheme, like the one presented in this section.

To exemplify a real world setting we make some assumptions:

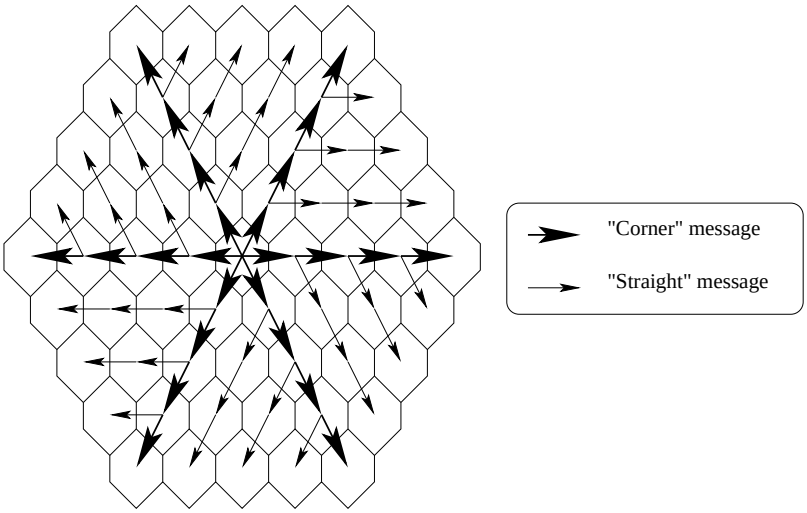

Figure 5.2: A local broadcasting scheme

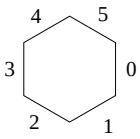

Figure 5.3: How directions are numbered

- the system is purely asynchronous: all kind of synchronization must be obtained through message passing;
- the system is modeled as a hexagonal tiling, where every cell only communicates with its six neighbors;
- a message can be spread through the system only by relaying it from neighbor to neighbor.

5.2.1 The algorithm

Let the message be structured as a tuple (c, r, h, m), where c is a flag indicating if the message must be duplicated: if c is set, the message shall be called a "corner" message; r is the maximum distance from the source the message must reach, h is the number of steps the message has taken up to now and m is the message itself. Suppose that in each cell the directions of incoming and outgoing messages are numbered clockwise modulo 6 (see figure 5.3). Then if a message arrives on direction d, the opposite direction will be $d + 3 \pmod 6$.

The algorithm can be implemented as in Fig. 5.4. The source sends six "corner messages" to its neighbors (lines 2–3); when a cell receives a message, if it is not far enough (line 6), it must relay it to the cell on the opposite side (line 7); if it is a corner message, the cell must also propagate a non-corner ($c = 0$) copy of the message to

```
1.  source:
2.      for d in 0 . . . 5 do
3.          send (1, 4, 1, m) along direction d;
4.  relay:
5.      upon receipt of (c, r, h, m) from direction d do
6.          if h < r then
7.              send (c, r, h + 1, m) along direction d + 3 (mod 6);
8.              if c then
9.                  send (0, r, h + 1, m) along direction d + 4 (mod 6);
10.         Act according to the received message (c, r, h, m);
```

Figure 5.4: The local broadcasting algorithm

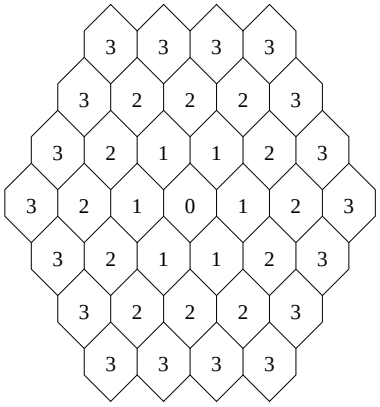

Figure 5.5: The tagging scheme

the clockwise-next direction (lines 8–9). This copy shall be subsequently propagated only in straight line. After the propagation of the message, of course, the relay cell must modify its record about the broadcasting cell according to the message content m (line 10).

5.2.2 Properties

Correctness

Let us consider a setting such as in figure 5.2, and let us label the cells according to their proximity to the central one, as in figure 5.5, with the usual greedy sequential tagging algorithm shown in figure 5.6. From the greedy nature of the algorithm it is clear that every cell is tagged with the minimum number of hops necessary to get to it from the 0-tagged cell.

Definition 5.1. Let $d = 0, \ldots, 5$. The *domain* of a message M sent from cell 0 along direction d, denoted by $D(d)$, is the set of cells that will be reached by some message triggered by M according to the local broadcasting algorithm 5.4.

In other words, the domain of the message sent along direction d by cell 0 is the set

```
1. All cells are initially untagged.
2. Tag 0 the central cell
3. for i = 0 to r do
4.     for each cell c tagged i do
5.         tag i + 1 every untagged neighbor of c
```

Figure 5.6: The tagging algorithm

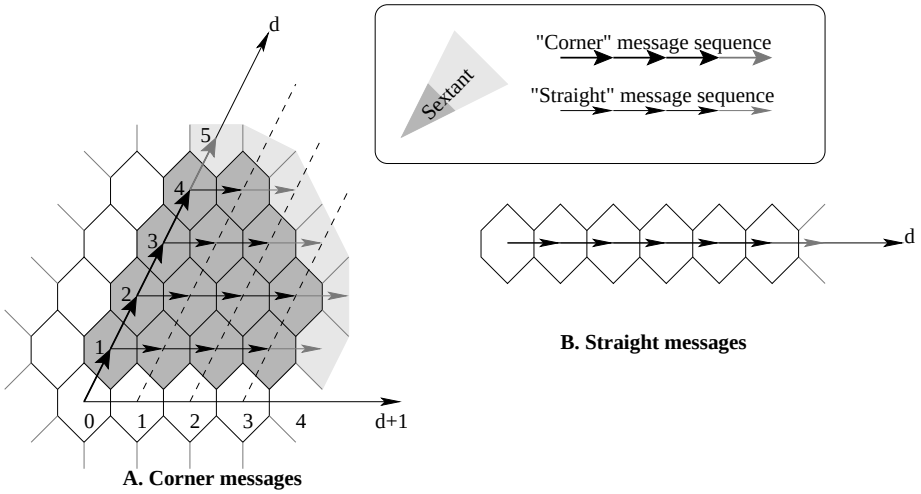

Figure 5.7: The d-th hex coordinate system and its positive sextant

of all cells that would receive a message by the local broadcasting algorithm 5.4 if M were the only message were sent from cell 0. Actually, cell 0 sends six messages, and each of them has its own domain.

Definition 5.2. Let $d = 0, \ldots, 5$, and let c be a cell. The *d-th hex coordinate system of cell c*, denoted by $H(c, d)$, is the reference system whose origin lies in the center of the cell and whose axes have directions d and $d + 1 \pmod 6$.

Definition 5.3. Let $d = 0, \ldots, 5$, and let c be a cell. The *positive sextant* of the d-th hex coordinate system of cell c, denoted by $P(c, d)$, is the set of all cells having a positive d coordinate and a nonnegative $d + 1 \pmod 6$ coordinate:

Lemma 1. *Let $d = 0, \ldots, 5$. The domain of the message sent along direction d by cell 0 is a subset of the positive sextant of the d-th hex coordinate system of cell 0.*

Proof. The target of the message sent along direction d by cell 0 is the cell having coordinates $(1, 0)$ in $H(0, d)$, thus it belongs to $P(0, d)$. Messages originated by a "corner" message in direction d will only take directions d ("corner") or $d+1 \pmod 6$ ("straight"). Since "straight" messages only generate other "straight" messages along the same direction, no other directions will ever be involved. It follows that the $H(0, d)$ coordinates of cells in the domain $D(d)$ can only increase, thus remaining within the positive sextant. □

Lemma 2. *In one run of the local broadcasting algorithm 5.2 every cell receives at most one message.*

Proof. Cell 0 doesn't belong to any positive sextant, so it doesn't belong to any domain of its own generated messages. As a consequence, it will never receive a message during the current run of the algorithm.

Consider cell $c \neq 0$. If it does not belong to any domain, then it never receives a message in the current run.

If the cell belongs to the domain $D(d)$ for some direction $d = 0, \ldots, 5$, then by lemma 1 it belongs to the positive sextant $P(0, d)$. Let $(i, j), i > 0, j \geq 0$, be the coordinates of c in $H(0, d)$. The chain of messages leading to cell c from cell 0 will start with the message sent by cell 0 to cell $(1, 0)$. Every chain is composed by a sequence of "corner" messages (causing the increment of the first coordinate) followed by a sequence of "straight" messages (causing the increment of the second coordinate), thus there is only one path going from $(0, 0)$ to (i, j), i.e. a sequence of i "corner" messages and j "straight" messages. Since the first message is sent only once, the chain will be followed only once, so cell c will receive only one message. □

Lemma 3. *In one run of the algorithm, no cell at distance $r + 1$ or more from cell 0 receives any message.*

Proof. Let us compare the local broadcasting algorithm 5.4 with the greedy tagging algorithm 5.6. Let us call h-tag of cell c the value of field h in the message received by that cell, and assume that cell 0 has a null h-tag.

Every cell c having an h-tag equal to h has received a message from a cell having a h-tag equal to $h - 1$, and this is true for all positive values of h, so that c has been reached from cell 0 in exactly h hops. Since the tagging algorithm 5.6 tags every cell with the smallest number t of hops needed to go from cell 0 to it, we have $t \leq h$ for every cell with an h-tag.

Since the algorithm stops when $h = r$, only cells up to a certain tag $t \leq r$ have been concerned by the message passing. □

Lemma 4. *Consider the greedy tagging algorithm 5.6. Given direction $d = 0, \ldots, 5$, in the hex coordinate system $H(0, d)$ a cell in the positive sextant having coordinates $(i, j), i > 0, j \geq 0$ is tagged $i + j$.*

Proof. Let us proceed by induction on the tag value t. In every hex reference system cell 0 has coordinates $(0, 0)$ and its tag is $t = 0$, so the thesis is true.

Let us consider the thesis true up to a certain value of t in the positive sextant of every hex reference system. Then cells whose coordinates are (i, j), where $i > 0$, $j > 0$ and $i + j \leq t$, have been tagged by the algorithm and no other cell will be tagged t. In the next step, the algorithm will choose the cells to tag $t + 1$. They must be all neighbors of cells tagged t that haven't been tagged yet. This implies that their coordinates (i, j) must verify $i + j > t$, since all the other cells have already been tagged by induction hypothesis.

Given direction d, in the positive sextant of the hex reference system $H(0, d)$ every cell having coordinates $(i, j), where i > 0, j \geq 0 and i + j = t + 1$, has at least one neighbor tagged t, namely the cell $(i - 1, j)$ (if $i > 1$) or the cell $(i, j - 1)$ (if $j > 0$). The only exception is cell $(1, 0)$, which has a neighbor tagged 0 outside the positive sextant. No cell having coordinates (i, j) such that $i + j > t + 1$ is in contact with any cell tagged t or less. □

Theorem 5.1. *The total number of cells being tagged t or less by the greedy tagging algorithm 5.6 is*

$$N_{tag} = 3t(t+1) + 1.$$

Proof. According to lemma 4, in every positive sextant the number of cells having tag $s > 0$ is precisely s, since they correspond to the coordinate set

$$\{(i,j)|i > 0 \wedge j \geq 0 \wedge i + j = s\} = \{(i, k-i)|0 < i \leq s\}.$$

So, in every positive sextant the total number of cells tagged t or less is

$$\sum_{s=1}^{t} s = \frac{t(t+1)}{2}.$$

By multiplying by the number of different sextants and adding the 0-tagged central cell, we obtain the result. □

Theorem 5.2. *Only all the cells up to distance r from the central cell (i.e. cell 0) receive a message in the local broadcasting algorithm 5.4.*

Proof. Let us count the number of messages sent during one run of the algorithm. Consider the algorithm performing in a synchronized way, so that all messages with a certain value of h will be transmitted and received before any message with $h + 1$ is sent. Let us call any of such checkpoints a *step.*

For simplicity consider only one positive sextant; multiplying by six will give the desired result.

At the first step ($h = 1$), the central node will send one message to its adjacent node in direction d. Let us call $T(h)$ the number of messages transmitted during steps from 1 to h. Clearly,

$$T(1) = 1. \tag{5.1}$$

At every subsequent step, the number of messages increases; in particular, at step h only one cell receives (and then relies) a "corner" message, while every past "corner" message triggered a new chain of "straight" messages, each of which generates one new message at every step. At step h there are exactly $h - 1$ previous chains, each one generating a new message, plus a newly generated chain, initiated by the cell that has just received the "corner" message:

$$T(h+1) = T(h) + h. \tag{5.2}$$

By solving the recurrence (equations 5.1 and 5.2 define the sequence of triangular numbers) we have

$$T(h) = \frac{h(h+1)}{2}.$$

Considering all the six sextants, and remembering that the algorithm proceeds until $h = r$, we obtain the total number of messages N_{msg} sent during one run of the algorithm:

$$N_{\text{msg}} = 6T(r) = 3r(r+1). \tag{5.3}$$

Notice that the central cell (cell 0) has not been considered, and since by lemma 2 no cell receives more than one message, the total number of involved cells is exactly $N_{\text{msg}} + 1$, which is precisely the number of cells that are tagged r or less by the greedy tagging algorithm 5.6, as shown in theorem 5.1.

Since, according to lemma 3, no cell with a tag higher than r can receive any message, the proof follows. □

Efficiency

According to lemma 2 no cell receives more than one message, so the total number of messages is minimal.

Let us consider a synchronized setting where every message takes a unit of time to travel and no other action takes time. So every step, being composed of messages traveling at the same time, takes exactly a unit of time to complete, while we do not consider the time required to execute the algorithm code inside a cell. Then every cell having distance h from the center will receive the message at time h. Since messages are restricted to travel between adjacent cells, no shorter time is possible.

The length of the message is as small as possible, since we impose a overhead of only two integers (the maximum distance r and the actual distance h) and a bit (the "corner" tag c). Moreover, while these parameters are useful to discuss the algorithm, none of them is really necessary to make the algorithm work: the value of r is presumably fixed for all the system, the values of h and of c can be computed locally once the central cell is known (presumably its ID is stored inside the message m) by considering its relative position.

Load balance

With the exception of the initial cell, which sends six messages, every other cell must transmit no more than two messages, so the message load for every cell is at most constant (0, 1, 2 or 6), and it is independent from the maximum distance r.

5.3 A Mutual Exclusion technique

5.3.1 The algorithm

To avoid conflicts in channel choice we must ensure that, when a cell is changing its configuration, none of its neighbors up to the reuse distance does the same thing simultaneously. For this we can implement a multiple token-passing protocol such that no two tokens are nearer than the reuse distance.

In the following, we shall call "critical state" the execution of the channel assignment code by the cell. To let cells enter the critical state safely, we let some tokens circulate through the system in such a way to ensure a few properties:

1. every cell receives a token from time to time;
2. two tokens are never nearer than the reuse distance;
3. a cell can enter the critical section only when it possesses the token;
4. a cell must pass the token as soon as possible.

If the reuse distance is two and the network is a regular hexagonal grid, let us refer to Fig. 5.8. The grey cells possess the token; when a cell is done with it, it sends a "token" message "upwards" (following the thick arrow) and two "free" messages along the thin arrows (this requires two other cells to act as relays or custom wiring). Before entering the critical state, a cell must wait for one "token" and two "free" messages. The "token" message ensures that all preceding cells in the token-passing chain are safe, while the two "free" messages declare the safety of the two potentially blocking cells which are possibly using the token at the same time in its neighborhood. To take

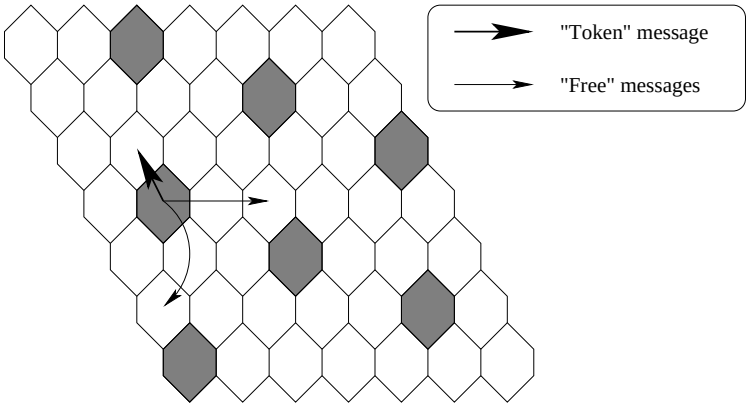

Figure 5.8: A multiple token-passing scheme

account of border effects, however, the two leftmost columns and the two uppermost rows should wait for just one “free” message, the cells in the upper left corner don’t have to wait for any “free” message, while the lower row cells should generate a token (without waiting for one) whenever they get enough “free” messages.

The underlying idea is to have a token composed by three parts, traveling in three different directions. A cell has the token only when it has collected the three messages, and only then it is allowed to enter the critical section and to pass the token. Another way to understand the algorithm is to consider the actual token as if it were carried by the “token” message in an inactive form, and the cell possessing it is allowed to “activate” it and to pass it only after receiving two auxiliary “free” messages. The “free” messages act as a rearguard communication system: cells are allowed to receive and pass the token only after other more advanced tokens have been forwarded.

The actual algorithm for a 7×7 grid with a reuse distance of 2 is presented in Fig. 5.9 (the directions are numbered clockwise from 0 to 5 starting from west). Three variables are used to store the status of the cell. The boolean *Token_flag* is **true** if and only if the cell is holding the token; the counter *Free_count* contains the number of “free” messages that still have to be received before the cell is able to use the token; the flag *Critical_section_flag* is **true** if and only if a channel request or release has been issued in the cell. The entry to the critical section is only allowed when *Token_flag* is **true** and *Free_count* is 0.

All cells must initially call the *Init* procedure to initialize their status, except those initially possessing the token (the grey ones in Fig. 5.8). The cells that initially possess the token must start calling the procedure *Have_token_at_the_beginning*. Cells that need to process a channel request (or release) must call the procedure *Enter_critical_section* that sets the *Critical_section_flag*, so that the algorithm runs the channel-assignment procedure when possible, and waits until that same bit is reset.

Of course the main loop is executed concurrently, and the *Check* procedure must end before any other message is received. For routing purposes, since they have to travel two cells, the “Free” messages have an integer part. When a cell receives a “free(0)” message, it just has to retransmit it as “free(1)” to a direction that depends on the incoming path (lines 30–33), while a “free(1)” message must be operated on place (lines 34–36) by procedure *Check*, that verifies if the cell has to enter the critical

```
1.  Procedure Have_token_at_the_beginning:
2.      Token_flag ← true;
3.      Free_count ← 0;
4.      do Check

5.  Procedure Init:
6.      if node is in the first row then
7.          Token_flag ← true
8.      else
9.          Token_flag ← false;
10.     Free_count ← 2;
11.     if node is in the two upper rows then
12.         decrease Free_count;
13.     if node is in the two leftmost columns then
14.         decrease Free_count;
15.     Critical_section_flag ← false;

16. Procedure Enter_critical_section:
17.     Critical_section_flag ← true;
18.     wait until Critical_section_flag = false;

19. Procedure Check:
20.     if Token_flag and Free_count = 0 then
21.         if Critical_section_flag then
22.             Run the channel assignment procedure;
23.         Send Token along direction 1;
24.         Send Free(0) along directions 3 and 4;
25.         do Init

26. Main polling loop:
27.     Upon receipt of Token do
28.         Token_flag ← true;
29.         do Check
30.     Upon receipt of Free(0) from direction 1 do
31.         Send Free(1) along direction 5;
32.     Upon receipt of Free(0) from direction 0 do
33.         Send Free(1) along direction 3;
34.     Upon receipt of Free(1) do
35.         decrease Free_count;
36.         do Check
```

Figure 5.9: The token-passing algorithm

section, skip it or wait because it isn't ready.

5.3.2 Properties

While the algorithm is relatively simple to describe and to understand, some of its properties can be fairly elusive, mainly because we are not allowed to synchronize the cells. So most interesting results have been obtained by experiments. The algorithm we have described is available as a Java applet embedded in the "local mutual exclusion" page which can be found in the Cellular Channel Assignment Tutorial (see appendix B) at the following address:

`http://www.science.unitn.it/~brunato/radio/`

Let us define the terminology we are going to use next.

Definition 5.4. A *configuration* of the system is defined by the disposition of tokens.

Of course, we can define a partial order in the set of all configurations.

Definition 5.5. Given two configurations, A and B, we say that A is a *subset* of B (and B is a *superset* of A) if every cell possessing a token in configuration A possesses it in configuration B.

Definition 5.6. A *sequence* of system configurations is the sequence of configurations arising from repeated application of the token-passing algorithm 5.9.

Definition 5.7. A system configuration is *empty* if no tokens are in the system.

Of course, an empty configuration leads to a sequence of empty configurations, because no messages are ever sent by any cell.

Definition 5.8. A system configuration is *correct* if tokens are never mutually nearer than the reuse distance.

Note that an empty configuration (no tokens at all) is correct by definition. Anyway, the system does not need to be correct since the beginning to be useful:

Definition 5.9. A sequence of configurations is *ultimately correct* if it contains only a finite number of incorrect configurations, i.e. if there exists a time T such that all configurations after time T are correct.

The first fact arising from extensive simulation of the algorithm is the following.

Fact 1. *Every initial configuration evolves to an ultimately correct sequence.*

That is, if we start by assigning tokens at random, without respecting any rule, then we run the algorithm, we shall find only a finite number of incorrect configurations, after which the system will only show correct, and eventually empty, token dispositions.

By starting from a random token assignment, then, the probability of coming to a sequence of correct configurations is 1. What is the chance that, starting from a random assignment, the system evolves to a nonempty (i.e. working) sequence? The histogram in figure 5.10 illustrates the results of some experiments where the probability of obtaining a working configuration is plotted against the initial filling factor, i.e. the probability that a cell receives a token in the initial configuration. If, for instance, a

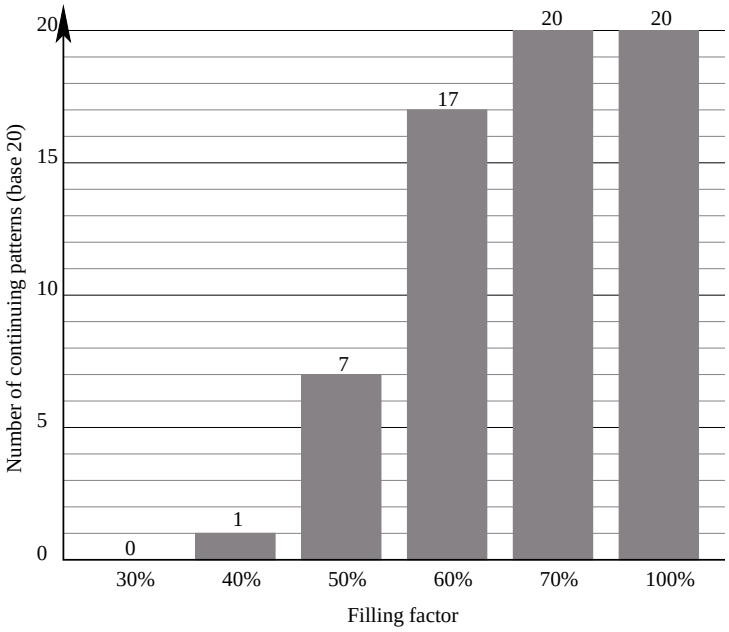

Figure 5.10: Fraction of self-stabilizing random configurations

cell is initially given the token with a .5 probability, the system has only a $7/20$ probability of developing a working configuration. However, if all cells initially receive a token then the system will very probably develop a working configuration.

Indeed, in our experiments it never happened that a completely filled system resulted in an empty configuration. This can be explained by the following argument. Let configuration B be a superset of configuration A. Every message sent in configuration A is also sent in B, possibly in different order, which ensures that if a cell develops a token at a certain time starting from configuration A it will develop it also starting from B, possibly earlier, due to the larger number of messages. Thus we have the following:

Proposition 5.1. *Given an initial configuration A leading to a nonempty ultimately correct sequence of configurations, every superset of A leads to a nonempty ultimately correct sequence of configurations.*

The fact stated by proposition 5.1, together with the fact that the partial order relation among configurations is a lattice (i.e. closed with respect to the maximum), explains why a completely filled configuration evolves into a ultimately correct nonempty sequence: it is a superset of any other initial configuration. Moreover, it explains why the histogram of figure 5.10 is growing.

Proposition 5.1 also gives a hint about the fault tolerance of the system: if a cell develops an unwanted token, the system will eventually return to a working configuration. On the contrary, if a token is lost in a correct configuration, there won't be enough messages to maintain a nonempty sequence.

Chapter 6

Conclusions

6.1 Achieved Goals

6.1.1 Heuristics

In chapter 3 we have applied the Reactive Search heuristic to the Channel Assignment problem, seen as a generalization of the Graph Coloring problem. The heuristic has shown its strength against other simpler techniques. Its strength lies in the heavy use of the past search history to trim the parameters and to fit the search to the particular configuration space.

6.1.2 The BBB Algorithm

The function we chose to minimize in chapter 4 achieves a double goal: finding its global minimum is straightforward, yet it captures enough of the complexity of the problem to challenge the best combinatorial algorithms we know. Moreover, the function could be computed locally, with a small amount of communication among base stations.

To show this we have introduced the penalty-function algorithm BBB, simulating it over a large number of runs. As we have seen, our heuristic behaves better than the others we tried, at least when compared on regular hexagonal patterns.

In chapter 5 we have shown how the distributed version dBBB could be built, by providing the necessary synchronization mechanisms.

6.2 Future Developments

The interference model adopted by almost every paper about the channel assignment subject is a *binary* one, in the sense that constraints always concern *couples* of transmitters. However, real-world interference is *additive*, in the sense that also the number of interfering stations should be taken into account. Consider the situation depicted in figure 6.1, where cell A is far enough from cells 1, 2, 3 and 4 to be allowed to use the same channels. What happens if cells 1, 2, 3 and 4 all use the same channel? It happens that, even though every single cell would not be able to generate a signal strong enough to interfere with cell A, the combined power of the four cells would sum up and cause a significant decrease of the signal to noise ratio for that channel in cell A. So,

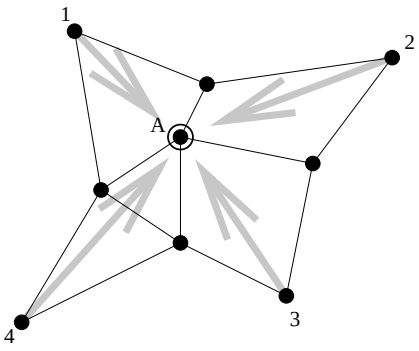

Figure 6.1: The additive interference constraint

binary constraints are not powerful enough to describe many real world phenomena, and other structures should be employed. For example, an interference graph could be validly replaced by a hypergraph.

Due to their nature, most algorithms have only been studied with such restrictive assumptions as boolean (non-additive) interference, and so our tests had to restrict to that case. However, our method can easily be applied to a large variety of channel assignment problems with additive interference constraints, adjacent-channel interference and so on. In fact, all that is required for the minimization to be polynomial is that every channel provides a *linear* contribution to the total penalty function.

A great amount of work must also be carried out to complete the theoretical analysis of the distributed algorithms in sections 5.2, page 32 and 5.3, page 38. Of course, theory of distributed systems is still at its early stages, and it does not provide, at present, a large theoretical background, if not in special cases.

Appendix A

An Integrated Library for Local Search

A.1 Introduction

We present a powerful, understandable and upgradable library for local search problem solving. We introduce an actual problem and we illustrate its solution by means of the library.

A.1.1 How to read the code

We have written the library code in C++, using the literate programming noweb system, which permits to write a single source file both for programming and for human-readable documentation; the latest has been included as follows.

The concept of *literate programming* has been introduced by Donald E. Knuth[Knu92] in many publications. The program is divided into code chunks that can be nested; every code chunk has a name, and it is identified by a unique string (possibly its description) and a code formed by the number of the page where it appears eventually followed by a letter to distinguish it from other code chunks on the same page. A code chunk that contains another code chunk refers to it by its identifier (string and code). Of course, a program is given to create a machine-understandable code from this system. This way, every logical building block, be it a separate function or not, can be described separately, and its position within the whole program can be easily traced.

As an example, consider the following *hello world* program.

At first we give the overall skeleton of the program.

45a ⟨*helloWorld.c* 45a⟩≡

```
⟨inclusions 45b⟩
int main (void)
{
  ⟨function call to write the hello world message 46a⟩
}
```

Then we define the code blocks that have been cited in the above chunk.

45b ⟨*inclusions* 45b⟩≡

```
#include <stdio.h>
```

46a ⟨*function call to write the hello world message* 46a⟩≡
```
printf ("Hello, World!\n");
```

A.2 The problem descriptor skeleton: the `GenericProblem` class

The first class we describe is a virtual class containing the definition which are common to most binary-string local-search problems, for instance a binary (boolean) array to contain the current string, another array to store the best found solution, the number of bits, a fitness function, a random string generation function. We also need a method to identify legal moves (i.e., to describe the *neighborhood* of the current string.

A.2.1 The header file `GenericProblem.H`

The header file includes a few standard libraries and declares the `GenericSolver` class, in order to allow a pointer to a `GenericSolver` object to be stored. The `GenericProblem` class does not contain private members, since it is a virtual class and must be extended before use.

46b ⟨*GenericProblem.H* 46b⟩≡
```
#ifndef _GENERICPROBLEM_H
#define _GENERICPROBLEM_H

#include <iostream.h>
#include "Rand48.h"

class GenericSolver;

class GenericProblem {
protected:
  ⟨GenericProblem - protected definitions 46c⟩
public:
  ⟨GenericProblem - public definitions 47a⟩
};
#endif
```

The protected definitions contain all member variables, which will be accessed by foreign code only by interface functions.

46c ⟨*GenericProblem - protected definitions* 46c⟩≡
```
GenericSolver *solver;
bool * const x, * const bestX;
int nbits, fitness;
int seed;
Rand48 rand;
```

The public `status` enumeration type is used as a response to declare if a move is feasible. All other public members of the class are functions: the constructor and destructor, the interface functions to operate on the protected members and the more comples functions to manage the local search of a problem solution.

47a ⟨*GenericProblem - public definitions* 47a⟩≡
```
enum status {
  PROHIBITED, INVALID, OK
};
```
⟨*GenericProblem - constructor and destructor* 47b⟩
⟨*GenericProblem - inline interface functions* 47c⟩
⟨*GenericProblem - problem management functions* 47d⟩

The construction of the problem requires only the length of the binary string and, optionally, a seed for the random number generator.

47b ⟨*GenericProblem - constructor and destructor* 47b⟩≡
```
GenericProblem (int, int = 777);
virtual ~GenericProblem ();
```

Most problem parameters can be accessed through the following functions.

47c ⟨*GenericProblem - inline interface functions* 47c⟩≡
```
int getNbits () const { return nbits; }
int  getSeed () const { return seed; }
bool operator[] (int i) const { return x[i]; }
void setSolver (GenericSolver *s) { solver = s; }
virtual void storeAsBest ()
  { memcpy (bestX, x, nbits * sizeof(bool)); }
```

The functions that manage the navigation through problem states are now introduced. Notice that all the functions are pure virtual, since they all depend on the specific nature of the problem.

47d ⟨*GenericProblem - problem management functions* 47d⟩≡
```
virtual int getMoves () const = 0;
virtual status randomConfiguration (long &) = 0;
virtual status newFitness (int, long &) const = 0;
virtual void execMove (int, long &) = 0;
virtual void writeCurrent (ostream& = cout) const;
virtual void writeBest (ostream& = cout) const;
```

A.2.2 The library file `GenericProblem.C`

The implementation of the `GenericProblem` class is very small, in fact most functions are pure virtual.

47e ⟨*GenericProblem.C* 47e⟩≡
```
#include "GenericProblem.H"
#include "GenericSolver.H"
```

⟨*GenericProblem - constructor and destructor implementation* 48a⟩
⟨*GenericProblem - Debugging tools* 48b⟩

The constructor of the class will just set some parameters and reserve the memory for the configuration strings. The destructor will free it. Both functions will usually be overridden by extensions of the class.

48a ⟨*GenericProblem - constructor and destructor implementation* 48a⟩≡

```
GenericProblem::GenericProblem (int n, int s):
  x (new bool[n]),
  bestX (new bool[n]),
  nbits (n),
  seed (s),
  rand (s)
{}

GenericProblem::~GenericProblem ()
{
  delete [] x;
  delete [] bestX;
}
```

The only generic functions we provide implement a very raw output of the stored binary strings on an output stream provided by the user.

48b ⟨*GenericProblem - Debugging tools* 48b⟩≡

```
void GenericProblem::writeCurrent (ostream &o) const
{
  for ( int i = 0; i < nbits; i++ )
    o << (x[i]?'T':'F');
}

void GenericProblem::writeBest (ostream &o) const
{
  for ( int i = 0; i < nbits; i++ )
    o << (bestX[i]?'T':'F');
}
```

A.3 The problem solver skeleton: the `GenericSolver` class

Next we define a class which stores the common features of prohibition-based local search techniques, such as a move-generation mechanism and a past history storage. Anyway, the prohibition mechanism can be ignored by making it allow any move towards legal configurations, so that prohibition-independent search schemes can also be implemented as extensions of this class.

A.3.1 The header file `GenericSolver.H`

This header declares the class `GenericProblem`, in order to include a reference to an instance in a `GenericSolver` object. The class `GenericSolver` itself contains protected variables and public functions.

49a ⟨*GenericSolver.H* 49a⟩≡

```
#ifndef _GENERICSOLVER_H
#define _GENERICSOLVER_H

#include "Rand48.h"

class GenericProblem;

class GenericSolver {
protected:
  ⟨GenericSolver - protected definitions 49b⟩
public:
  ⟨GenericSolver - public definitions 49c⟩
};

#endif
```

The protected members are a reference to the problem being solved, a random number generator, the size of the binary string, the current iteration, the iteration at which the best was found, the current fitness and the best fitness found up to now.

49b ⟨*GenericSolver - protected definitions* 49b⟩≡

```
GenericProblem  &problem;
Rand48 rand;
const int nbits;
int currentIteration, bestT;
long fitness, bestFitness;
```

The public members of the class are functions.

49c ⟨*GenericSolver - public definitions* 49c⟩≡

```
⟨GenericSolver - constructor and destructor 50a⟩
⟨GenericSolver - inline interface functions 50b⟩
⟨GenericSolver - problem solving functions 50c⟩
```

The constructor of the class requires to receive a reference to the problem it shall operate upon and an optional random seed. The destructor is not necessary in this class, since heap is not used explicitly, but is declared as a virtual function in order to define it in the derived classes. At present, it is declared as an inline empty function.

50a ⟨*GenericSolver - constructor and destructor* 50a⟩≡
```
GenericSolver (GenericProblem &, int = 777);
virtual ~GenericSolver () {}
```

Interface functions are used to extract information from the class, because all variables are protected.

50b ⟨*GenericSolver - inline interface functions* 50b⟩≡
```
int  getT           () const { return currentIteration; }
int  getBestT       () const { return bestT; }
int  getIteration   () const { return currentIteration; }
long getFitness     () const { return fitness; }
long getBestFitness () const { return bestFitness; }
```

All functions that actually solve the problem are algorithm-specific, so we just declare them as pure virtual.

50c ⟨*GenericSolver - problem solving functions* 50c⟩≡
```
virtual bool tabu (int) const = 0;
virtual void flip (int) = 0;
virtual void start () = 0;
virtual void restart () = 0;
virtual int bestMove (long int &) = 0;
virtual void move (int) = 0;
virtual void step () = 0;
virtual void run () = 0;
```

A.3.2 The library file `GenericSolver.C`

The only function to be implemented out of line is the constructor, which uses the parameters (a reference to the `GenericProblem` object describing the problem to be solved, and an optional random seed) to initialize the protected members to the right value.

50d ⟨*GenericSolver.C* 50d⟩≡
```
#include "GenericSolver.H"
#include "GenericProblem.H"

GenericSolver::GenericSolver (GenericProblem &prob, int seed = 777) :
  problem (prob),
  rand (seed),
  nbits (prob.getNbits()),
  currentIteration (0),
  bestT (0)
{
  problem.setSolver(this);
}
```

A.4 The implementation of a problem: the `AssignmentProblem` class

Now we are ready to implement a problem definition by extending the `GenericProblem` class.

A.4.1 The header file `AssignmentProblem.H`

51a ⟨*AssignmentProblem.H* 51a⟩≡

```
#ifndef _ASSIGNMENTPROBLEM_H
#define _ASSIGNMENTPROBLEM_H

#include "GenericProblem.H"

class AssignmentProblem: public GenericProblem {
private:
  ⟨AssignmentProblem - private definitions 51b⟩
public:
  ⟨AssignmentProblem - public definitions 51c⟩
};

#endif
```

We declare all parameters used to define the problem, as we defined it in the course of our work. A couple of private functions are also defined.

51b ⟨*AssignmentProblem - private definitions* 51b⟩≡

```
const int rows, cols, nce, nch, traffic, free, kappa, swaps, nmoves;
int * const usedChannels, * const bestUsedChannels,
  * const freeChannels, * const penalties, * const adjacency;
void writeVector (ostream&, const int*) const;
long int fitnessFromScratch ();
```

51c ⟨*AssignmentProblem - public definitions* 51c⟩≡

⟨*AssignmentProblem - constructor and destructor* 51d⟩
⟨*AssignmentProblem - implementation of previously virtual functions* 52a⟩

We define a more complex constructor; also the destructor will be modified, as many dynamic allocations are required to store all problem data.

51d ⟨*AssignmentProblem - constructor and destructor* 51d⟩≡

```
AssignmentProblem (int, int, int, int, int, int = 777);
~AssignmentProblem ();
```

Functions that were declared as pure virtual in the base class are now redeclared in order to implement it in the `.C` file.

52a ⟨*AssignmentProblem - implementation of previously virtual functions* 52a⟩≡

```
int getMoves () const { return nmoves; }
status randomConfiguration (long&);
status newFitness (int, long&) const;
void execMove (int, long&);
void storeAsBest ();
void writeCurrent (ostream& = cout) const;
void writeBest (ostream& = cout) const;
```

A.4.2 The library file `AssignmentProblem.C`

52b ⟨*AssignmentProblem.C* 52b⟩≡

```
#include "AssignmentProblem.H"
#include "GenericSolver.H"

⟨AssignmentProblem - utility functions 52c⟩
⟨AssignmentProblem - constructor implementation 53a⟩
⟨AssignmentProblem - destructor implementation 54b⟩
⟨AssignmentProblem - Initialize the problem with a random configuration 55a⟩
⟨AssignmentProblem - Incremental fitness evaluation 56b⟩
⟨AssignmentProblem - Execution of a move 57a⟩
⟨AssignmentProblem - best value storage 57b⟩
⟨AssignmentProblem - various debugging tools 58⟩
⟨AssignmentProblem - Calculate fitness of a new configuration 59⟩
```

Some generic functions are declared to help calculations.

52c ⟨*AssignmentProblem - utility functions* 52c⟩≡

```
template <class T> T MAX (T a, T b)
{
  return (a > b) ? a : b;
}

template <class T> T MIN (T a, T b)
{
  return (a < b) ? a : b;
}

int abs (int a)
{
  return (a > 0) ? a : -a;
}
```

The constructor requires the number of rows and columns in the hexagonal tile, the total number of channels, the traffic parameter, the radius of interference and an optional seed for the random number generator.

53a ⟨*AssignmentProblem - constructor implementation* 53a⟩≡

```
AssignmentProblem::AssignmentProblem (int r, int c, int ch, int t,
                int k, int s = 777) :
  ⟨AssignmentProblem constructor - Initialization list 53b⟩
{
  ⟨AssignmentProblem contructor - compute the adjacency matrix 54a⟩
}
```

Private members of the class are initialized by an initialization list. The base class constructor is called first.

53b ⟨*AssignmentProblem constructor - Initialization list* 53b⟩≡

```
GenericProblem (r*c*ch, s),
rows (r),
cols (c),
nce (r*c),
nch (ch),
traffic (t),
free (nch-traffic),
kappa (k+1),
swaps (traffic*free),
nmoves (nce*traffic*free),
usedChannels (new int [nce*traffic]),
bestUsedChannels (new int [nce*traffic]),
freeChannels (new int [nce*free]),
penalties (new int [nce*nch]),
adjacency (new int [nce*nce])
```

The second task of the constructor is the initialization of the adjacency matrix according to the parameters.

54a ⟨*AssignmentProblem contructor - compute the adjacency matrix* 54a⟩≡

```
for ( int i = 0; i < r; i++ )
  for ( int j = 0; j < c; j++ ) {
    int
      ipos = i * c + j,
      jpos = ipos,
      pos = ipos * nce + jpos,
      p = kappa;
    for ( int k = 0; k < c-j; k++ ) {
      adjacency[pos++] = (p>=0)?p:0;
      p--;
    }
    p = kappa;
    int
      pcol = j,
      ncol = 1;
    for ( int i1 = i+1; i1 < r; i1++ ) {
      if ( p > 0 )
        p--;
      if ( !(i1 % 2) )
        pcol--;
      ncol++;
      int j1;
      for ( j1 = 0; j1 < pcol; j1++ )
        adjacency[pos++] = (p > pcol-j1) ? (p-pcol+j1) : 0;
      for ( ; j1 < c && j1 < pcol+ncol; j1++ )
        adjacency[pos++] = p;
      for ( ; j1 < c; j1++ )
        adjacency[pos++] = (p > j1-ncol-pcol+1) ? (p-j1+ncol+pcol-1) : 0;
    }
  }
for ( int i = 1; i < nce; i++ )
  for ( int j = 0; j < i; j++ )
    adjacency [i*nce+j] = adjacency[j*nce+i];
```

The destructor frees all the reserved heap memory.

54b ⟨*AssignmentProblem - destructor implementation* 54b⟩≡

```
AssignmentProblem::~AssignmentProblem ()
{
  delete [] usedChannels;
  delete [] freeChannels;
  delete [] penalties;
  delete [] adjacency;
}
```

Usual solving sessions require the generation of a random string. Since not all strings are legal, the work is devolved to the problem descriptor (it is the only object that "knows" something about the nature of the problem). The function returns the configuration status (always OK) and its fitness, through the reference parameter.

55a ⟨*AssignmentProblem - Initialize the problem with a random configuration* 55a⟩≡

```
GenericProblem::status AssignmentProblem::randomConfiguration (long &f)
{
  ⟨GenericProblem random configuration - initialize an empty vector 55b⟩
  ⟨GenericProblem random configuration - fill the vector with a legal binary string 55c⟩
  ⟨GenericProblem random configuration - Calculate the initial fitness 56a⟩
  return OK;
}
```

55b ⟨*GenericProblem random configuration - initialize an empty vector* 55b⟩≡

```
for ( int i = 0; i < nbits; i++ ) {
  x[i] = false;
  penalties[i] = 0;
}
```

55c ⟨*GenericProblem random configuration - fill the vector with a legal binary string* 55c⟩≡

```
bool *xI = x;
int
  *usedI = usedChannels,
  *adjI = adjacency,
  *freeI = freeChannels;
for ( int i = 0; i < nce; i++ ) {
  for ( int j = 0; j < traffic; j++ ) {
    int ch;
    do
      ch = rand.asUnsignedInt() % nch;
    while ( xI[ch] );
    xI[ch] = true;
    *(usedI++) = ch;
    int *penI = penalties;
    for ( int i1 = 0; i1 < nce; i1++ ) {
      if ( adjI[i1] ) {
        const int
          chmin = MAX (0, ch-adjI[i1]+1),
          chmax = MIN (nch, ch+adjI[i1]);
        for ( int j1 = chmin; j1 < chmax;
          penI[j1++]++ );
      }
      penI += nch;
    }
  }
  for ( int j = 0; j < nch; j++ )
    if ( !*(xI++) )
      *(freeI++) = j;
  adjI += nce;
}
```

56a ⟨*GenericProblem random configuration - Calculate the initial fitness* 56a⟩≡

```
fitness = 0;
for ( int i = 0; i < nbits; i++ )
  if ( x[i] )
    fitness += --penalties[i];
fitness /= 2;
f = fitness;
```

56b ⟨*AssignmentProblem - Incremental fitness evaluation* 56b⟩≡

```
GenericProblem::status AssignmentProblem::newFitness (int m, long &f) con
{
  int
    cell = m / swaps,
    whichSwap = m % swaps,
    fromPlace = whichSwap / free,
    toPlace = whichSwap % free,
    fromChannel = usedChannels[cell*traffic+fromPlace],
    toChannel = freeChannels[cell*free+toPlace],
    fromBit = cell*nch + fromChannel,
    toBit = cell*nch + toChannel,
    change = penalties[toBit] - penalties[fromBit]
    - ((abs(fromChannel-toChannel)<adjacency[cell*(nce+1)]) ? 1 : 0);
  f = fitness + change;
  return (solver->tabu(fromBit) || solver->tabu(toBit)) ?
    GenericProblem::PROHIBITED : GenericProblem::OK;
}
```

57a ⟨*AssignmentProblem - Execution of a move* 57a⟩≡

```
void AssignmentProblem::execMove (const int m, long &f)
{
  const int
    cell = m / swaps,
    whichSwap = m % swaps,
    fromPlace = whichSwap / free,
    toPlace = whichSwap % free,
    fromChannel = usedChannels[cell*traffic+fromPlace],
    toChannel = freeChannels[cell*free+toPlace],
    fromBit = cell*nch + fromChannel,
    toBit = cell*nch + toChannel,
    change = penalties[toBit] - penalties[fromBit] -
    ((abs(fromChannel-toChannel)<adjacency[cell*(nce+1)])? 1 : 0);
  f = fitness = fitness + change;
  int *adjI = adjacency + cell*nce;
  for ( int i = 0; i < nce; i++ ) {
    int * const penI = penalties + i*nch;
    if ( *adjI ) {
      int
  chmin = MAX (0, fromChannel-*adjI+1),
  chmax = MIN (nch, fromChannel+*adjI);
      for ( int j = chmin; j < chmax; penI[j++]-- );

      chmin = MAX (0, toChannel-*adjI+1),
  chmax = MIN (nch, toChannel+*adjI);
      for ( int j = chmin; j < chmax; penI[j++]++ );
    }
    adjI++;
  }
  penalties[cell*nch+fromChannel]++;
  penalties[cell*nch+toChannel]--;
  freeChannels[cell*free+toPlace] = fromChannel;
  usedChannels[cell*traffic+fromPlace] = toChannel;
  const int
    fc = cell*nch+fromChannel,
    tc = cell*nch+toChannel;
  x[fc] = false;
  solver->flip (fc);
  x[tc] = true;
  solver->flip (tc);
  f = fitness;
}
```

57b ⟨*AssignmentProblem - best value storage* 57b⟩≡

```
void AssignmentProblem::storeAsBest()
{
  memcpy (bestX, x, nbits*sizeof(bool));
  memcpy (bestUsedChannels, usedChannels, nce*traffic*sizeof(int));
}
```

58 ⟨*AssignmentProblem - various debugging tools* 58⟩≡

```
void AssignmentProblem::writeVector (ostream &o, const int *v) const
{
  for ( int i = 0; i < rows; i++ ) {
    if ( !(i%2) )
      cout << "   ";
    for ( int j = 0; j < cols; j++ ) {
      const int cell = i*cols+j;
      o.form ("(%2d", v[cell*traffic]);
      for ( int k = 1; k < traffic; k++ )
  o.form (",%2d", usedChannels[cell*traffic+k]);
      o << ") ";
    }
    o << endl;
  }
}


void AssignmentProblem::writeCurrent (ostream &o) const
{
  writeVector (o, usedChannels);
}


void AssignmentProblem::writeBest (ostream &o) const
{
  writeVector (o, bestUsedChannels);
}
```

59 ⟨*AssignmentProblem - Calculate fitness of a new configuration* 59⟩≡

```
long int AssignmentProblem::fitnessFromScratch ()
{
  long f = 0;
  int *adjI = adjacency, *usedI = usedChannels, *freeI = freeChannels, *usedI1;
  bool * xI = x;
  for ( int i = 0; i < nce; i++ ) {
    for ( int j = 0; j < traffic; j++ )
      if ( !xI[usedI[j]] )
        return -1;
    for ( int j = 0; j < free; j++ )
      if ( xI[freeI[j]] )
        return -2;
    int c = 0;
    for ( int j = 0; j < nch; j++ )
      if ( xI[j] )
        c++;
    if ( c != traffic )
      return -3;
    usedI1 = usedI;
    for ( int i1 = i; i1 < nce; i1++ ) {
      if ( adjI[i1] )
        for ( int j = 0; j < traffic; j++ )
          for ( int j1 = ((i==i1)?(j+1):0);
                j1 < traffic; j1++ )
            if ( abs(usedI[j]-usedI1[j1])
                 < adjI[i1] )
              f++;
      usedI1 += traffic;
    }
    adjI += nce;
    usedI += traffic;
    freeI += free;
    xI += nch;
  }
  return f;
}
```

A.5 Implementation of a solver: the `ReactiveSearch` class

A.5.1 The header file `ReactiveSearch.H`

60a ⟨*ReactiveSearch.H* 60a⟩≡

```
#ifndef _REACTIVESEARCH_H
#define _REACTIVESEARCH_H

#include <math.h>
#include "Rand48.h"
#include "GenericSolver.H"
#include "History.H"

⟨ReactiveSearch - exceptions 60b⟩

class ReactiveSearch: public GenericSolver {
private:
  ⟨ReactiveSearch - private variables 61a⟩
  ⟨ReactiveSearch - private functions 61b⟩
public:
  ⟨ReactiveSearch - public definitions 61c⟩
};

#endif
```

60b ⟨*ReactiveSearch - exceptions* 60b⟩≡

```
class IntegerException {
private:
  const long int n;
public:
  IntegerException (long int num) : n (num) {}
  long int getN () const { return n; }
};


class NoAllowedMoves: public IntegerException {
public:
  NoAllowedMoves (long int num) : IntegerException (num) {}
};


class ProhibitionTooLarge: public IntegerException {
public:
  ProhibitionTooLarge (long int num) : IntegerException (num) {}
};
```

61a ⟨*ReactiveSearch - private variables* 61a⟩≡
```
History history;
const bool aspiration;
const int initialProhibition, maxVisits,
  maxFrequentlyVisited, maxMoves, maxChecks;
const double increase, decrease;
int * const timeChangedVar;
int * const varList, prohibition,
  nFrequentlyVisited, tProhibitionChanged, tEscape;
double movingAverage;
long long int * const frequentlyVisited;
```

61b ⟨*ReactiveSearch - private functions* 61b⟩≡
```
void timeChangedVarClear ();
bool reaction ();
void increaseProhibition ();
void decreaseProhibition ();
bool problemConsistency ();
void updateBestFitness ();
```

61c ⟨*ReactiveSearch - public definitions* 61c⟩≡
```
ReactiveSearch (GenericProblem &, int, int, int = 777);
virtual ~ReactiveSearch();
bool tabu (int) const;
void flip (int);
int getProhibition () const { return prohibition; }
void start ();
void restart ();
int bestMove (long&);
void move (int);
void step ();
void run ();
```

A.5.2 The library file `ReactiveSearch.C`

62a ⟨*ReactiveSearch.C* 62a⟩≡

```
#include <iostream.h>
#include <stdio.h>
#include "ReactiveSearch.H"
#include "GenericProblem.H"

⟨ReactiveSearch - Constructor 62b⟩
⟨ReactiveSearch - destructor 63a⟩
⟨ReactiveSearch - reset modification times 63b⟩
⟨ReactiveSearch - check if a variable is tabu 63c⟩
⟨ReactiveSearch - change the value of a bit 63d⟩
⟨ReactiveSearch - initialize the search 63e⟩
⟨ReactiveSearch - restart the search from a new configuration 64a⟩
⟨ReactiveSearch - find the move with the best increase of the objective function 64b⟩
⟨ReactiveSearch - perform a move 64c⟩
⟨ReactiveSearch - update the best fitness 65a⟩
⟨ReactiveSearch - perform a search step 65b⟩
⟨ReactiveSearch - execute a number of steps 65c⟩
⟨ReactiveSearch - check whether it's time to change parameters 66⟩
⟨ReactiveSearch - modify the prohibition times 67a⟩
⟨ReactiveSearch - check if the problem description is consistent 67b⟩
```

62b ⟨*ReactiveSearch - Constructor* 62b⟩≡

```
ReactiveSearch::ReactiveSearch (GenericProblem &prob, int mv,
        int chk, int seed) :
  GenericSolver (prob, seed),
  history (nbits),
  aspiration (true),
  initialProhibition (1),
  maxVisits (3),
  maxFrequentlyVisited (3),
  maxMoves (mv),
  maxChecks (chk),
  increase (1.1),
  decrease (0.9),
  timeChangedVar (new int[problem.getNbits()]),
  varList (new int [nbits]),
  prohibition (initialProhibition),
  tProhibitionChanged (0),
  tEscape (0),
  movingAverage (-1.0),
  frequentlyVisited (new long long int [maxFrequentlyVisited +1])
{}
```

63a ⟨*ReactiveSearch - destructor* 63a⟩≡
```
ReactiveSearch::~ReactiveSearch ()
{
  delete [] varList;
  delete [] timeChangedVar;
  delete [] frequentlyVisited;
}
```

63b ⟨*ReactiveSearch - reset modification times* 63b⟩≡
```
void ReactiveSearch::timeChangedVarClear ()
{
  for ( int n = 0; n < nbits; n++ )
    timeChangedVar[n] = -MAXINT;
}
```

63c ⟨*ReactiveSearch - check if a variable is tabu* 63c⟩≡
```
bool ReactiveSearch::tabu (int var) const
{
  return !problem[var]
    && timeChangedVar[var] >= currentIteration - prohibition;
}
```

63d ⟨*ReactiveSearch - change the value of a bit* 63d⟩≡
```
void ReactiveSearch::flip (int var)
{
  history.updateCurrentX (var, currentIteration);
  timeChangedVar[var] = currentIteration;
}
```

63e ⟨*ReactiveSearch - initialize the search* 63e⟩≡
```
void ReactiveSearch::start ()
{
  problem.randomConfiguration (fitness);
  bestFitness = fitness;
  problem.storeAsBest ();
  bestT = 0;
  currentIteration = 0;
  nFrequentlyVisited = 0;
  prohibition = initialProhibition;
  history.clear();
  for ( int nn = 0; nn < nbits; nn++ )
    if ( problem[nn] )
      history.updateCurrentX (nn, currentIteration);
  timeChangedVarClear();
}
```

64a ⟨*ReactiveSearch - restart the search from a new configuration* 64a⟩≡

```
void ReactiveSearch::restart ()
{
  problem.randomConfiguration (fitness);
  history.clear();
  for ( int nn = 0; nn < nbits; nn++ )
    if ( problem[nn] )
      history.updateCurrentX (nn, currentIteration);
  timeChangedVarClear();
  nFrequentlyVisited = 0;
  prohibition = initialProhibition;
  updateBestFitness ();
  currentIteration++;
}
```

64b ⟨*ReactiveSearch - find the move with the best increase of the objective function* 64b⟩≡

```
int ReactiveSearch::bestMove (long &minF)
{
  int bestCandidate = -1;
  long f;
  minF = MAXINT;
  int m;
  GenericProblem::status s;
  for ( int nm = 0;
  nm < maxChecks
    || (bestCandidate < 0 && nm < problem.getMoves());
  nm++ ) {
    m = rand.asUnsignedInt() % problem.getMoves();
    s = problem.newFitness (m, f);
    if ( (s == GenericProblem::OK
    || (aspiration
        && s == GenericProblem::PROHIBITED
        && f < bestFitness))
   && f < minF )
      {
  bestCandidate = m;
  minF = f;
      }
  }
  return bestCandidate;
}
```

64c ⟨*ReactiveSearch - perform a move* 64c⟩≡

```
void ReactiveSearch::move (int m)
{
  problem.execMove (m, fitness);
  updateBestFitness ();
  currentIteration++;
}
```

65a ⟨*ReactiveSearch - update the best fitness* 65a⟩≡

```
void ReactiveSearch::updateBestFitness ()
{
  if ( fitness < bestFitness ) {
    bestFitness = fitness;
    problem.storeAsBest ();
    bestT = currentIteration;
  }
}
```

65b ⟨*ReactiveSearch - perform a search step* 65b⟩≡

```
void ReactiveSearch::step ()
{
  long f;
  if ( reaction() ) {
    tEscape = currentIteration;
    restart ();
  }
  int m = bestMove (f);
  if ( m < 0 ) {
    restart ();
    m = bestMove (f);
  }
  move (m);
}
```

65c ⟨*ReactiveSearch - execute a number of steps* 65c⟩≡

```
void ReactiveSearch::run ()
{
  start ();
  while ( currentIteration < maxMoves && fitness > 0 )
    step ();
}
```

66 ⟨*ReactiveSearch - check whether it's time to change parameters* 66⟩≡

```
bool ReactiveSearch::reaction ()
{
  int lastTime, nVisits;
  if ( history.lookupCurrent(&lastTime, &nVisits) ) {
    if ( nVisits >= maxVisits ) {
      int i;
      long long int fp = history.getCurrentFingerprint();
      for ( i = 0; i < nFrequentlyVisited && frequentlyVisited[i] != fp;
      if ( i == nFrequentlyVisited ) {
  if ( nFrequentlyVisited >= maxFrequentlyVisited ) {
    nFrequentlyVisited = 0;
    movingAverage = -1.0;
    return true;
  }
  frequentlyVisited[nFrequentlyVisited++] = fp;
      }
    }
    const int repetitionInterval = currentIteration - lastTime;
    if ( repetitionInterval < 2*(nbits - 1) ) {
      if( movingAverage > 0 )
  movingAverage = 0.1 * repetitionInterval + 0.9 * movingAverage;
      else
  movingAverage =  repetitionInterval;
      increaseProhibition ();
    }
  }
  else
    history.installCurrent();
  const int interval = currentIteration - tProhibitionChanged;
  if ( interval > 2*movingAverage && interval > nbits )
    decreaseProhibition ();
  return false;
}
```

67a ⟨*ReactiveSearch - modify the prohibition times* 67a⟩≡

```
void ReactiveSearch::increaseProhibition ()
{
  const int newProhibition = int(prohibition * increase);
  if ( prohibition == newProhibition )
    prohibition++;
  else
    prohibition = newProhibition;
  if ( prohibition > MAXINT || prohibition < 0 )
    prohibition = MAXINT;
  tProhibitionChanged = currentIteration ;
}

void ReactiveSearch::decreaseProhibition ()
{
  const int newProhibition = int(prohibition * decrease);
  if ( prohibition == newProhibition )
    prohibition--;
  else
    prohibition = newProhibition;
  if ( prohibition < 1 )
    prohibition = 1;
  tProhibitionChanged = currentIteration ;
}
```

67b ⟨*ReactiveSearch - check if the problem description is consistent* 67b⟩≡

```
bool ReactiveSearch::problemConsistency ()
{
  int i;
  for ( i = 0; i < nbits && problem[i] == history[i]; i++ );
  return i == nbits;
}
```

Appendix B

Web Radio Tutorial

As a side product of our research, some algorithms have been developed to effectively test the techniques proposed in this work.

These programs have been subsequently ported to Java and they have been given a graphical interface in order to be used and appreciated by other people. These algorithms have been included in an WWW-accessible HTML framework.

These web pages can be found at the URL

`http://www.science.unitn.it/~brunato/radio/`

They are organized as a fast tutorial on the problem of channel assignment in cellular networks. As a result of this, our work has been more visible, and interaction with some researchers around the world has been possible.

B.1 Tutorial index

Most of the information contained in the tutorial is slightly outdated, due to the lack of spare time dedicated to it.

1. **Introduction** — a fast introduction to the central concepts of cellular networks and to Channel Assignment in particular.
2. **Mathematical models of the problem** — hexagonal tilings, graph coloring.
3. **Common algorithms** — The various combinatorial algorithms: Fixed Allocation, browsing techniques, dynamic assignment.
4. **Comparison among algorithms** — a Java applet lets the user compare the refusal probability of three major techniques at various traffic rates and at different numbers of available channels. Its use is discussed in section B.2.1.
5. **Research algorithms** — a resume of the objective-function approach shown in chapter 4 and a fast (*very* fast!) overview of some local search heuristics.
6. **Related problems** — the local broadcasting and local token passing schemes shown in chapter 5.

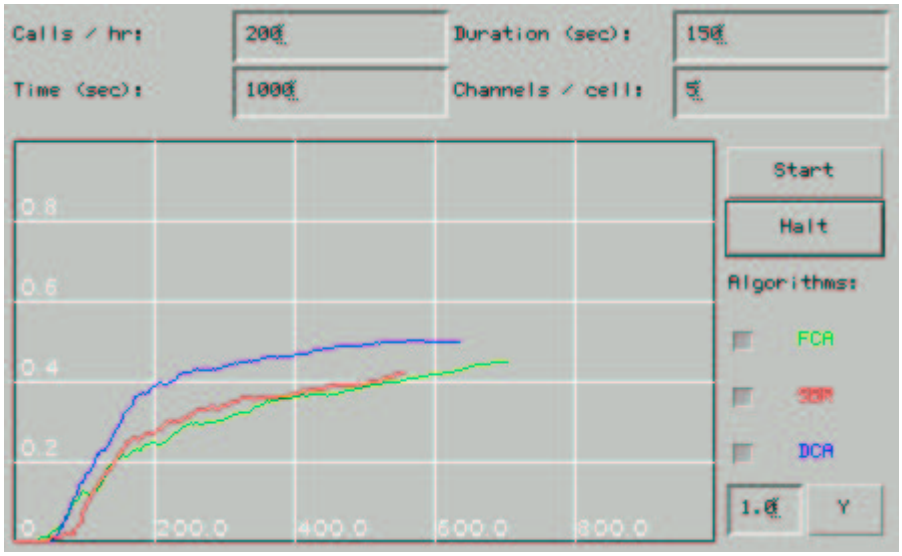

Figure B.1: The algorithm comparison applet

(a) **Local broadcast** — we report the algorithm in section 5.2 with explanations about it. We also provide a simple Java applet that allows the user to experiment with various start configurations and with step by step simulation. See section B.2.2 for details about the applet.

(b) **Local mutual exclusion** — The multiple token passing algorithm of section 5.3 is illustrated, then a Java applet allows the visitor to check the performance of the algorithm. The applet is explained in section B.2.3 and on the page itself.

7. **Ongoing work** — a yet-to-come chapter about the current state of the work.

8. **Bibliography**.

B.2 Java Applets

B.2.1 The Algorithm Comparison Applet

The applet, which can be seen in figure B.1, shows a graph panel where the blocking probability is reported against execution time for three different algorithms: FCA, SBR and a simple DCA technique with random channel choice (see chapter 2). Other more complex techniques, such as BDCL or BBB (see chapter 4) could be implemented, but their execution time would be too slow on an average machine, due to the unoptimized execution time of most Java virtual machines.

The three algorithms can be selected for execution and some parameters can be set: the number of call requests per hour, the overall number of channels, the simulation time and the average call duration.

The simulation can be started and stopped at will, and the vertical scale can be adjusted by typing the maximum Y value in the bottom right text window and then

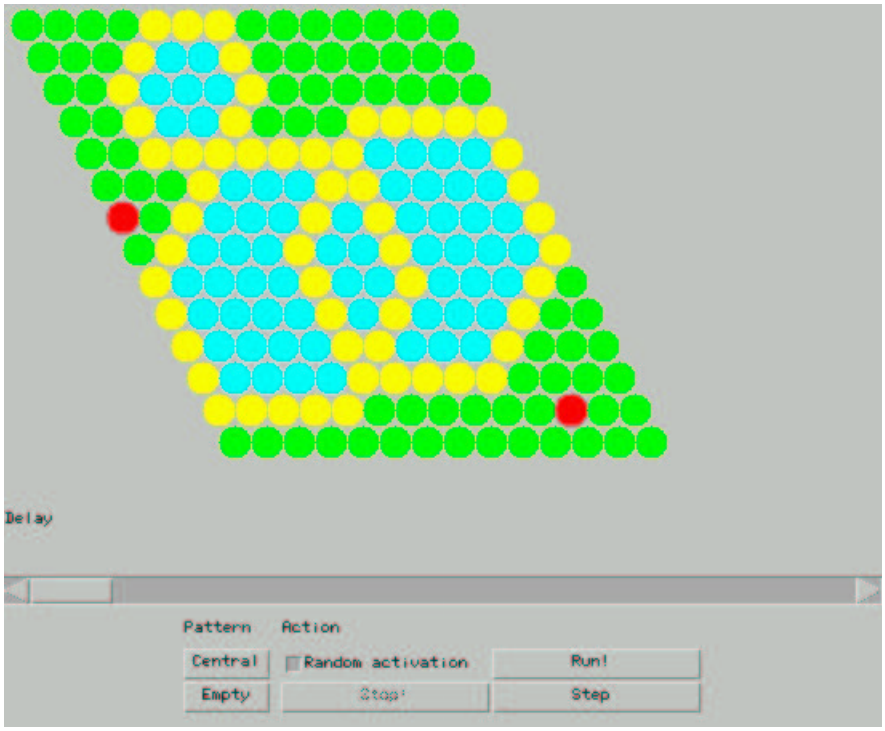

Figure B.2: The local broadcast test applet

pressing the “Y” key.

B.2.2 The Local Broadcast Test Applet

In figure B.2 we can see a snapshot of the simulation of the local broadcast algorithm of section 5.2 (see figure 5.4 at page 34) on a hexagonal grid.

All cells are initially green (empty). Clicking on a cell makes it the initiator of a broadcast. Hitting the button ”Step” forces an algorithm step at every cell. A cell becomes yellow when it has received a message and needs to process it; it is cyan when it has received and eventually forwarded the message according to the algorithm. The button ”Run” makes the system run continuously with sequential or random activations of the cells, depending on the status of the “Random Activation” checkbox.

More than one cell can be made a broadcast initiator at the same time: broadcasts can also intersect without interfering. The cells can be emptied and a start configuration with only a single initiating cell at the center of the board can be executed.

B.2.3 The Multiple Token Passing Test Applet

Finally, a multiple token passing test applet has been written to test the algorithm found in section 5.3; figure B.3 reports it with a regular configuration (the same that can be seen at page 39 repeated four times).

The board can be initialized with arbitrary token distribution by clicking on the cells, with a random configuration (in this case we can set a random seed to ensure

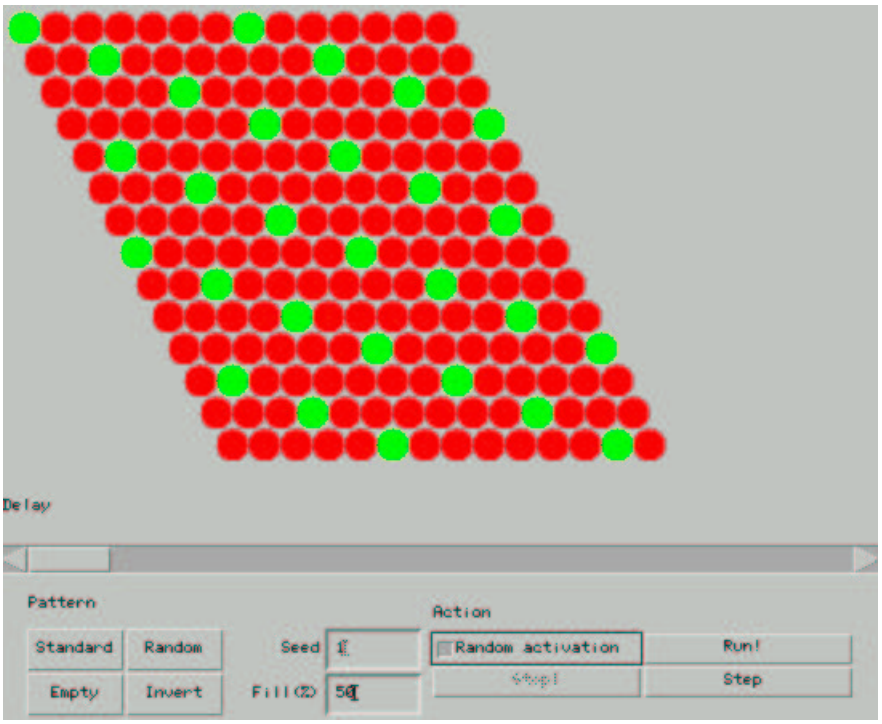

Figure B.3: The multiple token passing test applet

reproducibility and with a percent filling factor) or it can be filled with the regular distribution that can be seen in figure.

The execution can apply algorithm 5.9 in sequential order to all cells, or at a random order, to verify independence from execution order. The cell order can be set by switching the “Random Activation” checkbox.

By pressing the “Run” button, we see the results of continuous application of the algorithm (it can be stopped with the “Stop” button), while the “Step” button executes a single sweep over all cells of the system.

The system configuration can also be changed on the fly to show how the system responds to a failure. For example, one can start with a regular token distribution by pressing the “Standard” button. Then the system can be started with the “Run” button and left alone for a while. Then, if we push the “Stop” button we freeze the system. Choose a red cell (meaning that it possesses the token) and click on it to erase its token, thus emulating a failure on that cell. Re-run the system: it will soon lose every token by a cascade effect of missing messages. Now we can push again the “Stop” button in order to stop the algorithm (which at present is not doing anything). If we restore the token on the cell from which we had stolen it before by clicking again on it, then we press the “Run” button, we shall see the whole system coming back to life.

Bibliography

[Bat96] Roberto Battiti, *Reactive search: Toward self-tuning heuristics*, Modern Heuristic Search Methods (V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, eds.), John Wiley and Sons Ltd, 1996, pp. 61–83.

[BB95] Alan A. Bertossi and Maurizio A. Bonuccelli, *Code assignment for hidden terminal interference avoidance in multihop packet radio networks*, IEEE/ACM Transactions on Networking **3** (1995), 441–449.

[BBB97] Roberto Battiti, Alan A. Bertossi, and Mauro Brunato, *Cellular channel assignment: Comparing and simplifying heuristics*, Proceedings of the IEEE/ACM Workshop DIAL M for Mobility (Budapest), 1997, pp. 19–28.

[BBB99] Roberto Battiti, Alan A. Bertossi, and Maurizio A. Bonuccelli, *Assigning codes in wireless networks: Bounds and scaling properties*, Wireless Networks **5** (1999), 195–209.

[BBB00] Roberto Battiti, Alan A. Bertossi, and Mauro Brunato, *Cellular channel assignment: a new localized and distributed strategy*, to appear on Mobile Networks (2000).

[BF96] Onelio Bertazioli and Lorenzo Favalli, *Gsm — il sistema europeo di comunicazione mobile: Tecniche, architettura e procedure*, ATES, Hoepli, Milan, 1996.

[BT94a] Roberto Battiti and Giampietro Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), no. 2, 126–140.

[BT94b] ———, *Simulated annealing and tabu search in the long run: a comparison on qap task*, Computers and Mathematics with Applications **28** (1994), no. 6, 1–8.

[DAKR93] Manuel Duque-Antón, Dietmar Kunz, and Bernhard Rüber, *Channel assignment for cellular radio networks using simulated annealing*, IEEE Transactions on Vehicular Technology **42** (1993), no. 1, 14–21.

[DFR96] Enrico Del Re, Romano Fantacci, and Luca Ronga, *A dynamic channel allocation technique based on hopfield neural networks*, IEEE Transactions on Vehicular Technology **45** (1996), no. 1, 26–32.

[ES84] Eli Upfal Eli Shamir, *Sequential and distributed graph coloring algorithms with performance analysis in random graph spaces*, Journal of Algorithms **5** (1984), 488–501.

[Glo89] F. Glover, *Tabu search — part i*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[Glo90] ———, *Tabu search — part ii*, ORSA Journal on Computing **2** (1990), no. 1, 4–32.

[Glo94] ———, *Tabu search: Improved solution alternatives*, Mathematical Programming, State of the Art 1994 (J. R. Birge and K. G. Murty, eds.), The University of Michigan, 1994, pp. 64–92.

[HJ90] Hansen and B. Jaumard, *Algorithms for the maximum satisfiability problem*, Computing **44** (1990), 279–303.

[JS96] Scott Jordan and Eric J. Schwabe, *Worst-case performance of cellular channel assignment policies*, Wireless Networks **2** (1996), 265–275.

[KN96] Irene Katzela and Mahmoud Nagshineh, *Channel assignment schemes for cellular mobile telecommunication systems: A comprehensive survey*, IEEE Personal Communications (1996), 10–31.

[Knu92] Donald Ervin Knuth, *Literate programming*, CSLI Lecture Notes, no. 27, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

[Lin65] S. Lin, *Computer solutions of the travelling salesman problem*, BSTJ **44** (1965), no. 10, 2245–2269.

[Occ99] Gianluca Occhetta, *Extremal rays of smooth projective varieties*, PhD Thesis Series, Dipartimento di Matematica, Università degli Studi di Trento (1999), no. 27.

[SB96] Satinder Singh and Dimitri Bertsekas, *Reinforcement learning for dynamic channel allocation in cellular telephone systems*, Submitted to NIPS96 (1996).

[SW68] K. Steiglitz and P. Weiner, *Some improved algorithms for computer solution of the travelling salesman problem*, Proceedings of the Sixth Allerton Conference on Circuit and System Theory (Urbana, IL), 1968, pp. 814–821.

[YY94] Kwan Lawrence Yeung and Tak-Shing Peter Yum, *Compact pattern based dynamic channel assignment for cellular mobile systems*, IEEE Transactions on Vehicular Technology **43** (1994), no. 4, 892–896.

[ZY91] Ming Zhang and Tak-Shing Peter Yum, *The nonuniform compact pattern allocation algorithm for cellular mobile systems*, IEEE Transactions on Vehicular Technology **40** (1991), no. 2, 387–391.