2

Constraints Satisfaction Problems - CSPs

Constraints Satisfaction Problems (CSPs) have been studied intensely since the early 1980s, as a paradigm on their own. The first important papers on certain fundamental properties of Constraints Networks were by Montanari and by Freuder [21]. Papers on search algorithms were published by Elliot and Haralick [26] and later by Pearl and Dechter [15]. The body of work accumulated during the 1980s has driven a community of researchers and a special track on constraints-based reaoning in all important AI conferences. The field became known as CSP and the next milestone was a paper by Patrick Prosser, which formalized a large family of search algorithms for CSPs [51]. Several search algorithms that fall under the term *Intelligent Backtracking* were proposed early on. These include ideas like Lookahead [26] and BackJumping. Some of these algorithms were proposed in the general AI search context. As we will see, CSPs provide a well-defined search space definition, which enables exact formulations of all search algorithms. This makes the field an excellent laboratory for formulating new and complex ideas for search strategies. For the same reason, *distributed* CSPs are a fundamental formalism for advancing the whole field of distributed search.

The paper by Prosser [51] succeeded in defining a uniform structure for the different algorithms that were proposed in a variety of forms during the previous decade. The main result of [51] was a large set of hybrid algorithms that were combined from fundamental ones by using the uniform structures of the paper. The main core of search algorithms for CSPs are termed nowadays on the basis of the terminology of [51]. These are the Backjumping (BJ) algorithm and Conflict-based Backjumping (CBJ), and Forward-Checking (FC), which is the simplest lookahead algorithm (cf. [26, 51]) and their best combination, FC-CBJ. In the following sections we will present these algorithms briefly, to set up a basis for the distributed algorithms that use versions of the same ideas.

2.1 Defining CSPs

Introductory texts on Constraints Satisfaction Problems (CSPs) can be found in [16, 58] and the basic papers are [15, 51]. A CSP is a well-known NP-complete problem [16], which is often used to represent and solve problems such as timetabling [41, 43], meeting scheduling [60] and a variety of other scheduling problems [12]. Formally, a CSP is a tuple $< X, D, R >$, where X is a finite set of variables X_1, X_2, ... , X_m, and D is a set of domains D_1, D_2,... , D_m. Each domain D_i contains a finite set of values which can be assigned to variable X_i. R is a set of relations (constraints) that specify for each value $v_j \in D_i$, the set of allowed combinations for it to be assigned to variable X_i. More formally, constraints or **relations** R are subsets of the Cartesian product of the domains of the constrained variables. For a set of constrained variables $X_{i_k}, X_{j_l}, ..., X_{m_n}$, with domains of values for each variable $D_{i_k}, D_{j_l}, ..., D_{m_n}$, the constraint R is defined as $R \subseteq D_{i_k} \times D_{j_l} \times ... \times D_{m_n}$. A **binary constraint** R_{ij} between any two variables X_j and X_i is a subset of the Cartesian product of their domains - $R_{ij} \subseteq D_j \times D_i$.

An assignment (or a label) is a pair $< var, val >$, where *var* is a variable and *val* is a value from the domain of *var* that is assigned to it. A *compound label* is a set of assignments of values to a set of variables. A **solution** P to a CSP is a compound label that includes all variables, and which satisfies all the constraints [16, 58].

Constraints can be either explicit, in the form of a listing of all forbidden assignments combinations, or implicit. An implicit constraint can be, for example, a set of rules that the assignments must follow. For example, $X_1 > X_2$ can be such a constraint if the value domains are numeric. The most commonly used constraints are binary constraints. A CSP with only binary constraints is called a binary CSP. A CSP can have more than a single solution. A CSP may also have no solutions at all, in which case it is declared to be unsolvable.

In Figure 2.1 we have two CSPs. Each CSP has four variables X_1, X_2, X_3, X_4. The domains of values of all variables contain the three values: r, g, b (not shown). Constraints are represented by lines connecting variables. As we can see, all constraints are binary. All constraints are inequality constraints ($\neq$). The CSP on the left-hand side is solvable. The assignment $(X_1 = r, X_2 = g, X_3 = b, X_4 = r)$, for example, is a solution. There is more than one solution to this CSP, for example $(X_1 = g, X_2 = r, X_3 = b, X_4 = g)$ is also a solution. In contrast, the CSP on the right-hand side (RHS) is unsolvable. There does not exist an assignment of values to all variables that satisfies all constraints. One can say that the CSP on the RHS of Figure 2.1 is overconstrained.

The size of the search space for solving CSPs is exponential in the number of variables, in the worst case. The base of the exponential is the domain size of variables (e.g., the number of possible assignments per variable). This led researchers to propose methods that have the potential to find equivalent versions of the problem that have smaller domains of values. The general idea is to check values of the CSP and remove those values that can never be

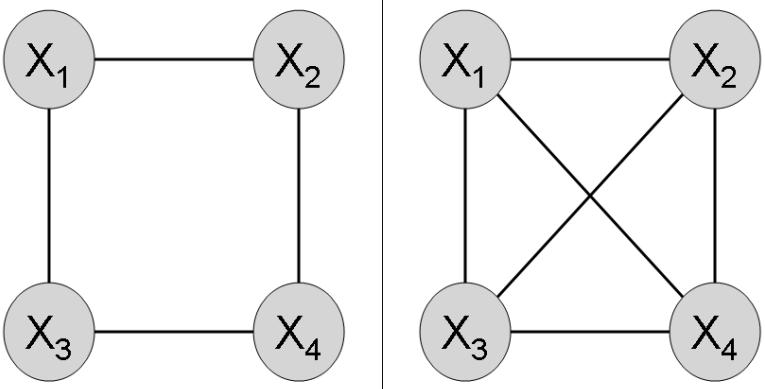

Fig. 2.1. A solvable CSP (LHS) and an unsolvable CSP (RHS)

part of a complete assignment that is a solution to the CSP. If such values are removed from the domains of their variables, one achieves two goals. On the one hand, the number of solutions to the problem remains exactly the same. On the other hand, some of the domains of values become smaller and thus the search space is reduced. The operation of removal of values that are inconsistent (e.g., cannot participate in any solution) is termed achieving local consistency. It is important to note that during a process of achieving local consistency the domain of a variable may become empty. In such a case the problem has been proved to be unsolvable [16]. One can say intuitively that in such a case a real saving of computation has been achieved, instead of exponential in the number of variables just exponential in the number of variables in a local neighbourhood (related to the specific local sonsistency that was used).

Consider the problem in Figure 2.2. There are three variables, each with its domain of values and all constraints are inequalities (e.g., a coloring problem). The value R of the variable A at the top of the graph is not compatible with any of the values of the variable B, at the bottom left of the graph. One can say that if the value R is removed from the domain of A, the resulting problem will have exactly the same solutions as the original problem. The process of removal of values that do not participate in solutions of a CSP is

based on checking consistencies. In our example the consistency is that of pairs of values in constrained variables. The constrained pair of variables (A, B) is checked for compatible pairs of assignments and (as we explained) the value R in A is found to have no compatible value in the domain of variable B. This particular type of check is called *Arc Consistency*, as it checks consistency over "arcs" of the constraints graph (for extensive discussion of arc-consistency see [6, 16, 58]). The process of enforcing arc-consistency can be done as a preprocessing phase, before a search for a solution to the CSP is initiated. It can also be performed at multiple stages during search. Consistencies can also be of higher degree than a single arc. One could check, for example, triplets of variables for consistent assignments [16]. In our description of distributed constraints we will refer to ideas of enforcing local consistency during search in a similar manner in which they are used in centralized CSPs (cf. [31, 51]).

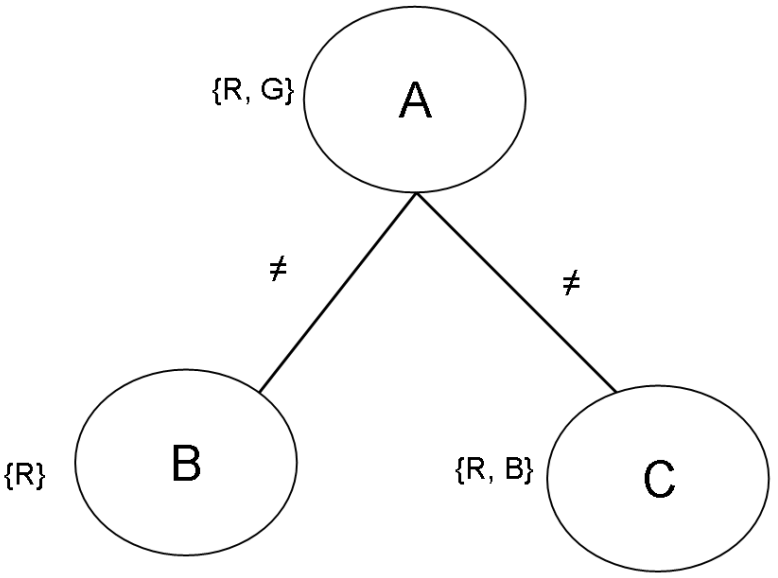

Fig. 2.2. A graph-coloring (solvable) CSP that is not arc-consistent

2.2 CSP Algorithms and Techniques

Practically all complete search algorithms for CSPs are based on the *backtracking* algorithm. In the backtracking algorithm, there are two possible steps,

an assignment step and a backtrack step. In an assignment step, the algorithm assigns a value to one of the variables, and checks that no constraint is violated (broken) due to this new assignment. If this is true, then the next step is another assignment step, or termination if all variables are assigned (a solution was found). If a constraint was violated, a backtrack step undoes the last assignment made and the next value in this unassigned variable's domain is assigned instead, if such a value exists. If the domain of the current variable is exhausted and there is no next value to try to assign, another backtrack step is taken. The backtracking algorithm is a depth-first search, on the search tree of possible value assignments to all variables. In the worst case, the algorithm requires exponential time in the number of variables, but only linear space [16, 31, 58].

Backtracking algorithms can be improved in two general ways. One way is to select a variable to backtrack to, that is not necessarily the last variable to have been assigned [31, 51]. The other way is to prune the search space by the use of lookahead methods. Lookahead methods were proposed by Haralick and Elliot in [26], categorized by depth, and will be described in more detail below.

Other methods of selective backtracking involve the maintenance of NoGOODs and explanations [23]. The family of algorithms that prune the search space by selective backward moves is commonly termed *backjumping*. These algorithms include simple backjumping (BJ), conflict-based backjumping (CBJ) [51] and Dynamic Backtracking (DBT) [23]. They have been shown by [31] always to visit no more nodes in the search space than simple backtracking.

The simplest form of lookahead, as analyzed by Haralick and Elliot, was later termed Forward Checking (FC) [51]. When the search process performs an instantiation (assignment) of a variable, it looks ahead towards the future (unassigned) variables, and removes from their current domain values that are incompatible with the tested instantiation. If the current domain of some future variable becomes empty, the combination of the assignments made so far would conflict with all value assignments of that variable, and thus the current state is inconsistent, and a backtrack is performed [51]. The current partial assignment cannot be extended to a solution of the CSP because the domain of some variable became empty and no assignment of this variable is compatible with the current partial assignment (cf. [31]). The current partial assignment is called a NOGOOD and the notion will be used extensively in all DISCSP algorithms.

When backtracking, it is important to undo all the value eliminations that were performed because of the trial instantiation. This is an immediate overhead of all lookahead algorithms (such as FC). The goal of the FC algorithm is to fail early by detecting inconsistencies as early as possible, thus saving exploration of dead ends. Forward checking performs more computation per assignment than the standard backtracking algorithm. The hope is that this will pay off by performing less assignments overall during the search.

Forward checking is the simplest method of *maintaining local consistency* (i.e., checking that the current state of the unassigned part of the search space is consistent). Exactly what is consistent is defined by the method used. Besides FC, many other methods for maintaining local consistency of the unassigned part of the search space exist. One can define two types of local consistency that can be used to induce a stronger lookahead than forward-checking.

- **Arc consistency**. The assignment $X_i = a$ is arc-consistent with respect to constraint C_{ij} (the constraint between X_i and X_j) if there is a value $b \in D_j$ such that $X_i = a, X_j = b$ satisfy C_{ij}. Such a value b is called a support of a. Variable X_i is arc-consistent if all its values are arc consistent with respect to every binary constraint involving X_i. A CSP is arc-consistent (AC) if every variable is arc-consistent.
- **Directed arc consistency (DAC)**. A variable X_i is DAC if all its values are arc-consistent with respect to every binary constraint involving X_i and X_j where $j > i$. This is a directed consistency and the check for existence of support values is only performed in one direction of the constraints. A CSP may be DAC for one ordering of the variables but not for another. Obviously AC is a stronger property than DAC.

Arc-inconsistent values can be removed, because they cannot participate in any solution. Enforcing AC or DAC does not guarantee that the CSP contains a solution. However, if a CSP is not AC or DAC, then it can be transformed into an equivalent CSP that is AC or DAC and that contains all the solutions of the original CSP [16]. AC (or DAC) can be achieved by removing arc inconsistent values until a fixed point is reached. If enforcing AC (or DAC) yields an empty domain, the problem is proven to be unsolvable. A major advantage of transforming the CSP into an equivalent AC (or DAC) CSP is that the resulting equivalent CSP may be of smaller size (e.g., its domains contain fewer values). There are several AC-enforcing algorithms. Two well-known algorithms are: AC3 [1] and AC2001 [5].

It is important to note that stronger methods for maintaining consistency do not always produce better performance. A tradeoff exists between the number of assignments performed, and the computational effort following each assignment. Enforcing stronger consistency may lead to fewer assignments done, as fewer dead-ends will be explored. However, to check and maintain consistency, some computational effort is required following each assignment.

We will refer back to these methods later on, when discussing how to maintain local consistency in optimization problems. Obviously the present exposition does not come near to summarizing all CSP algorithms. We choose to mention only what is most relevant for distributed search. For more information, one can consult the books by Dechter and Tsang [16, 58].

2.3 Behavior of CSP solving algorithms

CSPs are NP-complete problems. Therefore, evaluating the performance of algorithms by theoretical measures will not be of much use. In order to evaluate CSP solvers, empirical evaluation is often used. Performance is measured over a predefined set of problems. In order not to rely on implementation details that might influence results (such as implementation efficiency, processor speed, background processes running, etc.) performance is evaluated using a logical measure. The number of constraint checks (CCs) is one such measure that is frequently used. Another is the number of assignments made [31, 52].

The above two measures relate to the search process and search space, rather than to the specific implementation. A search algorithm that belongs to the backtracking family is a depth-first procedure. It assigns variables sequentially, checking each assignment in turn for consistency with former assignments [16]. Algorithms differ in their decision of how to backtrack (e.g., of forms of *backjumping*) or in their forms of checking the consistency of unassigned variables (e.g., *lookahead*). The main part of computation that is common to all search algorithms is to check consistency against the current partial assignment. This computation is composed of a series of checks of pairs of assignments (for binary CSPs). Each such check can be thought of as an $O(1)$ operation, accessing the constraint matrix to find whether a pair of values are compatible. By counting these operations all CSP search algorithms can be measured on a uniform scale.

The number of assignments performed by a search algorithm is also an implementation-independent measure. Here one considers the trajectory of the algorithm in the search space. Assigning variables one by one, any search algorithm checks for consistency and then proceeds to the next variable. When it gets to a dead-end it backtracks and each reassignment of a variable is counted by this measure. It is clear that the number of assignments will be lower for algorithms that prune the search space efficiently. However, if pruning the search tree is based on additional computations at each node visited, the resulting procedure does not necessarily perform fewer computations. In other words, it is an interesting question whether a lookahead CSP search algorithm will be faster. It has been shown theoretically that forward checking (FC) performs no more assignments than simple backtracking [31]. Independently, it has been shown *empirically* that FC outperforms BT on randomly generated CSPs.

It is of interest to investigate the dependency of a random problem's difficulty on the parameters of the problems. The number of variables and the domain size obviously influence the problem difficulty, as the search space grows exponentially with these two parameters. However, even for a fixed number of variables and a fixed domain size, the problems vary wildly, due to the nature of the constraints. For a fixed-size problem, one problem can be extremely easy to solve, if there are almost no constraints, while another may be highly constrained and require more effort until a solution is found. Over the

last decade a uniform set up for testing CSP algorithms has been established. Experiments are commonly performed on randomly generated binary CSPs. The problems are randomly generated using four parameters: $< n, k, p_1, p_2 >$, where n is the number of variables, k is the uniform domain size, and p_1 is the probability of a constraint existing between any two variables X_i and X_j. If X_i and X_j are constrained, any two value assignments $a \in D_i$ and $b \in D_j$ are forbidden with probability p_2 [52]. p_1 is called the **constraint density**, and p_2 is called the **constraint tightness**.

When evaluating algorithms on such randomly generated problems, it was discovered that one of the four parameters is a critical parameter. When three of the parameters n, k, and p_1 are held fixed, the critical parameter is p_2 (the *tightness*). When p_2 varies between 0 and 1 the problem difficulty exhibits a **phase transition**. The average difficulty of problems goes from easy, to difficult, to easy. When measuring the run time of the algorithm by counting the number of constraint checks, the run time varies with the value of the tightness p_2 through three regions - low-high-low. All algorithms finish their execution very fast for very low values of p_2 (only a handful of value combinations are forbidden and most full assignments are in fact solutions). Run time increases exponentially with increasing p_2 up to some peak, after which it starts to decrease. Figure 2.3 shows the phase transition for $n = 20$, $k = 10$, $p_1 = 1.0$ (presented in Prosser [52]). Figure 2.4 presents the phase transition for similar random problems with a density of $p_1 = 0.5$, the phase transition peak is at roughly $p_2 = 0.4$ (for these parameters).

To the left of the left vertical bar in Figure 2.5 ($p_2 = 0.35$) all problems are solvable, to the right of the right vertical bar ($p_2 = 0.41$) all problems are unsolvable. The area between the two vertical bars contains both solvable and unsolvable problems, this area is also called the "mushy region" by [52, 56]. It is in this region that the average search effort is maximal. The algorithm used, for Figure 2.5 is an enhanced variation of the forward-checking algorithm.

The existence of the phase transition is explained intuitively as follows. For low values of the tightness p_2, the constraints are very loose and the problems are easy. Increasing the constraint tightness naturally increases the difficulty. As fewer full assignments become solutions, it becomes increasingly hard to find a full assignment that is consistent (a solution). When the value of p_2 is high enough, the problems become easier, in that the algorithm does not need to search the entire exponential search space. The problems are overconstrained and have no solution. The algorithm reaches an inconsistent state higher up the search tree, thus pruning more of the search space.

The phase transition (mushy region) at the center of the peak contains a mix of solvable and unsolvable problems. Intuitively these are problems with either few solutions (so they are hard to find and require searching through most of the search space to reach), or unsolvable problems (that are consistent up until a very late stage of the assignment process, thus forcing the search process to explore most of it before declaring there is no solution).

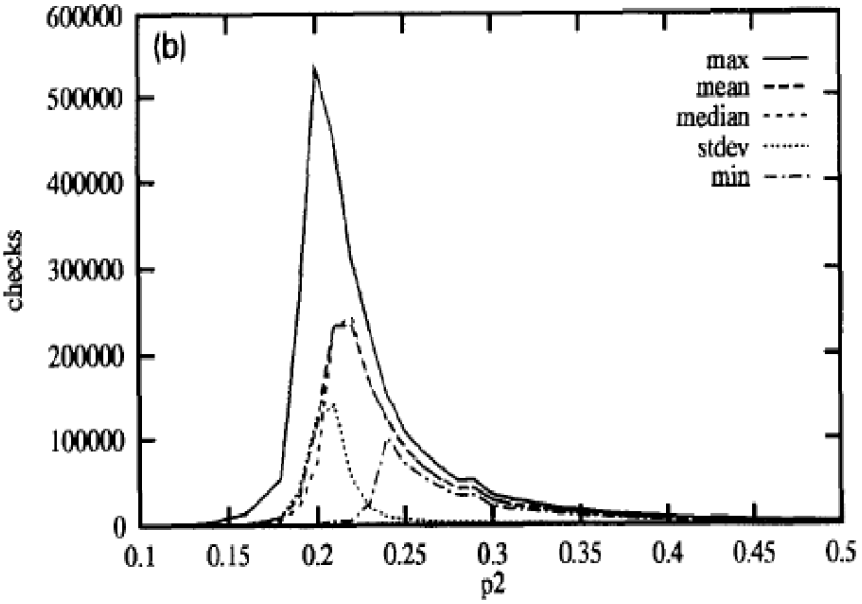

Fig. 2.3. The phase transition for $p_1 = 1.0$ (from [52]).

The phase transition of problem difficulty is very important. It was also discovered for distributed CSPs and can serve as a very good feature for checking the validity of performance measures in the distributed domain. Furthermore, the exponential growth in difficulty of finding the first solution to a CSP is an excellent differentiating criterion among search algorithms. In order to achieve a substantial gain in efficiency, algorithms need to be tested on hard problem instances. For randomly generated CSPs these are at p_2 values that are near the critical value (e.g., near the peak). Throughout this book when the performance of algorithms is compared we will use random CSPs and observe the different behavior of the algorithms being compared for hard problem instances. It is for these problems that one wants to design a more-intelligent algorithm and a better heuristic (see Chapter 11).

It is also important to mention that a similar phenomenon has been also discovered for some constraint optimization (COP) and distributed constraints optimization (DisCOP) *algorithms*. The important difference is that a phase transition exists for CSPs for all algorithms. For constraint optimization problems (COPs) a phase transition was found only for certain lookahead search algorithms [33, 36]. Chapter 3 describes constraints optimization problems (COPs) and algorithms for finding an optimal solution. These algorithms belong to the Branch and Bound family, but use additional lookahead tech-

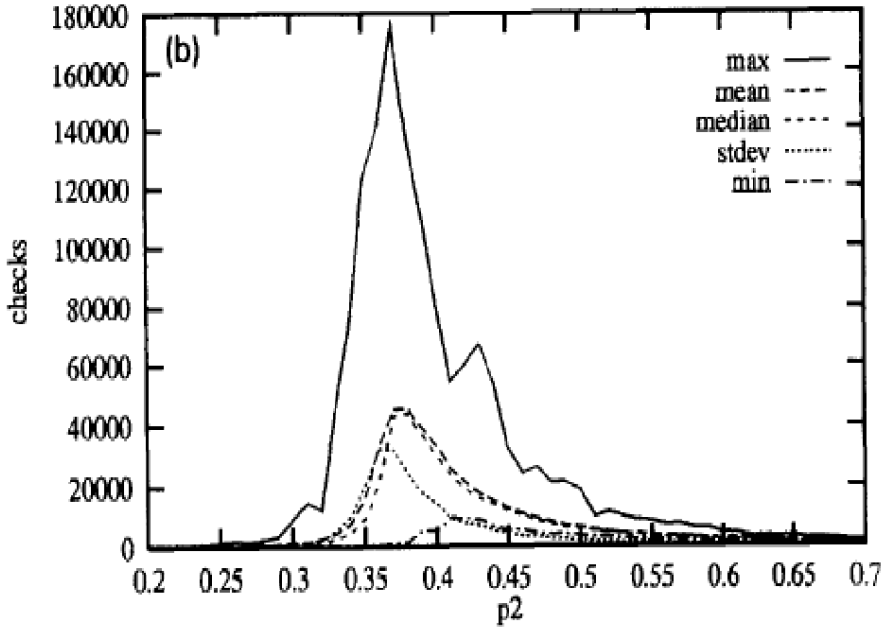

Fig. 2.4. The phase transition for $p_1 = 0.5$ (from [52]).

niques. The deeper methods of lookahead turn out to produce a specific form of a phase transition for COPs [33]. Chapter 3 presents the latest results on COP algorithms. Chapter 16 presents the first distributed optimization algorithm that demonstrates a phase transition for *Distributed COPs* [22]. As will be shown in Chapter 18, standard asynchronous distributed optimization algorithms do not produce a phase transition for hard instances of DisCOPs [22].

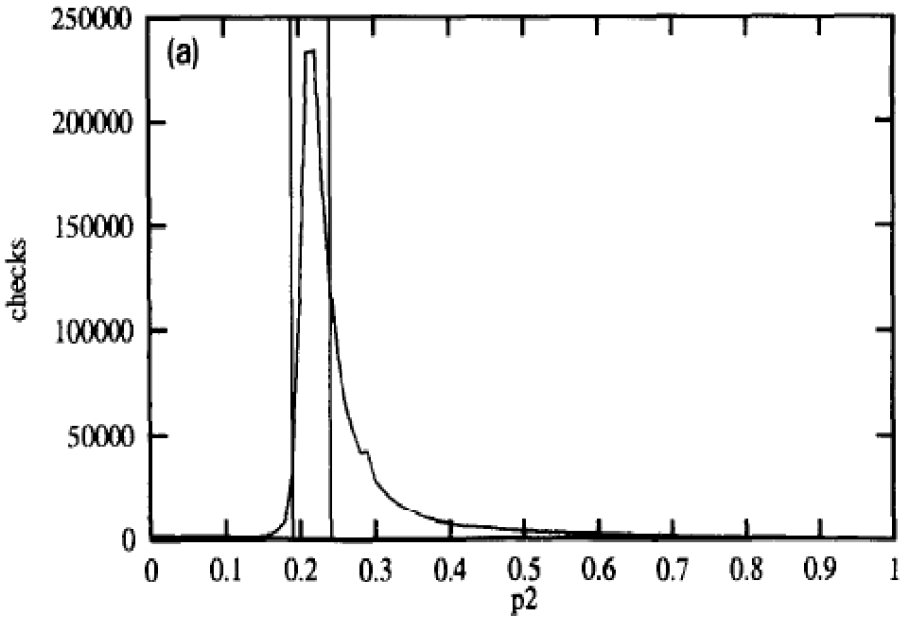

Fig. 2.5. The phase transition region for $p_1 = 1.0$

http://www.springer.com/978-1-84800-039-1

Distributed Search by Constrained Agents
Algorithms, Performance, Communication
Meisels, A.
2008, XX, 216 p., Hardcover
ISBN: 978-1-84800-039-1